Stochastic Simulation Optimization for Districting Problems with Application to Police Patrol District Design

A Dissertation

Presented to

the faculty of the School of Engineering and Applied Science

University of Virginia

in partial fulfillment

of the requirements for the degree

Doctor of Philosophy

by

Yue Zhang

August

2014

APPROVAL SHEET

The dissertation

is submitted in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

[illegible]

AUTHOR

The dissertation has been read and approved by the examining committee:

Donald Brown

Advisor

Gerard Learmonth

Matthew Gerber

John Porter

Matthew Trowbridge

Accepted for the School of Engineering and Applied Science:

[illegible]

Dean, School of Engineering and Applied Science

August

2014

Abstract

Districting or territory design is the procedure of partitioning small geographic units into larger areas in order to facilitate the operations performed over the region and optimize planning criteria under certain constraints, such as contiguity and compactness. The size of the possible solution space is large and the corresponding graph-partitioning problem is NP-hard. District design problems arise in a broad range of real-world applications such as political redistricting, sales and services territory alignment, school districts and emergency services. This dissertation develops a simulation-based optimization districting methodology for public services with stochastic demand and dynamic service operations. Stochastic inputs and constraints make the optimization problem difficult to solve. We apply the new approach to police patrol district design, which is one of the most important decisions that affect the effectiveness of patrol operations. The objective is to find the best districting plan so that the long-term average response time and workload variation can be minimized. A heuristic parameterized districting algorithm is developed to generate various district plans. The parameters in the seeds-growing algorithm can control the pattern or layout of the district plans.

In this dissertation, an Agent-Based (AB) simulation model and a Discrete-Event (DE) simulation model are built to evaluate districting plans generated by the parameterized districting algorithm. The AB model is developed using Java Repast in Geographical Information System (GIS) environment. It simulates the patrolling, dispatching and responding activities of police cars in actual road network and building locations of the city. The calls for services (CFS) incidents in the simulation are generated based on spatial and temporal patterns extracted from historical data. The high fidelity simulation provides good visualization for model validation but limits the computational efficiency. Inspired by the Hypercube Queuing Model, a Discrete-Event simulation model is developed using Java SE. It only keeps track of major events of police cars and thus improves the evaluation efficiency. A comparative study is conducted between the AB and DE simulation model.

In the simulation optimization procedure, three methodologies are developed to find the best district plan under simulation evaluations. In the Response Surface Methodology (RSM), the relationship between districting parameters in the districting algorithm and performance measures is studied using an iterative experimental design method for computer simulation. Fractional Factorial Design, Space Filling Design and Kriging Prediction are used to find optimal settings of important districting parameters. Good districting plans can be generated more efficiently and thus support the decision making process of police department. Another Simulated Annealing (SA) approach is developed based on an existing SA algorithm. The new approach defines the solution neighborhood by making relatively big changes on current plan, such as merging and cutting districts. Compared with the existing SA method, the new approach uses less iteration to converge and is more robust to deal with different adjacency patterns of geographical atoms. Genetic Algorithm (GA) is also applied to the optimization of police districting. The three simulation optimization methodologies are tested and compared in the application study of police districting of Charlottesville, VA, USA.

Acknowledgements

I would like to express the deepest appreciation to my academic advisor Prof. Donald Brown. Thank you for giving me such a wonderful opportunity to do research on Agent-Based Simulation and Simulation Optimization. Thank you for giving me direction in my Ph.D. study. I find myself improved a lot under your guidance. It is you that leads me to the research world. I have learned a lot from you and my Ph.D. study is one of the most valuable experiences of my life. I would like to thank my Ph.D. advisory committee members Prof. Gerard Learmonth, Prof. Matthew Gerber, Prof. John Porter and Prof. Matthew Trowbridge. Thank you for your advice and suggestion for my research work. I would like to thank Samuel Huddleston, one of our research group members. Thank you for your ideas and writing help for our paper. It was really a pleasant experience. In addition, I would like to thank my roommate Yue Sun. We had so many interesting talks and discussions, some of which inspired me in my research work. Last but the most importantly, I would like to thank my parents Maoxin Zhang and Bing Ling. Thank you for your support and encouragement during my Ph.D. study.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Introduction to Districting Problem

Districting problem, also referred to as territory design, is the problem of partitioning small geographic areas (atomic units such as census blocks, counties, zip code or company trading areas) into larger geographic clusters called districts or territories in order to facilitate the operations performed over the region and optimize planning criteria under certain constraints [65][51]. An exhaustive literature review on general district design problems can be found in [65]. District design problems arise in a wild range of real-world applications such as political redistricting, sales and services territory alignment, school districts and emergency services. Spatial constraints such as contiguity and compactness of the districts are common to these problems. Contiguity requires that the atomic geographic units in a district should be connected to each other. A district is geographically compact if it is round-shaped and undistorted [65]. Compactness of districts can reduce the possibility of gerrymandering in political districting and shorten unproductive travel time when designing sales and service territory. Other districting criteria and constraints are dependent on specific problems. For example, demographic criteria such as population and voter equality, minority representation should be considered in political districting. In commercial territory design problem, the number of customers and product demand should be balanced among all territories.

The districting problem can be modeled as a graph-partitioning problem. Atomic geographic units can be considered as nodes in a graph while the adjacency relationships of these atoms can be represented by the edges between nodes. A district design is a partitioning plan that divides the nodes into several sub-groups given node information such as location, population or customer demand depending on the context. The objective of the districting problem is to find the optimal graph-partition plan that has the best overall performances on all the criteria. The evaluation of district design is in a deterministic manner for some problems. For example, political districting uses a single function (e.g. standard deviation) of district attributes to measure the population equality criteria. However, in some districting problem related to public services and logistics, the demand for services is usually uncertain over the region and the activities related to service operations show stochastic pattern (e.g. vehicle routing and traveling). This dissertation focuses on the districting problem with stochastic inputs and constraints, and describes a methodology to evaluate and optimize the districting plan for police patrol operations, which is an important urban emergency service.

1.2 Police Patrol District Design Problem

1.2.1 Urban Emergency Services

Emergencies are incidents that pose immediate risks to public safety, health, life, welfare or environment. Most emergency incidents require urgent responses or interventions to mitigate the impact of these incidents and prevent worsening of the situation [1]. If emergencies are severe or prolonged, they may exceed the capacity of first responders, law enforcement officials or other emergency response units [5]. Operations Research (OR) methodologies are widely used in urban emergency services. The OR researches of the emergency response services can be grouped into four major categories: 1) urban services (police patrol, firefighting services and emergency medical services, et al.), 2) disaster services (disaster evacuation & rescue) and 3) specific hazards (hurricane, flood, earthquake et al.) and 4) general emergency [55].

Most urban emergency services are provided through the use of emergency vehicles (e.g. fire engines, ambulances and police cars). Fire engines wait at fire stations until there is a call for service. After completing the service, fire engines return to base to wait another call [17]. Unlike fire engines, ambulances often wait at rudimentary location (e.g. parking lot) for calls for services. They are also periodically relocated to insure a good coverage at all times. Sometimes ambulances responding to a call may be diverted to a more important call [24]. Police cars regularly perform preventive patrol tasks since their presence on city streets can deter potential crimes. After responding to calls for services incident, they return to their beats and continue preventive patrols. Different cities have different temporal and spatial pattern of calls for emergency services. The allocation strategy of emergency vehicles should match the characteristics of the cities and is the essential problem faced by emergency service officers. This study focuses on spatial allocation of urban emergency response resources, especially in police patrol services. The major methodology is to use agent-based and discrete-event simulation to model the emergency response system, and then find optimal or sub-optimal solution of resource allocation strategy through simulation optimization procedure. The procedure and application details are fully provided for police patrol system. Similar ideas can be applied for firefighting system and emergency medical services.

1.2.2 Introduction to Police Patrol

Police patrols play an important role in public service by responding to incidents, deterring and preventing crimes. It can give a sense of security to people who need protection and discourage those who may commit crimes in the absence of a patrol [89]. Police patrolling is an indispensable component and function of police departments [104]. However, since police resources are limited, there is an understandable interest in patrol strategies and operations that provide safety at minimum cost. Typically, a city is partitioned into command districts or precincts. Each command district usually has a headquarters and a commanding officer to manage and supervise the police patrol operations. Each command district is further divided into several beats or sectors [33]. Police can effectively manage their operations through the design of the command districts and the choice of patrol strategies of the police units within those districts. Basically, there are three major types of patrol strategies for patrol officers: active patrol, random patrol, and directed patrol. Patrol officers may choose one strategy or combine them to accommodate the specific conditions in their area [54].

- **Active Patrol**

  The patrol officer should use every available opportunity to discover, detect, observe, and interdict the unusual event so that they can predict and prevent the crime.

- **Random Patrol**

  It is also referred to as varied patrol. The patrol route should be random and varied so that it cannot be predicted by the potential criminals.

- **Directed Patrol**

  The patrol officer should pay more efforts to those areas where crimes or problems are most likely to occur. The increase of police patrol in hot spots can lead to significant reductions in the number of crimes committed in these same areas.

Another important patrol principle is beat integrity. Patrol officer is expected to remain within assigned patrol district. Beat integrity can be absolute or relative, depending on the number of patrol units assigned, the size of the patrol district and the activity within the area [54]. For absolute beat integrity, patrol officers should remain in the assigned patrol district at all times. In relative beat integrity, patrol officers remain largely within the patrol area and leave it only for good reason, such as back up another officer or respond to a call for service (CFS) incident in another patrol district [54]. When there is a CFS incident, many police departments use the principle that "the nearest police car responds to the call", which belongs to the case of relative beat integrity. The police car may cross the patrol boundary to respond an incident.

Police patrol district design or region design is one of the important resource allocation strategies of police department. The deployment of patrol force through designated patrol districts is a standard management method to enhance the deterrent capability of uniformed patrol force [54]. Better districting plans can lead to lower response times, officers' familiarization with their assigned area, more efficient utilization of personnel, equalized distribution of workload, uniform police visibility, enhanced officer safety, balanced police response to calls and officer accountability [54]. For small and medium-sized cities, patrol district design determines the patrol boundaries for each patrol unit, for example, police car. For large cities, patrol region design usually starts from higher level of resource allocation. Larger patrol districts for several patrol units are determined first then these districts are further divided for each patrol unit. Patrol districts should be designed in a rational and systematic manner [54]. Several factors need to be considered when designing patrol districts: 1) the size of the area to be patrolled should not exceed the limit that a single officer can cover, depending on types of the area (urban, suburban or rural), 2) natural or man-made barriers, (e.g., rivers, railroad tracks, interstate highways, valleys), 3) workload indicators (e.g., CFS, criminal incidents, traffic patterns), 4) significant area characteristics (e.g., local neighbor hoods, major shopping centers) [54]. Different cities have different CFS and crime pattern. CFS incidents or crime events are more likely to happen in some area than other areas. So, the police patrol resources should be spatially allocated based on the characteristics of cities. The size, shape, demographics, and geography of the patrol district are major determinants of the effectiveness of patrol operations and is the focus of this paper.

1.2.3 Performance Measures of Police Districting

There are two common performance measures used for police district design: the average response time and the variation of workload [33]. Quick response to citizen calls for service can 1) improve the chances of catching the offender at the scene or nearby 2) increase the chances of identify and locate witnesses, 3) provide immediate gathering of physical evidence, 4) provide immediate lifesaving first aid, 5) enhances the reputation of police department, 6) creates citizen satisfaction with the police [31][55]. The spatial distribution and allocation of police cars to districts affect both performance measures. In a 1971 study of the New York Police Department, more than half of all dispatches were inter-sector dispatches (between districts). Usually, the nearest police car responds to the CFS incident so it may cross the patrol boundaries to respond. The average response time for these inter-sector responses was approximately 40% greater than that for intra-sector responses and the average travel distance was about 53% greater [71]. This large proportion of inter-sector cases indicates that the district design may not be efficient with regard to the first measure, response time.

Our own interviews with members of the Charlottesville, VA Police Department have shown that the response times do not vary much between or within districts. This is because Charlottesville is small city (approximately 7 miles in diameter) with a population of about 40,000. However, these interviews did reveal large variations in the workloads of officers in different districts. This shows a lack of efficiency with respect to the second objective: variation in workload.

1.2.4 Scope & Objectives

Typically, for a small and median-sized city (e.g.Charlottesville, VA, USA), the city area is directly divided into several patrol districts and one patrol unit can conduct patrol activities autonomously in its district. Under such background, this study mainly develops a systematic approach of looking for optimal or sub-optimal patrol boundaries for each patrol unit to minimize the long-term average response time and workload variation among patrol units. For large cities, (e.g. New York), the methodology described cannot be applied directly to the whole city because the first task for central police department is usually to allocate police resources to several command districts (e.g. Bronx, Brooklyn, Manhattan, et al.). However, if given the police resources (e.g. number of police cars) for each command district, each district can be treated as a small city and the methodology described can be applied. In other words, this study focuses more on the micro-level patrol district design for each patrol unit rather than the macro-level command district design and resource allocation for large cities. So, the objective is to find optimal or sub-optimal districting plan (patrol boundaries for each patrol unit) that can minimize the long-term average response time and workload variation for police patrol system.

1.2.5 Relevant Problems in Emergency Services

1.2.5.1 Fire Station Location Problem

Perhaps the most important resource allocation decision faced by any chief fire officer is to determine the number and locations of fire stations. The optimal plan is the one that minimizes the sum of losses from fire and the cost of providing the services [58]. Proper fire station locations can shorten the response distance and response time, help determine the reasonable number of fire stations and utilize these resources efficiently [105]. The cost of locating a new or moving an existing fire station is high so careful analysis and decision of fire stations locations are necessary for strategic planning purposes [17]. A fire station (also called a fire house, fire hall or firemen's hall) is an around-the-clock facility of firefighters together with fire suppression apparatus, such as fire engines or ladder trucks, and personal protective equipment, fire hoses and other specialized equipment [17]. The main task of fire stations is to operate in case of fire and some other emergencies [109]. It is assumed that the fire engines wait at the fire stations until they are called for services. After completing service, the fire engines return to the fire stations to wait calls [17]. The selection of optimal fire station locations is a multi-objective decision problem. Based on literature, there are two major categories of objectives: cost and response distance (time). Objectives related to cost are to minimize the total setup cost of fire stations, annual operating cost and total loss cost of an incident. Objectives related to response distance and response time are minimize the average response distance (time) and worst-case response distance (time) [105][17]. Other objectives may include: attain targeted number of fire stations required, minimize service overlaps of fire stations and minimize locating where water availability could be a problem [17].

1.2.5.2 Ambulance Location Problem

Emergency medical services (EMS) provide out-of-hospital acute medical care, transport to definitive care, and other medical transport to patients which prevent the patient from transporting themselves [2]. Basically, the ambulance services include four steps: 1) incident detection and reporting, 2) call screening, 3) vehicle dispatching and 4) actual intervention by paramedics. In step 1 and 2, the severity and degree of urgency of the incident are determined. Then, the emergency services manager should determine the type and number of ambulances to dispatch [24]. Typically, the decision of ambulance services arise at three different levels [44]: 1) strategic level (e.g. location and construction of fixed facilities; equipment, staff training) 2) tactical level (staff scheduling, location and relocation of emergency vehicles at any time), 3) operational level (paramedical procedure followed by staff). The standard set by United States Emergency Medical Services Act is that 95% of requests should be served within 10 minutes in urban areas [18]. The response time is very important in medical emergency services. So, the ambulances should be well located at all times to guarantee an adequate coverage and a quick response time [108].

More recent researches study the redeployment or relocation of ambulance. When a vehicle is responding to a call for medical service, it may leave a significant area without coverage. Once another incident occurs in this area, the response time may exceed the standard. So, it is necessary to relocate other ambulances to maintain a proper coverage level [46]. The development of new technology (computer, GPS) makes the redeployment problem tractable. The information of ambulance locations and status is reported to central department at all times. The redeployment plans are solved through real-time dynamic models [45]. More constraints should be considered for the relocation problem. For example, only a limited number of ambulances can be moved when relocation is conducted. Also, repeated round trips between two locations must be avoided. Long trips should also be avoided. More examples of constraints can be seen in [45].

1.3 Dissertation Overview

The rest of the dissertation is organized as follows: Chapter 2 reviews the existing methods of generic territory design problem, police patrol district design, the complexity of the problem, and the major methodologies applied in this dissertation, such as modeling & simulation, experimental design, response surfaces and simulated annealing. Chapter 3 introduces a parameterized algorithm to generate districting plans, describes agent-based and discrete-event simulation models to evaluate district designs and compares different evaluation methods. Chapter 4 describes a response surface method for the simulation optimization of police patrol district design as well as its application in Charlottesville police department case study. Chapter 5 applies two heuristic optimization methods (simulated annealing and genetic algorithm) to police districting and compares them with response surface method. Chapter 6 concludes the dissertation and suggests future work.

Chapter 2

Literature Review

2.1 Review of Districting Problem

2.1.1 Introduction to Problems

As mentioned in Section 1.1, there are a broad range of applications in districting or territory design. Many literatures can be found in political districting and sales territory design. In political districting, a governmental area is partitioned into a given number of territories for parliament election and presidential election [74][65]. In addition to contiguity and compactness, the main planning criterion is to have approximately the same population in each district. Minority representation is another demographic criterion. The intention is to ensure the minority voter have equal opportunity to participate in the political process [65]. The districting problem is important in real world because the district design directly affect the election results [29]. Some politicians want to maximize the advantages of the political party and win the election by controlling the districting process. This "gerrymandering" problem leads to egregious and unnatural district shapes. So, the districting method should be fair and avoid gerrymandering [74]. In sales territory design problem, the companies subdivide the market area into regions of responsibility to operate the sales forces [65]. The territory design should balance the number of customers and sales volume among districts so that the workload and travel times of sales person can be evenly distributed [94][93][87]. Maximizing profit is another activity-related criterion when designing the sales regions. Time-effort allocation and territory design can be integrated to maximize profit using limited resources of call time or effort [37][65].

Other applications of districting problem are related to public services. In some problems, territories or districts need to be designed for facilities providing service at a fixed location, e.g. schools or hospitals. Customers and inhabitants have to visit a public facility to obtain services. In school districting, residential areas are assigned to schools. Districting criteria and constraints are usually related to school capacity, balance of school utilization, travel distances of students and racial balance [65][25]. Similarly, for other social facilities like hospitals and public transportation, criteria such as capacity limit, utilization, good accessibility should be taken into consideration [65][79].

Some public services are not provided at fixed locations but distributed over a region or on-site where the calls for services incident happens [65]. The main design criteria are related to balance of workload, response time and travel distances. Examples are district design for police patrol services, winter services and solid waste collection services. The evaluation of such on-site services territory design is more complicated than other districting problems. There is much uncertainty of the temporal and spatial pattern of the service incidents. The vehicle routing, dispatching and responding activities are strongly related to the occurrences of the incidents and also show stochastic pattern. To evaluate the response time we need to compute the travel times for the incidents. The locations and times of these incidents are stochastic and must be modeled by an empirical distributions based on current incident data. Further the travel times depend on the characteristics of the road networks in the regions which cannot be conveniently modeled by single variable functions of the region weights. Similarly, the workload metric depends on travel times and the complexities described above hold for workload. Additionally the types of incidents and their spatial distribution within and across the districts affect workload variation among districts.

2.1.2 Basic Model and Computational Complexity

A convenient formulation for the police district design problem is as the aggregation of smaller geographic units into larger units that form the districts. This problem reduces to the graph-partition problem with the constraints of contiguity and compactness. The polygons within each district must be spatially contiguous. Compactness provides shorter travel times between members of the district. Evaluation of compactness depends on the distance metric used. For Euclidean distance a circle is the most compact shape, while for Manhattan distance a square is most compact. The graph partitioning problem can be reduced to the problem of "Cut into Connected Components of Bounded Weight", which is NP-hard [62][9].

As with all optimization problems we can employ either exact or heuristic methods [9]. Exact methods systematically exam districting plans explicitly or implicitly. Since the problem is NP-hard, we can use explicit methods only for small districting problems that have limited practical applications. Heuristic methods, such as, simulated annealing, genetic algorithms, and stochastic gradient ascent can find local optima for the districting problem. These local optima can provide good operating solutions for police departments.

2.1.3 Districting Methods

Since the nature of the territory design or districting problem is a graph-partitioning problem, the solution or the partitioning plan can be represented by integers (use integer to specify which district each atomic unit is assigned to). Mathematical programming techniques can be applied to solve this optimization problem. The spatial restriction of contiguity and compactness as well as other problem-specific criteria can be included in either objective function or constraints. The first mathematical programming method was proposed by [57]. The problem was modeled as a capacitated p-median facility location problem. Then this location-allocation model was improved and modified by [40] and [47]. Another mathematical programming method is based on a set partitioning model [78]. Divisional methods can also be applied to districting problem [41]. The idea is to partition the whole region iteratively into smaller sub problems until the number of the desired territories is met. In recent years, more heuristic methods are developed for districting problems, such as Simulated Annealing [33], Tabu Search [20][23], Genetic Algorithm [30], and GRASP [86]. Voronoi diagrams can also be applied to solve the territory design problem [85][83]. Geographical Information System (GIS) can be integrated with districting algorithm so that rich variety of maps, spatial databases and geographical objects in GIS can be utilized [65] [64][25].

2.2 Operation Research in Emergency Response

Generally, police patrol operations belongs to emergence response management. Based on Simpson et al.'s review [55] of the development of operational research and emergency response in the recent fifty years, the relevant researches and studies can be mapped into four focus areas: urban services, disaster services, specific hazards and general emergency. Among the 361 emergency response-related OR articles between 1965 and 2007, one third of them is in the category of urban services. Police patrol problem belongs to this category, which also contains fire station location planning, unit assignment and ambulance queuing models, etc. During the fifty years, operation research is widely used to solve the emergency response problems. The top three popular OR methodologies in emergency response are mathematical programming, probabilities and statistics, and simulation. 70% of the articles are related to these three methods. Some other OR methodologies applied in emergency response are decision theory, system dynamics, fuzzy sets and expert systems, queuing theory, and soft OR, etc. Most foundational work in urban service appeared in the period of 1978 to 1984, which can be considered as the golden era' of management science. Then, in 1990s, the focus of emergency response in research shifted from the urban services to the investigations of disaster services and general emergency category. But there are still a small increasing number of articles related to urban services. According to [55], the current opportunities in this area are 'soft' versus 'hard' OR, information and decision support systems, volunteers and temporary organizational structures and performance measurement. They also mentioned that most OR work addresses the well-structured problems of emergency services but most of emergency response itself is semi-structured. Therefore, the management of disorganization of emergency response is another potential growth area in the future.

2.3 Approaches to Police District Design

2.3.1 Manual Methods

Historically, the geographic patrolling boundaries were drawn by hand based on police department's knowledge and experience of the total area and the availability of the police resources [104][80]. Police department also considered the natural boundaries, such as the hills or rivers, the locations of hotspots of crime as well as the administrative boundaries, such as neighborhoods and communities [32]. The limitation of the hand drawn boundaries is that the human is limited in the number of options they can consider and in their formal evaluation of the alternatives. Each alternative represents a different possibility in terms of workload and response times, but assessing these by hand can be very time consuming. The assessment is particularly difficult since both workload and response times are stochastic. This inability to measure the efficiency of the districting alternatives means that manual methods are not appropriate and good computer-based methods are required in order to create district that can positively impact high-level decision making by the police [104][32].

2.3.2 Operations Research and GIS Methods

Operation research(OR) and geographic information system (GIS) methods can be used to design the police patrol districts. According to [32], the first OR application was to use p-Median clustering as in [80] to minimize the total weighted travel distance to service the expected calls. In 1979, Aly and Litwhiler used an interchange heuristic method to allocate police briefing stations to districts [7]. In 2002, a simulated annealing approach was used by Amico et al., to search for a good partitioning of the police command districts by assessing the average response time and variation of workload of police officers with a simulation called PCAM [33]. GIS methods have been used more recently in police district design. In 2010 [32] used GIS methods with a maximal covering formulation to determine optimal police patrol areas. This approach produced good alternatives for police districts design. However, the criteria for evaluation were based on static calculation of distances, weights of incidents, and queuing model statistics. Police patrols are highly dynamic activities. It is difficult to use static approaches to find effective answers for most police. Instead, simulation methods of police patrol can provide more useful evaluations.

2.4 Simulation of Police Patrol

2.4.1 Introduction to Simulation

Simulation is the technique of using computers to imitate, or simulate the operations of different kinds of real-world systems. To study a system, the relationships between input variables, environmental variables and response variables are of great interest. If these relationships are simple enough, it is possible to use mathematical methods to find the analytic solution. However, if the real-world system is a complex system, the relationships that compose the system are difficult to analyze mathematically. In such situation, computer simulation can be used to build and evaluate the model and estimate the characteristics of the model [73].

In a sound simulation study, the first step is to formulate the problem and plan the study. In this step, the scope of the system and the objective of study are determined. Also, system variables of interest and performance measurements are discussed. Then, the next step is information collection on the system structure and operating procedures. These observations are used to make assumptions or rules in the simulation system. Data of system variables should also be collected. The statistical patterns of the variables are extracted from the data and are used to generate incidents or system attributes in simulation. Two necessary steps in simulation are validation of assumptions and verification of computer implementation. Then, design of experiment is conducted for simulation model. In this step, the simulation run length, length of warm-up period and number of repetitions are determined. If the system is a complex system and many system variables are studied, design of experiment (DOE) method such as factorial design, response surface method can be applied. The final step is to analyze the output data of simulation. In this step, absolute performance of certain system configurations is determined and alternative configurations are compared in a relative sense [73]. Sensitivity analysis, ”What-If” analysis and risk analysis are examples of the output analysis for simulation [69]. These analyses can help understanding more about the system and provide suggestions of actions to take.

2.4.2 Motivation of Simulation Method

Simulation is another tool to investigate the police patrol problem. Within the law enforcement environment, it is difficult to test and evaluate new dispatch and patrol strategies by doing direct field experiment. The direct "let's try it out" approach faces high risk and cost of money, morale, and public confidence [70]. In addition, the police dispatch and patrol is a complicated problem with the uncertainty of the amount and location of crime, the dynamics between decisions and the consequences, and the complex relations among many variables. Sometimes, theoretical research and mathematical models cannot provide comprehensive solutions to the problem. Either the new design of police region or the new scheduling strategies are difficult be tested directly in field [70]. However, computer simulation imitates and represents major characteristics or behaviors of police patrol system. It can provide a platform to conduct scientific experiment to test and evaluate the alternative police patrol strategies [70]. The optimal solution can be found through simulation optimization.

2.4.3 Agent-Based Simulation

Agent-based simulation can explain the real-world phenomena and solve the practical problems of complex systems by simulating the behaviors and interactions of agents in the system. It relies more on natural decision rules of agents rather than the complex or approximate equations. Usually, these agents have their own logic and rules to make decisions of their behaviors based on their environment and the status of the agents near them. In this way, the real-world complex system is translated to the simulation model in the computer naturally and the output is the long-term phenomena or trend of the whole system. It has been proved that Agent-based Modeling and

Simulation can be applied into many domains such as population studies, biology, ecology, economics and urban planning.

In police patrol system, Agent-based simulations can capture of the behaviors of patrol units in an environment, through the use of decision rules. These decision rules govern the interactions between objects in the simulation. For example, when a police car object interacts with a road object the rules specify the rate of transition to the next road object. These rules can also represent static properties of the object, for example the speed limit, and the dynamic properties of the environment, such as weather, construction, and traffic conditions.

The interactions between multiple objects governed by the rule sets in the simulation produce emergent behaviors or properties that cannot be predicted before running the simulations. For police patrol system, the most important emergent properties are the response times to CFS and police workloads. These properties are the metrics that allow us to score the effectiveness of different patrol districting plans. Neither of these properties can be accurately anticipated a priori using only a districting plan and the numbers of CFS within the districts.

2.5 Introduction to Simulation Optimization

Simulation optimization is the procedure of looking for the best input variables values to make the output optimal without running through each possibility of input variable during the process of simulation [26]. If the simulation model is complex and it has a large number of input variables, the simulation experiment may be limited by the computational power of the computer conducting the experiment. It is also possible that the sub-optimal rather than the optimal solutions is selected. A good simulation optimization can use fewer resources to obtain more information in a simulation experiment. Based on the type of the input parameters of simulation model, simulation optimization methods can be basically divided into two major categories: continuous parameter and discrete parameter simulation optimization [12]. For continuous parameter simulation optimization, gradient-based method is an important category. Examples of gradient-based search methods are Finite Difference Estimation [14], Likelihood Ratio Estimators [48], Perturbation Analysis [101] and Frequency Domain Experiments [82]. Other techniques for continuous parameters include Stochastic Approximation [88] and Sample Path Optimization [90]. One major category of discrete parameter simulation optimization is Random Search [12]. In each iteration of the random search method, neighboring feasible solutions are evaluated via simulation and the alternative yielding better estimate is kept for the next iteration [10][11]. Some other methodologies can be applied to both continuous and discrete parameter cases. Response surface methodology is an iterative searching procedure of fitting a series of regression models between input parameters and output responses, and optimizing the resulting regression function [19][34][36]. Examples of Heuristic Methods include Genetic Algorithms [50], Evolutionary Strategies [97], Simulated Annealing [39] and Tabu Search [84]. Other statistical methods include Importance Sampling Methods [98], Ranking and Selection [52] and Multiple Comparisons With The Best [59]. More literature reviews of simulation optimization can be seen in [12][102][13][42].

2.6 Simulation Optimization Using DOE

2.6.1 Introduction to DOE

Design of experiments (DOE) is a systematic and scientific method of planning experiments in which principles and techniques are applied at the data collection stage, resulting in valid, defensible, and supportable conclusions. In addition, this process is carried out with minimum cost of time and money [81] [3]. There are four general areas of problems in which DOE may be used: 1) Comparative 2) Screening & Characterizing 3) Modeling and 4) Optimization [3].

- **Comparative**

  In a simple comparative experiment, it is of interest to study and identify whether a single factor can result in a significant change or improvement to the response variable [3].

- **Screening & Characterizing**

  In a complex system, many factors can affect the response variables. The screening process helps understanding the system by identifying the important factors. It can provide a ranked list of important through unimportant factors that affect the response variable [3].

- **Modeling**

  In modeling procedure, the relationships between input variables and response variables can be represented by mathematical equations. The coefficients of input variables need to be estimated. The mathematical equation can help understanding the system and can provide prediction of output variable if given the values of input variables [3].

- **Optimization**

  For some systems, it is of great interest to maximize or minimize the output variable. The optimization is the process of looking for the best settings of input variables in order to optimize the response variable [3].

Three basic principles of experimental design are randomization, replication and blocking.

- **Randomization**

Randomization means both allocation of experimental material and the order of individual runs of experiment are randomly determined. The motivation for the randomization principle is that there exist some extraneous noise factors affecting the response variable [81]. For example, the materials under the same factor level might be slightly different and it may influence the experimental result. Another example is that the machine's performance may be significantly different during different phases of experiment while we expect its performance to be uniform. We are not interested in this factor but it may affect the response variable. By properly randomizing the experiments, the effects of those noise variables can be averaged out and will not affect experiment result significantly [81].

- **Replication**

  Replication means independent repetitions of the combinations of factor settings. With the help of replications, the estimation of experimental errors can be obtained. Within replications of a certain factor combination, since the settings of factors are exactly the same, the difference of response variables comes from the noise variables or random errors. The replication can also provide experimenters a more accurate estimation of the sample mean under certain factor settings. The variance of sample will decrease with larger sample size [81].

- **Blocking**

  Blocking is another technique of reducing the noise variables (or nuisance factors) effects. Some noise variables are not easy to control, (e.g. temperature), we can randomize the experimental condition or order of runs to average out their influence to response variable. Some noise variables can be controlled in experiments. For example, there are two batches of materials for the same level of material factor. We suspect there is a difference between these two batches. In such case, each batch of material can form a block and subsequent statistical analysis can identify whether the block factor affects the experiment result [81].

2.6.2 DOE for Computer Simulation

The statistical theory of DOE was developed for real and non-simulated experiments. In real experiments it is impractical to study more than ten input factors. Moreover, it is also hard to experiment with many factors that have more than five values. However, a computer simulation model may have hundreds of inputs and parameters each with many values. A multitude of different settings of running conditions need to be evaluated. Moreover, sequential experiments can be more easily conducted for computer simulation than real experiments. For example, agricultural experiments need a single growing season to run. The sequential design will cost more time. Computer experiments, on the other hand, do not need such long time to run and therefore more well-suited to sequential designs instead of ”one shot” designs [69].

In a simulation study, the problem is solved by simulate the internal structure or mechanism of a real-world system rather than find the analytical solution. Computer simulation can be deterministic or stochastic. In determinist simulation, same setting of input variables will lead to same output variables. In such case, the principle of replication is meaningless since there is no variation of response variable for same input values. Stochastic simulation or random simulation uses Pseudo-Random Numbers (PRNs) to generate the values of some systems variables. So, the same system input may have different output unless the PRN streams or seeds are identical [68]. M/M/1 queuing system is an example of stochastic simulation. The interval time and service time follow a certain distribution such as exponential distribution [69]. In the simulation, these times are randomly drawn from corresponding distribution extracted from collected data. We are interested in the long-term performance of the system such as average waiting time. For stochastic simulation, the DOE principle of replication makes senses due to the variation of response variable for same inputs. The accurate estimation of sample mean can also be obtained by increasing the run length of simulation.

Simulation analysts need to try different values of input variables or parameters of the model in order to learn the relationship to output variables. If an input factor is kept constant, then we cannot estimate its effect on output. If experimenters change only one input at a time while keeping all other inputs fixed at base values, such experiments are inefficient and they will fail to estimate the interaction effects between input factors. The simulation experiments should be scientifically designed. If the simulation model has many input variables or parameters, the number of different combinations of input factors can be large and it is computational expensive to evaluate all the possibilities. In such case, screening experiments should be designed to identify the important factors and further experiments can be conducted on these factors. Metamodels are widely used in design and analysis of experiment to approximate the Input/Output (I/O) function defined by the simulation model. The polynomial of first or second order is the most popular metamodel. Other examples of metamodel include Classification And Regression Trees (CART), Generalized Linear Models (GLM), Multivariate Adaptive Regression Splines (MARS), (artificial) neural networks, nonlinear regression models, nonparametric regression analysis, splines, support vector regression, etc [69].

2.6.3 Factorial Design

2.6.3.1 Full Factorial Design

A full factorial design, also referred as a fully crossed design, is an experimental design in which all possible combinations of levels of all the factors are investigated. For example, if factor A has a levels and factor B has b levels, the experiment contains all ab treatment combinations [81]. Using such experimental design, we can identify the main effect of each factor on the response variable as well as the interaction effects between factors on the response variable. The main effect of a factor is defined as the change in response variable produced by a change in the values of the factor. In some experiments, the interaction effects between factors exist. It refers to the case that the difference in response between the levels of one factor is different at all levels of the other factors [81]. If there are many factors involved, say, k factors and each factor takes two possible levels, then the total number of combinations will be 2^k and the design is called 2^k factorial design. The 2^k design is very useful in the early stages of experimental work, when many factors are investigated. It provides the smallest number of runs with which k factors can be studied in a complete factor design. So, it is widely used in factor screening experiments [81].

Two common methods to analyze factorial experiments are analysis of variance (ANOVA) and regression analysis. Some graphical methods can also be applied such as main effects plots, interaction plots as well as normal probability plots. If the factors are continuous, two-level factorial designs assume the effects to be linear. When a quadratic effect is expected for a factor, more complicated methods such as central composite design. The optimization of the factor settings can be achieved by using response surface methodology [38].

2.6.3.2 Fractional Factorial Design

When the number of factors k increases in a 2^k factorial design, the combination number of all factors' levels outgrows the experimental resources, such as time and money. However, a large proportion of degrees of freedom in a full factorial design correspond to higher order interactions and only a few correspond to the main effects and low-order interaction effects of factors. If certain high-order interactions can be reasonably assumed to be negligible, only a fraction of the full factorial design needs to be run in order to get the information on the main effects and low-order interactions. The major use of fractional factorial design is in screening experiments. The factors identified as important can be investigated more thoroughly in subsequent experiments. Three key ideas of a fractional factorial design are 1) the sparsity of effects principle (the system or process is driven by some main effects and low-order interactions), 2) the projection property (fractional design can be projected into stronger designs in the subset of significant factors) and 3) sequential experimentation (fractional design can be assembled sequentially to a larger design) [81].

Fractional designs can be expressed as I^{k-p}, where I is the number of levels of each factor, k is the number of factors, and p is the size of the fraction of the full factorial used. p is also the number of generators. The generators define which effects or interactions are confounded. Resolution is an important property of a fractional design. It represents the ability to separate main effects and low-order interactions from one another. It can be defined as the minimum word length in the defining relation excluding (1). Resolution II is not useful because main effects are confounded with other main effects. Resolution III experiments can estimate main effects, but they may be aliased with two-way interactions. In resolution IV experiments, main effects are not confounded by two-way interactions. Two-way interactions can be estimated, but they may be aliased with other two-way interactions [21].

2.6.4 Analysis of Variance

Analysis of variance (ANOVA) is a statistical method that partitions the observed variance of a particular variable into components representing different sources or causes of variation. It uses statistical test to determine whether a significant relationship exists between input factors and response variable in an experiment [81]. Before using ANOVA, some statistical graphical method such as box plots, histograms and scatter plots, can be applied to compare the means and standard deviation among groups. ANOVA analyzes the data more objectively and formally. For single factor ANOVA, the data records can be grouped by different levels of the factor. Then the variance of the response variable can be decomposed into variation between groups and variation within group. The variation between groups is the sum of squares of the differences between the treatment averages and the grand average while the variation within group is the sum of squares of the differences of observations within treatments from treatment averages. The ANOVA F-statistic is a ratio of the Between Group Variation divided by the Within Group Variation. A large F value indicates that not all the group means are equal and thus this factor is important to the response variable. To further study the difference between treatment means, multiple comparison techniques such as Tukey's test and Fisher Least Significant Difference (LSD) method can be applied. ANOVA usually has three assumptions: 1) Independence of observations, 2) Normality (the error item follows normal distribution), 3) Equality of variance (the variance of different groups should be the same). The statistical test is valid if three assumptions are met. If the normality assumption is unjustified, nonparametric methods such as Kruskal-Wallis test can be used to test the equality of treatment means. The multiple-factor ANOVA provides whether main effects or interactions between factors are significant or not [81].

2.6.5 Fitting Regression Models

In design of simulation experiments, the simulation model is viewed as a black box rather than a white box. In the black-box view, the input and output variables of the simulation model are observable while the internal transformation between input and output are not observable. The general black-box representation of simulation experiment model is $\hat{\Theta} = s(d_1, ..., d_k, r_0)$ where $\hat{\Theta}$ is the simulation output vector ; the function $s(.)$ is the mathematical function implicitly defined by the simulation model; d_j with $j = 1, ...k$ is the jth input variable of the simulation program, and r_0 is the vector of PRN seeds [69]. Regression model can be applied to model and explore the relationship between input factors and output variables. In general, suppose there is a single response variable y that depends on k independent or regressor variables, $x_1, x_2, ..., x_k$. The regression model is a mathematical model that characterizes the relation between y and $x_1, x_2, ..., x_k$, for example $y = \phi(x_1, x_2, ..., x_k)$. In most cases, an appropriate function is chosen to approximate ϕ. Low-order polynomial models are widely used as the approximation of functions. A multiple linear regression model with k regressor variables can be represented as $y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + ... + \beta_k x_k + \epsilon,$. It assumes the linear relationship between input and output variables. The error item ϵ is assumed to be normally distributed. $\beta_j, j = 0, 1, ..., k$ are the regression coefficients, representing the expected change in response y per unit change in x_j when all other input variables are kept constant. They can be estimated by using least square method on experimental data. If non-linear relationship is assumed in the experimental area, second-order items and interaction items can be added to the linear regression model [81]. The regression model can help understanding the relationship between input and output variables. It can also be used for optimization of response variable.

2.6.6 Response Surface Methodology

Response surface methodology (RSM) uses a sequence of designed experiments to explore the relationships between input variables and response variables. The objective is to rapidly and efficiently improve the response variable and finally reach the general vicinity of the optimum [81].Usually, RSM starts from a small area in the design space by conducting a factorial design or fractional factorial design, depending on the number of factors and the sample size the experimenter can afford. By analyzing the experiment result, the relation between factors and response is approximated in the small area. If the relation is mainly linear relation, it means the response can be improved by changing the factor value towards the steepest accent direction if we want to maximize the response value. So, the next iteration of factorial experiment can be designed along this direction until the curvature of the approximation of function is found, indicating that the design region is close to the local optima of response surface. Then, next iteration of experiment can be designed in the region (central composite design (CCD)) and second-order polynomial model can be built to further understand the relation between factors and response in this region. Finally, optimization of response variable can be achieved in this area.

2.6.7 Space Filling Design

Space-filling designs can be used when run-to-run variability is small, for example, deterministic computer simulation. Two major objectives of space-filling designs are: 1) spread the design points over the operation space of factors 2) space the points uniformly for each factor. Generally, there are three major types of space-filling designs of experiments: 1) sampling methods, 2) distance methods, 3) uniform distribution methods. In addition, there are some hybrids methods.[6][95]

The basic idea of sampling methods is to put the design points in the grid of operational space of factors. If we put one design point into each grid, the number of runs will grow fast if the number of factors tends to be large. One approach is to randomly select subset of these points in the grid. Latin Hypercube Sampling is such a method that chooses subset of all the points in the grid. It extends the idea of Latin Squares. A Latin square is an n by n array filled with n different symbols (treatments), each occurring exactly once in each row and exactly once in each column. Latin Hypercube Sampling randomly chooses one treatment from n. The corresponding n grids are filled with design points. In this way, it ensures even spread across each of the factor. However, the design points may not be evenly spread across the operational space of all the factors. To overcome this drawback, Randomized Orthogonal Arrays extends Latin Hypercube Design and try to make the projections into larger sub-spaces of operational area more diverse or space filling [95]. Although more efficient, orthogonal sampling strategy is more difficult to implement since all random samples must be generated simultaneously. The motivation of distance methods is to spread the design points in the operational space of the factors to make them far away from each other. The Sphere-Packing design method maximizes the minimum distance between two design points so that the points can be spread out as much as possible in the operational space. Other distance methods include Minimax Distance Design and Minimum of the Average Distance Design [6]. The uniform distribution method minimizes the discrepancy between the design points (empirical distribution) and a theoretical uniform distribution. So, it can make the design points as uniform as possible in the design space [95].

There is trade-off between the "spreading points out" performance and uniformity in design space among space-filling designs. Sphere Packing maximizes the minimum distance between design points. It has the best performance of spreading the points out in the design space. Uniform minimizes has the best performance of uniformity of design points and Latin Hypercube produces designs that mimic the uniform distribution but with the constraint of even spacing of the levels of each factor. It is a compromise between the Sphere-Packing method and the Uniform design method.[6]

2.6.8 Comparison Between RSM & SFD

Both RSM and space filling design explore the relationships between several explanatory variables and one or more response variables. But these two methods are different in many aspects.

First of all, the basic ideas and procedures of the two methods are totally different. The space-filling designs explore the relationships between factors and response in the broad operational space of factors. It spreads the design points out in the space and also tries to ensure the uniformity of the design points on each factor. So, the method has a "global" view of the design space. So, the experiment result can provide general relation between predictors and response with a limited resolution. The response surface designs, on the other hand, is to use a sequence of designed experiments to obtain an optimal response. The response surface method has a relatively "local" view of the design space compared with space-filling designs.

Secondly, Scopes of applications are different for these two methods. The space-filling method is very useful when the run-to-run variability is small, for example, deterministic computer simulation system. In such case, randomization, blocking and replication are not necessary because repeating the same run yields the same response. Thus, the design points can be spread out in the overall design space to have a general understanding between predictors and response. If it is used for stochastic system, replication should be applied in design points. Then, the number of runs tends to be large. The response surface method, on the other hand, is more suitable in this case. Because it focuses on a small area so that limited sample size can be used to get better understand in this area. After several iterations, the local optima can be found to improve the response. However, this method has limitation when the actual relationship between predictors and response is complicated. For example, there are many local optima in the design space. The local optima found by response surface method may be worse than some other local optima. In such situation, a global view of the design space is important. The space-filling method can be used to identify the promising small region then response surface method can find the local optima in this region.

2.6.9 Kriging Metamodel

Kriging is another metamodel that can approximate the Input/Output (I/O) function implied by the simulation model. low-order polynomial regression model can be fitted to data obtained from small experimental areas while Kriging model can be used for larger experimental area. Kriging is a global method for the whole experimental area rather than local method for a small experimental area. The objectives of using metamodel are prediction, sensitivity analysis and optimization [68].

Originally Kriging was developed in the field of geostatistics or spatial statistics by a South African mining engineer Krige . Then Matheron developed the mathematics of Kriging model [76]. In geostatistics, Kriging method is used to interpolate the value of a random field (e.g., the elevation as a function of geographic location) at an unobserved location from observations at nearby locations. Later, the two-dimension input in geostatistics was extended to k-dimensional input where k is a given positive integer and Kriging models were used in the I/O data of deterministic simulation models [91]. In 2003, Van Beers and Kleijnen started with the application of Kriging to random simulation models [67][68].

2.6.10 Pareto Frontier

Pareto Optimization, also referred as Multi-objective optimization (or multi-objective programming, multi-criteria or multi-attribute optimization) is the methodology of simultaneously optimizing two or more conflicting objectives subject to certain constraints [100][96]. For multi-objective problems, it is difficult to identify a single solution that optimized all the objectives at the same time. Usually, when one of the objectives is improved, other objectives becomes worse while searching for the solutions. There is trade-off between different objectives. A Pareto optimal or Pareto efficient solution is a tentative solution that cannot be replaced with another solution which improves one objective without worsening other objectives. So, a Pareto solution cannot be dominated by any other solution for all objectives. The Pareto frontier or Pareto set or Pareto front is the set of solutions that are Pareto efficient. So, for the decision makers, their attention can be restricted to the solution set of Pareto

Frontier and consider the tradeoffs within this set rather than the whole solution space.

2.7 Simulation Optimization Using Heuristics

2.7.1 Genetic Algorithm

Genetic algorithm (GA) is an adaptive search heuristic based on natural selection. This heuristic (or meta-heuristic) generates population of candidate solutions (also called individuals) and make them evolve toward better solutions iteratively by simulating the natural evolutionary process such as inheritance, mutation, selection, and crossover. Each solution is represented by a set of properties (chromosomes or genotype) which can be recombined, mutated and altered in the evolutionary process. Usually, the evolution starts from a population of individuals generated randomly. The fitness of each individual in this generation is evaluated by calculating the objective function of the optimization problem. In each iteration, the individuals with better fitness are more likely to be selected from current population and form the next generation through the evolutionary process (mutation and crossover). Then the same procedure is conducted on the new generation in this iterative algorithm. Commonly, the GA terminates when a pre-specified number of iterations has been finished or a satisfactory optimal solution has been reached. To apply GA in different optimization problems, we need to define a genetic representation of the solution domain and a fitness function to evaluate the solution domain depending on the specific problem. In simulation optimization, the individual fitness is obtained by running the simulation model. More detailed introduction and its applications can be seen in [35] [107].

2.7.2 Simulated Annealing

Simulated Annealing (SA) is a probabilistic metaheuristic optimization method analogous to the process of slowing cooling a physical system so that a minimal energy state is achieved [26][53]. It is used to locate a good approximation to the global optimum of a given objective function in a large search space (often discrete space). In order not to be trapped in local minima, the SA procedure sometimes accepts movements in uphill directions (for minimization problem). The acceptance probability of an uphill direction is controlled by a sequence of random variables [15]. To simulate the slowing cooling process, the probability of accepting inferior solutions decreases slowly as the SA algorithm explores the solution space. To apply the SA algorithm to a specific problem, the following parameters should be specified: 1) the solution space, 2) the objective function, 3) the solution neighborhood, 4) the acceptance probability function, 5) the annealing schedule (how temperature decreases) and 6) the initial temperature. The algorithm's efficiency can be greatly affected by the choices of these parameters. Unfortunately, there is no general way to find the best choices of these parameters for all problems. Usually, one needs to design experiments to find the best settings of these factors. When SA is applied in simulation optimization, the simulation model can be used as the objective function. The SA method was independently developed by [66] and [27]. More detailed procedures and its applications can be seen in [8][106][103][49].

Chapter 3

Simulation Evaluation of Police Patrol District Plans

This chapter mainly describes the generation and evaluation of police patrol district designs. Section 3.1 describes the temporal and spatial pattern in historical CFS data in Charlottesville police department application study. Section 3.2 describes a parameterized redistricting algorithm that can generate district plans of various layouts. Section 3.3 and Section 3.4 introduces an agent-based police patrol simulation model and a discrete-event simulation model, respectively. Section 3.5 compares the two simulation models as well as a close-form method to evaluate district plans. Relevant publications can be seen in [111][112][114].

3.1 Application Study Data

The Charlottesville Police Department (CPD) provided the data used as the application study for this analysis. The City of Charlottesville is a mid-size city centrally located in the state of Virginia, USA. The city has a diameter of about 7 miles and a year-round population of about 40,000, which swells to about 66,000 during the academic year due to the presence of a major university.

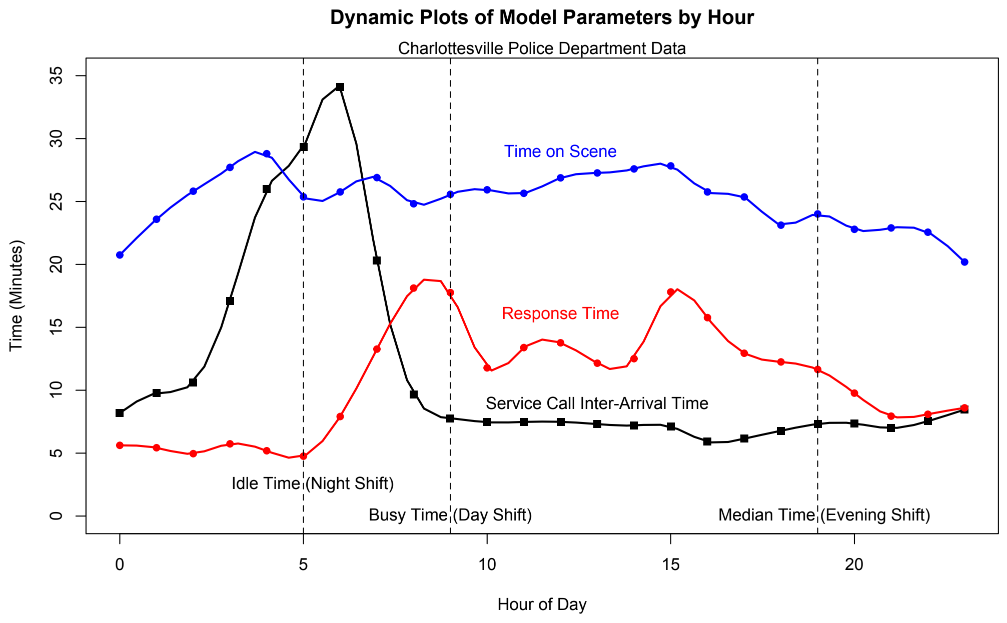


Figure 3.1: Dynamic Plots of Model Parameters by Hour

The current districting plan used by the CPD is about 20 years old. The city uses eight city patrol districts, with one car routinely assigned to each patrol district during each patrol shift. The police department operates three shifts a day: morning, evening, and overnight. As is the policy in many police departments, the CPD always dispatches the nearest available car to the scene of a CFS in an effort to minimize response time, rather than relying on each police car to respond to all calls within its district. Figure 3.1 provides an illustration of how the demand for police assets varies over the 24 hour period. This graph references 330,000 CFS incidents observed over a four year period. The averages of the CFS inter-arrival times, times on scenes and response time per hour are plotted in blue, red and green, respectively. During the night shift, the inter-arrival time for Calls-For-Service (CFS) is high, meaning that the CFS intensity is low. During the day and evening hours, CFS intensity is high, placing greater demands on the police patrols. As Figure 3.1 illustrates, the response time for police responding to CFS is highly correlated with traffic volume in the city. At night, police can respond relatively quickly to CFS because there is little traffic.

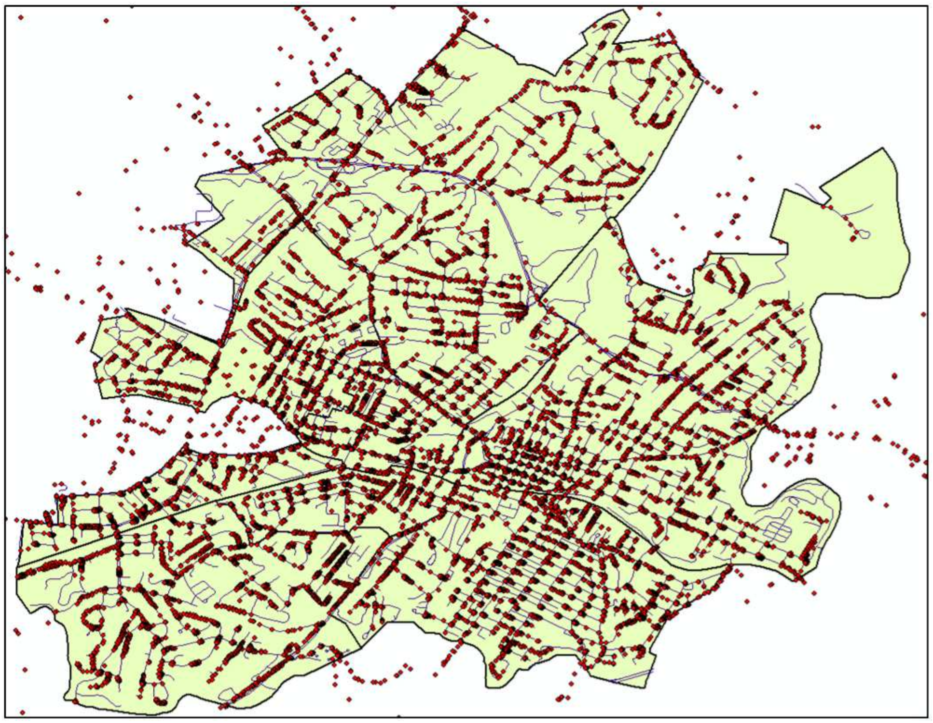


Figure 3.2: Locations of Historical CFS Incidents in Charlottesville

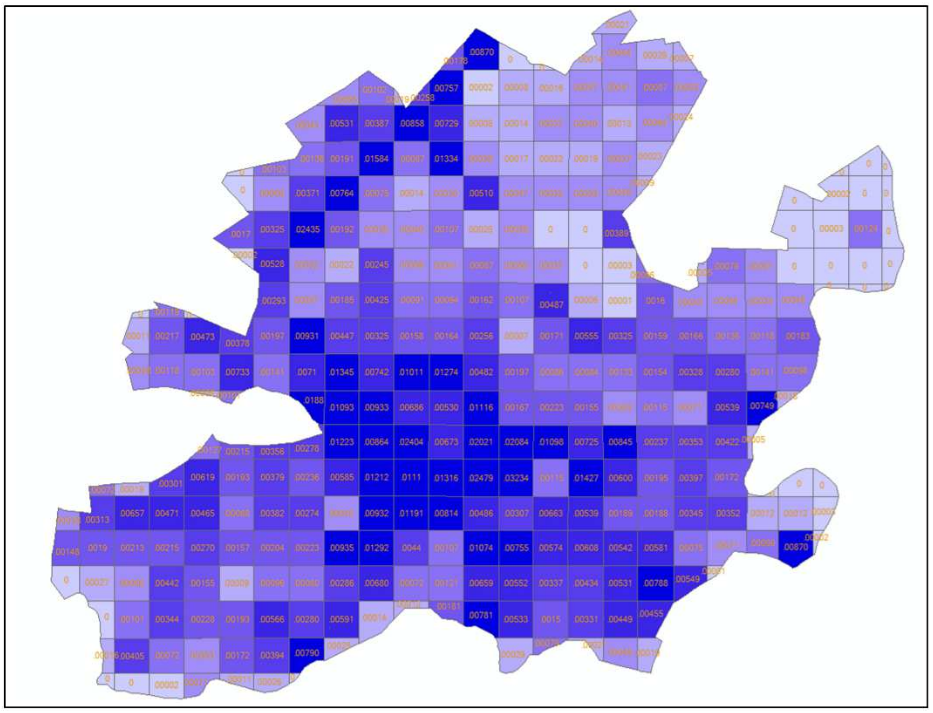


Figure 3.3: Call For Service Probabilities in Charlottesville Grid Network

During the morning and evening rush hour periods, it takes much longer for police to navigate traffic to the scene of calls for service. The time on the scene for a CFS remains relatively stable over the 24 hour period. Figure 3.3 is the CFS probability density map for Charlottesville. The locations of 330,000 historical CFS incidents are spatially joined to 323 grids. The CFS probability for each grid is calculated based on the counts of CFS incidents in the grid. The grids with greater CFS probability are drawn in darker color. It can be seen that the CFS incidents are more likely to happen in the downtown area and along the US-29 road. Such spatial and temporal CFS information can be used to generate CFS incidents and model patrol behaviors in the simulation model.

3.2 Generation of Districting Plans

The generation of districting plans is based on atomic geographical units. There are some existing geographical units such as police beats or census blocks. Usually, these geographical units consider administrative boundaries, important roads, or some natural boundaries (mountains, rivers). The redistricting procedure can start from these units and re-group them into several districts. When developing districting plans for large areas containing hundreds of such geographical units, police beats or census blocks are good choices for atomic units. However, some cities only have 20 or 30 census blocks. Police beats or census blocks are not small enough for optimal patrol boundaries. In such case, grid network can be used instead. The city can be divided into several hundreds of grids and they are small enough to be atomic units. Clearly, more atomic units represent more possible districting plans. Such representation is more suitable for systematic and scientific study of districting problem. The output districting plans based on grid boundaries can be adjusted according to existing boundaries such as important roads, administrative boundaries and natural boundaries.

The main idea of our districting algorithm is similar to the Constraint-Based Polygonal Spatial Clustering (CPSC) algorithm developed by [63]. The two main steps of the algorithm are 1) select seed polygons (one atomic geographical unit) and 2) select the polygon in neighborhood to be added to the existing cluster until all units are assigned to districts. The major difference of our redistricting algorithm is the selection of seed for each district. Rather than totally randomizing the locations of the seeds, we locate the seeds on several concentric circles over the urban region. The underlying intuition is that there is a general trend that downtown area of city has more population than suburbs. So the CFS incidents and crime events are also more likely to happen in central region of city. In the concentric circles model, central region of city has more seeds than suburbs and the patrol districts in downtown area tend to be small, which may facilities quick response and reduce workload variation. When selecting the seeds, the minimum distance between these seeds should be greater than a threshold to allow space for the districts to grow. Then, each seed alternates in acquiring adjacent atomic units until a stopping criterion is reached. The stopping criterion is that the sum of CFS probabilities of units in the district is greater than a user selected bound.

During the growing process, the districts or existing cluster of polygons select one adjacent unit to develop. If there are several alternatives, it chooses the one that can maximize the compactness score. Some randomness can be added to this procedure. The difference between random growth and compact growth can be seen in Figure 3.5. The district can randomly choose one adjacent unit to develop. After a number of iterations of development, all districts stop growing because the stopping condition is met or because there are no more adjacent units for growth. Usually, there are still some unassigned units. They are assigned to the adjacent districts based on compactness principle. Then, the districting plan is adjusted to balance the CFS

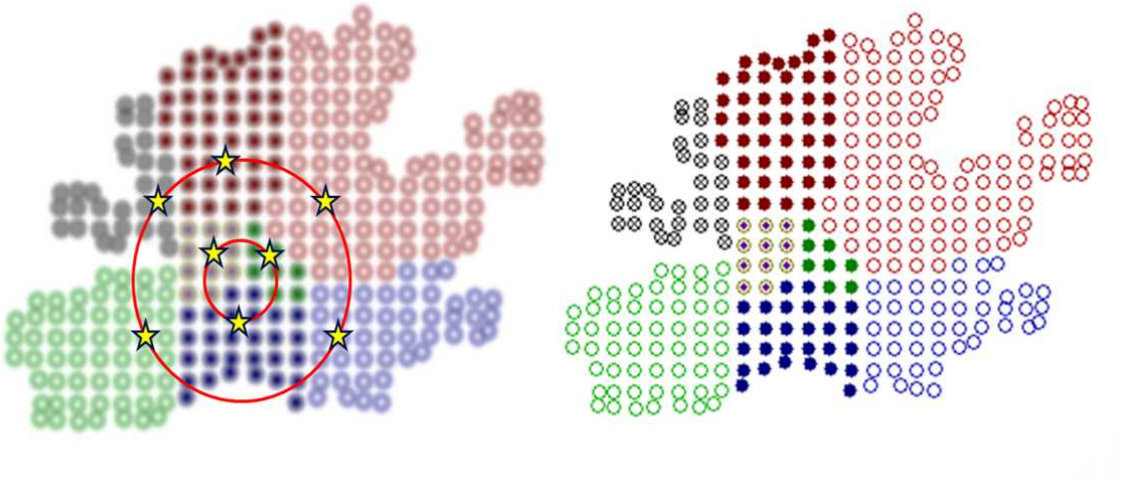


Figure 3.4: Seeds Selection of Police Districting for Charlottesville

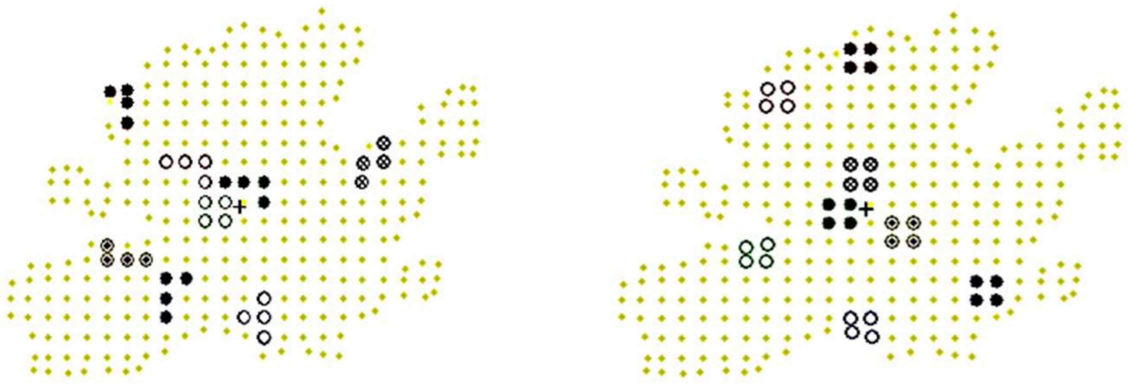


Figure 3.5: Random Growth vs. Compact Growth

probabilities in each district and smooth the boundary between districts.

The locations of the seeds for districts determine the framework and basic structure of the districting plans. The relevant parameters for starting the growing process are:

- The center of the concentric circles
- The number of circles
- The radius of each circle
- The number of seeds on each circle

Additional parameters determine the course of district growth. Examples are:

- The stopping criterion for the growing process
- The number of growth iterations
- Growth randomness vs. compactness
- The number of iterations that balance the CFS probability between districts
- The number of iterations that smooth the boundary between districts

In this way, a districting plan can be described and represented by a set of districting parameters. Once a districting plan is generated, some measurements can be quickly calculated without detailed simulation evaluation, such as compactness of plans and the variation of CFS probability of all districts. These intermediate measurements of districting plans can be used to select top proportion of plans for further simulation evaluation.

3.3 Agent-Based Simulation Model

3.3.1 Introduction

We cannot use closed form expressions because the assessment of response times and workloads requires the incorporation of multiple factors that interact in complex ways. Also, field experiments in the law enforcement and safety management are clearly not feasible because of the risks and costs, not to mention, the public relations problems [70]. This means that evaluation of the police patrol districting plans requires a high fidelity simulation. A key feature needed in this simulation is the ability to accurately represent behaviors of the police in response to calls-for-service. Agent-based simulations afford the ability to effectively represent these behaviors.

Agent-based simulations capture of the behaviors of objects in an environment, such as police patrols in city, through the use of decision rules. These decision rules govern the interactions between objects in the simulation. For example, when a police car object interacts with a road object the rules specify the rate of transition to the next road object. These rules can also represent static properties of the object, for example the speed limit, and the dynamic properties of the environment, such as weather, construction, and traffic conditions. Other example rules used in our simulation include:

- The nearest police car to a new CFS responds to that CFS;
- The responding police car takes the shortest path to the location of the CFS;
- If a police car is servicing a CFS then it is no longer available to respond to a new CFS;
- If the nearest available police car is in a different district it will cross the district boundary and respond to the CFS.

The interactions between multiple objects governed by the rule sets in the simulation produce emergent behaviors or properties that cannot be predicted before running the simulations. For our purposes the most important emergent properties are the response times to CFS and police workloads. These properties are the metrics that allow us to score the effectiveness of different patrol districting plans. Neither of these properties can be accurately anticipated a priori using only a districting plan and the numbers of CFS within the districts. As we indicated in Subsection 2.1.1 if we could simply develop districting plans that equalize the expected CFS in each district then our problem reduces to the graph-partitioning problem. Although graph-partitioning is NP-hard; nonetheless, there are available heuristics than can be applied.

Instead the police districting problem has evaluation complexity (see Subsection 2.1.1) in addition to the computational complexity of the graph-partitioning problem. In fact, when we use our high fidelity simulation to evaluate police districting plans that minimize the difference in CFS between districts they actually do worse than some other plans. The same is true for workload. The ability to discover these emerging properties is an important feature of the agent-based simulation we built and a critical requirement in the assessment of competing districting plans.

3.3.2 Implementation

We implemented the simulation using Java Repast. Java Repast is an open source, agent-based modeling and simulation platform [4]. It uses object-oriented model and has a source library of classes for creating and running agent-based simulations and for displaying and collecting data from these simulations. Geographic data, such as the data expressed in shapefiles, can be imported into the Java Repast model. Using these geographic data the behaviors and movements of the agents can be controlled according to rule sets that exploit these data. For example, the shapefile data layer on secondary roads can have attributes that provide speed limits in different segments of this layer.

We built our police patrol simulation model as an extension to Malleson's RepastCity prototype [75]. The inputs to our simulation model consist of the shapefiles of the city, the patrol district plan, the police patrol allocation plan and a data set of CFS times, locations, and severity (this last attribute determines the distribution of the service time for the CFS). The shapefiles for the city include primary and secondary roads, major highways, and obstacles or impediments (e.g., construction or road work).

In order to simulate the time and spatial pattern of actual CFS and maintain the randomness of the city environment, the time between incidents and the locations are randomly chosen based on the distribution of actual CFS. Rather than use a bootstrap approach which would resample from actual CFS, we instead use an empirical fit of

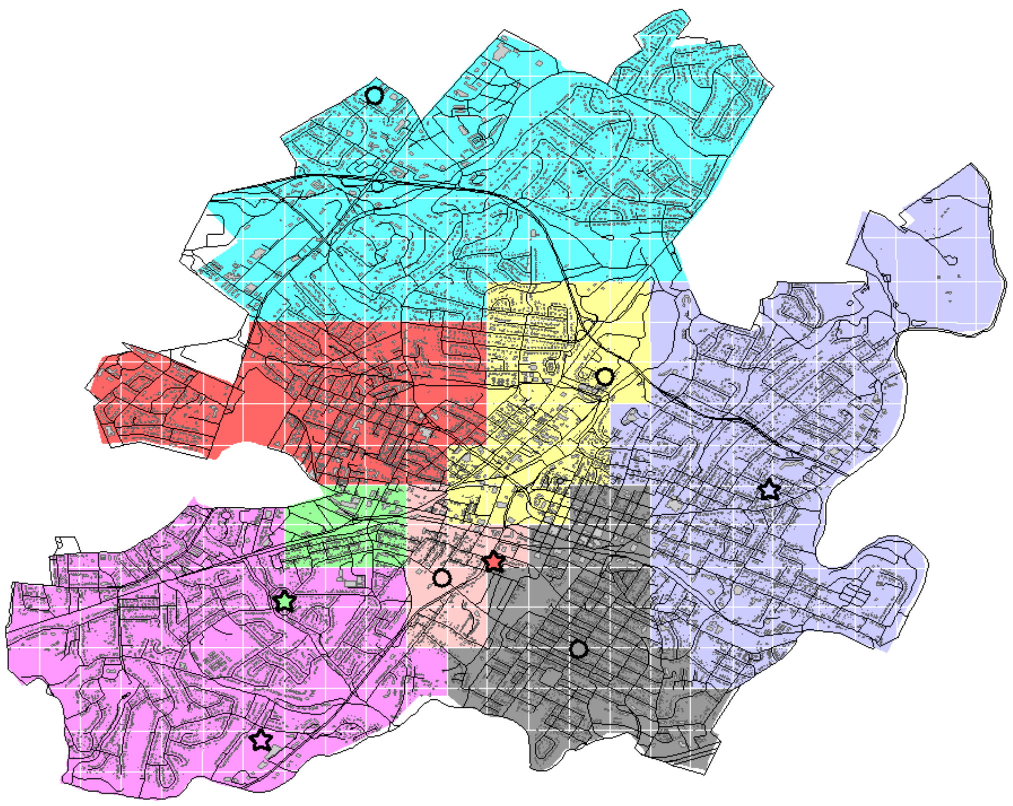


Figure 3.6: Static View of Visualization of Agent-Based Simulation

the distributions of the CFS in space and time. We then chose CFS values based on random draws according to these distributions.

Police cars are on patrol in their districts until they are dispatched by a CFS. Their patrol routes are randomly chosen from the network of roads in their assigned districts. An incoming CFS will generate the dispatch of the nearest police car and that car will follow the shortest route to the location of the CFS. In following this route it will use the maximum safe speed for the route which is greater than or equal to the speed limit on the route.

After the police car reaches the CFS location, the police car will remain at that location for the service time of that CFS. We obtain this service time as random draw from the service time distribution for the type and severity of the particular CFS. The service time distributions are empirical distributions found from the data set of actual CFS. When the service time ends the police car returns to its patrol route and again becomes available for dispatch to the next CFS.

We run the simulation for chosen number of runs or for a selected amount of simulation time. For each CFS in a run we record

- The time and location of the CFS;
- The identity of responding police car;
- The time of dispatch for the responding police car;
- The time of arrival at the CFS location for the responding police car;
- The time of departure from the CFS by the responding car (i.e., the time of arrival plus the service time);
- The travel distance of the responding car.

Using these data we can calculate the average response time and workload for each run and, hence, for each districting plan.

3.4 Discrete-Event Simulation Model

3.4.1 Introduction

The discrete event simulation model is based on the Hypercube Queuing Model (HQM), a well-known descriptive model used to analyze emergency response systems as spatially distributed queueing systems [72]. In the HQM model, each server (patrol car, fire engine, ambulance, etc.) has two states: idle (0) and busy (1). The state of the whole system is represented as a binary sequence of server statuses. When the number of servers exceeds three, all possible system states form a hypercube.

Historical CFS incidents data and traffic information provide estimates for the arrival rates of the servers into each geographical atom. As long as the aggregate service rate of the system exceeds the total arrival rates of CFS incidents (i.e., supply exceeds the demand), calculating the steady-state probability of the resulting Markov chain provides the probability of being in each possible system state in the hypercube. System performance metrics such as average response time and workload variation are calculated from the hypercube probabilities. While the basic HQM provides a very flexible framework for modeling emergency response systems, the size of the problem grows exponentially with the number of servers. Solving each instance requires solving a linear system with an exponential number of variables [22].

[22] demonstrates that Monte Carlo discrete-event simulations based on HQM converge to the steady-state probabilities estimated by HQM very quickly. Therefore, discrete event simulations provide an alternative method for solving for the HQM steady-state probabilities. The discrete-event model can more easily be extended to simulate complex situations, such as multiple cars responding, different priorities of CFS incidents, different CFS arrival rate at different times of day, as well as various patrol and dispatch rules. We developed the simulation model for the CPD districts in Java 1.6 SE using pseudocode provided in [22]. [92] provides the method we used to calculate the expected locations of CFS and patrol cars in the city using the Charlottesville data.

3.4.2 Implementation

We developed the simulation model in Java 1.6 SE using pseudocode provided in [22]. [92] provides the method we used to calculate the expected locations of CFS and patrol cars in the city. The inputs and outputs for the discrete event simulation model are shown in Figure 3.7. The simulation model generates CFS using the exponential distribution model defined by the inter-arrival rate parameter. The CFS incidents are spatially distributed within the city according to the geographic probability model generated from historical data (i.e., the CFS probabilities for each atom). CFS service times are randomly selected from the exponential model defined by the service time parameter. The discrete-event simulation model tracks the occurrence of four types of events: 1) Calls For Service (CFS), 2) patrol car arrival at CFS , 3) patrol Car departure from CFS , 4) patrol car arrival at base (Idle Position). When a CFS occurs in the simulation model, the nearest idle patrol car is dispatched by changing the server availability status from idle (0) to busy (1). The patrol car can be assumed to be at the centroid of the district. An alternative way to generate idle police car's current position is to randomly select an atom centroid in patrol district based on the police car's preventive patrol frequency. The travel time can be estimated by the Euclidean distance between the car's current position and the CFS incident location divided by the responding speed. When police car arrives at scene, service time is randomly generated from the exponential distribution estimated from historical records. When police car leaves the scene, the travel time is estimated in the similar way. The patrol car status returns to idle (0) once the car returns to its base location within the patrol sector after each event.

To simplify the problem, the discrete-event model assumes zero line capacity; if all servers are busy when an incident happens, the incident is dropped or considered as being responded to by units outside the modeled system. As can be seen in Figure 3.7, the simulation output includes 1) average response time, 2) workload variation, 3) cross-boundary rate, 4) incident drop rate. The first two measures are discussed in Section 1.2.3. Cross-boundary rate is the proportion of incidents responded by police cars from other districts. Incident drop rate is the proportion of generated incidents that are not responded because all servers are busy at that time. Since these "dropped" incidents are usually responded by the police units from neighborhoods, they are not included in the calculation of average response time and

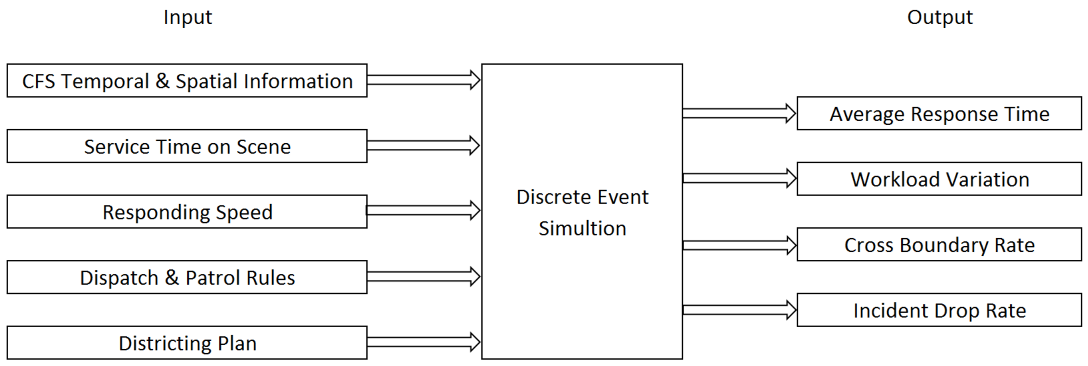


Figure 3.7: Input and Output of Discrete-Event Simulation

workload standard deviation.

3.4.3 Sensitivity Analysis

As noted in Subsection 3.4.2, the simulation output depends on several factors. But our focus is to study how districting plans affect performance measures. So, it is necessary to understand how other factors affect responses and configure them in a reasonable way. In this study, simulation experiment is designed for the following factors: dispatch and patrol rules, CFS inter-arrival time, service time on scene and average responding speed. The districting plan is fixed at current plan. The spatial CFS probability for each atom is also fixed at current distribution (Figure 3.3). In this experiment, two types of patrol and dispatch rules are implemented and compared. One is to assume idle police cars are always in the centroid of the districts when CFS incident occurs. The other is to randomly generate police car's position based on historical patrol frequency. In the simulation, 1 tick of simulation time represents 1 second of actual time. The levels of CFS inter-arrival time and service time are chosen based on the range of historical data. The range of responding speed is calibrated so that the range of response time of simulation output is in the same level of historical data. Four responses of the experiments are shown in Figure 3.7.

After simulation experiment, ANOVA is conducted for each response. The items related to main effects and two-way interactions are included in the model. Figure 3.8 shows the main effects plots for each response. The abbreviations and the corresponding meaning are listed below.

- **AveRespTime:** average response time of a district plan. The unit is tick in simulation.
- **WorkloadStd:** standard deviation of workload proportions of police cars among districts
- **Rule:** The factor of patrolling rule. There are two levels for this categorical variable: CarStayDistrictCenter and RandomPatrol.
- **CarStayDistrictCenter:** The police cars do not conduct preventive patrol. Instead, police cars stay at the centroid of the district if they are idle when no incidents happen.
- **RandomPatrol:** The police cars conduct random preventive patrol when they are in the idle state.
- **SerTimeScene:** The service time police cars spend on scene. The unit is tick in simulation
- **RespSped:** The responding speed of police cars when they respond to incidents. The unit is Euclidian distance based on coordinates per tick.

For average response time, all factors are significant at 95% confidence level. CFS inter-arrival time and responding speed explain most of the variance and are more sensitive to this response. Higher inter-arrival time and responding speed lead to lower response time. Higher CFS inter-arrival time leads to lower cross-boundary rate and higher workload variation. When CFS intensity is low, incidents are more

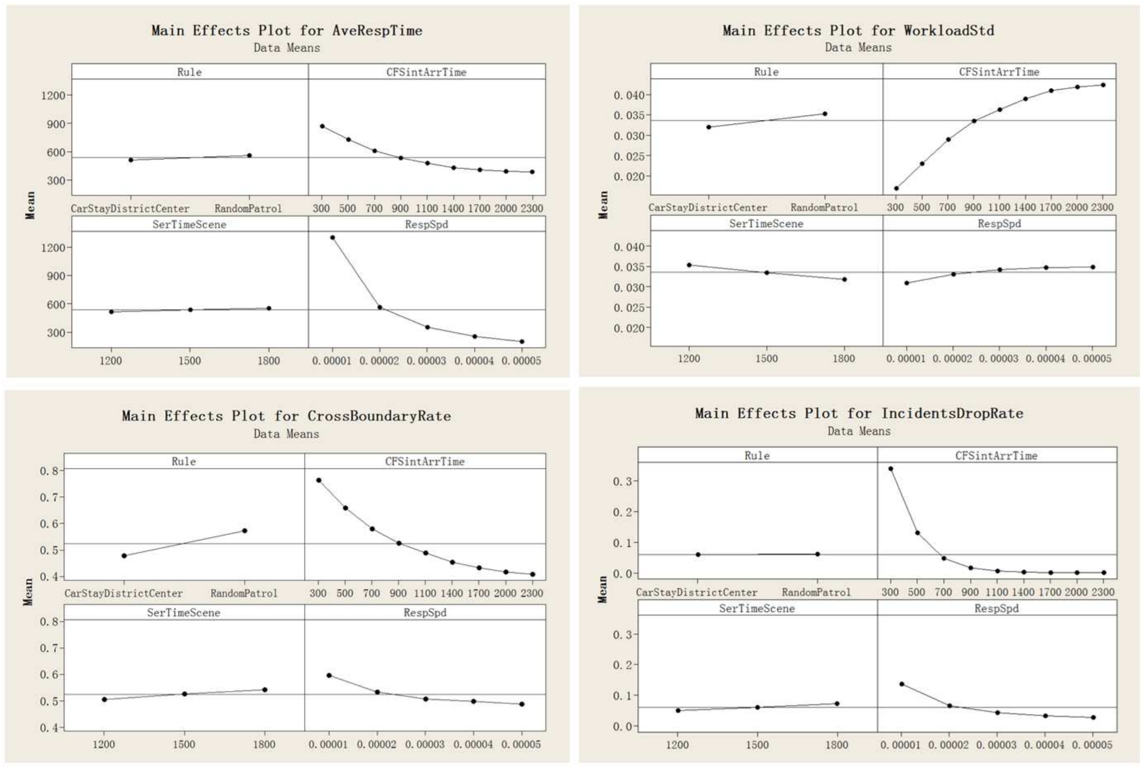


Figure 3.8: Sensitivity Analysis of Discrete-Event Simulation

likely to be responded by police cars within that district so cross-boundary rate is low. The incidents near the downtown area (a crime hot-spot) are always responded by the police cars in the nearby districts. These cars can handle them because the calls intensity is low. However, when CFS inter-arrival time is low and CFS intensity is high, the police cars near downtown area cannot handle the calls so that the cars in other districts are frequently dispatched to respond to calls in the downtown area. Thus, the cross-boundary rate is high and the workload variation is low.

Both response variables differ significantly ($p < 0.05$) across the two dispatch and patrol rules. However, although statistically significant the differences are subtle and would have few practical effects. Idle police cars staying at district centroid seem to perform better since both average response time and workload variation are lower than randomly patrol scenario. Since preventive patrol is a necessary function of the police department, the random patrol rule makes more sense and is closer to the actual situation. Though it is not practical for police cars to always stay in the district center all the time, higher frequency of visits to the central area may improve both average response time and workload variation. For all the responses, CFS inter-arrival time and responding speed and their interaction items explain most of the variance and are more sensitive. So, when the simulation model is used for evaluating districting plans, these two factors should be modeled and configured carefully since little changes in them will lead to large changes in responses. When evaluating the districting plans, the inter-arrival time and service time are modeled in the hour-by-hour manner based on the historical data (left panel of Figure 3.1). The responding speed is also set hourly so that the response time in simulation output is equal to the historical means of that hour. In this way, the generation of CFS incidents and overall queuing statistics in simulation are calibrated to actual situation.

3.5 Comparison of Evaluation Methods

3.5.1 A Closed Form Evaluation Method

The closed-form evaluation method relies on the relationship between location event CFS probability and the observed CFS counts over a geographic area. [61] demonstrate that criminal hot-spot (probability) maps can be used to accurately forecast future crime counts within police patrol districts. These criminal hot-spot maps are two dimensional probability density functions that can be estimated using kernel density estimation [56], predictive crime models [99][60], or by binning historical crime counts by atom [112]. We use the binning approach in this paper. These criminal hot-spot maps provide estimates for the probability of crime occurrence within each atom, with the notation π_i. For district k, the workload score W_k is estimated as the sum of atom event probabilities π_i within the district.

$$W_k = \sum_{i \in k} \pi_i \tag{3.1}$$

The district workload score W_k represents the proportion of work each district patrol is expected to perform. Since the objective is to provide equal workloads across the districts, district plans are scored using the sample standard deviation of the district workload scores σ_{W_k}. Lower workload standard deviation scores equate to better performance.

Criminal hot-spot maps can also be used to estimate the response time for officers to service calls within their districts. The response time score R is calculated as the sum of the probability weighted distances between each district centroid C_k and each atom location i.

$$R = \sum_{k=1}^{K} \left[\sum_{i \in k} (\pi_i ||C_k - i||) \right] \tag{3.2}$$

In the formula above, the notation $||C_k - i||$ denotes the norm (distance) between the district centroid and atom i. Depending on the situation, Euclidean, Manhattan, or travel (road) distance can be used to estimate the travel cost. In this application, we used the Euclidean distance. Lower response time scores equate to better district plan performance. This method assumes that there will be very limited cross-boundary service by the patrols within the sectors.

3.5.2 District Plan Selection

The three district plan evaluation methods above each provide an approach for scoring district plan response time and workload variation performance. The goal is to identify the district plans that provide good performance in both objectives. For multi-objective problems such as this one, there usually does not exist a single solution that simultaneously optimizes both objectives. Instead, there exists a (possibly infinite) set of Pareto-efficient solutions. A solution is Pareto-efficient (also called non-dominated or Pareto-optimal) for two objectives if one cannot improve performance in one performance measure by selecting a different alternative without sacrificing performance in another. Graphing the performance of the solutions provides a simple way to identify the Pareto-efficient frontier [43]. Figure 3.9 illustrates the trade-off space and resulting Pareto-efficient solutions identified by the agent-based model in both the low-intensity and high-intensity settings. The agent-based model identifies five non-dominated solutions for the low-intensity scenario and two non-dominated solutions for the high-intensity setting. We used the same approach to identify the non-dominated solution set in each scenario for the closed-form and discrete-event simulations. As is shown in Figure 3.9, Pareto analysis using the scores from the agent-based simulation in the low-intensity scenario (left panel) and the high-intensity scenario (right panel). Black circles represent non-dominated solutions identified by the agent-based model while colored points represent the non-dominated solutions identified by the other approaches. The dashed lines provide a visual reference for average performance in each evaluation measure as defined by the agent-based model. Note that all three methods provide highly scored non-dominated solutions in the low-intensity scenario but that solutions recommended by the CF and DE methods are rated as relatively average by the agent-based method in the high-intensity scenario. Also note that the average response times on these graphs for the current district plan correlate closely to the observed historical response times for the two scenarios in Figure 3.1, indicating a well-calibrated simulation model. Figure 3.9 also identifies these solutions as well as the current district plan used by the CPD.

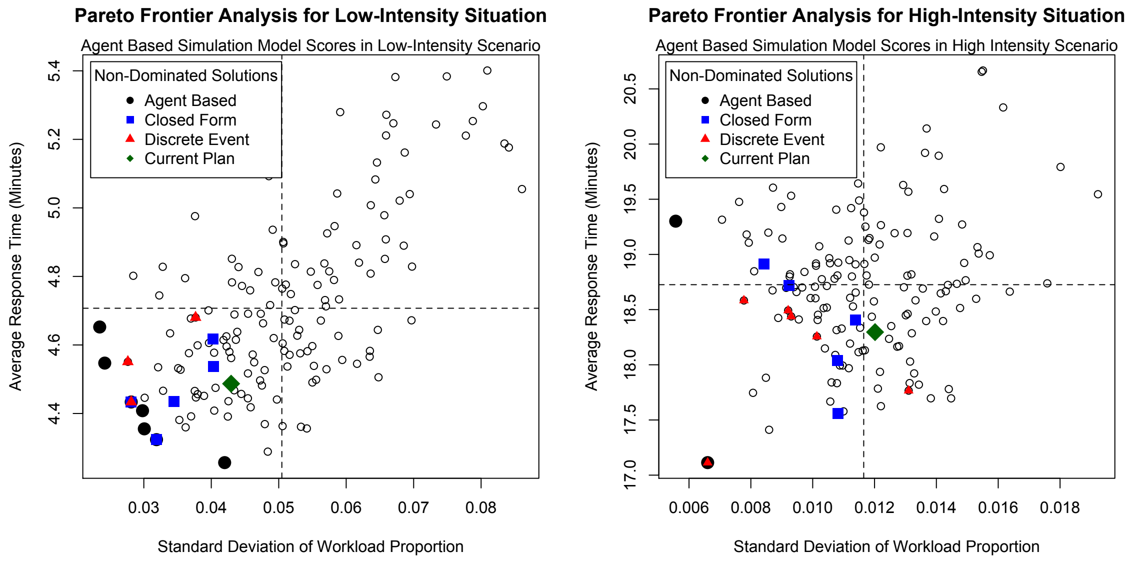


Figure 3.9: Pareto Analysis of Average Response Time for Low and High Intensity Situation

3.5.3 Comparison Results

As Figure 3.9 illustrates, there is some disagreement between the three different methodologies about which plans are best. Note that in the low-intensity scenario, the agent-based simulation model scores all of the non-dominated solutions by the other two methods relatively highly (they are all clustered in the lower-right hand corner). There is also some agreement on plans that are Pareto-optimal, with some districting plans on the Pareto frontiers of all three methods in the low-intensity scenario. However, in the high-intensity scenario, the Pareto-efficient plans identified by the closed-form and discrete-event methods tend to be rated as relatively average in at least one measure by the agent-based simulation.

Table 3.1 and Figure 3.10 provide an explanation for these differences. Table 3.1 provides the coefficient of determination (R^2) statistic comparing Closed Form (CF), Discrete Event (DE) and Agent-Based (AB) scores for workload variation and response time under low, medium, and high event intensity conditions. Figure 3.10

provides a pair-wise scatterplot for the most correlated (workload variation in the low-intensity scenario) and least correlated (workload variation in the high-intensity scenario) situations in this table for a visual reference. All methods provide highly correlated workload variation scores in the low intensity scenario but highly uncorrelated scores for workload variation in the high-intensity scenario. The response time scores are less correlated than workload variation scores in the low-intensity scenario but more correlated than workload variation in the high-intensity scenario. For both performance measures, correlation between methods decreases as event intensity increases.

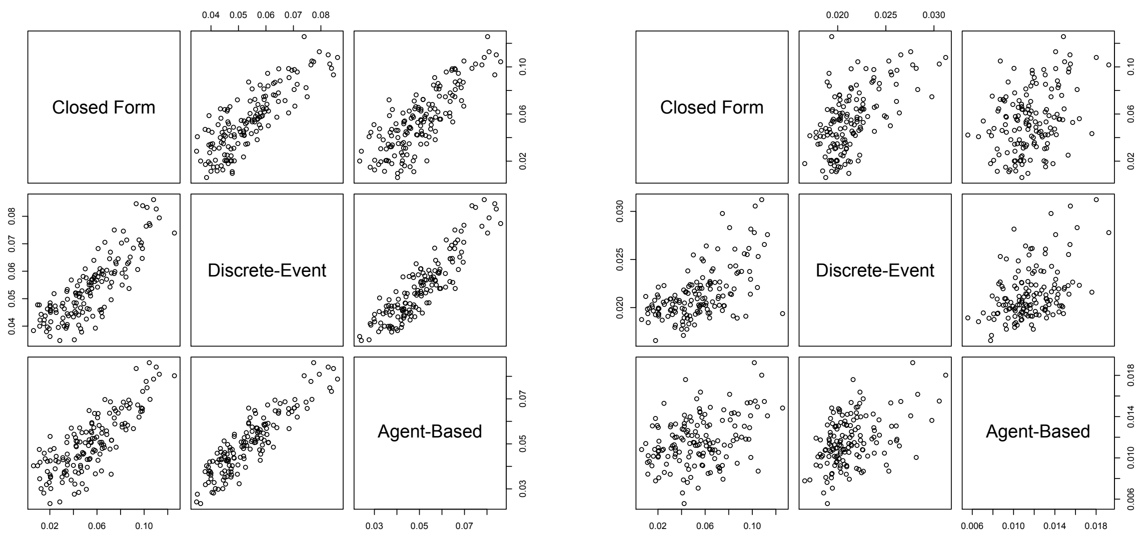


Figure 3.10: Pairwise Correlation of Workload Variation among CF, DE and AB Model in Low (left) and High (right) Intensity Conditions

Table 3.1: Pairwise R^2 CF, AB and DE scores for Workload Variation and Response Time under Low, Medium, and High Intensity Conditions.

Event	Workload Variation			Response Time		
Intensity	CF- AB	CF - DE	DE - AB	CF - AB	CF - DE	DE - AB
Low	0.65	0.70	0.81	0.45	0.51	0.74
Medium	0.33	0.44	0.47	0.30	0.44	0.72
High	0.13	0.35	0.19	0.21	0.43	0.52

Table 3.2: Pairwise R^2 Showing How the Performance Measures Correlate Within Methods Across Three Scenarios

Pairwise	Workload Variation			Response Time		
Comparison	CF	DE	AB	CF	DE	AB
Low-Med	1	0.69	0.45	1	0.92	0.55
Low - High	1	0.60	0.24	1	0.91	0.41
Med - High	1	0.91	0.11	1	0.98	0.46

Table 3.2 further explains the results observed in Table 3.1. Table 3.2 provides the within-method coefficient of determination (R^2) across the three scenarios. As can be seen, the closed form method provides the exact same scores for every scenario, the discrete event scenarios are highly correlated across scenarios, but the agent-based scores change significantly.

These results prompted further analysis to understand why the agent-based simulation model scores change so significantly in the high-intensity scenario. We identified two dynamics within the system that cause the agent-based simulation model to significantly alter the scores as CFS intensity increases. The first insight the agent-based model provides concerns the effect of the patrolling behavior of the police cars. When the police cars randomly patrol within their districts, they are often far from the patrol district centroid (as can be seen in the snapshot of the agent-based simulation model in Figure 3.6). The CPD always dispatches the nearest available police car to the scene of a CFS. Thus, cross-boundary support is quite frequent. In the low-intensity scenario, cross-boundary response averages about 42%. However, this cross-boundary support rises to 70% in the medium intensity scenario and 75% in the high-intensity scenario. These rates roughly correspond to the rates observed in a 1971 New York City study that found that cross-boundary support accounted for more than half of police dispatches [71].

The second significant system dynamic is the effect CFS intensity and slow response times due to traffic have on the workload variation during the busy periods of the day. The difference in workload variation (standard deviation of the workload proportion) among districting plans during the high-intensity period around the morning rush hour is very low (note the difference in scales on the horizontal axis in Figure 3.9). During the high-demand period, all police cars experience a high workload due to the high CFS intensity and slow response speeds due to traffic. Thus, the districting plan has little to do with the workload officers experience during this busy time; for the most part, the police cars are all responding to CFS. This observation yields an important insight for the CPD. Counter-intuitively, the districting plan becomes most relevant when CFS intensity is low and less important when CFS intensity is high. This is because when CFS intensity is low, the officers spend most of their time patrolling, but when CFS intensity is high, all officers are responding to calls rather than patrolling (on average, 80% of available officer man-hours are employed responding to calls during this period). During the peak rush-hour periods, it may be possible to significantly reduce the average CFS response time by positioning police cars throughout the city near those locations most likely to need CFS during this busy time instead of having officers attempt to both patrol throughout the districts and respond to calls, especially since officers spend relatively little time patrolling the districts they are assigned. In discussions with the CPD, they verified this effect and commented that the system dynamics observed in the agent-based model seemed to correspond closely to that experienced by their officers. In this case, the agent-based simulation model reveals complexities in behavior and applicable insights that the other two evaluation methods do not provide.

3.5.4 Conclusion and Discussions

Our results indicate that all three evaluation methods produce very similar scores for workload variation when CFS intensity is low enough that the car patrols can meet the demand in their own sectors. However, when the in-district demand exceeds in-district supply, police patrols begin crossing boundaries to meet demand in other police sectors at a very high frequency. This scenario produces a level of complexity that the closed form and discrete event approaches are not well-equipped to handle. Only the agent-based simulation model accurately represents the resulting complexities and significantly changes the workload variation scores to reflect the behavior of the system. The significant insight the agent-based model provides is that, because call volume is so high, officers rarely patrol their sectors in this period, instead spending most (on average about 80%) of their time responding to calls both in and out of sector. The visualization of the system's complexities the agent-based model provides was also helpful in validating the performance of the simulation with the CPD client.

The scores the three methods provide for response time were less correlated with each other in the low intensity setting than they were for workload variation (ranging between 0.45 and 0.74). However, the correlations between response time scores for the three methods were less sensitive to changes in intensity than the workload variation scores, and the discrete event and agent-based simulations maintained relatively high correlation with each other throughout all three scenarios. The closed form approach did not seem to provide good estimates as it did not have high correlation with either of the other two methods in any of the scenarios. This is probably due to the fact that this method does not account for cross boundary support, and therefore underestimates the effect out of sector CFS has on the average response time.

Future work for this study includes extending the discrete event and agent-based simulation models to dynamically change the modeling parameters for response speed, service time, and inter-arrival time over the 24 hour cycle to correspond with the rates seen in Figure 3.1. Using this approach will provide an estimate for how well the various districting plans perform over a 24 hour period in actual practice. Planned extensions to the current simulation models include more complex response rules such as call prioritization and multiple car response for certain types of calls. Planned extensions to the closed form approach include performance comparisons using other distance measures (i.e., road-network distance, Manhattan distance, etc.) and development of methods for estimating the effect of cross-boundary support on the average response time performance measure.

Chapter 4

Optimization of District Design Using Response Surfaces

This chapter introduces the application of response surface methodology (RSM) in simulation optimization of police patrol district design problem. Section 4.1 describes the iterative searching procedure of the RSM. Section 4.2 and Section 4.3 show the application of the methodology in discrete-event and agent-based simulation models, respectively. Section 4.4 summarizes the method and discusses the limitations and future work. Relevant publication can be seen in [110]

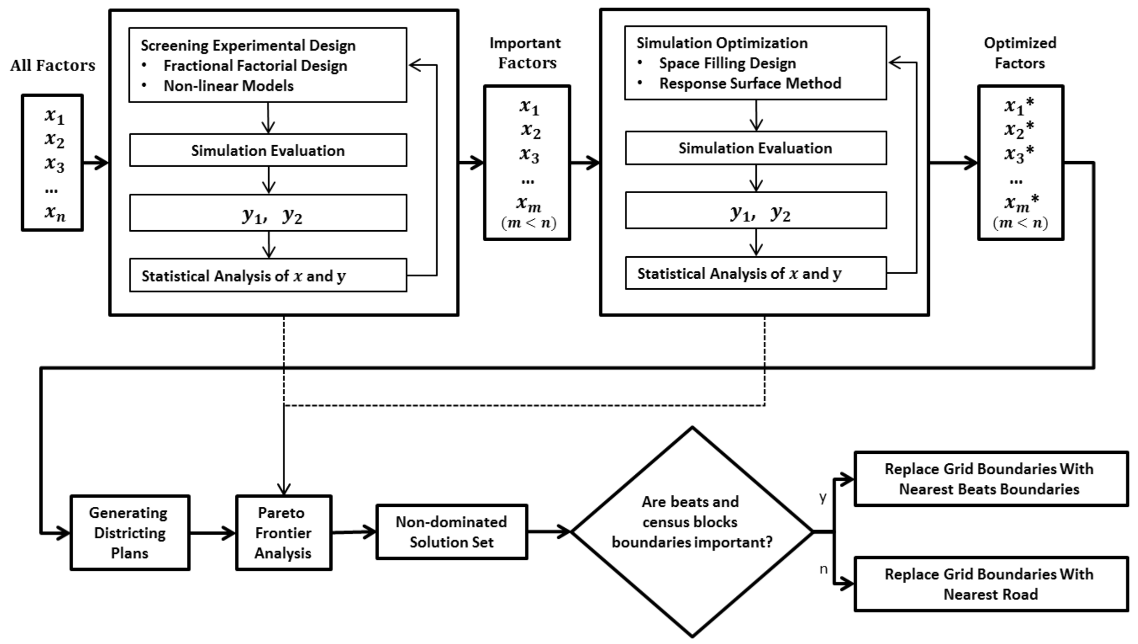
4.1 Iterative Searching Procedure

The objective of the iterative searching procedure is to study the relationship between districting parameters and the two performance measurements: average response time and variation of officers' workload among different patrol districts. Based on the relationship, the settings or values of the parameters can be improved so that better districting plans can be generated efficiently by the parameterized districting algorithm.

4.1.1 Response Variables of Simulation Experiment

As we noted in Subsection 3.2, both determinism and randomness exist in the parameterized districting algorithm. For example, some factors determine the locations and sizes of the concentric circles, such the x and y coordinates of the circles' centers, the radius of the circles. Same settings of these factors lead to same locations and sizes of concentric circles. The randomness comes from two parts: 1) the seeds on the concentric circles are randomly located, 2) the random development strategy when the districts are growing. So, overall speaking, the districting algorithm is stochastic. Therefore, the same settings of all the districting parameters can generate many different districting plans with similar concentric circles structure. The similar structure basically determines the spatially allocation of the police patrol units and the performances of patrol behaviors. There exists a limit or an upper bound for the patrol performances of different districting plans generated by same settings of parameters. So, from the perspective of experimental design, the districting parameters can be considered as the input factors of an experiment and the response variables are the top performances of a certain setting of these input factors.

Due to the evaluation complexity, we cannot use simulation model to evaluate too many different districting plans generated by same settings of districting parameters. Since the computational time for generating districting plans is much less than that for simulation evaluation, for each setting of parameters, we can use districting algorithm to generate many districting plans (e.g., hundreds of plans). Then, instead of using simulation model to evaluate all the plans, a simple and quick screening process can be conducted. As was mentioned in Subsection 3.2, some intermediate measurements can be quickly calculated immediately after a districting plan is generated, such as compactness score and standard deviation of cumulative CFS probabilities among districts. So, hundreds of districting plans can be ranked by a combined weighted score of the intermediate metrics. Based on preliminary experiments and results, there



All Factors
x_1 x_2 x_3 … x_n
Screening Experimental Design
• Fractional Factorial Design
• Non-linear Models
Simulation Evaluation
$y_1,\ y_2$
Statistical Analysis of x and y
Important Factors
x_1 x_2 x_3 … x_m $(m < n)$
Simulation Optimization
• Space Filling Design
• Response Surface Method
Simulation Evaluation
$y_1,\ y_2$
Statistical Analysis of x and y
Optimized Factors
x_1* x_2* x_3* … x_m* $(m < n)$
Generating Districting Plans
Pareto Frontier Analysis
Non-dominated Solution Set
Are beats and census blocks boundaries important?
y
n
Replace Grid Boundaries With Nearest Beats Boundaries
Replace Grid Boundaries With Nearest Road

Figure 4.1: Iterative Searching Procedure

is linear trend between variation of cumulative CFS probabilities among districts and both performance measurements. Then, only the top proportion of the ranked plans is evaluated by the simulation model. Finally, the average response times and variations of workload of the evaluated plans are used as response variables of a certain setting of parameters.

4.1.2 Description of Searching Procedure

As can be seen from Figure 4.1, the searching procedure starts from a screening experimental design for all districting parameters. Since there are about 10 parameters in the districting algorithm and their relationships with response variables are totally unknown to us, it is natural to conduct screening experiment to identify important or significant factors. Then further optimization process focuses more on the important factors. In the screening experimental design, fractional factorial design is recommended to apply. For each factor, two values are selected in a reasonable range.

The full factorial design for 10 factors has $2^{10} = 1024$ runs of experiments, which are too many for simulation evaluation. Under the assumption that some higher order interactions among factors are not significant, 1/16 or 1/32 fractional factorial design is recommended. The number of runs can be reduced to 64 or 32, respectively. If more powerful computational resources are available, more runs of experiments can be conducted. Then, using the ranking method mentioned in Subsection 4.1.1 and simulation evaluation of top proportion, response variables can be obtained. To study the relationship between input factors x and responses y, Analysis of Variance (ANOVA) can be applied. From the main effects model, we can identify the significant districting parameters. Another aspect is how much variance of response variables can be explained by all the factors. Interactions of factors can be added to the main effects model to see the significance. The important factors selected in this screening procedure should explain much proportion of the response variances.

Since important districting parameters identified in screening experimental design can explain much proportion of the response variances, the simulation optimization process can focus more on these factors. The objective is to find the best settings of these important districting parameters. The simulation optimization procedure is similar to the screening procedure. It also designs iterations of simulation experiments. Design of new experiments is based on the statistical analysis of previous iterations. Two basic searching processes can be used: 1) Space Filling Design, 2) Response Surface Methodology. Space filling design such as Latin Hypercube design can spread experimental points over the entire experimental region. After simulation evaluation, Kriging metamodel can be used to predict the response variables in the region where experiments are not conducted. The Kriging prediction can help us have a general understanding of the relationship between input factors and responses. The optimized setting of factors can also be found using the Kriging model. Response surface methodology, on the other hand, starts from a small area where the relationship between input factors and responses is assumed to be linear. Factorial design can be applied and first-order model is built. Based on the first-order model, steepest decent direction of input factors can be found. Several consecutive factorial experiments can be designed along the steepest decent direction. Central Composite Design (CDD) can be used in the experimental area where curvature of responses may occur. Curvature test needs to be conducted and second-order model can be built. Based on second-order model, the best settings of input factors can be predicted. The best settings provided by both methods can be compared. Simulation evaluation can be conducted to verify the best settings.

With optimized settings of important districting parameters, better districting plans can be generated much more efficiently than random searching. After generating many plans, all the evaluated plans, including the ones generated in screening experimental design and simulation optimization procedure, construct a pool of alternatives for final decision making. Since there are two performance measurements for a districting plan, it is a multi-objective decision making problem. No single districting plan can optimize both responses at the same time. In such situation, Pareto Frontier Analysis can be used to find the non-dominated solution set. The police department can choose one districting plan from the non-dominated solution set, based on their actual preference between response time and workload considerations. As was noted in Subsection 3.2, the districting algorithm is based on grid network. So, the districting plan is given by boundaries based on grids, which may violate some existing boundaries, such as important roads and natural boundaries. It is necessary to convert the gird boundaries to those boundaries. This procedure needs consulting with police department. If they must use some geographical units such as beats or census blocks as base units of redistricting, the grid boundaries can be replaced with nearest census blocks or beats boundaries. However, if the census blocks or beats are large, the final districting plan may be further away from optimal performances.

Another method is to replace boundaries based on grids with nearest roads. In this way, the final solution may be close to optimal performances.

4.2 Application in Discrete-Event Simulation

4.2.1 Latin Hypercube Design and Regression Analysis

In Charlottesville case study, two concentric circles are used to generate seeds in redistricting algorithm due to the shape and size of the city. 5 districting parameters in the concentric structure are involved in the districting algorithm. The descriptions of experimental factors and corresponding reasonable range of values are listed below:

- cen_x: x value of concentric circle center's geographical coordinate: (-78.508, -78.462)
- cen_y: y value of concentric circle center's geographical coordinate: (38.015, 38.051)
- n_1: the number of seeds on inner circle: $\{1,2,3,4\}$
- r_1: the radius of inner circle, represented by proportion of city radius: (0.1, 0.2)
- r_2: the radius of outer circle, represented by proportion of city radius: (0.3, 0.6)

The location of concentric circle center is represented by two factors: cen_x and cen_y. Since the total number of districts n is fixed at 8, the number of seeds on the outer circle n_2 is automatically determined by $n - n_1$. In screening experimental design, space filling design is conducted and 200 experimental points are spread over the operational space of the five factors. A Latin Hypercube method was used with the criterion of maximizing minimum distance between points. These experimental points can be viewed in the dimensions of cen_x and cen_y in Figure 4.2. We notice that some experimental points are outside the Charlottesville city boundary. The districting algorithm can find the nearest atom to the seeds on the concentric circle but outside the city region. So, the algorithm is robust and can still generate plans that satisfy the constraints of continuity and compactness.

For each experimental point, 20 districting plans are generated. After simulation evaluation, responses are obtained by averaging the performance measures of the 20 plans. Based on the observations of the scatter plots between input factors and responses, non-linear relation may exist between cen_x, cen_y and responses. So, square items of both input factors are added to the linear regression models after standardizing all factors. Linear models are built for both responses (see Listing A.1). The F tests for both models are significant at 95% confidence level. Adjusted R^2 statistics for both models are 0.95 and 0.80, respectively. Common significant factors for both responses are cen_x and cen_y. ANOVA is conducted for both linear models. cen_x, cen_y and their squared items can explain 94% variance of average response time and 79% variance for workload variation. Therefore, the location of the concentric circle center is the most important districting parameter that affects the performance measurements. Further optimization procedure can focus on cen_x and cen_y. Regression analysis also helps us understand how other factors affect responses. For example, larger radius of outer circle in reasonable range may generate plans that have better response time. More seeds in inner circle and larger radius of inner circle may lead to lower workload variation. Since the major goal of the design is to identify important factors, interaction items are not added to the model. Including them may lead to better understanding of the relationship between input factors and responses. Another important observation is that there is a strong linear relationship between two responses and the R^2 statistic is up to 0.80. So, there is no big conflict between the two responses. Parameter settings leading to better average response time may also have better performance in workload variation.

4.2.2 Kriging Model for Concentric Circle Center

Using the 200 experimental points (Figure 4.2) in Latin Hypercube design, Kriging model can be used to predict the optimal concentric circle center, the most important districting factor as noted in Subsection 4.2.1. After building the Kriging model, predictions were made for both response variables on a 40 by 40 grid structure. The 3D views of the response surfaces are shown in Figure 4.2. The ranges of x and y in middle and right panel are consistent with the range of red points in the left panel. In this way, we have a general understanding of the response surface, which can be used as guidance for further experiments. The response surfaces for two output variables show similar trend. The performances of points in central region are better than those in peripheral region of the design space. Using the Kriging model, the optimal location of concentric circles' center for each response can be found and prediction of the performances can be made. We use x_1 and x_2 to represent cen_x and cen_y. y_1 and y_2 are used for average response time and workload variation. For average response time, $(x_1^*, x_2^*) = (-78.486, 38.030)$ after conversion to original scale and $y_1^* = 267.556$ with $SE_1 = 11.705$. For workload proportion standard deviation, $(x_1^*, x_2^*) = (-78.489, 38.027)$ and $y_2^* = 3.049$ with $SE_2 = 0.118$. The two optimal points for two responses are close to each other, which coincides with the strong linear relationship between two responses as noted in Subsection 4.2.1

4.2.3 Central Composite Design

Based on the response surface predicted by Kriging model, it is conjectured that the curvature between factors and responses may exist near the optimal setting in Kriging prediction. Since the predicted optimal concentric circle centers for two responses are close to each other, one CCD can be used for both responses. We locate the CCD center in the midpoint between the two optimal concentric circle centers predicted in Subsection 4.2.2. 10 experimental points are generated. 2 of them are in the CCD

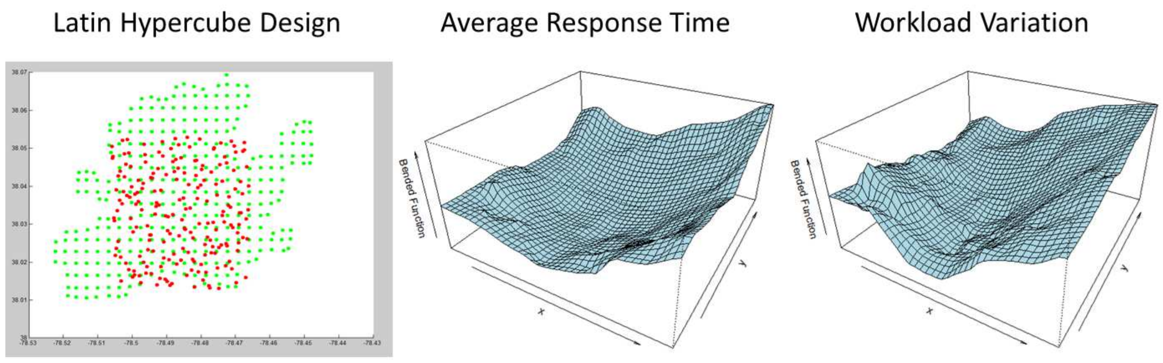


Figure 4.2: Latin Hypercube Design (red points) and 3D View of Kriging Prediction

center and others are located around the center with radius of 0.008 in Euclidean distance of geographical coordinates. Again, 20 districting plans are generated and evaluated for each experimental point. First order and second order models are built for both responses. It can be seen from Listing A.2 that the factors in first order models and models themselves do not show significance. But the second order items in second order models show significance. For example, the square items for x_1 and x_2 are significant to average response time. The square item for x_1 and the item x_1x_2 show significance for workload variation. These observations indicate the existence of curvatures for both responses in the central region of the city. Based on the second order models, prediction of responses can be made and optimal values of input variables can be found. For average response time, $(x_1^*, x_2^*) = (-78.488, 38.031)$ after conversion to original scale and $y_1^* = 270.222$ with $SE_1 = 2.463$. For workload proportion standard deviation, $(x_1^*, x_2^*) = (-78.489, 38.029)$ and $y_2^* = 3.316$ with $SE_2 = 0.099$. The locations of the best centers of concentric circles are very close to the ones found by the Latin hypercube design and Kriging method as is shown in Figure 4.3.

After comparison with the best area of concentric circles' centers and the CFS density map, it can be found that best center area is very close to the CFS density center of the city. (Figure 4.3) The possible explanation is that the size for the

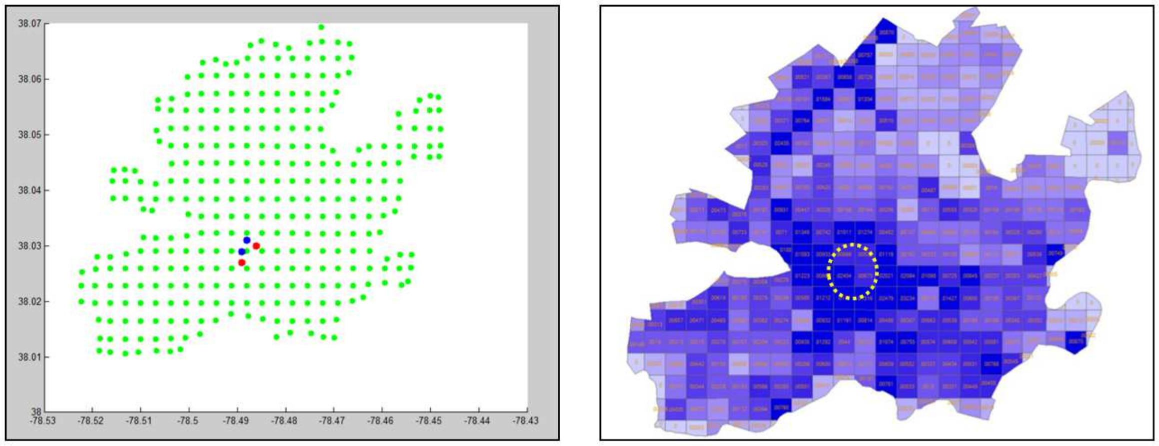


Figure 4.3: Best centers found by LHC & Kriging (red) and CCD (blue), CFS density map (right)

districts near the CFS density center should be small because it can help to balance the workload variation between districts. In the districting algorithm, if the concentric center is close the CFS density center, then the districts around the inner circle cannot grow very large because they are limited by the districts around the outer circles. On the other hand, if the center of concentric circles in the districting algorithm is located in the region where CFS density is low, the districts around the inner circle cannot grow to a large size and the CFS probabilities they cover are small. The districts around the outer circle will cover larger proportion of CFS probabilities and will lead to larger workload variation.

4.2.4 Pareto Analysis

Using improved districting parameters, more better districting plans can be generated efficiently. The comparison with plans generated by random parameters is shown in Figure 4.4 and Figure 4.5. Each point in the figure represents a districting plan. The x and y axes represent simulation evaluation of average response time (ticks in simulation) and workload variation (standard deviation of districts' workload percentages). The strong linear relationship between two responses mentioned in Subsection 4.2.1 is

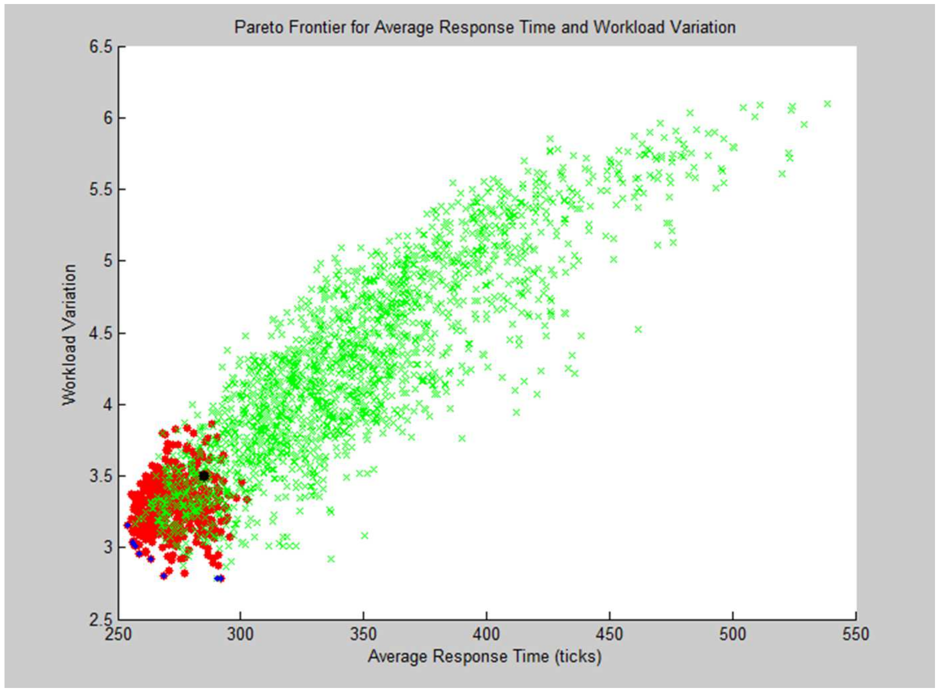


Figure 4.4: Pareto Analysis for Discrete-Event Models

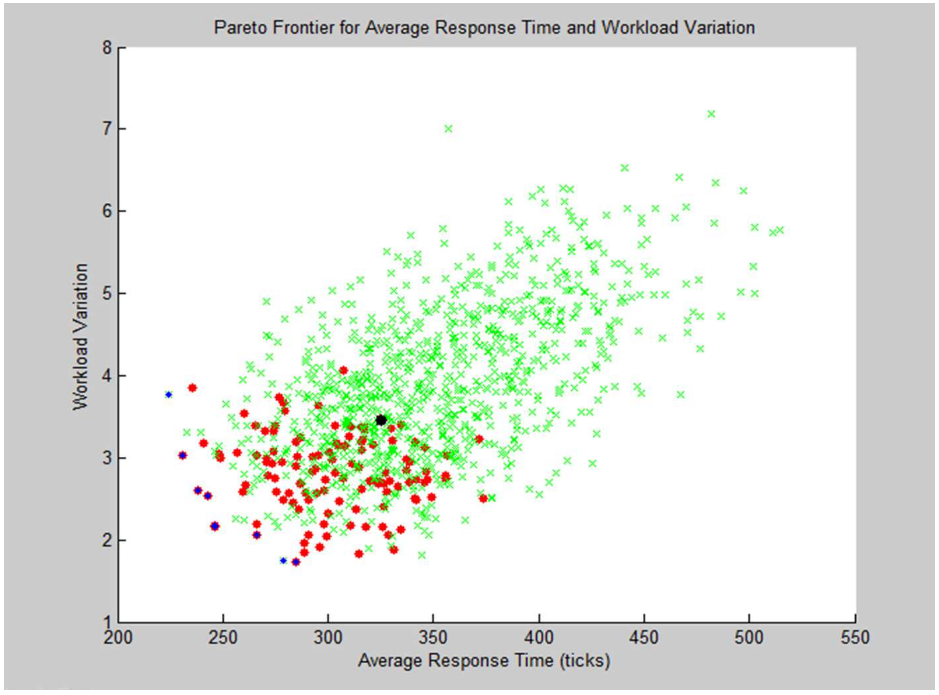


Figure 4.5: Pareto Analysis for Agent-Based Models

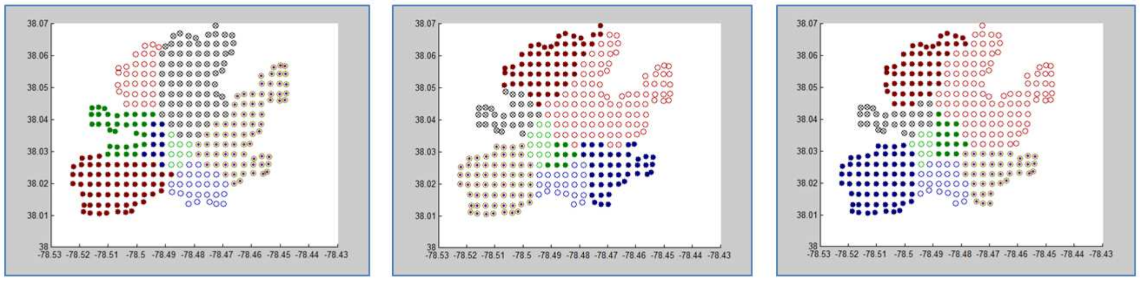


Figure 4.6: Sample Districting Plans From Pareto Non-dominated Solution Set

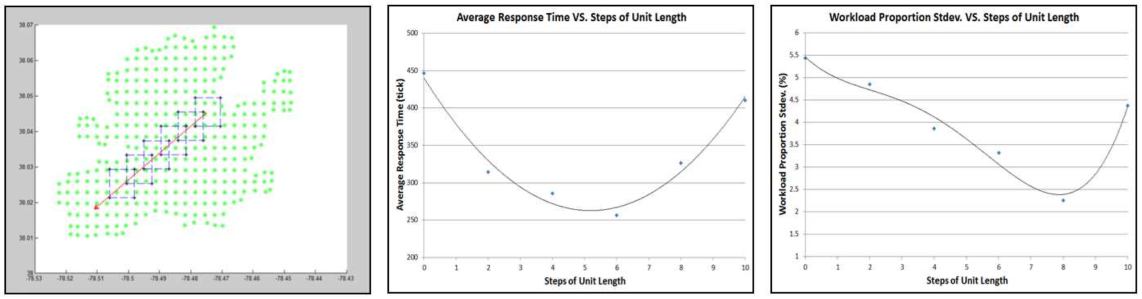


Figure 4.7: Optimization Using Response Surface Methodology for Agent-Based Model

clearly shown in Figure 4.4. It can be seen that plans generated by improved parameters (red points) are significantly better than randomly generated plans (green points) and current plan (black point). Pareto analysis is applied to find the non-dominated solution set (blue points). Most plans in this set are generated by improved districting parameters. Samples of these plans are shown in Figure 4.6. They all have the similar concentric circle structure. These plans provide alternatives of districting plans for police department and the final decision depends on the trade-off between two responses. The last step is to replace the boundaries based on grids with boundaries based on existing geographical units (police beats, census blocks) or nearest roads, depending on police department's practical considerations [112].

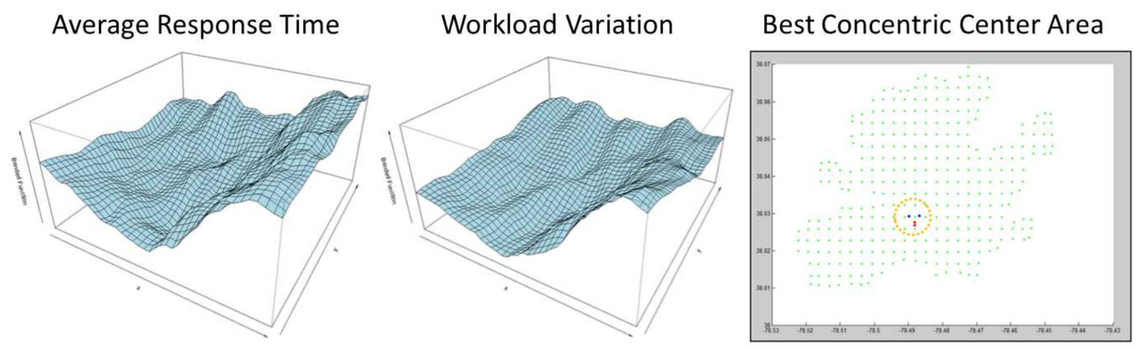


Figure 4.8: 3D View of Kriging Prediction and Best Concentric Circle Area

4.3 Verification Using Agent-Based Simulation

The iterative searching procedure can be tested and verified using agent-based police patrol simulation model (dynamic view: http://www.youtube.com/watch?v=PfdM8Ob1TuM). In the screening experimental design phase, factorial design is conducted for districting parameters. All factors are significant to responses and cen_x and cen_y can explain 82% variance of average response time and 50% for workload variation. The location of concentric circle center is the most important parameter. In simulation optimization procedure, Response Surface Methodology (RSM) is applied to optimize cen_x and cen_y. First, a 2^2 factor design of experiment was conducted for cen_x and cen_y in a small area in peripheral region of city. Based on the first order model, steepest decent direction was identified and step length was calculated. Then, a series of 2^2 experiments were conducted along the steepest direction at 2, 4, 6, 8 and 10 steps. The average performances versus the step number are plotted (Figure 4.7). Both responses were reduced along the steepest decent direction. The lowest average response time occurred at step 6 and workload variation reached its best value at step 8. Then, two CCD experiments were conducted for two responses and second-order models were built for them. The predicted optimal location of concentric circle center (red points) is shown in the right panel Figure 4.8. Latin hypercube design is also conducted for verification. The response surfaces and optimal concentric circle location predicted by Kriging model can be seen in Figure 4.8. The response surfaces are similar to the ones for discrete-event model. The optimal circle centers (blue points) are close to the ones in RSM, which coincides with ones found using discrete-event simulation (Figure 4.3). Districting plans generated by improved parameters have significantly better performance than randomly generated plans (Figure 4.5). Therefore, the iterative searching procedure also works when using agent-based simulation for evaluation of districting plans.

4.4 Conclusion and Discussion

An iterative searching procedure is proposed to find optimal or sub-optimal districting plans. Experimental design methods are used to study the districting parameters in a redistricting algorithm and the performance measures of the districting plans generated. Screening experimental design such as fractional factorial design is used to identify important parameters. Then further simulation optimization procedure focuses on the important parameters. Then improved districting plans can be used to generate good districting plans efficiently. This iterative searching procedure is tested in both DE and AB simulation. Latin hypercube design and central composite design are used for DE model while factorial design and response surface methodology are applied for AB model. Among the five parameters in the districting algorithm, the location of concentric circle is the most important factor. It can explain a large proportion of response variance. There is a strong linear relationship between two responses so the best concentric circle centers for two responses are close to each other. The best region for the center found using AB and DE models are similar and coincide with downtown area of the city. The performance of districting plans generated by improved parameters is significantly better than randomly generated plans. Compared with an adjusted simulated annealing (SA) approach, response surface method has better searching efficiency while the evaluation of optimal plans is slightly worse than SA method. Thus, using experimental design method, we achieve the goal of generating good districting plans efficiently.

The experimental design methods in iterative searching procedure are based on the parameterized districting algorithm. The original solution space of graph partitioning problem is transformed into the space represented by several districting parameters in reasonable continuous ranges. This conversion makes it convenient to use experimental design methods for optimization of districting plans. In addition to design of simulation experiment, other simulation optimization methods, such as genetic algorithm and simulated annealing can also be applied with the parameterized districting algorithm for optimization. In Charlottesville case study, seeds of districts are located on two concentric circles. But the pattern or structure of seeds is not limited to the concentric circle structure in a general sense. Ellipses can be used for cities with rectangular shape. For cities with more than one business district with high CFS probability density, the circles for seeds may not be in concentric framework. Each circle can have its own parameters such as center and radius. This method can be tested in cities with different shapes and CFS patterns in the future. In this way, the parameterized districting algorithm and iterative searching procedure will be more general to apply in the decision making process of police department. The approaches included in this study such as discrete-event and agent-based simulation, parameterized districting algorithm and experimental design method for simulation optimization can also be developed and applied in other urban emergency response services such as fire-fighting and ambulance services. Similar methodologies can be developed for resource allocation problems, districting problems and location optimization problems.

Chapter 5

Optimization of District Design Using Heuristic Algorithms

This chapter shows the application of two heuristic optimization algorithms in police patrol district design problem. Section 5.1 describes an adjusted simulated annealing approach and its application in Charlottesville Police case. Section 5.2 applies genetic algorithm in the police districting problem. Comparative study is conducted between heuristic methods and response surface methodology. Section 5.3 summarizes the methods and discusses the limitations and future work. Relevant publication can be seen in [113].

5.1 Simulated Annealing

This Section shows how to apply simulated annealing algorithm in police patrol district design problem. Sub-section 5.1.1 reviews an existing simulated annealing (SA) approach in police patrol district design. Sub-section 5.1.2 describes the adjustment made based on the existing method and Sub-section 5.1.3 shows its application in discrete-event simulation model in Charlottesville case. Sub-section 5.1.4 compares the adjusted SA method with the response surface method introduced in Chapter 4.

5.1.1 Existing Simulated Annealing Method

The existing simulated annealing approach [33] uses Patrol Car Allocation Model (PCAM) [28] as the evaluation tool of districting plans. Given the total number of police cars and a patrol district design represented by partition of atoms (atomic geographical units), the PCAM can first allocate police cars to districts, then evaluate the patrol performances such as average response time and workload variation. The simulated annealing algorithm starts with an initial districting plan as current solution. In each iteration, a new solution is generated from the current plan by reassigning an atom on the border between two adjacent districts. After evaluation of PCAM, if the new districting plan is better than the current solution, the new plan becomes the current solution. If it is not better than the current solution, then it may become the current solution with an acceptance probability. The probability of accepting the inferior solution is determined by the temperature parameter and the difference of evaluation between the new solution and current solution. The simulated annealing process starts searching the solution space in a highly random manner to prevent confinement to local optima. As the algorithm progresses, the temperature gradually decreases and the probability of inferior solution acceptance also decreases. So the current solution tends to avoid moving dramatically, and the algorithm searches locally for a better solution in the neighborhood of current solution. Eventually, it is desired that the algorithm stops with an optimal or near-optimal solution.

In each iteration, the neighborhood solution is defined by making slight changes on current solution. In the current districting plan, one atom is randomly selected on the border between two adjacent districts. After swapping the atom between the two districts, the new plan is generated. However, not each new plan can be considered feasible to replace the current solution. The algorithm has several conditions and constraints to constitute feasibility. These constraints include: 1) The ratio of areas of the biggest and smallest districts should not exceed a specified bound; 2) The atoms of any district should be connected to each other; 3) Compactness of any district (the ratio of the longest Euclidean path and the square root of the area) should not exceed a specified bound; 4) Convexity constraint. The district which receives a new atom should not have a protrusion. Similarly, the district that loses an atom should not have an indentation. More specifically, the following three situations are considered infeasible situations that violate the convexity constraints: "(i) an exchange which causes an atom to be too far away from the closest atom in its district; (ii), an exchange which causes an atom to be adjacent to only one other atom in its district; (iii), if A is the exchanged atom and B is an atom adjacent to A in the receiving district, then B must be adjacent to at least four atoms (including A) in its district."[33]

These constraints ensure the solution generated is implementable. The size constraint avoids the possibility of generating districts that are too large or too small. The contiguity constraint keeps patrol cars from crossing district boundaries. The compactness and convexity constraints avoid gerrymandering and ensure that a district has a relatively round rather than long and slender shape. In the Buffalo police case study, 409 R-districts (atomic geographical unit) are partitioned into 5 districts. The simulated annealing algorithm begins with the current district design in Buffalo and runs for 400K iterations. The actual computational time is 2 hours in a 450MHz personal computer.

In our study, it is of great interest to integrate the existing simulated annealing algorithm with the discrete-event simulation for district design evaluation. However, the current simulated annealing approach cannot be applied directly in Charlottesville police case study. First of all, the discrete-event simulation takes much longer time to evaluate a districting plan than PCAM. The PCAM uses less than 0.018 second to evaluate a district design while the discrete-event model uses 5 seconds. The estimated total computational time is 23 days for 400K iterations, which is not realistic in

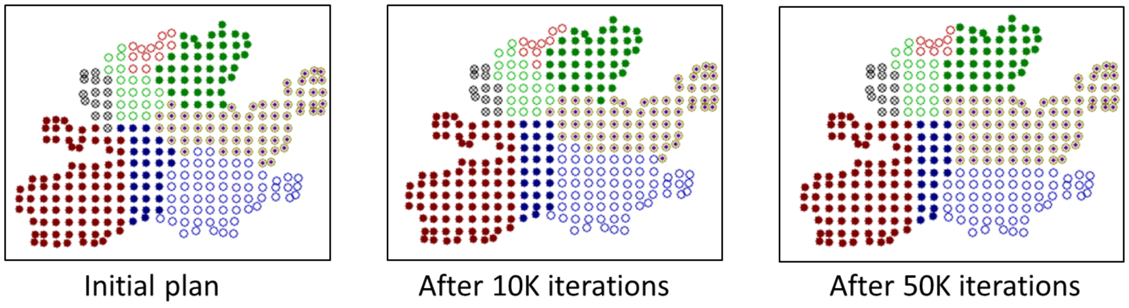


Figure 5.1: Districting Plan is confines to a local optima

practice. Secondly, the convexity constraints cannot be applied to atoms in the form of grid network. The convexity constraints are so strong that the solution tends to get stuck in a local optimal. For example, the convexity constraint (ii) alone will lead to the situation shown in Figure 5.1. It can be seen that the layout or structure of districting plan does not change so much after 50K iterations of simulated annealing process. The initial plan is a bad plan and the final plan is still far from the optimal configuration based on previous study. Similar problem occurred if convexity constraint only includes situation (iii) even if the "at least four atoms" condition is relaxed to 3 or 2 atoms. These convexity constraints work well in Buffalo case because the base atomic units are "R-districts" and most of them have more than 4 adjacent atoms. However, in Charlottesville case, most atom grids are squares which have at most 4 adjacent atoms. The convexity constraints limit the alternatives of possible changes and thus confine the solution in local optima. If we remove the convexity constraints (ii) and (iii), gerrymandering problem will occur. Therefore, small adjustments on the existing simulated annealing approach may not work in Charlottesville grid network case. In the next section, an adjusted simulated annealing algorithm is described, which works for atoms in grid network (and other types of atoms) and is well-suited for discrete-event simulation evaluation.

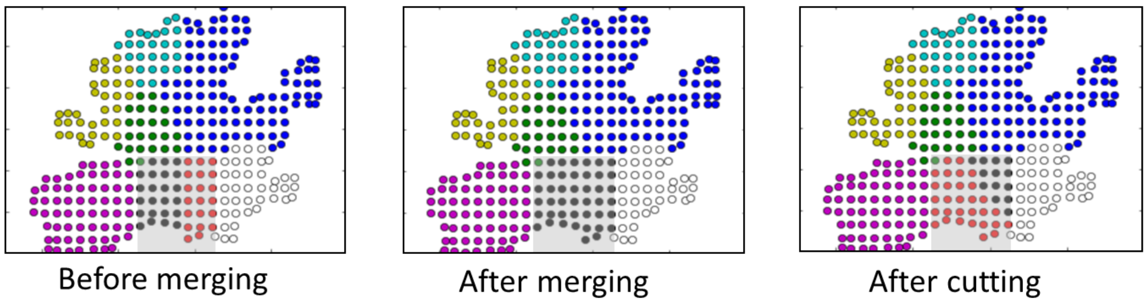


Figure 5.2: Merging and Cutting Process

5.1.2 Adjusted Simulated Annealing Algorithm

The basic simulated annealing process in the new algorithm is the same as the existing approach in Section 5.1.1 The major adjustment made for the original approach is the definition of solution neighborhood. In existing method, the new districting plan is generated by changing only one atom based on current plan. In adjusted algorithm, relatively "big" changes are made to current solution in each iteration.

In step 1 of the algorithm (Figure 5.2), relatively big changes are made to current districting plan by merging and cutting 2 adjacent districts. To ensure the randomness of the algorithm, the two adjacent districts are randomly selected and the cutting angle is also randomly chosen. When cutting the merged district, the x and y coordinates of the "center of mass" are calculated by taking the weighted sum of all atoms' coordinates based on CFS probabilities. In this way, the two districts after cutting will have almost equal CFS probabilities. Due to various shapes of the combined district, the random cutting process may produce districts that violate the contiguity constraint. If that happens, the changes made by the merging and cutting process are canceled. Since it is important to make a big change to the current plan, the merging and cutting procedure is tried on all pairs of adjacent districts until there is a successful one.

Since the merging and cutting process of step 1 is in a highly random manner,

Algorithm 1 Adjusted Simulated Annealing Algorithm

```
1: Merge and cut random 2 adjacent districts
       ∗ Randomly select two adjacent districts from all pairs
       ∗ Merge them into one and randomly cut it into 2 districts
           · Find the “center of mass” of the combined district
           · Select a random angle and cut through the center
       ∗ Conduct contiguity check. If it fails, cancel the changes
       ∗ Repeat this process on all pairs of districts until there is a successful “merge and cut”
       ∗ If all pairs fail, the program goes to the next step
2: Cut one “bad” district and merge two smallest districts
       ∗ “Bad” districts: high CFS probability, bad compactness
       ∗ Cut one “Bad” district into two districts
       ∗ Choose the “smallest” two adjacent districts to merge
       ∗ Conduct contiguity check. If it fails, cancel the changes
3: Seeds-growing procedure
       ∗ Find “center of mass” of each district
       ∗ Find atoms closest to “centers of mass” and use them as seeds in the re-growing procedure
       ∗ Mark All atoms as “unassigned”
       ∗ Assign each seed to a district
       ∗ An iterative process of seeds growing
           · Develop the district with min CFS probability
           · The district’s unassigned neighbor atoms are assigned to the nearest district
           · Repeat this process until it stops
           · The smallest district cannot grow anymore (all its neighbor atoms are assigned to districts)
       ∗ An iterative process for remaining unassigned atoms
           · From unassigned atoms adjacent to all current districts, randomly select one atom
           · Assign it to the nearest adjacent district
           · Repeat this process until all atoms are assigned
```

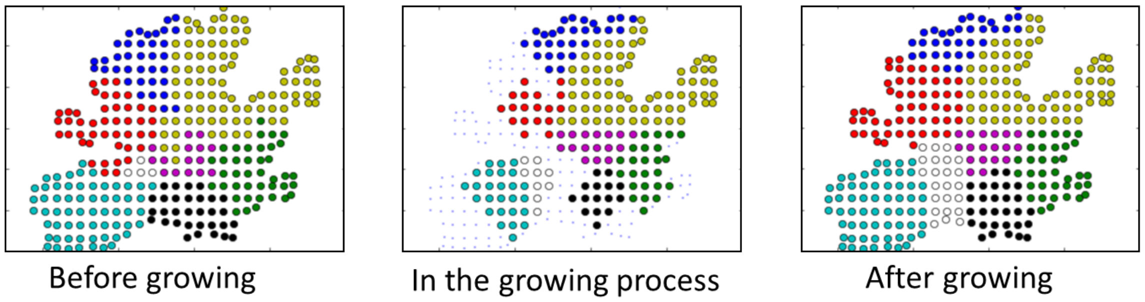


Figure 5.3: Seeds Growing Process

the derived districting plan may have very large districts or districts with bad compactness. If the district's cumulative CFS probability or compactness exceeds a pre-specified upper bound, it is considered as a "bad" district. If there is no "bad district", step 2 can be skipped. If there are many "bad" districts, the worst district is cut into 2 districts. The cutting procedure is similar that in step 1. Usually, the worst district has gerrymandering issue and the cutting tends to produce districts with contiguity violation. So a pre-specified number of trials of cutting can be conducted with random cutting angles. In this way, the chance of successful cutting is increased. Based on previous study, the districting plans with lower variation of CFS probabilities among all districts tend to have better performances on average response time and workload variation. Step 2 can help balance the CFS probabilities among districts so that the search for optimal plans are kept in a solution set that has relatively low CFS probability variation. Step 2 can also break the gerrymandering district into 2 districts and in most cases each of them has better compactness.

Step 1 makes big random changes to the original districting plan and step 2 adjusts the plan so that districts become more compact and the CFS probabilities are balanced among districts. However, the random cutting and merging process may make the district boundaries serpentine. Step 3 of the algorithm uses the "seed-growing" procedure to smooth the district boundaries while keeping the original pattern or structure of the districting plan (Figure 5.3). The "center of mass" of each district serves as a "seed" and these seeds ultimately grow to districts. In each iteration of the growing procedure, the district with the lowest cumulative CFS probability has the priority to acquire adjacent neighbor atoms. So, the CFS probabilities can be balanced among districts. When districts grow, the neighbor atoms are usually assigned to the nearest adjacent district. In this way, the shape of the district tends to be round and thus the compactness of the districting plan can be guaranteed.

5.1.3 Application in Charlottesville Police Case

As is noted in Section 3.1, the goal of Charlottesville case study is to find an optimal or near-optimal districting plans that minimize both average response time and workload variation among districts. The districting plan is drawn by partitioning the 323 atoms in a grid network (Figure 3.3) into 8 districts with the constraints of contiguity and compactness. The adjusted simulated algorithm described in Section 5.1.2 is applied for the optimization of districting designs. The discrete-event simulation described in Section 3.4 is used for evaluation of district plans.

The discrete-event simulation outputs two measurements (average response time and workload variation) for a districting plan while simulated annealing algorithm only needs one single evaluation score. So it is necessary to develop a procedure to convert two simulation measures into one evaluation score. This procedure is developed based on 3000 random plans generated by parameterized districting algorithm described in [112]. These plans have various layouts and structures and include very good and very bad plans. They are generated before the simulated annealing algorithm and are used as a reference to develop the evaluation score. First, these plans are evaluated by discrete-event simulation so that the two measures of each plan are obtained. Then, the evaluation score of each plan is calculated based on weighted sum of standardized performance measures (0 for worst and 1 for best). The weights for average response time and workload variation is [0.4, 0.6] since reducing workload

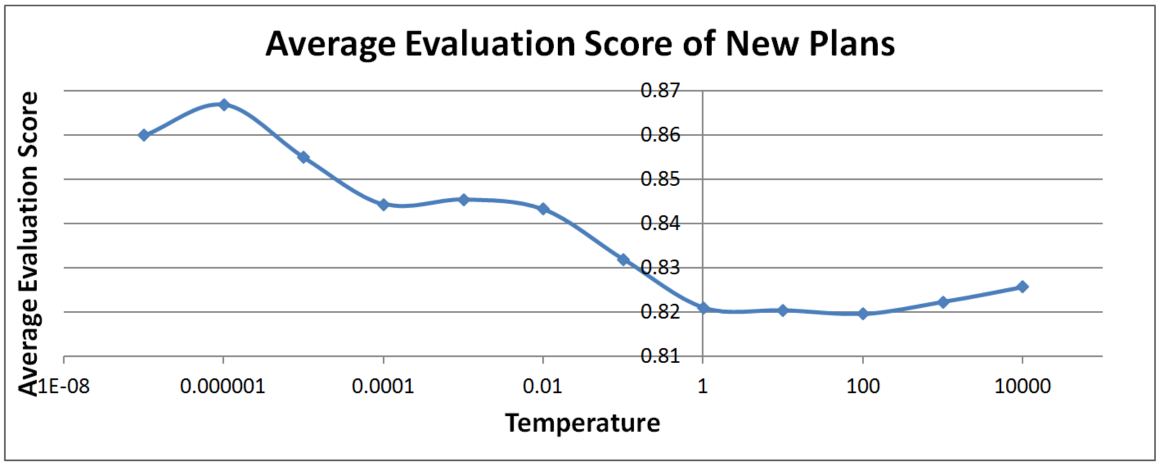


Figure 5.4: Temperature Experiments

variation is more important to Charlottesville police department. The evaluation score is 0 for the worst plan and 1 for the best plan. Lastly, the relationship between simulation measures and one evaluation score can be obtained by running regression analysis on these sample plans. Thus, this relationship can be used to convert two simulation measures into one evaluation score. Based on this relationship, if the districting plan generated by simulated annealing algorithm is better than the best plan in the random samples, the evaluation score will be greater than 1. If it is worse than the worst plan in the samples, the evaluation score will be less than 0. In most cases, the evaluation score is between 0 and 1.

The adjusted simulated annealing algorithm, instead of minimizing the objective value of a solution, maximizes the evaluation score of districting plans. Correspondingly, the probability of inferior solution acceptance p in the simulated annealing algorithm is given by: $p = exp((v(s) - v(s_0))/t)$ where $v(s)$ is the evaluation score of the new solution s while $v(s_0)$ is the evaluation score of the current solution s_0. t is the temperature of the current iteration. The parameters in the simulated annealing are set in the following way. In step 2 of the algorithm, if the cumulative CFS probability of a district is greater than 0.16, this district is considered as "bad" district and will be cut into two smaller districts. We set this upper bound to be greater than 0.125, which is the balanced CFS probability for 8 districts. For compactness, if the ratio of the square root of a district's area to its longest Euclidean path is greater than 1.5, this district will be "bad" district. These parameters can be configured based on practical requirements and considerations of police departments of different cities. The initial and the ending temperatures are determined by a temperature experiment, which is similar to the procedure described in [33]. In the experiment, the temperature is kept constant and simulated annealing algorithm runs for a large number of iterations. The average evaluation score is calculated for all the new districting plans generated in the process. Repeat this procedure for different levels of temperatures. The annealing curve in Figure 5.4 shows the increasing trend of average evaluation score when the temperature changes from high to low. It can also be seen that the average score at temperature 1000 and 10000 are slightly higher than 100. Since we desire more randomness on the prospective solutions, 100 is a better choice of the initial temperature. Similarly, since the evaluation score at temperature 1e-6 is slightly higher than 1e-7, the ending temperature can be set as 1e-6.

In the simulated annealing process, temperature is exponentially reduced from 100 to 1e-6. Experimental result shows that exponential reduction function is better than the linear one. The starting districting plan is selected from the 3000 random plans generated by the parameterized districting algorithm. A very bad plan is selected as the starting plan to show the robustness of the algorithm. It has very bad performance on average response time and workload variation, whose evaluation score is only 0.066. The simulated algorithm runs for 3000 iterations using 4 hours in a personal computer. Most of the time is spent on simulation evaluation of plans and the process of changing a current solution is relatively quick. The trends of evaluation scores of new plans, current plans and best plans in the simulated process are shown in Figure 5.5. It can be seen that the evaluation score of best plan increases very quickly at

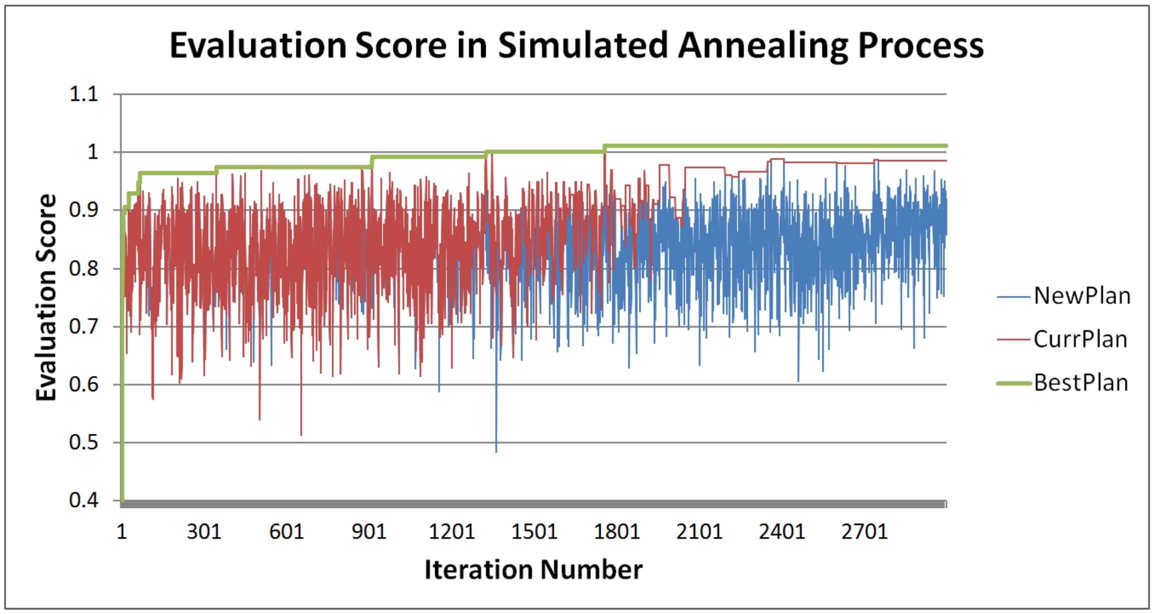


Figure 5.5: Evaluation Scores in SA Process

the first several iterations and has a stable improvement in the rest of the process. The reason why the algorithm can jump out of the bad plans very quickly is that the algorithm cuts large districts into small districts and very small districts are merged into median-sized districts so that the CFS probabilities of districts are balanced. Districting plans that has lower variation of CFS probabilities among districts tend to have better performances measures. The final best plan has a evaluation score that is greater than 1, which is better than the best plan in the set of random plans. The current plan changes in a highly random manner in the first half of the process due to the high temperature and high probability of accepting inferior solutions. The current plan tends to be stable in the rest of the process because the temperature gradually decreases which lowers the acceptance probability. The evaluation score of new plans shows drastic fluctuation in the whole process because the solution neighborhood is defined to make relatively big changes to current plan.

The computer program records all the districting plans generated in the simulated annealing process. Figure 5.6 shows the top 4 plans after ranking them by the evalua-

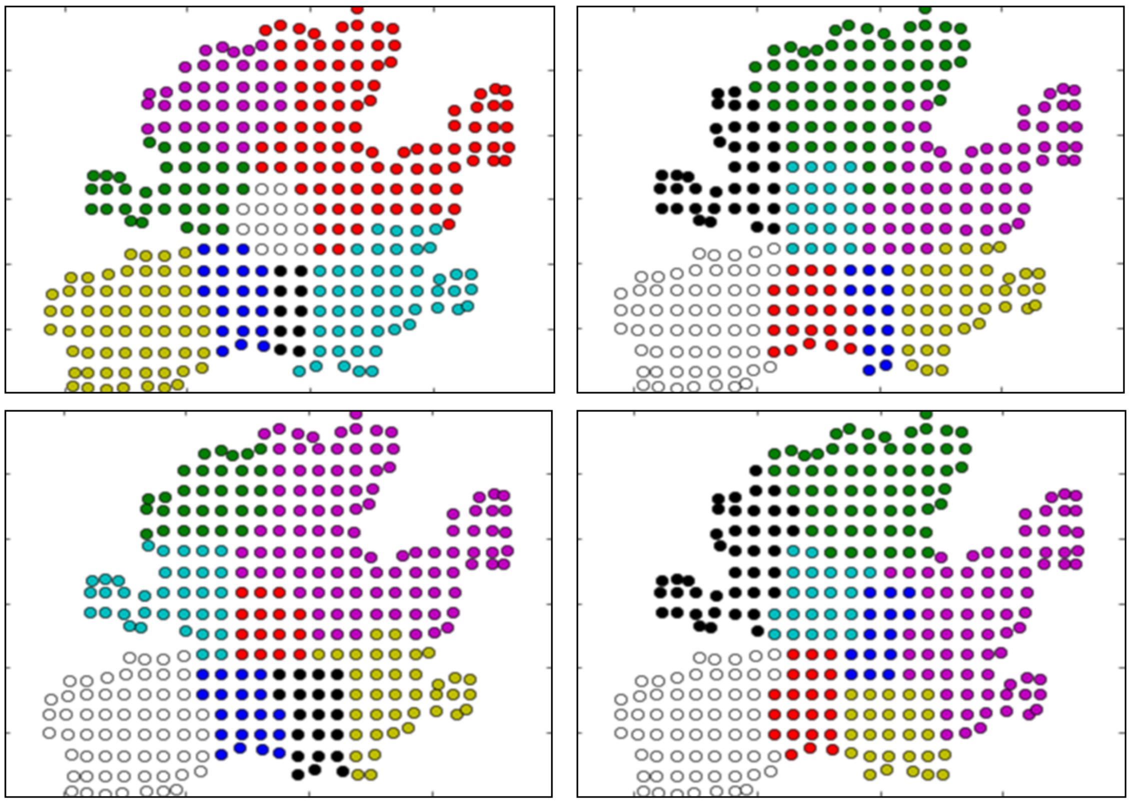


Figure 5.6: Best 4 Districting Plans Generated in SA Process

tion score. It can be seen that they all have similar patterns. Three small districts are near the downtown area of the city where the CFS probability is relatively high and the other five districts are on the periphery of the city where the CFS probability is lower than the central region. This pattern of districts found by simulated annealing algorithm is very similar to the one found in previous study [112]. Compared with the districting plan generated in the old simulated annealing approach [33], the compactness of some districts is not very good and the boundaries of districts are not very smooth and sometimes convex. However, the top plans generated by new algorithm can provide general structure of districting plans for the police departments. The final police patrol districts can be drawn by replacing grid boundaries by nearest roads or boundaries of existing geographical units such as police beats or census blocks.

5.1.4 Comparison with Response Surface Districting Method

Figure 5.7 shows the comparison of districting plans generated by the adjusted simulated annealing algorithm, response surface optimization approach and parameterized districting plans. Each point represents a districting plan which is evaluated by the same discrete-event simulation. Since the goal is to minimize both average response time and workload standard deviation, the points on the left bottom part of the graph represent better districting plans. The blue points are the 3000 plans generated by randomizing districting parameters of the parameterized districting plans [112]. These plans include very good and very bad plans. The performance measures of them have a large range so that the random search is very inefficient. The green points are districting plans generated by a response surface optimization methodology.

An iterative search procedure is conducted in a series of simulation experiments on districting parameters in parameterized districting algorithm. In this procedure, important districting parameters are identified and optimized by experimental design and response surface methodology. Using improved districting parameters, the range

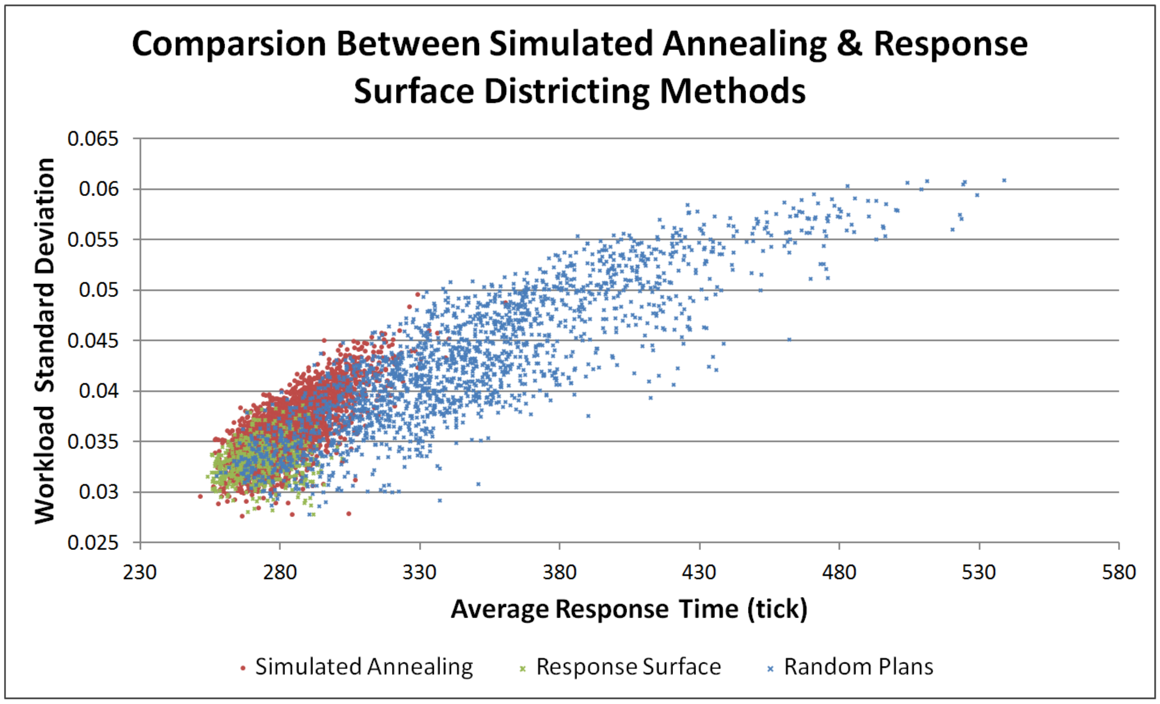


Figure 5.7: Comparison with Response Surface Districting Method and Random Plans

of performances is relatively small and in the top proportion of random plans. So, the response surface optimization method can generate good districting plans efficiently. The red points represent plans generated in the adjusted simulated annealing approach. The range of performance of these plans is much smaller than the random plans but is bigger than that in response surface method. So, the search efficiency of the response surface method is better than the simulated annealing approach. But it should be noticed that the searching procedure in response surface method is a iterative process. One iteration of experimental design depends on the statistical analysis of simulation output of the previous step. On the contrary, the simulated annealing algorithm searches districting plans in an automatic manner. The user only needs to set the parameters of the simulated annealing algorithm and let the program run. After the program is running for a pre-specified number of iterations, districting plans can be generated automatically. There is no need to analyze the intermediate result in the middle of the process. In addition, Figure 5.7 shows that the optimal plans in simulated annealing algorithm are slightly better than those in response surface method. More red points are in the Pareto frontier of plans. Therefore, we conclude that the adjusted simulated annealing approach is better for practical use than the response surface optimization method.

5.2 Genetic Algorithm

5.2.1 Apply GA to Police Districting Optimization

Genetic Algorithm (GA) is another heuristic algorithm that can be used in district design optimization. The basics of GA can be seen in sub-section 2.7.1. [16] shows how to apply Genetic Algorithm to the political districting problem. When applying GA to a specific problem, we need to encode the solution into chromosomes and select the proper fitness function. [16] uses the centroid of each district to represent the configuration of districting plan. The solution is a sequence of district centroid represented by the numbered atomic geographic unit or the coordinates of each centroid. In this optimization problem, compactness and population equality are treated as objectives while the contiguity is treated as a quasi-hard constraint [16].

Inspired by the encoding method in [16], we use the coordinates of district seeds to represent the configuration of district plan in police patrol districting problem. The latitude and longitude of each seed are included in the solution. The solution is encoded as $\{x_1, y_1, x_2, y_2, ..., x_n, y_n\}$ where x_i and y_i represent the longitude and latitude of each seed, respectively, while n is the number of districts to be designed. Then the "seeds-regrowing" sub-routine described in sub-section 5.1.2 is used to generate districting plans given seed locations. After simulation evaluation, the average response time and workload variation as well as compactness are combined into a weighted score as the fitness of the individual. The procedure to form the objective

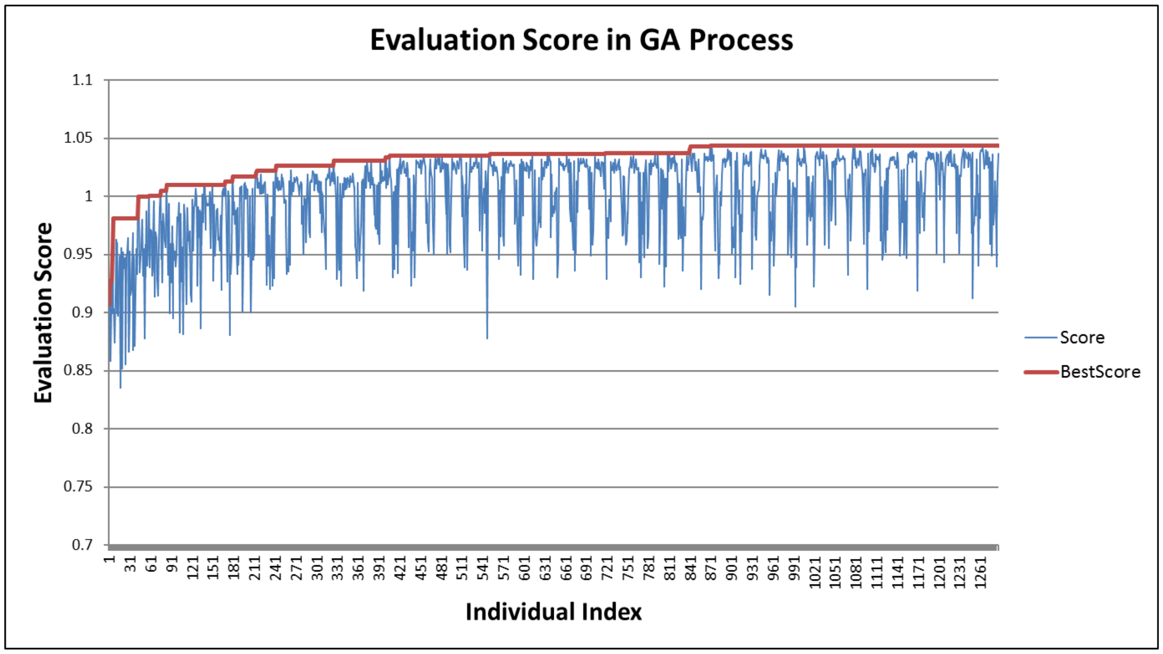


Figure 5.8: Evaluation Scores in GA Process

function is similar to that in the adjusted simulated annealing approach described in sub-section 5.1.3.

To apply the GA to Charlottesville Police districting study, the Java Genetic Algorithms Package (JGAP) [77] is integrated with existing Seeds-growing Algorithm as well as Police Patrol Discrete-event Simulation Model. Since we need to design 8 districts for Charlottesville, the individual solution is encoded using 16 variables representing the longitudes and latitudes of 8 seeds. The ranges of longitudes and latitudes are the same for all seeds. The range of x_i is (-78.508, -78.462) and y_i is taken from (38.015, 38.051), which is the same as the range of concentric circle location in Figure 4.2. The number of generations is set to be 50 and each generation has 20 individuals. All other configurations are set as default values in the package. It take 1 hour of actual computational time to finish the process in the normal PC. After the GA process, the evaluation score of individuals versus the order of individual being evaluated is plotted (Figure 5.8). It can be seen that the initial individuals already

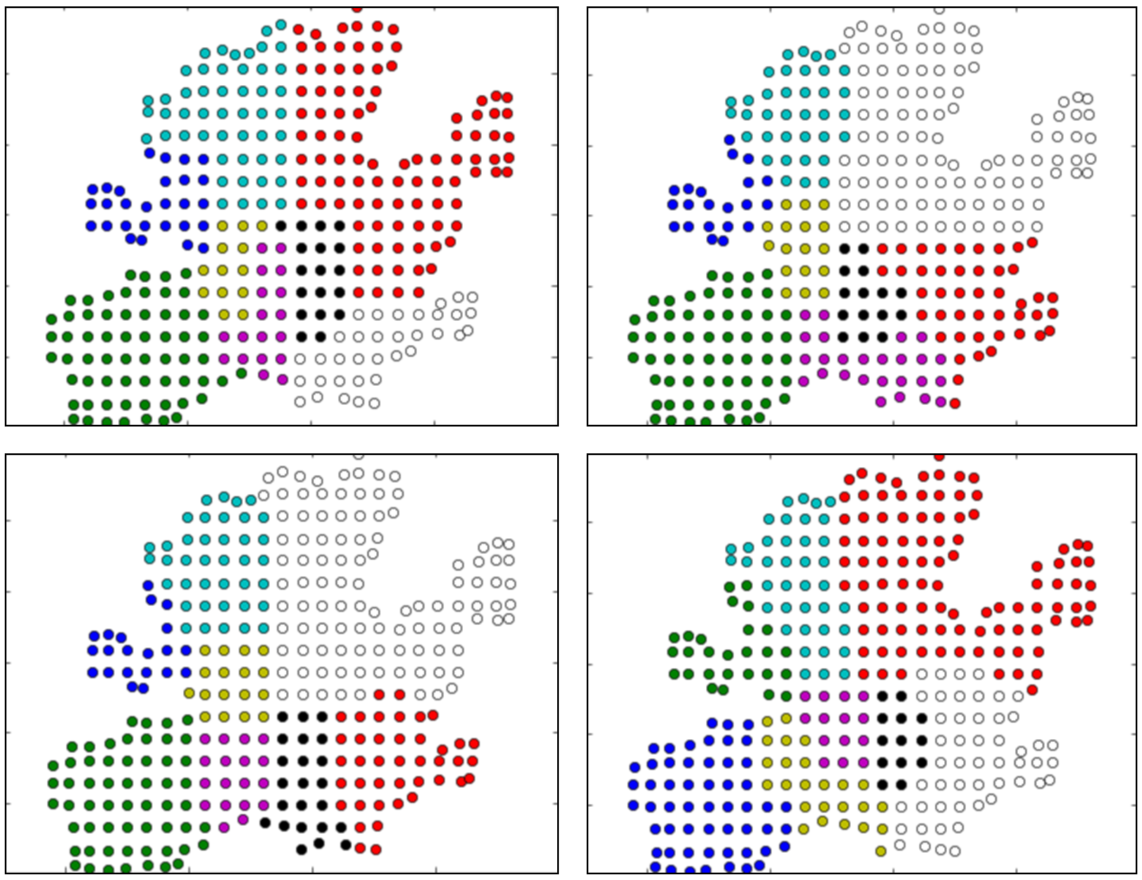


Figure 5.9: Best 4 Districting Plans Generated in GA Process

have relatively good performances. This is because each seed location is randomly selected in the given range so the seeds tend to spread evenly in the reasonable scope and the corresponding districting plans are more likely to balance the workload. So very bad plans are not likely to occur in the beginning phase. There is a periodic fluctuations of individual score because in each generation there are some good plans and bad plans due to the evolutionary process such as crossover and mutation. The best score increases significantly in the first 20 generations and then tends to be stable. There is another small increase of best score in generation 33. The top 4 plans ranked by fitness are plotted in Figure 5.9. Basically, the configuration or layout of these best plans are similar to those found by other optimization approaches.

5.2.2 Comparative Study

The simulation evaluation of average response time and workload variation of districting plans generated in the GA process are plotted together with randomized plans and plans generated using response surface methods and adjusted simulated annealing approach (Figure 5.2.2). The black dots represent the plans generated by GA process. It can be seen that the range of GA search is much smaller than that of randomized plans and is similar to range of SA search. Unlike RS and SA, more search points tend to be in the optimal part and the range of dense area is similar to that in RS search. The best performances on response time of GA plans are a little worse than RS and SA but GA performs better in reducing workload variation. The relative weights for different performance measures in the GA objective function can be adjusted to satisfy users' requirements. Therefore, the search efficiency of GA is worse than RS but is better than SA. Compared with RS, GA is a more automatic method. Compared with SA, the GA districting method is relatively easy to implement because there is no cut and merge process, which also affects the computational efficiency of each iteration.

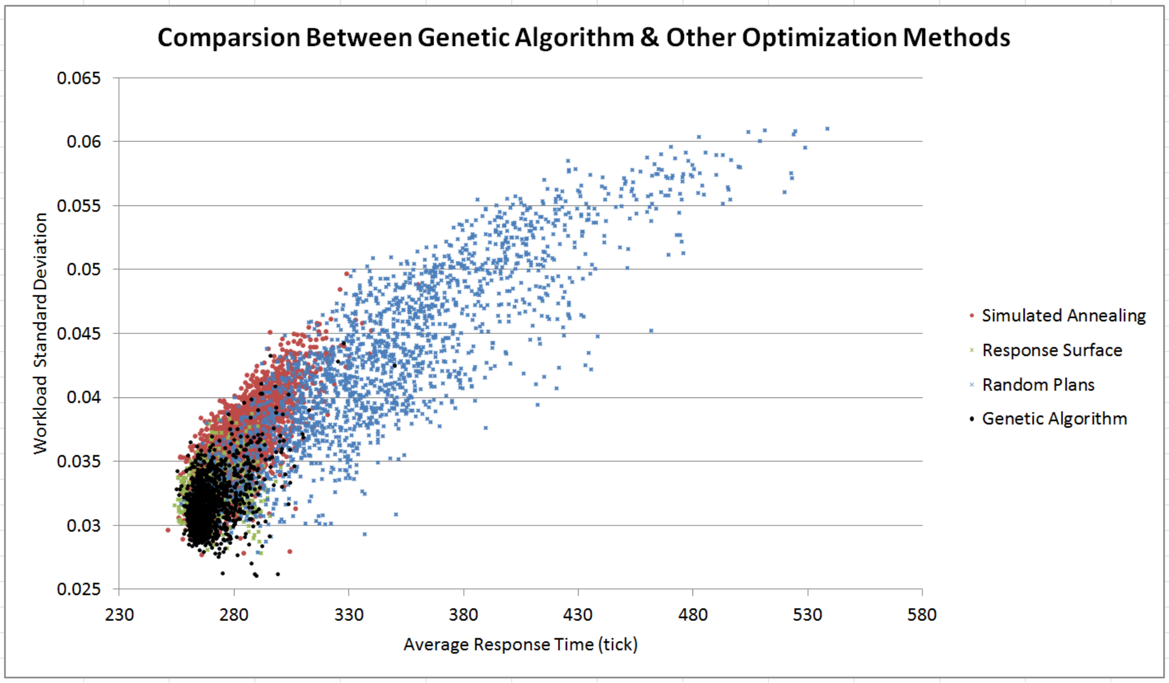


Figure 5.10: Comparison Between GA and other Optimization Methods

5.3 Conclusion and Discussion

In this Chapter, two heuristic optimization methods are applied to find optimal or near-optimal police patrol districting plans. In the adjusted simulated annealing approach, the police patrol discrete-event simulation is used to evaluate the average response time and workload variation among districts. The adjusted approach defines the solution neighborhood in a new way. Instead of changing the assignment of only one atom, the new approach makes relatively big changes to current solution through a cutting and merging process of districts in current district plan. The experimental result of the Charlottesville case study shows that the new approach uses less iteration to reach good solutions, which is well-suited for discrete-event simulation evaluation of districting plans. Another benefit of the new approach is the robustness for the connectivity and adjacency pattern of atoms. The old method's convexity constraint works well for atoms in the form of "R-Districts" in the Buffalo case but it is too strong and cannot work well for atoms in grid network where atoms have relatively fewer adjacent neighbors. The consequence is that the current plan changes very little after much iteration. Small adjustment on the convexity constraint does not work well either. In the new approach, the adjacency pattern of atoms has little influence on the algorithm. The adjusted simulated annealing approach is also compared with the response surface optimization method based on a parameterized districting algorithm in previous study. Experimental results show that the simulated annealing approach searches solutions over a bigger range of districting plans. It is an automatic procedure and the user does not need to conduct statistical analysis during the process. Thus, it is better for practical application. Experimental results also show that it can generate better plans than response surface method.

Inspired by a study of political districting optimization, the genetic algorithm is applied to the police districting problem. The districting solution is encoded in a similar way as the existing method. Seed coordinates of each district are used to represent the configuration of the districting plan. Fitness of individuals is formed as the weighted sum of average response time, workload variation and compactness. Java Genetic Algorithms Package (JGAP) is integrated with existing Seeds-growing Algorithm as well as Police Patrol Discrete-event Simulation Model. The optimal plans generated by GA show similar pattern or layout as RS and adjusted SA approach. The search efficiency of GA is worse than RS but is better than SA. GA optimal plans have better performances on workload variation but worse performances on average response time. Compared with RS, GA is a more automatic method. Compared with SA, the GA districting method is relatively easy to implement because there is no cut and merge process, which also affects the computational efficiency of each iteration.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

The dissertation reviewed the characteristics of the generic territory design problem and police patrol district design problem from the perspective of past and current work. We also showed the complexity of the problem from both computational and evaluation perspectives. The dissertation described a simulation optimization methodology to find the optimal or close-optimal district design that can minimize the long-term average response time and workload variation among districts. The district design can be represented as a partition plan of atomic geographical units with the constraint of contiguity and compactness. A parameterized districting algorithm was developed to generate district plans of various patterns or layouts.

To evaluate districting plans, an agent-based (AB) simulation model and a discrete-event (DE) model were built. The agent-based model simulates basic patrolling, dispatching and responding behaviors of police cars in actual geographical information system environment (road network and locations of buildings). Calls for service (CFS) incidents are generated based on temporal and spatial pattern extracted from historical dataset. The two performance measures tend to converge as the simulation proceeds. The converged measures can be used as evaluation results of districting plans. The AB model was developed using Java Repast, which provides good visualization for system validation. The dynamic view of the AB model was validated by local police department and can be considered to represent basic patrol activities. Thus it is meaningful to use it to evaluate the long-term measures of district plans. The good visualization of the agent-based modeling tool is guaranteed by tick-by-tick updates of agents' location and status information, which is computational expensive. It usually takes more than 10 minutes to use AB model to evaluate a district plan. To improve the evaluation efficiency, a discrete-event (DE) simulation model was built based on Hypercube queuing model. Instead of tick-by-tick updates of police cars' information, the DE model only keeps track of important events of police cars. In addition, the travel time in DE model is estimated by Euclidean distance rather than the movements of police cars along the road network in AB model. In this way, the DE model can evaluate districting plans in a much more efficient way. To compare evaluations of the two models, several hundreds of random districting plans were generated and evaluated by both AB and DE model under three levels of CFS demand intensities. Experimental results show that there is strong linear relationship between AB and DE evaluations in most cases. Therefore, the DE model is recommended to use in the simulation optimization procedure because it is more computational efficient and provides similar results as AB model.

The dissertation describes how to use response surface methodology (RSM) to search the optimal district plan in a systematic way. Experimental design methods are used to study the districting parameters in a redistricting algorithm and the performance measures of the districting plans generated. Screening experimental design such as fractional factorial design is used to identify important parameters. Then further simulation optimization procedure focuses on the important parameters. Then improved districting plans can be used to generate good districting plans efficiently.

This iterative searching procedure is tested in both DE and AB simulation. Latin hypercube design and central composite design are used for DE model while factorial design and response surface methodology are applied for AB model. Among the five parameters in the districting algorithm, the location of concentric circle is the most important factor. It can explain a large proportion of response variance. There is a strong linear relationship between two responses so the best concentric circle centers for two responses are close to each other. The best region for the center found using AB and DE models are similar and coincide with downtown area of the city. The performance of districting plans generated by improved parameters is significantly better than randomly generated plans.

The dissertation also describes an adjusted simulated annealing searching algorithm to find optimal or near-optimal police patrol districting plans. The police patrol discrete-event simulation is used to evaluate the average response time and workload variation among districts. The adjusted approach defines the solution neighborhood in a new way. Instead of changing the assignment of only one atom, the new approach makes relatively big changes to current solution through a cutting and merging process of districts in current district plan. The experimental result of the Charlottesville case study shows that the new approach uses less iteration to reach good solutions, which is well-suited for discrete-event simulation evaluation of districting plans. Another benefit of the new approach is the robustness for the connectivity and adjacency pattern of atoms. The old method's convexity constraint works well for atoms in the form of "R-Districts" in the Buffalo case but it is too strong and cannot work well for atoms in grid network where atoms have relatively fewer adjacent neighbors. The consequence is that the current plan changes very little after many iterations. Small adjustment on the convexity constraint does not work well either. In the new approach, the adjacency pattern of atoms has little influence on the algorithm. The adjusted simulated annealing approach is also compared with the response surface optimization method based on a parameterized districting algorithm in previous study. Experimental results show that the simulated annealing approach searches solutions over a bigger range of districting plans. It is an automatic procedure and the user does not need to conduct statistical analysis during the process. Thus, it is better for practical application. Experimental results also show that it can generate better plans than response surface method.

This dissertation also applies genetic algorithm to the police district design. Inspired by a study of political districting optimization, the genetic algorithm is applied to the police districting problem. The districting solution is encoded in a similar way as the existing method. Seed coordinates of each district are used to represent the configuration of the districting plan. Fitness of individuals is formed as the weighted sum of average response time, workload variation and compactness. Java Genetic Algorithms Package (JGAP) is integrated with existing Seeds-growing Algorithm as well as Police Patrol Discrete-event Simulation Model. The optimal plans generated by GA show similar pattern or layout as RS and adjusted SA approach. The search efficiency of GA is worse than RS but is better than SA. GA optimal plans have better performances on workload variation but worse performances on average response time. But we can use weights of performance measures to adjust the relative performances on different measures. Compared with RS, GA is a more automatic method. Compared with SA, the GA districting method is relatively easy to implement because there is no cut and merge process, which also affects the computational efficiency of each iteration.

6.2 Discussion & Future Work

One contribution of this dissertation is the development of the agent-based and discrete-event simulation of police patrol system. The agent-based model considers

GIS features and provides dynamic visualization for model validation and verification. The discrete-event model is much more computational efficient and provides similar evaluation results as the agent-based model. In this study, the simulation models are mainly used to test and evaluate district designs. They can be further extended to model more complex situations, such as call prioritization and multiple car response for certain types of calls. In addition to district plans, other operational patrol strategies can be compared and tested. In the current model, patrol cars conduct preventive random patrol if the cars are idle. Other possible preventive patrol rules are directed patrol and active patrol. It is interesting to compare different preventive patrol strategies. Similar agent-based and discrete-event models can be built for the fire-fighting services and EMS to solve the spatial allocation problem of urban emergency resources.

In discrete-event and agent-based simulation models, the CFS incidents are generated from the empirical distribution based on temporal and spatial pattern of historical CFS data. In Charlottesville application study, we use all the incidents that happened in a period of 4 years to obtain the empirical distribution. When we apply the simulation evaluation method to other cities, it is necessary to check if the CFS pattern shows significant year-by-year variation. If the yearly pattern is very similar to each other, then we can use CFS pattern in cumulative time period to generate incidents in the simulation model. If the yearly pattern shows significantly difference, it is better to use the most recent data to model the incidents in simulation model. In this way, it is more likely to find the optimal districting plan that fits the current characteristics of the city.

In current discrete-event simulation model, the estimation of travel time is based on the Euclidean distance between the origin atom and the destination atom. It is possible that there are some natural boundaries such as mountains and rivers in the city. Obviously the current manner in DE model will lead to inaccurate estimation of travel time if the route crosses the mountain or river. The agent-based simulation model does not have such problem because the actual road structure and speed limits on road segments are considered in the AB model. To improve the DE model, we can use a matrix to record the travel time between any two atoms. Each travel time is estimated based on the actual road structure and speed limits in GIS system. In DE simulation, the travel time can be read from the matrix which is already calculated before the simulation runs. In this way, the accuracy of travel time estimation in DE model can be improved. Similarly, we can use this matrix to develop a method that can quickly measure the average travel time within districts. If the district has a natural boundary in the middle, the average travel time between any two atoms within the district tend to longer. We can use it as a criteria for selecting districting plans.

The response surface methodology (RSM) described in this dissertation provides a systematic way to find the optimal or close-optimal district design. The parameterized districting algorithm generates district plans by locating seeds in concentric circle structure. The iterative searching procedure works well in Charlottesville police case study. However, it is still a question that how well it works for other cities. Different cities have different CFS patterns. The concentric circle structure works for Charlottesville but may not work for other cities. The seeds can be spread on squares or eclipse structure, depending on the shape of the specific city. Therefore, it is of great interest to test the RSM districting method on other cities. If it does not work well, it is necessary to study the CFS patterns of a number of cities and try to develop a more general way to parameterize the redistricting procedure. In addition, similar RSM can be developed for relevant problems of other urban emergency services.

In Charlottesville application study, the number of districts and police cars are pre-determined for the district design optimization problem. It is possible that the number of districts is not fixed and is also a decision variable for other cities. It should be noticed that more districts mean more police cars and there will be more costs brought by extra police resources. Usually, more police cars conducting patrol activities in the city lead to better patrol performances. But it is also possible that the marginal increase of performance will be less when we keep adding police cars to the city. For the district design problem, if the number of districts is included in the model as a decision variable, we can use the budget of police resources as a constraint or include it in the objective function to model the trade-off between the financial cost and the patrol performances achieved.

Appendix A

Regression Analysis & ANOVA Results

Listing A.1: Regression Analysis and ANOVA for Factors and Responses

```
--------------------------------------------
Regression Analysis of Average Response Time
--------------------------------------------

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  280.635      1.544 181.728  < 2e-16 ***
cenX          78.999      2.887  27.363  < 2e-16 ***
cenX.sq      436.560     10.705  40.779  < 2e-16 ***
cenY          31.991      2.793  11.454  < 2e-16 ***
cenY.sq      349.963     10.639  32.894  < 2e-16 ***
n1             2.442      2.617   0.933    0.352
r1             1.472      2.760   0.533    0.594
r2           -16.747      2.803  -5.974  1.1e-08 ***
---

Residual standard error: 11.3 on 192 degrees of freedom
Multiple R-squared: 0.9513,     Adjusted R-squared: 0.9495
F-statistic: 535.5 on 7 and 192 DF,  p-value: < 2.2e-16

Analysis of Variance Table

Response: Average Response Time
           Df Sum Sq Mean Sq   F value    Pr(>F)
cenX        1 107267  107267  840.1928 < 2.2e-16 ***
cenX.sq     1 206024  206024 1613.7198 < 2.2e-16 ***
cenY        1  20276   20276  158.8156 < 2.2e-16 ***
cenY.sq     1 140312  140312 1099.0201 < 2.2e-16 ***
n1          1     88      88    0.6919    0.4066
r1          1     53      53    0.4132    0.5211
r2          1   4557    4557   35.6920 1.102e-08 ***
Residuals 192  24513     128
---

-----------------------------------------
Regression Analysis of Workload Variation
-----------------------------------------

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  3.68128    0.04020  91.570  < 2e-16 ***
cenX         1.27422    0.07516  16.954  < 2e-16 ***
cenX.sq      4.91701    0.27870  17.643  < 2e-16 ***
cenY         0.53475    0.07271   7.355 5.41e-12 ***
cenY.sq      2.30866    0.27697   8.335 1.46e-14 ***
n1           0.21706    0.06814   3.185  0.00169 **
r1           0.14242    0.07186   1.982  0.04891 *
r2          -0.10596    0.07298  -1.452  0.14813
---

Residual standard error: 0.2942 on 192 degrees of freedom
Multiple R-squared: 0.8095,     Adjusted R-squared: 0.8026
F-statistic: 116.6 on 7 and 192 DF,  p-value: < 2.2e-16

Analysis of Variance Table

Response: Workload Variation
           Df  Sum Sq Mean Sq  F value    Pr(>F)
cenX        1 29.7313 29.7313 343.6156 < 2.2e-16 ***
cenX.sq     1 28.2575 28.2575 326.5818 < 2.2e-16 ***
cenY        1  5.3577  5.3577  61.9211 2.543e-13 ***
cenY.sq     1  5.8987  5.8987  68.1730 2.378e-14 ***
n1          1  0.8349  0.8349   9.6493  0.002181 **
r1          1  0.3492  0.3492   4.0356  0.045950 *
r2          1  0.1824  0.1824   2.1084  0.148128
Residuals 192 16.6128  0.0865
---
```

Listing A.2: First and Second Order Models for Both Responses in CCD

```
-------------------------------------------
First order model for average response time
-------------------------------------------

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  301.151      6.330  47.572 4.74e-10 ***
x1             6.018     10.009   0.601    0.567
x2            -7.846     10.009  -0.784    0.459
---

Residual standard error: 20.02 on 7 degrees of freedom
Multiple R-squared: 0.1224,     Adjusted R-squared: -0.1284
F-statistic: 0.488 on 2 and 7 DF,  p-value: 0.6333

----------------------------------------
First order model for workload variation
----------------------------------------

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  3.84551    0.14147  27.182 2.34e-08 ***
x1           0.16228    0.22367   0.726    0.492
x2           0.02458    0.22367   0.110    0.916
---

Residual standard error: 0.4474 on 7 degrees of freedom
Multiple R-squared: 0.07143,    Adjusted R-squared: -0.1939
F-statistic: 0.2692 on 2 and 7 DF,  p-value: 0.7715

--------------------------------------------
Second order model for average response time
--------------------------------------------

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  271.015      2.514 107.799 4.44e-08 ***
x1             6.018      1.778   3.385 0.027650 *
x2            -7.846      1.778  -4.414 0.011568 *
x1.sq         48.542      3.325  14.597 0.000128 ***
x2.sq         26.787      3.325   8.055 0.001290 **
x1x2           3.304      3.554   0.930 0.405217
---

Residual standard error: 3.555 on 4 degrees of freedom
Multiple R-squared: 0.9842,     Adjusted R-squared: 0.9644
F-statistic: 49.77 on 5 and 4 DF,  p-value: 0.001078

----------------------------------------
Second order model for workload variation
----------------------------------------

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  3.32302    0.09954  33.382  4.8e-06 ***
x1           0.16228    0.07038   2.306  0.08243 .
x2           0.02458    0.07038   0.349  0.74457
x1.sq        0.99759    0.13167   7.576  0.00163 **
x2.sq        0.30844    0.13167   2.342  0.07916 .
x1x2         0.40523    0.14074   2.879  0.04504 *
---

Residual standard error: 0.1408 on 4 degrees of freedom
Multiple R-squared: 0.9475,     Adjusted R-squared: 0.8818
F-statistic: 14.43 on 5 and 4 DF,  p-value: 0.01145
```

Appendix B

Java Codes for DE Simulation and SA Process

B.1 Entities

B.1.1 Atom

```
// Atom.java

package Entities;

import java.text.DecimalFormat;
import java.util.ArrayList;
import java.util.Collections;

public class Atom implements RunSim.SimParameters{
      private int id; // start from 0

      private static int count = 0;
      private Coordinate coor;
      private double cfsProb;
      private int districtNum;
      private double patrolFrequency;
      private double distToCenter;
      private ArrayList<Integer> respCarPreferList = new ArrayList<Integer>();
      private ArrayList<Atom> nbAtomsList = new ArrayList<Atom>();

      public Atom(double x, double y) {
            super();
            coor = new Coordinate(x, y);
            this.id = count++;
      }

      public int getMostFreNbDisrict(){
            ArrayList<Integer> districtList = new ArrayList<Integer>();
            int fre[] = new int [DISTRICT_NUM];
            for(Atom nbAtom: this.getNbAtomsList()){
                  districtList.add(nbAtom.districtNum);
```

```
        }
        for(Integer districtNum: districtList){
              fre[districtNum]++;
        }
        int maxFre = Integer.MIN_VALUE;
        int maxIndex = -1;
        for (int i = 0; i < fre.length; i++) {
              if(fre[i] > maxFre){
                    maxFre = fre[i];
                    maxIndex = i;
              }
        }
        return maxIndex;
  }

  public ArrayList<Atom> getNbAtomsListSameDistrict(){
        ArrayList<Atom> nbAtomsListSameDistrict = new ArrayList<Atom>();
        for(Atom nb: this.getNbAtomsList()){
              if(nb.getDistrictNum() == this.getDistrictNum()){
                    nbAtomsListSameDistrict.add(nb);
              }
        }
        return nbAtomsListSameDistrict;
  }

  public void addNbAtom(Atom atom){
        nbAtomsList.add(atom);
  }

  public ArrayList<Atom> getNbAtomsList(){
        return nbAtomsList;
  }

  public static void setCountZero(){
        count = 0;
  }

  public int getDistrictNum() {
        return districtNum;
  }

  public void setDistrictNum(int districtNum) {
        this.districtNum = districtNum;
  }

  public double getCfsProb() {
        return cfsProb;
  }

  public void setCFSprob(double cfsProb) {
        this.cfsProb = cfsProb;
  }

  public double getPatrolFrequency() {
        return patrolFrequency;
  }

  public void setPatrolFrequency(double patrolFrequency) {
        this.patrolFrequency = patrolFrequency;
  }

  public Coordinate getCoor() {
        return coor;
  }

  public void setCoor(Coordinate coor) {
        this.coor = coor;
  }
```

```
	public int getId() {
		return id;
	}

	public double getDistToCenter() {
		return distToCenter;
	}

	public void setDistToCenter(double distToCenter) {
		this.distToCenter = distToCenter;
	}

	@Override
	public String toString() {
		DecimalFormat df = new DecimalFormat();
		df.setMaximumFractionDigits(3);
		df.setMinimumFractionDigits(3);
		return "Atom [id=" + id + ", " + coor.toString() + ", districtNum="
				+ districtNum + ", cfsProb=" + df.format(cfsProb)
				+ ", patrolFrequency:" + df.format(patrolFrequency)
				+ ", respCarPreferList:" + respCarPreferList+"]";
	}

	public void setRespCarPreferList(ArrayList<District> districtsArrayList) {
		ArrayList<IndexDistPairs> indexDistPairsList = new
		    ArrayList<IndexDistPairs>();
		for (District district : districtsArrayList) {
			double dist = Coordinate.calDistTwoPoints(this.getCoor(),
					district.getExpectedCarLocation());
			indexDistPairsList.add(new IndexDistPairs(district.getID(), dist));
		}
		Collections.sort(indexDistPairsList);
		for (IndexDistPairs pair : indexDistPairsList) {
			respCarPreferList.add(pair.getIndex());
		}

	}

	public ArrayList<Integer> getRespCarPreferList() {
		return respCarPreferList;
	}

}

class IndexDistPairs implements Comparable {

	int index;
	double distance;

	public IndexDistPairs(int index, double distance) {
		super();
		this.index = index;
		this.distance = distance;
	}

	public int getIndex() {
		return index;
	}

	public void setIndex(int index) {
		this.index = index;
	}

	public double getDistance() {
		return distance;
	}
	public void setDistance(double distance) {
		this.distance = distance;
	}

	@Override
	public int compareTo(Object o) {
		IndexDistPairs pair = (IndexDistPairs) o;
		if (this.getDistance() < pair.getDistance())
			return -1;
		else if (this.getDistance() > pair.getDistance())
			return 1;
		else
			return 0;
	}
}
```

B.1.2 Police Car

```
// Car.java
package Entities;

import java.text.DecimalFormat;
import java.util.ArrayList;

public class Car {
	private int id;
	private static int count = 0;
	private int districtNum;
	private Coordinate baseLocation; //expected location
	private Coordinate taskLocation;
	private Coordinate tempLocation;
	private int status; // 0: idle, 1:busy
	private int idleTime;
	private int busyTime;
	private double availability;
	private int taskNum;
	private CFSincident cfsIncident;

	public Car(ArrayList<District> districtsArrayList) {
		this.id = count++;
		this.districtNum = districtsArrayList.get(id).getID();
		this.baseLocation = districtsArrayList.get(id).getExpectedCarLocation();
		this.status = 0;
	}

	public Coordinate getTempLocation() {
		return tempLocation;
	}

	public void setTempLocation(Coordinate tempLocation) {
		this.tempLocation = tempLocation;
	}

	public static void setCountZero(){
		count = 0;
	}

	public int getTaskNum() {
		return taskNum;
	}

	public void addOneTaskNum() {
		this.taskNum++;
	}
	public int getIdleTime() {
		return idleTime;
	}

	public int getBusyTime() {
		return busyTime;
	}

	public void updateIdleBusyTime() {
		if (this.status == 1) {
			this.busyTime++;
		} else {
			this.idleTime++;
		}
		this.availability = (double) (idleTime)
				/ (double) (idleTime + busyTime);
	}

	public double getAvailability() {
		return availability;
	}

	public int getID() {
		return id;
	}

	public int getDistrictNum() {
		return districtNum;
	}

	public Coordinate getBaseLocation() {
		return this.baseLocation;
	}

	public int getStatus() {
		return status;
	}

	public void setStatus(int status) {
		this.status = status;
	}

	public Coordinate getTaskLocation() {
		return taskLocation;
	}

	public void setTaskLocation(Coordinate taskLocation) {
		this.taskLocation = taskLocation;
	}

	public CFSincident getCfsIncident() {
		return cfsIncident;
	}

	public void setCfsIncident(CFSincident cfsIncident) {
		this.cfsIncident = cfsIncident;
	}

	@Override
	public String toString() {
		DecimalFormat df = new DecimalFormat();
		df.setMaximumFractionDigits(3);
		df.setMinimumFractionDigits(3);
		return "Car [id=" + id + ", districtNum=" + districtNum
				+ ", baseLocation=" + baseLocation + "]";
	}
}
```

B.1.3 CFS Incident

```
// CFSincident.java
package Entities;

public class CFSincident {
	private int createTime;
	private int dispatchTime;
	private int carArrTime;
	private int leaveScnTime;
	private Coordinate coor;
	private int atomNum;
	private int districtNum;
	private int carNum;
	private int isCrsBdrResp;
	private int isDropped = 0;

	public int getCreateTime() {
		return createTime;
	}
	public void setCreateTime(int createTime) {
		this.createTime = createTime;
	}
	public int getDispatchTime() {
		return dispatchTime;
	}
	public void setDispatchTime(int dispatchTime) {
		this.dispatchTime = dispatchTime;
	}
	public int getCarArrTime() {
		return carArrTime;
	}
	public void setCarArrTime(int carArrTime) {
		this.carArrTime = carArrTime;
	}
	public int getLeaveScnTime() {
		return leaveScnTime;
	}
	public void setLeaveScnTime(int leaveScnTime) {
		this.leaveScnTime = leaveScnTime;
	}
	public Coordinate getCoor() {
		return coor;
	}
	public void setCoor(Coordinate coor) {
		this.coor = coor;
	}
	public int getDistrictNum() {
		return districtNum;
	}
	public void setDistrictNum(int districtNum) {
		this.districtNum = districtNum;
	}
	public int getAtomNum() {
		return atomNum;
	}
	public void setAtomNum(int atomNum) {
		this.atomNum = atomNum;
	}
	public int getCarNum() {
		return carNum;
	}
	public void setCarNum(int carNum) {
		this.carNum = carNum;
	}
	public int isCrsBdrResp() {
		return isCrsBdrResp;
	}
	public void setCrsBdrResp(int isCrsBdrResp) {
		this.isCrsBdrResp = isCrsBdrResp;
	}
	public int getIfDropped() {
		return isDropped;
	}
	public void setIfDropped(int isDropped) {
		this.isDropped = isDropped;
	}
	public String toString(){
		return "CreateT: " + createTime + ", dispatchT: " + dispatchTime + ",
		    carArrT: " + carArrTime + ", leaveScnT: " + leaveScnTime+ "
		    "+coor.toString()+ ", atomNum: " + atomNum
		      + ", districtNum: " + districtNum + ", carNum: " + carNum + ",
		          isCrsBdrResp: " + isCrsBdrResp + ", isDropped: " + isDropped;
	}
}
```

B.1.4 CFS Incident Queue

```
// CFSqueue.java
package Entities;

import java.util.LinkedList;

public class CFSqueue {

	private LinkedList<CFSincident> queue = new LinkedList<CFSincident>();
	private int capacity;

	public CFSqueue(int capacity){
		this.capacity = capacity;
	}

	public boolean enQue(CFSincident incid){
		if(isQueFull()){
			return false;
		}else{
			queue.addLast(incid);
			return true;
		}
	}

	public CFSincident deQue(){
		return queue.removeFirst();
	}

	public int getQueLen(){
		return queue.size();
	}

	public boolean isQueEmpty(){
		return (getQueLen()==0);
	}

	public boolean isQueFull(){
		return (getQueLen()==capacity);
	}
	public int getCapacity() {
		return capacity;
	}

	public void setCapacity(int capacity) {
		this.capacity = capacity;
	}

	public void clearQue(){
		queue.clear();
	}

	public String toString(){
		StringBuilder sb = new StringBuilder();
		sb.append("DistrictNum; {");
		for (int i = 0; i < this.getQueLen(); i++) {
			sb.append(queue.get(i).getDistrictNum() + " ");
		}
		sb.append("}");
		return sb.toString();
	}
}
```

B.1.5 Coordinate

```
// Coordinate.java

package Entities;

import java.text.DecimalFormat;

public class Coordinate {
	private double x;
	private double y;

	public double getX() {
		return x;
	}

	public void setX(double x) {
		this.x = x;
	}

	public double getY() {
		return y;
	}

	public void setY(double y) {
		this.y = y;
	}

	public Coordinate() {
		super();
		this.x = 0;
		this.y = 0;
	}

	public Coordinate(double x, double y) {
		super();
		this.x = x;
		this.y = y;
	}
	public static double calDistTwoPoints(Coordinate coor1, Coordinate coor2) {
		double deltaX = coor1.getX() - coor2.getX();
		double deltaY = coor1.getY() - coor2.getY();
		return Math.pow(Math.pow(deltaX, 2) + Math.pow(deltaY, 2), 0.5);
	}

	@Override
	public String toString() {
		DecimalFormat df = new DecimalFormat();
		df.setMaximumFractionDigits(3);
		df.setMinimumFractionDigits(3);
		return df.format(x) + "\t" + df.format(y);
	}
}
```

B.1.6 District

```
// District.java

package Entities;

import java.text.DecimalFormat;
import java.util.ArrayList;
import java.util.Random;
import java.util.Set;
import java.util.HashSet;

public class District implements RunSim.SimParameters{
	private int id;
	private static int count = 0;
	private ArrayList<Atom> districtAtomsArrayList = new ArrayList<Atom>();
	private ArrayList<Integer> atomIndexList = new ArrayList<Integer>();

	// constructor
	public District(ArrayList<Atom> totalAomsArrayList) {
		id = count++;
		for (Atom atom : totalAomsArrayList) {
			if (atom.getDistrictNum() == this.id) {
				districtAtomsArrayList.add(atom);
				atomIndexList.add(atom.getId());
			}
		}
	}

	public Coordinate getMassCenter() {
		double x = 0;
		double y = 0;
		double probSum = 0;
		for (Atom atom : districtAtomsArrayList) {
			x += atom.getCoor().getX() * atom.getCfsProb();
			y += atom.getCoor().getY() * atom.getCfsProb();
			probSum += atom.getCfsProb();
		}
		x /= probSum;
		y /= probSum;
		return new Coordinate(x, y);
	}

	public Coordinate getRandomMassCenter() {
		double x = 0;
		double y = 0;
		double probSum = 0;
```

```
        for (Atom atom : districtAtomsArrayList) {
              x += atom.getCoor().getX() * atom.getCfsProb();
              y += atom.getCoor().getY() * atom.getCfsProb();
              probSum += atom.getCfsProb();
        }
        x /= probSum;
        y /= probSum;
        //add randomness
        double randNum = new Random().nextDouble();
        x += RAND_MASS_CENTER_TOL*randNum;
        y += RAND_MASS_CENTER_TOL*randNum;

        return new Coordinate(x, y);
  }

  public double getBorderRatio(){
        return
            (double)this.getBorderAtomLength()/(double)this.getDistrictAtomsArrayList().size();
  }

  public int getBorderAtomLength(){
        int count = 0;
        for(Atom atom: this.getDistrictAtomsArrayList()){
              boolean isOutsideBorder = (atom.getNbAtomsList().size() < 4);
              boolean isInsideBorder = (atom.getNbAtomsList().size() !=
                  atom.getNbAtomsListSameDistrict().size());
              if(isOutsideBorder || isInsideBorder){
                    count++;
              }
        }
        return count;
  }

  public void removeAtom(Atom atom){
        boolean exists = false;
        for (int i = 0; i < atomIndexList.size(); i++) {
              if(atomIndexList.get(i) == atom.getId()){
                    exists = true;
              }
        }
        if(exists){
              districtAtomsArrayList.remove(atom);
              atomIndexList.remove(new Integer(atom.getId()));
        }
  }

  public void receiveAtom(Atom atom){
        boolean exists = false;
        for (int i = 0; i < atomIndexList.size(); i++) {
              if(atomIndexList.get(i) == atom.getId()){
                    exists = true;
              }
        }
        if(!exists){
              districtAtomsArrayList.add(atom);
              atomIndexList.add(atom.getId());
        }
  }

  public ArrayList<District> getAdjDistrictArrList(){
        ArrayList<District> adjDistrictArrList = new ArrayList<District>();
        for(Integer districtId: this.getAdjDistrictIdList()){
              adjDistrictArrList.add(districtsArrayList.get(districtId));
        }
        return adjDistrictArrList;
  }

  public HashSet<Integer> getAdjDistrictIdList(){
        HashSet<Integer> adjDistrictIdList = new HashSet<Integer>();
        for(Atom atom: this.getDistrictAtomsArrayList()){
              for(Atom atomsNb: atom.getNbAtomsList()){
                    if(atomsNb.getDistrictNum() != this.id){
                          adjDistrictIdList.add(atomsNb.getDistrictNum());
                    }
              }
        }
        return adjDistrictIdList;
  }

  public double getMaxDistanceAreaRatio(){
        double maxDistance = Double.MIN_VALUE;
        for(Atom atomI: districtAtomsArrayList){
              for(Atom atomJ: districtAtomsArrayList){
                    double currDistance =
                        Coordinate.calDistTwoPoints(atomI.getCoor(), atomJ.getCoor());
                    if(currDistance > maxDistance){
                          maxDistance = currDistance;
                    }
              }
        }
        return maxDistance/Math.sqrt(getArea());
  }

  public double getArea(){
        return districtAtomsArrayList.size()*(ADJ_ATOM_DISTANCE*ADJ_ATOM_DISTANCE);
  }

  public void addAtom(Atom atom){
        districtAtomsArrayList.add(atom);
  }

  public void deleteAtom(Atom atom){
        districtAtomsArrayList.remove(atom);
  }

  public double getCmptness(){
        double maxDist = -9999;
        double cmptness;
        for(Atom atom1: districtAtomsArrayList){
              for(Atom atom2: districtAtomsArrayList){
                    double currDist = Coordinate.calDistTwoPoints(atom1.getCoor(),
                        atom2.getCoor());
                    if(currDist > maxDist){
                          maxDist = currDist;
                    }
              }
        }
        cmptness = maxDist/Math.sqrt(districtAtomsArrayList.size());
        return cmptness;
  }

  public double getRespDistance(){
        double sum = 0;
        for(Atom atom: this.districtAtomsArrayList){
              double distToDistrCenter = Coordinate.calDistTwoPoints(atom.getCoor(),
                  this.getGeoCenter());
              sum += distToDistrCenter * atom.getCfsProb();
        }
        return sum;
  }

  public ArrayList<Atom> getDistrictAtomsArrayList() {
        return districtAtomsArrayList;
  }

  public void setDistrictAtomsArrayList(ArrayList<Atom> districtAtomsArrayList) {
        this.districtAtomsArrayList = districtAtomsArrayList;
  }

  public ArrayList<Integer> getAtomIndexList() {
        return atomIndexList;
  }

  public void setAtomIndexList(ArrayList<Integer> atomIndexList) {
        this.atomIndexList = atomIndexList;
  }

  public static void setCountZero(){
        count = 0;
  }

  public int getID() {
        return this.id;
  }

  public double getCfsProb() {
        double cfsProb = 0;
        for (Atom atom : districtAtomsArrayList) {
              cfsProb += atom.getCfsProb();
        }

        return cfsProb;
  }

  public int getAtomNum() {
        return districtAtomsArrayList.size();
  }

  public Coordinate getGeoCenter() {
        double x = 0;
        double y = 0;
        for (Atom atom : districtAtomsArrayList) {
              x += atom.getCoor().getX();
              y += atom.getCoor().getY();
        }
        x = x / getAtomNum();
        y = y / getAtomNum();
        return new Coordinate(x, y);
  }

  public Coordinate getExpectedCarLocation() {
        double x = 0;
        double y = 0;
        double freSum = 0;
        for (Atom atom : districtAtomsArrayList) {
              x += atom.getCoor().getX() * atom.getPatrolFrequency();
              y += atom.getCoor().getY() * atom.getPatrolFrequency();
              freSum += atom.getPatrolFrequency();
        }
        x /= freSum;
        y /= freSum;
        return new Coordinate(x, y);
  }

  @Override
  public String toString() {

        DecimalFormat df = new DecimalFormat();
        df.setMaximumFractionDigits(4);
        df.setMinimumFractionDigits(4);
        return "District [id=" + id + ", atomIndexList=" + atomIndexList
                    + ", CFSprob=" + df.format(this.getCfsProb()) + ", GeoCenter:"
                    + this.getGeoCenter().toString() + ", ExpCarLoc:"
                    + this.getExpectedCarLocation().toString() + "]";
```

```
      }
}
```

B.1.7 District Plan

```java
// DistrictPlan.java

package Entities;

import java.text.DecimalFormat;

public class DistrictPlan {
      private int [] planArr;
      public DistrictPlanMeasure measure;
      public DistrictPlan(int atomNum){
            planArr = new int[atomNum];
            measure = new DistrictPlanMeasure();
      }
      public DistrictPlan(int [] arr){
            planArr = new int[arr.length];
            for (int i = 0; i < arr.length; i++) {
                  planArr[i] = arr[i];
            }
            measure = new DistrictPlanMeasure();
      }
      public void setPlanArr(int [] arr){
            planArr = new int[arr.length];
            for (int i = 0; i < arr.length; i++) {
                  planArr[i] = arr[i];
            }
      }
      public int [] getPlanArr(){
            return planArr;
      }
      public void setMeasure(double aveRespTime, double workloadStdev) {
            this.measure.setAveRespTime(aveRespTime);
            this.measure.setWorkloadStdev(workloadStdev);
      }
      public DistrictPlanMeasure getMeasure(){
            return measure;
      }
      public String toString(){
            StringBuilder sb = new StringBuilder();
            sb.append(this.getMeasure().toString());
            return sb.toString();
      }
      public void showPlan(){
            StringBuilder sb = new StringBuilder();
            for (int j = 0; j < this.getPlanArr().length; j++) {
                  sb.append(this.getPlanArr()[j] + "\t");
            }
            System.out.println(sb.toString());
      }

      public int compareTo(DistrictPlan newPlan){
            return this.getMeasure().compareTo(newPlan.getMeasure());
      }
}
```

B.1.8 Performance Measure of District Plan

```java
// DistrictPlanMeasure.java
package Entities;

import java.text.DecimalFormat;

public class DistrictPlanMeasure {
      private double aveRespTime;
      private double workloadStdev;
      public DistrictPlanMeasure(){
            aveRespTime = 0;
            workloadStdev = 0;
      }
      public double getAveRespTime() {
            return aveRespTime;
      }

      public void setAveRespTime(double aveRespTime) {
            this.aveRespTime = aveRespTime;
      }

      public double getWorkloadStdev() {
            return workloadStdev;
      }

      public void setWorkloadStdev(double workloadStdev) {
            this.workloadStdev = workloadStdev;
      }

      public int compareTo(DistrictPlanMeasure newPlanMeasure) {
            if(this.getWeightedScore() < newPlanMeasure.getWeightedScore()){
                  return -1;
            }else if(this.getWeightedScore() > newPlanMeasure.getWeightedScore()){
                  return 1;
            }else
                  return 0;
      }
      public double getWeightedScore(){
            //stan_score = 1.91 - 0.00144 randY1 - 18.6 randY2
            return 1.91 - 0.00144*aveRespTime - 18.6*workloadStdev;
      }

      public String toString(){
            DecimalFormat df = new DecimalFormat();
            df.setMaximumFractionDigits(3);
            df.setMinimumFractionDigits(3);
//          return "\nMeasure: AveRespTime: " + df.format(aveRespTime) + " WorkloadStdev:
    " + df.format(workloadStdev) + " WeigthedScore: " + df.format(getWeightedScore());
            return "WeigthedScore: " + df.format(getWeightedScore());
      }
}
```

B.2 IO

B.2.1 Input

```java
// Input.java

package IO;

import java.io.BufferedReader;
import java.io.DataInputStream;
import java.io.FileInputStream;
import java.io.InputStreamReader;
import java.util.ArrayList;

public class Input {
      public static ArrayList<ArrayList<Double>> readDataFile(String filePath){
            try {
                  FileInputStream fstream = new FileInputStream(filePath);
                  DataInputStream in = new DataInputStream(fstream);
                  BufferedReader br = new BufferedReader(new InputStreamReader(in));
                  String strLine;
                  ArrayList<ArrayList<Double>> arrListMatrix = new
                      ArrayList<ArrayList<Double>>();
                  while ((strLine = br.readLine()) != null) {
                        if(!strLine.equals("")){
                              strLine = strLine.trim();
                              String[] strArray = strLine.split("\t");
                              ArrayList<Double> arrList = new ArrayList<Double>();
                              for (String str : strArray) {
                                    arrList.add(Double.parseDouble(str));
                              }
                              arrListMatrix.add(arrList);
                        }
                  }
                  in.close();
                  return arrListMatrix;
            } catch (Exception e) {
                  System.err.println("Error: " + e.getMessage());
                  return null;
            }
      }

      public static boolean readFileIntoDoubleArray(String filePath,
                  double[] doubleArray) {
            ArrayList<ArrayList<Double>> matrix = IO.Input.readDataFile(filePath);
            if (matrix.get(0).size() != doubleArray.length) {
                  return false;
            } else {
                  for (int i = 0; i < doubleArray.length; i++) {
                        doubleArray[i] = matrix.get(0).get(i);
                  }
                  return true;
            }
      }

      public static boolean readFileIntoIntArray(String filePath,
                  int[] intArray) {
            ArrayList<ArrayList<Double>> matrix = IO.Input.readDataFile(filePath);
            if (matrix.get(0).size() != intArray.length) {
                  return false;
            } else {
                  for (int i = 0; i < intArray.length; i++) {
                        intArray[i] = matrix.get(0).get(i).intValue();
                  }
                  return true;
            }
      }

      public static boolean readFileIntoDoubleMatrix(String filePath,
                  double[][] doubleMatrix) {
            ArrayList<ArrayList<Double>> matrix = IO.Input.readDataFile(filePath);
            if (matrix.size() != doubleMatrix.length || matrix.get(0).size() !=
                doubleMatrix[0].length) {
                  return false;
            } else {
                  for (int i = 0; i < matrix.size(); i++) {
                        for (int j = 0; j < matrix.get(0).size(); j++) {
                              doubleMatrix[i][j] = matrix.get(i).get(j);
                        }
                        }
                        return true;
                }

        }

        public static boolean readFileIntoIntMatrix(String filePath,
                        int[][] intMatrix) {
                ArrayList<ArrayList<Double>> matrix = IO.Input.readDataFile(filePath);
                if (matrix.size() < intMatrix.length || matrix.get(0).size() !=
                     intMatrix[0].length) {
                        return false;
                } else {
                        for (int i = 0; i < intMatrix.length; i++) {
                                for (int j = 0; j < intMatrix[0].length; j++) {
                                        intMatrix[i][j] = matrix.get(i).get(j).intValue();
                                }
                        }
                        return true;
                }
        }
}
```

B.2.2 Output

```
// Output.java

package IO;

import java.io.BufferedWriter;
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;

public class Output {
        public static void writeToFile(String FilePath, String content){
                try {
                        File file = new File(FilePath);

                        // if file doesnt exists, then create it
                        if (!file.exists()) {
                                file.createNewFile();
                        }

                        FileWriter fw = new FileWriter(file.getAbsoluteFile());
                        BufferedWriter bw = new BufferedWriter(fw);
                        bw.write(content);
                        bw.close();
                } catch (IOException e) {
                        e.printStackTrace();
                }
        }

        public static void writeLineToFile(String FilePath, String content){
                try {
                        PrintWriter out = new PrintWriter((new FileWriter(
                                        FilePath, true)));
                        out.print(content);
                        out.close();
```

```
            } catch (IOException e) {
                  // TODO Auto-generated catch block
                  e.printStackTrace();
            }
      }
}
```

B.3 RunSim

B.3.1 Run Parameterized Districting Algorithm

```
// DistrAlgProc.java

package RunSim;

import java.util.ArrayList;

import Entities.Atom;
import Entities.Coordinate;
import Entities.District;
import IO.Input;
import RunSim.MyMath;
import SimEvent.MyRandom;

public class DistrAlgProc{
      static int CON_READIN_COL = 7;
      static int CON_READIN_ROW = 10;
      static int CON_START_ROW = 1; //inclusive, start from 1
      static int CON_END_ROW = 1; //inclusive
      static int dPlanNumGennedEachCon = 1;

      final static int ATOM_NUM = 323;
      final static int DISTRICT_NUM = 8;
      static String CFS_PROB_EACH_ATOM_PATH = "IO\\Input\\CFS_PROB_EACH_ATOM.txt";
      static String COOR_EACH_ATOM_PATH = "IO\\Input\\COOR_EACH_ATOM.txt";
      static String ADJACENCY_MATRIX_PATH = "IO\\Input\\ADJACENCY_MATRIX.txt";
      static String DISTRICTING_PARAMETERS_MATRIX_PATH =
          "IO/Input/DISTRICTING_PARAMETERS_MATRIX.txt";
      static String DISTRICT_PLAN_MATRIX_PATH = "IO/Input/DISTRICT_PLAN_MATRIX.txt";
      static double CFS_PROB_EACH_ATOM[] = new double[ATOM_NUM];
      static double COOR_EACH_ATOM[][] = new double[ATOM_NUM][2];
      static int ADJACENCY_MATRIX[][] = new int[ATOM_NUM][4];
      static double DISTRICTING_PARAMETERS_MATRIX[][] = new double
          [CON_READIN_ROW][CON_READIN_COL];
      static ArrayList<Atom> atomsArrayList;
      static ArrayList<District> districtsArrayList;
      static Coordinate cityCenter = new Coordinate();
      static double cityRadius;
      static StringBuilder sb = new StringBuilder();

      //districting parameters
      static double cenX1, cenY1, cenX2, cenY2, r1, r2;
      static int n1, n2;
      static int smthIterNum = 3;
      static double swapBorderProbBdry = 0.06;

      public static void main(String[] args) {
      }

      public static int[] genOnePlanBySeeds(ArrayList<Coordinate> seedsCoorsList){
            genOnePlanWithoutCheck(seedsCoorsList);
            if(checkPlanConnectivity() && checkDistrictIntegrity()){
                return getDistrictPlanArr();
            }else{
                return getInvalidDPlanArr();
            }
        }

        public static int [] getInvalidDPlanArr(){
            int [] arr = new int[ATOM_NUM];
            for (int i = 0; i < arr.length; i++) {
                arr[i] = -1;
            }
            return arr;
        }

        public static void genOnePlanWithoutCheck(ArrayList<Coordinate> seedsCoorsList) {
            readDataSet();
            initAtoms();
            setAtomDistToCenter();
            generateSeedsProcedure(seedsCoorsList);
            initDistricts();
            fillBorderProcedure();
            swapDistrOnBorderProcedure();
            smoothBorderProcedure();
        }

        public static int [] getDistrictPlanArr(){
            int [] arr = new int[ATOM_NUM];
            for (int i = 0; i < arr.length; i++) {
                arr[i] = atomsArrayList.get(i).getDistrictNum();
            }
            return arr;

        }

        public static boolean checkPlanConnectivity(){
            boolean result = true;
            for(District distr: districtsArrayList){
                if(checkDistrictConnectivity(distr) == false){
//                      System.out.println("District " + distr.getID() + " fails in
    connectivity check");
                    result = false;
                }
            }
            return result;
        }

        public static boolean checkDistrictConnectivity(District distr){
            ArrayList<Atom> atomList = new ArrayList<Atom>();
            ArrayList<Atom> nbAtomListWithDups = new ArrayList<Atom>();
            int newAtomListSize, oldAtomListSize;
            if(distr.getDistrictAtomsArrayList().size() == 0)
                return false;
            Atom startAtom = distr.getDistrictAtomsArrayList().get(0);
            atomList.add(startAtom);
            do{
                oldAtomListSize = atomList.size();
                for(Atom atom: atomList){
                    nbAtomListWithDups.add(atom); //add itself
                    for(Atom nbAtom: atom.getNbAtomsList()){
                        if(atom.getDistrictNum() == nbAtom.getDistrictNum()){
                            nbAtomListWithDups.add(nbAtom);
                        }
                    }
                }
                ArrayList<Atom> nbAtomList = removeDupAtoms(nbAtomListWithDups);
                atomList = nbAtomList;
                newAtomListSize = atomList.size();
            }while(newAtomListSize > oldAtomListSize);
            int totalAtomNum = 0;
            for(Atom atom: atomsArrayList){
                  if(atom.getDistrictNum() == distr.getID()){
                        totalAtomNum++;
                  }
            }
//          System.out.println("totalAtomNum " + totalAtomNum + " newAtomListSize " +
    newAtomListSize );
            return totalAtomNum == newAtomListSize;
      }

      public static boolean checkDistrictIntegrity(){
            boolean isAllDistrsInteg = true;
            for(District distr:districtsArrayList){
                  if(distr.getDistrictAtomsArrayList().size() == 0){
                        isAllDistrsInteg = false;
                  }
            }
            return isAllDistrsInteg;
      }

      public static ArrayList<Atom> removeDupAtoms(ArrayList<Atom> atomListWithDups){
            ArrayList<Atom> atomListNoDups = new ArrayList<Atom>();
            for(Atom atom: atomListWithDups){
                  boolean isExists = false;
                  for(Atom atomInResultList: atomListNoDups){
                        if(atom.getId() == atomInResultList.getId()){
                              isExists = true;
                        }
                  }
                  if(isExists==false){
                        atomListNoDups.add(atom);
                  }
            }
            return atomListNoDups;
      }

      public static void smoothBorderProcedure() {
            for (int i = 0; i < smthIterNum; i++) {
                  smoothBorderOneIter();
            }
      }

      public static void smoothBorderOneIter() {
            //smooth procedure
            for(Atom atom: atomsArrayList){
                  District acpDistr = getBorderAtomMaxFrequentNbDistrict(atom);
                  District givDistr = districtsArrayList.get(atom.getDistrictNum());
                  if(acpDistr != null){
                        //change it temporarily
                        atom.setDistrictNum(acpDistr.getID());
                        givDistr.deleteAtom(atom);
                        acpDistr.addAtom(atom);
                  }
            }
      }

      public static District getBorderAtomMaxFrequentNbDistrict(Atom atom){
            int [] distrFreArray = new int [DISTRICT_NUM];
            for(Atom nbAtom: atom.getNbAtomsList()){
                  distrFreArray[nbAtom.getDistrictNum()]++;
            }
            District resultDistrict = null;
            int allNbNum = atom.getNbAtomsList().size();
            int itsDistrFre = distrFreArray[atom.getDistrictNum()];
            for (int i = 0; i < distrFreArray.length; i++) {
                  boolean con1 = allNbNum == 4 && distrFreArray[i] >= 3 && i !=
                      atom.getDistrictNum();
```

```
                boolean con2 = allNbNum == 3 && distrFreArray[i] >= 2 && i !=
                    atom.getDistrictNum();
                boolean con3 = itsDistrFre == 0 & distrFreArray[i] >= 1;
                boolean con4 = itsDistrFre == 1 & distrFreArray[i] >= 2;

                if( con1 || con2 || con3 || con4){
                      resultDistrict = districtsArrayList.get(i);
                }
          }
          return resultDistrict;
    }

    public static void swapDistrOnBorderProcedure() {
          //only adjust for promising plans
          if(getCfsProbStdev() > swapBorderProbBdry){
                return;
          }

          double cfsProbStdBefore, cmptnessBefore, cfsProbStdAfter, cmptnessAfter;
          do{
//                System.out.println("in swapDistrOnBorderProcedure");
                cfsProbStdBefore = getCfsProbStdev();
                cmptnessBefore = getDistrPlanCmptness();
                SwapDistrsForBorderAtoms();
                cfsProbStdAfter = getCfsProbStdev();
                cmptnessAfter = getDistrPlanCmptness();
//                System.out.println("cfsProbStdAfter: " + cfsProbStdAfter + "
    cmptnessAfter: " + cmptnessAfter );
          }while(cfsProbStdAfter < cfsProbStdBefore && cmptnessAfter < cmptnessBefore);
    }

    public static void SwapDistrsForBorderAtoms() {
          //release assigned border
          int changesMade = 0;
          for(Atom atom: getAllDistrsAssignedBorder()){
                ArrayList<District> atomNbDistrsList = getAtomNbDistrsList(atom);
                if(atomNbDistrsList.size() == 1) break;
                District givDistrict = districtsArrayList.get(atom.getDistrictNum());
                District acpDistrict = null;
                for(District distr: atomNbDistrsList){
                      if(distr.getID() != atom.getDistrictNum()){
                            acpDistrict = distr;
                      }
                }
                //swap district temporarily
                double cfsProbStdBefore = getCfsProbStdev();
                double cmptnessBefore = getDistrPlanCmptness();
                atom.setDistrictNum(acpDistrict.getID());
                givDistrict.deleteAtom(atom);
                acpDistrict.addAtom(atom);
                double cfsProbStdAfter = getCfsProbStdev();
                double cmptnessAfter = getDistrPlanCmptness();
                changesMade++;
                //need to check connectivity
                //if become worse, change back
                if(cfsProbStdAfter > cfsProbStdBefore || cmptnessAfter > cmptnessBefore
                    || checkDistrictIntegrity() == false ||
                    (checkDistrictConnectivity(givDistrict)== false) ){
//                      System.out.println("cancel change");
                      atom.setDistrictNum(givDistrict.getID());
                      acpDistrict.deleteAtom(atom);
                      givDistrict.addAtom(atom);
                      changesMade--;
                }
          }
//          System.out.println("changesMade: " + changesMade);
    }
```

```
	public static ArrayList<Atom> getAllDistrsAssignedBorder(){
		ArrayList<Atom> assignedBorder = new ArrayList<Atom>();
		for(Atom atom: atomsArrayList){
			ArrayList<District> atomNbDistrsList = getAtomNbDistrsList(atom);
			if(atomNbDistrsList.size() >= 2){
				assignedBorder.add(atom);
			}

		}
		return assignedBorder;
	}

	public static ArrayList<District> getAtomNbDistrsList(Atom atom){
		ArrayList<District> nbDistrListWithDups = new ArrayList<District>();
		ArrayList<District> nbDistrList = new ArrayList<District>();
		for(Atom nbAtom: atom.getNbAtomsList()){
			nbDistrListWithDups.add(districtsArrayList.get(nbAtom.getDistrictNum()));
		}
		//remove dups
		for(District distr1: nbDistrListWithDups){
			boolean isExits = false;
			for(District distr2: nbDistrList){
				if(distr1.getID() == distr2.getID()){
					isExits = true;
				}
			}
			if(isExits == false){
				nbDistrList.add(distr1);
			}
		}
		return nbDistrList;
	}

	public static void generateSeedsProcedure(ArrayList<Coordinate> seedsCoorsList) {

		ArrayList<Atom> seedsAtomList = getSeeds(seedsCoorsList);
//		//print
//		for(Atom atom: seedsAtomList){
//			System.out.print(atom.getId() + "\t");
//		}
//		System.out.println();

		while(checkArrayListDuplicate(seedsAtomList) == true){
//			System.out.println("in generateSeedsProcedure");
			seedsAtomList = removeArrayListDuplicate(seedsAtomList);
			int randAtomndex = MyRandom.randomInt(ATOM_NUM-1);
			seedsAtomList.add(atomsArrayList.get(randAtomndex));
		}
	}

	public static ArrayList<Atom> copyAtomList(ArrayList<Atom> oriAtomList){
		ArrayList<Atom> newAtomList = new ArrayList<Atom>();
		for(Atom atom: oriAtomList){
			newAtomList.add(atom);
		}
		return newAtomList;
	}

	public static ArrayList<District> copyDistrictList(ArrayList<District>
	    oriDistrictList){
		ArrayList<District> newDistrictList = new ArrayList<District>();
		for(District distr: oriDistrictList){
			newDistrictList.add(distr);
		}
		return newDistrictList;
	}

	public static void fillBorderProcedure() {
```

```
            //section 1: fill districtMinProb border
            double atomsAssignedRateBeforeFill, atomsAssignedRateAfterFill;
            do{
//                System.out.println("section 1: fill districtMinProb border");
                  atomsAssignedRateBeforeFill = getAtomsAssignedRate();
                  fillBorderOneIter(1);
                  atomsAssignedRateAfterFill = getAtomsAssignedRate();
            }while(atomsAssignedRateBeforeFill != atomsAssignedRateAfterFill);

            // print plans in the middle
            int []dPlanArr = getDistrictPlanArr();
            System.out.println("in fillBorderProcedure:");
            for (int i = 0; i < dPlanArr.length; i++) {
                  System.out.print(dPlanArr[i] + "\t");
            }
            System.out.println("");

            //section 2: fill all districts border
            while (getAtomsAssignedRate() != 1) { // until all atoms assigned
//                System.out.println("section 2: fill all districts border");
                  fillBorderOneIter(2);
            }

//          printCurrDistrictPlan();
      }

      public static void fillBorderOneIter(int sectionNum) {
            //fill the hole procedure, each unassigned atom belongs to the nearest
                district
            //get all districts' unassigned border
            ArrayList<Atom> distrUnassignedBorder = null;
            if(sectionNum == 1){
                  distrUnassignedBorder =
                      getOneDistrUnassignedBorder(selectDistrWithMinProb(districtsArrayList));
            }else if(sectionNum == 2){
                  distrUnassignedBorder = getAllDistrsUnassignedBorder();
            }

//          System.out.println("allDistrsUnassignedBorder.size()" +
    allDistrsUnassignedBorder.size());
//          for(Atom atom: allDistrsUnassignedBorder){
//                System.out.println(atom.getCoor().getX() + "\t" +
    atom.getCoor().getY());
//          }
            //assign each atom to nearest district
            for(Atom atom: distrUnassignedBorder){
                  ArrayList<District> atomNbDistrList = getAtomNbDistrList(atom);
                  District distrSelected = null;
                  if(atomNbDistrList.size() == 1){ // assign it directly
                        distrSelected = atomNbDistrList.get(0);
                  }else{ // 2 or more districts, choose the nearest one
                        double minDist = 99999;
                        for(District distr: atomNbDistrList){
                              double currDist =
                                  Coordinate.calDistTwoPoints(atom.getCoor(),
                                  distr.getGeoCenter());
                              if(currDist < minDist){
                                    minDist = currDist;
                                    distrSelected = distr;
                              }
                        }
                  }
                        atom.setDistrictNum(distrSelected.getID());
                        distrSelected.addAtom(atom);
            }
      }

      public static ArrayList<District> getAtomNbDistrList(Atom atom){
            ArrayList<District> atomAdjDistrListWithDups = new ArrayList<District>();
            for(Atom nbAtom: atom.getNbAtomsList()){
                  if(nbAtom.getDistrictNum() != -1){ //assigned
                        atomAdjDistrListWithDups.add(districtsArrayList.get(nbAtom.getDistrictNum()));
//                      System.out.println(nbAtom.getDistrictNum());
                  }
            }
            //remove dups
            ArrayList<District> atomAdjDistrList = new ArrayList<District>();
            for(District distrOld: atomAdjDistrListWithDups){
                  boolean isExists = false;
                  for(District distrNew: atomAdjDistrList){
                        if(distrOld.getID() == distrNew.getID()){
                              isExists = true;
                        }
                  }
                  if(isExists == false){
                        atomAdjDistrList.add(distrOld);
                  }
            }

            return atomAdjDistrList;
      }

      public static ArrayList<Atom> getAllDistrsUnassignedBorder(){
            ArrayList<Atom> allDistrsUnassignedBorder = new ArrayList<Atom>();
            for(Atom atom: atomsArrayList){
                  if(atom.getDistrictNum()==-1){
                        ArrayList<Atom> nbAtomList = atom.getNbAtomsList();
                        boolean isOneNbAtomAssigned = false;
                        for(Atom nbAtom: nbAtomList){
                              if(nbAtom.getDistrictNum() != -1){
                                    isOneNbAtomAssigned = true;
                              }
                        }
                        if(isOneNbAtomAssigned){
                              allDistrsUnassignedBorder.add(atom);
                        }
                  }
            }
            return allDistrsUnassignedBorder;
      }

      public static ArrayList<Atom> getOneDistrUnassignedBorder(District distr){
            ArrayList<Atom> oneDistrUnassignedBorder = new ArrayList<Atom>();
            for(Atom atom: atomsArrayList){
                  if(atom.getDistrictNum()==-1){
                        ArrayList<Atom> nbAtomList = atom.getNbAtomsList();
                        boolean isOneNbAtomBelongsGivenDistr = false;
                        for(Atom nbAtom: nbAtomList){
                              if(nbAtom.getDistrictNum() == distr.getID()){
                                    isOneNbAtomBelongsGivenDistr = true;
                              }
                        }
                        if(isOneNbAtomBelongsGivenDistr){
                              oneDistrUnassignedBorder.add(atom);
                        }
                  }
            }
            return oneDistrUnassignedBorder;
      }

      public static double getDistrPlanCmptness(){
            ArrayList<Double> dplanCmptArr = new ArrayList<Double> ();
            for(District distr: districtsArrayList){
                  dplanCmptArr.add(distr.getCmptness());
            }
            return MyMath.sum(dplanCmptArr);
```

```
        }

        public static double getCfsProbStdev(){
              ArrayList<Double> probArr = new ArrayList<Double>();
              for(District distr: districtsArrayList){
                    probArr.add(distr.getCfsProb());
              }
              return MyMath.stdev(probArr);
        }

        public static double getTolRespDistance(){
              ArrayList<Double> respDistArr = new ArrayList<Double>();
              for(District distr: districtsArrayList){
                    respDistArr.add(distr.getRespDistance());
              }
              return MyMath.sum(respDistArr);
        }

        public static District selectDistrWithMinProb(ArrayList<District> distrList) {
              District resultDistr = null;
              // choose the district with lowest cfsProb

              double minProb = 9999;
              for (District distr : distrList) {
                    if (distr.getCfsProb() < minProb) {
                          minProb = distr.getCfsProb();
                          resultDistr = distr;
                    }
              }
              return resultDistr;
        }

        public static double getAtomsAssignedRate(){
              //get atoms assigned rate
              double atomAssignedCount = 0;
              double atomAssignedRate;
              for(Atom atom: atomsArrayList){
                    if(atom.getDistrictNum()!=-1){
                          atomAssignedCount ++;
                    }
              }
              atomAssignedRate = atomAssignedCount/atomsArrayList.size();
              return atomAssignedRate;
        }

        public static void initDistricts() {
              District.setCountZero();
              // init districts
              districtsArrayList = new ArrayList<District>();
//            System.out.println("\n--initDistricts--");
              for (int i = 0; i < DISTRICT_NUM; i++) {
                    districtsArrayList.add(new District(atomsArrayList));
//                  System.out.println(districtsArrayList.get(i));
              }
              District.setCountZero();
        }

        public static void checkDupSeeds(ArrayList<Atom> seedsAtomList) {
              boolean isDuplicate = checkArrayListDuplicate(seedsAtomList);
              if(isDuplicate){
                    System.out.println("Duplicate seeds generated");
                    System.exit(0);
              }
        }

        public static boolean checkArrayListDuplicate(ArrayList<Atom> arrList) {
              //check duplicate
              boolean isDuplicate = false;
```

```
            for (int i = 0; i < arrList.size(); i++) {
                  for (int j = i + 1; j < arrList.size(); j++) {
//                      System.out.println("compare " + i + " " + j);
                        if(arrList.get(i).getId() == arrList.get(j).getId()){
                              isDuplicate = true;
                        }
                  }
            }
            return isDuplicate;
      }

      public static ArrayList<Atom> removeArrayListDuplicate(ArrayList<Atom> arrList){
            int duplicateIndex = -1;
            for (int i = 0; i < arrList.size(); i++) {
                  for (int j = i + 1; j < arrList.size(); j++) {
//                      System.out.println("compare " + i + " " + j);
                        if(arrList.get(i).getId() == arrList.get(j).getId()){
                              duplicateIndex = i;
                              break;
                        }
                  }
            }
            arrList.remove(duplicateIndex);
            return arrList;

      }

      public static ArrayList<Atom> getSeeds(ArrayList<Coordinate> seedsCoorsList) {


            ArrayList<Atom> seedsAtomList = findNearestSeedAtoms(seedsCoorsList);
            //set district num
//          System.out.println("seedsAtomList.size()"+seedsAtomList.size());
            for (int i = 0; i < seedsAtomList.size(); i++) {
                  seedsAtomList.get(i).setDistrictNum(i);
            }

//          System.out.println("Seeds Atoms Coors: ");
            for(Atom atom: seedsAtomList){
//                System.out.println(atom.getId() + "\t" + atom.getCoor().getX() + "\t" +
    atom.getCoor().getY());
            }
            return seedsAtomList;
      }

      public static ArrayList<Atom> findNearestSeedAtoms(ArrayList<Coordinate>
          seedsCoorsList) {
//          System.out.println("atomsArrayList.size(): " + atomsArrayList.size());

            ArrayList<Atom> seedsAtomList = new ArrayList<Atom> ();
            for(Coordinate coor: seedsCoorsList){
                  double minDist = 999999;
                  int minDistAtomID = -1;
                  for(Atom atom: atomsArrayList){
                        double currDist = Coordinate.calDistTwoPoints(atom.getCoor(),
                            coor);
                        if(currDist < minDist){
                              minDist = currDist;
                              minDistAtomID = atom.getId();
                        }
                  }
                  seedsAtomList.add(atomsArrayList.get(minDistAtomID));
            }
            return seedsAtomList;
      }

      public static ArrayList<Coordinate> getTwoCirclesSeedsCoords() {
            ArrayList<Coordinate> twoCircleSeedsCoordsList = new ArrayList<Coordinate>();
            ArrayList<Coordinate> Circle1SeedsCoordsList = new ArrayList<Coordinate>();
            Circle1SeedsCoordsList = genOneCircleSeedsCoors(cenX1, cenY1, n1,
                r1*cityRadius);
            for(Coordinate coor: Circle1SeedsCoordsList){
                  twoCircleSeedsCoordsList.add(coor);
//                System.out.println(coor.getX() + "\t" + coor.getY());
            }

            ArrayList<Coordinate> Circle2SeedsCoordsList = new ArrayList<Coordinate>();
            Circle2SeedsCoordsList = genOneCircleSeedsCoors(cenX2, cenY2, n2,
                r2*cityRadius);
            for(Coordinate coor: Circle2SeedsCoordsList){
                  twoCircleSeedsCoordsList.add(coor);
//                System.out.println(coor.getX() + "\t" + coor.getY());
            }
            return twoCircleSeedsCoordsList;
      }

      public static ArrayList<Coordinate> genOneCircleSeedsCoors(double centerX, double
          centerY, int n, double r) {
            ArrayList<Coordinate> oneCircleSeedsCoordsList = new ArrayList<Coordinate>();
            int randAngle = MyRandom.randomInt(360);

            for (int i = 0; i < n; i++) {
                  double randTheta = (double)randAngle*Math.PI/180.0;
                  double seedX = centerX + Math.cos(randTheta)*r;
                  double seedY = centerY + Math.sin(randTheta)*r;
                  Coordinate coor = new Coordinate(seedX, seedY);
                  oneCircleSeedsCoordsList.add(coor);
                  randAngle += 360/n;
            }
            return oneCircleSeedsCoordsList;
      }

      public static double getCityRadius() {
            ArrayList<Double> distToCenterList = new ArrayList<Double>();
            for(Atom atom: atomsArrayList){
                  distToCenterList.add(atom.getDistToCenter());
            }
            return MyMath.max(distToCenterList);
      }

      public static void setAtomDistToCenter() {
            for(Atom atom: atomsArrayList){
                  double distToCenter = Coordinate.calDistTwoPoints(atom.getCoor(),
                      cityCenter);
                  atom.setDistToCenter(distToCenter);
            }
      }

      public static Coordinate getCityCenter() {
            Coordinate result = new Coordinate();
            ArrayList<Double> xList = new ArrayList<Double>();
            ArrayList<Double> yList = new ArrayList<Double>();
            for(Atom atom: atomsArrayList){
                  xList.add(atom.getCoor().getX());
                  yList.add(atom.getCoor().getY());
            }
            result.setX(MyMath.average(xList));
            result.setY(MyMath.average(yList));
            return result;
      }

      public static void readDataSet() {
            Input.readFileIntoDoubleArray(CFS_PROB_EACH_ATOM_PATH, CFS_PROB_EACH_ATOM);
            Input.readFileIntoDoubleMatrix(COOR_EACH_ATOM_PATH, COOR_EACH_ATOM);
            Input.readFileIntoIntMatrix(ADJACENCY_MATRIX_PATH, ADJACENCY_MATRIX);
      }

      public static void initAtoms() {
            Atom.setCountZero();
            atomsArrayList = new ArrayList<Atom>();
            for (int i = 0; i < ATOM_NUM; i++) {
                  Atom atom = new Atom(COOR_EACH_ATOM[i][0], COOR_EACH_ATOM[i][1]);
                  atomsArrayList.add(atom);
            }

            // cfs prob for each atom, district plan, patrol frequency
            for (int i = 0; i < atomsArrayList.size(); i++) {
                  Atom atom = atomsArrayList.get(i);
                  atom.setCFSprob(CFS_PROB_EACH_ATOM[i]);
            }

            //init adjacency
            for (int i = 0; i < ADJACENCY_MATRIX.length; i++) {
                  for (int j = 0; j < ADJACENCY_MATRIX[0].length; j++) {
                        if(ADJACENCY_MATRIX[i][j] != -1){
                              atomsArrayList.get(i).addNbAtom(atomsArrayList.get(ADJACENCY_MATRIX[i][j]));
                        }
                  }
            }

            //init district to -1
            for (int i = 0; i < atomsArrayList.size(); i++) {
                  Atom atom = atomsArrayList.get(i);
                  atom.setDistrictNum(-1);
            }
            Atom.setCountZero();
      }
}
```

B.3.2 My Math Functions

```
// MyMath.java

package RunSim;

import java.util.ArrayList;
import java.util.List;

public class MyMath {

      public static <T> double average(ArrayList<T> arrList){
            double sum = 0;
            for(T item: arrList){
                  sum += Double.parseDouble(item.toString());
            }
            return sum/arrList.size();
      }

      public static <T> double max(ArrayList<T> arrList){
            double max = Double.MIN_VALUE;
            for(T item: arrList){
                  double currItem = Double.parseDouble(item.toString());
                  if(currItem > max){
                        max = currItem;
                  }
            }
            return max;
      }
```

```
	public static <T> double min(ArrayList<T> arrList){
		double min = Double.MAX_VALUE;
		for(T item: arrList){
			double currItem = Double.parseDouble(item.toString());
			if(currItem < min){
				min = currItem;
			}
		}
		return min;
	}

	public static <T> double sum(ArrayList<T> arrList){
		double sum = 0;
		for(T item: arrList){
			sum += Double.parseDouble(item.toString());
		}
		return sum;
	}

	public static <T> double stdev(ArrayList<T> arrList){
		double ave = average(arrList);
		double ss = 0;
		for(T item: arrList){
			ss += Math.pow(((Double)item-ave), 2);
		}
		return Math.pow(ss/(arrList.size()-1), 0.5);
	}

	public static void zeroIntArray(int [] array){
		for (int i = 0; i < array.length; i++) {
			array[i] = 0;
		}
	}

	public static void zeroDoubleArray(double [] array){
		for (int i = 0; i < array.length; i++) {
			array[i] = 0;
		}
	}

	public static void zeroDoubleMatrix(double [][] matrix){
		for (int i = 0; i < matrix.length; i++) {
			for (int j = 0; j < matrix[0].length; j++) {
				matrix[i][j] = 0;
			}
		}
	}

	public static void zeroIntMatrix(int [][] matrix){
		for (int i = 0; i < matrix.length; i++) {
			for (int j = 0; j < matrix[0].length; j++) {
				matrix[i][j] = 0;
			}
		}
	}
}
```

B.3.3 Simulation Parameters

```
// SimParameters.java

package RunSim;

import java.util.ArrayList;
import java.math.*;

import Entities.Atom;
import Entities.Car;
import Entities.District;
import SimEvent.EventList;

public interface SimParameters {

      //input paths
      static String CFS_PROB_EACH_ATOM_PATH = "IO\\Input\\CFS_PROB_EACH_ATOM.txt";
      static String PATROL_FREQUENCY_EACH_ATOM_PATH =
          "IO\\Input\\PATROL_FREQUENCY_EACH_ATOM.txt";
      static String COOR_EACH_ATOM_PATH = "IO\\Input\\COOR_EACH_ATOM.txt";
      static String DISTRICT_PLAN_MATRIX_PATH = "IO\\Input\\DISTRICT_PLAN_MATRIX_BAD.txt";
      static String ADJACENCY_MATRIX_PATH = "IO\\Input\\ADJACENCY_MATRIX.txt";
      static String SIM_RUN_CONDITIONS_PATH = "IO\\Input\\SIM_RUN_CONDITIONS.txt";

      //output paths
      static String PLAN_MEASURE_LOG_PATH = "IO\\Output\\PLAN_MEASURE_LOG.txt";
      static String CURR_PLAN_ARRAY_LOG_PATH = "IO\\Output\\CURR_PLAN_ARRAY_LOG.txt";
      static String NEW_PLAN_ARRAY_LOG_PATH = "IO\\Output\\NEW_PLAN_ARRAY_LOG.txt";
      static String NEW_PLAN_EVALUATION_LOG_PATH =
          "IO\\Output\\NEW_PLAN_EVALUATION_LOG.txt";

      //setting
      final static int SIM_ANL_ITER_NUM = 3000; //at least 10 times, or just 1 time
      final static long SIM_RUN_LENGH = 10000000; // usually 10000000 for convergence
      final static int STARTING_PLAN_MATRIX_ROW = 0; //cannot run: 9 //recommend 1,
          starting score 0.2
      final static int TOTAL_DISTRICT_PLAN_NUM = 100; //number of rows in matrix read
          into memory
      final static int DISTRICT_PLAN_READIN_TOL_NUM = 20;
      final static boolean DISPLAY_CONSOLE_INFO = false;
      final static int PRINT_BEST_PLAN_INTERVAL = SIM_ANL_ITER_NUM/10;

      final static int CAR_NUM = 8;
      final static int DISTRICT_NUM = 8;
      final static int ATOM_NUM = 323;
      static double CFS_PROB_EACH_ATOM[] = new double[ATOM_NUM];
      static double PATROL_FREQUENCY_EACH_ATOM[] = new double[ATOM_NUM];
      static int DISTRICT_PLAN_ARRAY[] = new int[ATOM_NUM];
      static int DISTRICT_PLAN_MATRIX[][] = new
          int[DISTRICT_PLAN_READIN_TOL_NUM][ATOM_NUM];
      static int ADJACENCY_MATRIX[][] = new int[ATOM_NUM][4];
      static double COOR_EACH_ATOM[][] = new double[ATOM_NUM][2];
      static ArrayList<Atom> atomsArrayList = new ArrayList<Atom>();
      static ArrayList<District> districtsArrayList = new ArrayList<District>();
      static ArrayList<Car> carsArrayList = new ArrayList<Car>();
      static EventList eventList = new EventList();

      //record information within a strategy evaluation
      static ArrayList<Double> responseTimeArrayList = new ArrayList<Double>();

      //record summary information for each district strategy
      static ArrayList<Double> strategyAveRespTimeList = new ArrayList<Double>();
      static ArrayList<Double> strategyWorkLoadPropStdList = new ArrayList<Double>();

      //simulated annealing parameters
      final static double INIT_TEMP = 10.0;
      final static double END_TEPM = 0.001;
      final static double ALPHA = Math.pow(END_TEPM/INIT_TEMP,
          1/((double)SIM_ANL_ITER_NUM-1));
//    final static double ALPHA = 0.999079297956651; //temperature reduction factor
      final static double RAND_MASS_CENTER_TOL = 0.005;
      final static int LONG_BORDER_DISTRICT_SIZE = 15;
      final static double LONG_DISTRICT_BORDER_RATIO = 0.7;
      final static int ATOM_CHAGNED_NUM_PER_TURN = 10;
```

```
	final static int ATOM_SMOOTHED_NUM_PER_TURN = 15;
	final static double BREAK_CMPT_BOUND = 1.5;
	final static double BREAK_PROB_BOUND = 0.16;
	final static int SIZE_LB = 2;
	final static int SIZE_UB = 130;
	final static double PROB_LB = 0.0001;
	final static double RPOB_UB = 0.52;
	final static double NB_DISTANCE_2ND = 0.0060; //0.0045
	final static double COMPACT_EACH_DISTRICT = 1.9;
	final static double ADJ_ATOM_DISTANCE = 0.0032;
	final static double DIG_ATOM_DISTANCE = 0.0045;
}
```

B.3.4 Run Simulated Annealing Algorithm

```
// SimRun.java
package RunSim;

import java.text.DecimalFormat;
import java.util.ArrayList;
import java.util.HashSet;
import Entities.Atom;
import Entities.CFSqueue;
import Entities.Car;
import Entities.Coordinate;
import Entities.District;
import Entities.DistrictPlan;
import SimEvent.CFS_ArrivalEvent;
import IO.Input;
import java.util.Random;
import java.util.Collections;

public class SimRun implements SimParameters {

	static int currentTick;
	static int currentCondiIndex;
	static int currentPlanIndex;
	public static int CFS_INTER_ARRIVAL_TIME_MEAN;
	public static double CAR_SPEED_MEAN;
	public static int TIME_ON_SCENE_MEAN;
	public static int PATROL_DISPATCH_RULE;//1: stay at district center, 2: random
	    patrol, 3: no cross boundary responding, stay at center
	public static int CFS_QUEUE_CAPACITY;
	public static CFSqueue cfsQueue;


	public static void main(String[] args) {
		setSimConditions();
		//main procedure begins
		//record information
		StringBuilder currPlanArrLogSb = new StringBuilder();
		StringBuilder newPlanArrLogSb = new StringBuilder();
		StringBuilder planMeasureLogSb = new StringBuilder();
		StringBuilder newPlanEvaluationSb = new StringBuilder();
		currPlanArrLogSb.append("InitTemp" + "\n");
		newPlanArrLogSb.append("InitTemp" + "\n");
		planMeasureLogSb.append("InitTemp" + "\t" + "NewPlan" + "\t" + "CurrPlan" +
		    "\t" + "BestPlan" + "\n");
		newPlanEvaluationSb.append("InitTemp" + "\t" + "AveRespTime" + "\t" +
		    "WorkloadStdev" + "\n");

//		//experiment on different initial temperature
//		double [] initTempList = {10};
//		for (int i = 0; i < initTempList.length; i++) {
```

```
//                simulatedAnnealingProcess(initTempList[i], currPlanArrLogSb,
    newPlanArrLogSb, planMeasureLogSb, newPlanEvaluationSb);
//          }
            simulatedAnnealingProcess(INIT_TEMP, currPlanArrLogSb, newPlanArrLogSb,
                planMeasureLogSb, newPlanEvaluationSb);
            IO.Output.writeToFile(CURR_PLAN_ARRAY_LOG_PATH, currPlanArrLogSb.toString());
            IO.Output.writeToFile(NEW_PLAN_ARRAY_LOG_PATH, newPlanArrLogSb.toString());
            IO.Output.writeToFile(PLAN_MEASURE_LOG_PATH, planMeasureLogSb.toString());
            IO.Output.writeToFile(NEW_PLAN_EVALUATION_LOG_PATH,
                newPlanEvaluationSb.toString());

            System.out.println("fin");
      } // end of main function

      private static void simulatedAnnealingProcess(double initTemp, StringBuilder
          currPlanArrLogSb, StringBuilder newPlanArrLogSb, StringBuilder
          planMeasureLogSb, StringBuilder newPlanEvaluationSb){
            DistrictPlan currPlan = new
                DistrictPlan(getRowFromMatrix(STARTING_PLAN_MATRIX_ROW));
            DistrictPlan bestPlan;
            DistrictPlan newPlan;
            //evaluate currPlan
            evaluateDistrictPlan(currPlan);
            System.out.println("currPlan: " + currPlan);
            //init bestPlan
            bestPlan = currPlan;
            System.out.println("bestPlan: " + bestPlan);
            //init temperature
            double temperature = initTemp;

            //simulated annealing procedure
            for (int i = 0; i < SIM_ANL_ITER_NUM; i++) {
                  //update temperautre
//                System.out.println("ALPHA: " + ALPHA);
                  temperature = temperature * ALPHA;
                  System.out.println("Current Temperature: " + temperature);

                  //adjust currentPlan to newPlan
                  newPlan = adjustPlan(currPlan);
                  //evaluate newPlan
                  System.out.println("");
//                System.out.println("\ninit newPlan");

                  evaluateDistrictPlan(newPlan);
                  System.out.println("\nCurrent SIM_ANL_ITER_NUM:" + (i+1) + "/" +
                      SIM_ANL_ITER_NUM);
                  System.out.println("current initTemp: " + initTemp);
                  System.out.println("newPlan: " + newPlan);
                  System.out.println("currPlan: " + currPlan);
                  if (newPlan.compareTo(bestPlan) > 0) {// better than bestPlan
                        bestPlan = newPlan;
                        System.out.println("newPlan > bestPlan, so bestPlan replaced.");
                  }
                  if (newPlan.compareTo(currPlan) > 0) { // better than currPlan
                        currPlan = newPlan;
                        System.out.println("newPlan > currPlan, so currPlan replaced.");
                  } else {//worse than currPlan
                        double probAccpt = updateProbAccpt(currPlan, newPlan,
                            temperature);
                        System.out.println("probAccpt = " + probAccpt);
                        if (new Random().nextDouble() < probAccpt) {
                              currPlan = newPlan;
                              System.out.println("accept newPlan under p = " + probAccpt
                                  + ", currPlan replaced.");
                        } else {
                              System.out.println("currentPlan does not change");
                        }
                  }
                  System.out.println("\nAfter this iteration: ");
                  System.out.println("currPlan: " + currPlan);
                  currPlan.showPlan();
                  System.out.println("bestPlan: " + bestPlan);
                  bestPlan.showPlan();

                  //record information each iteration
                  currPlanArrLogSb.append(initTemp + "\t");
                  newPlanArrLogSb.append(initTemp + "\t");
                  for (int j = 0; j < currPlan.getPlanArr().length; j++) {
                        currPlanArrLogSb.append(currPlan.getPlanArr()[j] + "\t");
                        newPlanArrLogSb.append(newPlan.getPlanArr()[j] + "\t");
                  }
                  currPlanArrLogSb.append("\n");
                  newPlanArrLogSb.append("\n");
                  planMeasureLogSb.append(initTemp + "\t" +
                      newPlan.getMeasure().getWeightedScore() + "\t" +
                      currPlan.getMeasure().getWeightedScore()+ "\t" +
                      bestPlan.getMeasure().getWeightedScore() + "\n");
                  newPlanEvaluationSb.append(initTemp + "\t" +
                      newPlan.getMeasure().getAveRespTime() + "\t" +
                      newPlan.getMeasure().getWorkloadStdev() + "\n");
            }// end of for loop

      }


      private static double updateProbAccpt(DistrictPlan currPlan, DistrictPlan newPlan,
          double temperature) {
            double deltaScore = newPlan.getMeasure().getWeightedScore() -
                currPlan.getMeasure().getWeightedScore();
            assert deltaScore <= 0;
            double acptProb = Math.exp(deltaScore/temperature);
            return acptProb;
      }

      private static DistrictPlan adjustPlan(DistrictPlan currPlan) {
            System.out.println("\nadjustPlan");
            DistrictPlan newPlan = new DistrictPlan(currPlan.getPlanArr());

            newPlan = mergeCutAnyTwoPlan(newPlan);
            newPlan = cutConvexDistrict(newPlan);
            newPlan = cutLongBorderDistrict(newPlan);
            newPlan = mergeCutPlanBasedOnProb(newPlan);
            newPlan = mergeCutPlanBasedOnCmpt(newPlan);
            newPlan = changeNewPlanRandomly(newPlan);
            newPlan = reGrowPlan(newPlan);
            return newPlan;
      }

      private static DistrictPlan reGrowPlan(DistrictPlan newPlan) {

            System.out.println("before seeds process:");
            int []dPlanArr1 = newPlan.getPlanArr();
            for (int i = 0; i < dPlanArr1.length; i++) {
                  System.out.print(dPlanArr1[i] + "\t");
            }
            System.out.println();

            int []dPlanArr= new int[ATOM_NUM];
            do{
                  System.out.println("reGrowPlan");
                  ArrayList<Coordinate> seedsCoorsList = new ArrayList<Coordinate>();


                  for(District district: districtsArrayList){
                        seedsCoorsList.add(district.getRandomMassCenter());
                  }
```

```
//                //print
//                for(Coordinate coor: seedsCoorsList){
//                      System.out.println(coor.toString());
//                }

                  dPlanArr = DistrAlgProc.genOnePlanBySeeds(seedsCoorsList);

                  //print
                  System.out.println("after seeds process:");
                  for (int i = 0; i < dPlanArr.length; i++) {
                        System.out.print(dPlanArr[i] + "\t");
                  }
                  System.out.println("");

            }while(dPlanArr[0] == -1);
            newPlan = new DistrictPlan(dPlanArr);

            return newPlan;
      }

      private static DistrictPlan mergeCutAnyTwoPlan(DistrictPlan newPlan) {
            DistrictPlan tempPlan = new DistrictPlan(newPlan.getPlanArr());
            System.out.println("\nmergeCutAnyTwoPlan");
            init(tempPlan.getPlanArr());
            int trialNum = 3;
            for (int i = 0; i < trialNum; i++) {
                  for(District districtA: getPorbDistrictListRevOrd(districtsArrayList)){
                        ArrayList<District> adjList = districtA.getAdjDistrictArrList();
                        for(District districtB: getPorbDistrictListSortedByProb(adjList)){
//                            System.out.println("\nmergeCutAnyTwoPlan from big to
    small\n");
                              tempPlan = new DistrictPlan(newPlan.getPlanArr());
                              init(tempPlan.getPlanArr());
                              //make them refer to updated districtsArrayList
                              districtA = districtsArrayList.get(districtA.getID());
                              districtB = districtsArrayList.get(districtB.getID());
                              double districtAprob = districtA.getCfsProb();
                              double districtBprob = districtB.getCfsProb();
//                            System.out.println("districtA: " + districtA.getID() + "
    prob: " + districtA.getCfsProb());
//                            System.out.println("districtB: " + districtB.getID() + "
    prob: " + districtB.getCfsProb());
//                            mergeTwoDistrict(tempPlan, mrgDistrict, mrgedDistrict);
                              System.out.println("before merge");
                              tempPlan.showPlan();
                              mergeTwoDistrict(tempPlan, districtA, districtB);
                              init(tempPlan.getPlanArr());
                              System.out.println("After merge");
                              tempPlan.showPlan();
                              //make them refer to updated districtsArrayList
                              //old statement
                              districtA = districtsArrayList.get(districtA.getID());

//                            //new way, random select one to break
//                            int randDistrictNum;
//                            do{
//                                  randDistrictNum = new Random().nextInt(DISTRICT_NUM);
//                            }while(randDistrictNum == districtB.getID());
//                            districtA = districtsArrayList.get(randDistrictNum);

                              districtB = districtsArrayList.get(districtB.getID());
//                            System.out.println("after random selection");
//                            System.out.println("districtA: " + districtA.getID());
//                            System.out.println("districtB: " + districtB.getID());
//                            cutPlanToTwo(tempPlan, mrgedDistrict, brkDistrict);
                              cutPlanToTwo(tempPlan, districtB, districtA);
                              init(tempPlan.getPlanArr());
                              System.out.println("After cut");
                              tempPlan.showPlan();

//                            //print check result
//                            if(!checkCompact()){
//                                  System.out.println("checkCompact failed");
//                            }else if(!checkSize()){
//                                  System.out.println("checkSize failed");
//                            }else if(!checkPlanConectivity()){
//                                  System.out.println("checkPlanConectivity failed");
//                            }else if(!checkBorderNbDistance()){
//                                  System.out.println("checkBorderNbDistance failed");
//                            }else if(!checkProb()){
//                                  System.out.println("checkProb failed");
//                            }



//                            System.exit(0);

                              if((checkCompact() && checkSize() &&
                                  checkPlanConectivity()&& checkBorderNbDistance() &&
                                  checkProb())){
                                    System.out.println("\tmerge and cut successful!");
                                    System.out.println("\tdistrictA: " +
                                        districtA.getID() + " prob: " + districtAprob);
                                    System.out.println("\tdistrictB: " +
                                        districtB.getID() + " prob: " + districtBprob);
                                    return tempPlan;
                              }
                        }
                  }
            }


            System.out.println("\talready tried all possibilities, failed......");
            return newPlan;
      }

      private static DistrictPlan cutConvexDistrict(DistrictPlan newPlan) {
            DistrictPlan tempPlan = new DistrictPlan(newPlan.getPlanArr());
            System.out.println("\ncutConvexDistrict");
            init(tempPlan.getPlanArr());

            ArrayList<Integer> cvxDistrictIdList = getConvexDistrictIdList();

            if(cvxDistrictIdList.size() == 0){
                  System.out.println("\tno need to cutConvexDistrict");
                  return tempPlan;
            }

            int brkDistrictID = cvxDistrictIdList.get(new
                Random().nextInt(cvxDistrictIdList.size()));
            District brkDistrict = districtsArrayList.get(brkDistrictID);
            System.err.println("\tDistrict " + brkDistrict.getID() + " is convex!");
            District mrgDistrict;
            District mrgedDistrict;
            for(District district: getPorbDistrictListSortedByProb(districtsArrayList)){
                  ArrayList<District> adjList = district.getAdjDistrictArrList();
                  for(District adjDistrict: getPorbDistrictListSortedByProb(adjList)){
                        tempPlan = new DistrictPlan(newPlan.getPlanArr());
                        init(tempPlan.getPlanArr());
                        mrgDistrict = districtsArrayList.get(district.getID());
                        mrgedDistrict = districtsArrayList.get(adjDistrict.getID());
                        mergeTwoDistrict(tempPlan, mrgDistrict, mrgedDistrict);
                        cutPlanToTwo(tempPlan, mrgedDistrict, brkDistrict);
                        init(tempPlan.getPlanArr());
                        if((checkCompact() && checkSize() && checkPlanConectivity()&&
                            checkBorderNbDistance() && checkProb())){
                              System.out.println("\tmerge and cut successful!");
                              return tempPlan;
                        }
                  }
            }
            System.out.println("\talready tried all possibilities, failed......");
            return newPlan;
      }

      private static DistrictPlan cutLongBorderDistrict(DistrictPlan newPlan) {
            System.out.println("\ncutLongBorderDistrict");
            DistrictPlan tempPlan = new DistrictPlan(newPlan.getPlanArr());
            init(tempPlan.getPlanArr());

            ArrayList <Integer> longBorderDistrictList =
                getLongBorderDistrictList(LONG_BORDER_DISTRICT_SIZE,
                LONG_DISTRICT_BORDER_RATIO);

            if(longBorderDistrictList.size() == 0){
                  System.out.println("\tno need to cutLongBorderDistrict");
                  return tempPlan;
            }
            int brkDistrictID = longBorderDistrictList.get(new
                Random().nextInt(longBorderDistrictList.size()));
            District brkDistrict = districtsArrayList.get(brkDistrictID);
            System.err.println("\tDistrict " + brkDistrict.getID() + " has large border
                ratio!");
            District mrgDistrict;
            District mrgedDistrict;
            for(District district: getPorbDistrictListSortedByProb(districtsArrayList)){
                  ArrayList<District> adjList = district.getAdjDistrictArrList();
                  for(District adjDistrict: getPorbDistrictListSortedByProb(adjList)){
//                      System.out.println("\nmergeCutPlanBasedOnProb");
                        tempPlan = new DistrictPlan(newPlan.getPlanArr());
                        init(tempPlan.getPlanArr());
                        mrgDistrict = districtsArrayList.get(district.getID());
                        mrgedDistrict = districtsArrayList.get(adjDistrict.getID());
//                      System.out.println("mrgDistrict: " + mrgDistrict.getID() + "
    prob: " + mrgDistrict.getCfsProb());
//                      System.out.println("mrgedDistrict: " + mrgedDistrict.getID() + "
    prob: " + mrgedDistrict.getCfsProb());
                        mergeTwoDistrict(tempPlan, mrgDistrict, mrgedDistrict);
                        cutPlanToTwo(tempPlan, mrgedDistrict, brkDistrict);
                        init(tempPlan.getPlanArr());
                        if((checkCompact() && checkSize() && checkPlanConectivity()&&
                            checkBorderNbDistance() && checkProb())){
                              System.out.println("\tmerge and cut successful!");
//                            System.out.println("\tbrkDistrict: " + brkDistrict.getID()
    + " prob: " + brkDistrict.getCfsProb());
                              return tempPlan;
                        }
                  }
            }
            System.out.println("\talready tried all possibilities, failed......");
            return newPlan;
      }


      private static DistrictPlan mergeCutPlanBasedOnProb(DistrictPlan newPlan) {

            DistrictPlan tempPlan = new DistrictPlan(newPlan.getPlanArr());
            System.out.println("\nmergeCutPlanBasedOnProb");
            init(tempPlan.getPlanArr());

//          for(District district: districtsArrayList){
```

```
//                System.out.println("District: " + district.getID() + " prob: " +
    district.getCfsProb());
//          }
//          System.out.println();

            District maxProbDistrict = getMaxProbDistrict(districtsArrayList);
//          System.out.println("maxProbDistrict: " + maxProbDistrict.getID());
//          System.out.println("prob: " + maxProbDistrict.getCfsProb());

            if(maxProbDistrict.getCfsProb() < BREAK_PROB_BOUND){
                  System.out.println("\tno need to merge and cut!");
                  return tempPlan;
            }
            //rewrite here
            District brkDistrict = maxProbDistrict;
            District mrgDistrict;
            District mrgedDistrict;
            for(District district: getPorbDistrictListSortedByProb(districtsArrayList)){
                  ArrayList<District> adjList = district.getAdjDistrictArrList();
                  for(District adjDistrict: getPorbDistrictListSortedByProb(adjList)){
//                      System.out.println("\nmergeCutPlanBasedOnProb");
                        tempPlan = new DistrictPlan(newPlan.getPlanArr());
                        init(tempPlan.getPlanArr());
                        mrgDistrict = districtsArrayList.get(district.getID());
                        mrgedDistrict = districtsArrayList.get(adjDistrict.getID());
//                      System.out.println("mrgDistrict: " + mrgDistrict.getID() + "
    prob: " + mrgDistrict.getCfsProb());
//                      System.out.println("mrgedDistrict: " + mrgedDistrict.getID() + "
    prob: " + mrgedDistrict.getCfsProb());
                        mergeTwoDistrict(tempPlan, mrgDistrict, mrgedDistrict);
                        cutPlanToTwo(tempPlan, mrgedDistrict, brkDistrict);
                        init(tempPlan.getPlanArr());
                        if((checkCompact() && checkSize() && checkPlanConectivity()&&
                            checkBorderNbDistance() && checkProb())){
                              System.out.println("\tmerge and cut successful!");
                              System.out.println("\tbrkDistrict: " + brkDistrict.getID()
                                  + " prob: " + brkDistrict.getCfsProb());
                              System.out.println("\tcurrent prob stdev: " +
                                  currentCfsProbStdev());
                              return tempPlan;
                        }
                  }
            }
            System.out.println("\talready tried all possibilities, failed......");
            return newPlan;
      }

      private static DistrictPlan mergeCutPlanBasedOnCmpt(DistrictPlan newPlan) {

            DistrictPlan tempPlan = new DistrictPlan(newPlan.getPlanArr());
            System.out.println("\nmergeCutPlanBasedOnCmpt");
            init(tempPlan.getPlanArr());

//          for (District district : districtsArrayList) {
//                System.out.println("District: " + district.getID() + " cmpt: "+
    district.getMaxDistanceAreaRatio());
//          }
//          System.out.println();

            District maxCmptDistrict = getWorstCmptDistrict(districtsArrayList);
//          System.out.println("maxCmptDistrict: " + maxCmptDistrict.getID());
//          System.out.println("cmpt: " + maxCmptDistrict.getMaxDistanceAreaRatio());

            if (maxCmptDistrict.getMaxDistanceAreaRatio() < BREAK_CMPT_BOUND) {
                  System.out.println("\tno need to merge and cut!");
                  return tempPlan;
            }
            // rewrite here
            District brkDistrict = maxCmptDistrict;
            District mrgDistrict;
            District mrgedDistrict;
            for (District district : getPorbDistrictListSortedByProb(districtsArrayList))
                {
                  ArrayList<District> adjList = district.getAdjDistrictArrList();
                  for (District adjDistrict : getPorbDistrictListSortedByProb(adjList)) {
//                        System.out.println("\nmergeCutPlanBasedOnCmpt_version2");
                        tempPlan = new DistrictPlan(newPlan.getPlanArr());
                        init(tempPlan.getPlanArr());
                        mrgDistrict = district;
                        mrgedDistrict = adjDistrict;
//                        System.out.println("mrgDistrict: " + mrgDistrict.getID()+ " prob:
    " + mrgDistrict.getCfsProb());
//                        System.out.println("mrgedDistrict: " + mrgedDistrict.getID()+ "
    prob: " + mrgedDistrict.getCfsProb());

                        mergeTwoDistrict(tempPlan, mrgDistrict, mrgedDistrict);
                        cutPlanToTwo(tempPlan, mrgedDistrict, brkDistrict);
                        init(tempPlan.getPlanArr());
                        if ((checkCompact() && checkSize() && checkPlanConectivity()&&
                            checkBorderNbDistance() && checkProb())) {
                              System.out.println("\tmerge and cut successful!");
                              System.out.println("\tcurrent max compact: " +
                                  currentMaxDistanceAreaRatio());
                              return tempPlan;
                        }
                  }
            }
            System.out.println("\talready tried all possibilities, failed......");
            return newPlan;
      }

      private static DistrictPlan changeNewPlanRandomly(DistrictPlan newPlan) {
            System.out.println("\nchangeNewPlanRandomly");
            int atomChanged = 0;
            for (int i = 0; i < ATOM_CHAGNED_NUM_PER_TURN; i++) {
//                  System.out.println(i + " 'th try~~~");
                  DistrictPlan tempPlan;
                  Atom selectedAtom;
                  int givDistrictID, acptDistrictID;
                  int failureTimes = 0;
                  double probBfStd, probAfStd;
                  boolean conectCondition, isSuccessFlag = true;
                  do {
//                        System.out.println("\nchangeNewPlanRandomly failureTimes: " +
    failureTimes);
                        failureTimes++;
                        tempPlan = new DistrictPlan(newPlan.getPlanArr());
                        init(tempPlan.getPlanArr()); // so the district atom variables
                            can be used
                        if(failureTimes > 10){
//                              System.out.println("Already tried 10 times, give up.....");
                              isSuccessFlag = false;
                              break;
                        }
                        // get atoms in border
                        ArrayList<Atom> borderAtoms = getBorderAtomsSet();
//                        System.out.println("borderAtoms length: " + borderAtoms.size());

                        // randomly choose an atom on border
                        selectedAtom = borderAtoms.get(new
                            Random().nextInt(borderAtoms.size()));
//                        System.out.println("selectedAtom: " + selectedAtom.getId());
                        // get neighbor district id
                        HashSet<Integer> nbDistrictList = getNbDistrictList(selectedAtom);
//                        System.out.println("nbDistrictList: " + nbDistrictList);
```

```
                        // randomly select a district from nbDistrictList
                        givDistrictID = selectedAtom.getDistrictNum();
                        acptDistrictID = selectAcptDistrictID(nbDistrictList);
//                      System.out.println("acptDistrictID: " + acptDistrictID);
                        //before adjust, get cfsProbStdev
                        probBfStd = currentCfsProbStdev();
                        // adjust districtPlan
                        changeOneAtom(tempPlan, selectedAtom,
                            givDistrictID,acptDistrictID);
                        //before adjust, get cfsProbStdev
                        probAfStd = currentCfsProbStdev();
                        //checking condition
                        conectCondition = checkDistrictConectivity(givDistrictID) &&
                            checkDistrictConectivity(acptDistrictID);
                  } while (!(conectCondition && checkBorderNbDistance() && checkCompact()
                      && checkSize() && checkProb() && checkProbImpov(probBfStd,
                      probAfStd)));
                  if(isSuccessFlag){
                        newPlan = tempPlan;
                        atomChanged++;
//                      System.out.println("pass the test!!!");
                  }
            } //end of for loop
            System.out.println("\tTotal atoms changed: " + atomChanged);
            System.out.println("\tCFSprobStdev: " + currentCfsProbStdev());
            return newPlan;
      }

      private static boolean checkProbImpov(double probBfStd, double probAfStd) {
            return (probAfStd < probBfStd);
      }

      private static DistrictPlan smoothNewPlan(DistrictPlan newPlan) {
            System.out.println("\nsmoothNewPlan");
            int atomChanged = 0;
            for (int i = 0; i < ATOM_SMOOTHED_NUM_PER_TURN; i++) {
                  DistrictPlan tempPlan;
                  Atom selectedAtom;
                  int givDistrictID, acptDistrictID;
                  int failureTimes = 0;
                  boolean conectCondition, isSuccessFlag = true;
                  do {
//                      System.out.println();
//                      System.out.println("\nsmoothNewPlan, failureTimes: " +
    failureTimes);
                        failureTimes++;
                        tempPlan = new DistrictPlan(newPlan.getPlanArr());
                        init(tempPlan.getPlanArr()); // so the district atom variables
                            can be used
                        if(failureTimes > 10){
//                            System.out.println("Already tried 10 times, give up.....");
                              isSuccessFlag = false;
                              break;
                        }
                        // get isolate atoms in border
                        ArrayList<Atom> isolatedAtomsList = getIsolatedAtomsList();
//                      System.out.println("isolatedAtomsList: " + isolatedAtomsList);
                        if(isolatedAtomsList.size()==0)
                              return tempPlan;
                        // randomly choose an atom on border
                        selectedAtom = isolatedAtomsList.get(new
                            Random().nextInt(isolatedAtomsList.size()));
                        givDistrictID = selectedAtom.getDistrictNum();
                        acptDistrictID = selectedAtom.getMostFreNbDisrict();
                        // adjust districtPlan
                        changeOneAtom(tempPlan, selectedAtom,
                            givDistrictID,acptDistrictID);
                        //checking condition
```

```
                    conectCondition = checkDistrictConectivity(givDistrictID) &&
                        checkDistrictConectivity(acptDistrictID);
                } while (!(conectCondition && checkBorderNbDistance() && checkCompact() &&
                    checkSize() && checkProb()));
                if (isSuccessFlag) {
                    newPlan = tempPlan;
                    atomChanged++;
                    // System.out.println("pass the test!!!");
                }
            } //end of for loop
            System.out.println("\ttotal atoms changed: " + atomChanged);
            ArrayList<Atom> isolatedAtomsList = getIsolatedAtomsList();
            System.out.println("\tisolatedAtoms num: " + isolatedAtomsList.size());
            System.out.println("\tCFSprobStdev: " + currentCfsProbStdev());
            return newPlan;
        }

        private static void cutPlanToTwo(DistrictPlan tempPlan, District mrgedDistrict,
            District brkDistrict) {

            Coordinate massCenter = brkDistrict.getRandomMassCenter();
            int randAngle = new Random().nextInt(179) - 89; //from -89 to 89
            //Math.sin(30*Math.PI/180)
            double k = Math.tan(randAngle * Math.PI / 180);
            //line: y = k * x
            for(Atom atom: brkDistrict.getDistrictAtomsArrayList()){
                if(atom.getCoor().getY()- massCenter.getY() < k *
                    (atom.getCoor().getX() - massCenter.getX()) ){
                    tempPlan.getPlanArr()[atom.getId()] = mrgedDistrict.getID();
                }
            }
        }

        private static void mergeTwoDistrict(DistrictPlan tempPlan, District mrgDistrict,
            District mrgedDistrict) {
            for(Atom atom: mrgedDistrict.getDistrictAtomsArrayList()){
                tempPlan.getPlanArr()[atom.getId()] = mrgDistrict.getID();
            }
        }

        public static District getRandomDistrict(ArrayList<District> districtList){
            ArrayList<Double> districtProbList = new ArrayList<Double>();
            District minProbDistrict = null;
            for(District district: districtList){
                districtProbList.add(district.getCfsProb());
            }
            Collections.sort(districtProbList);
//          System.out.println("randomint: "+ new Random().nextInt(2));
//          System.out.println("districtProbList" + districtProbList);
            int randomIndex = new Random().nextInt(DISTRICT_NUM - 1);
            for(District district: districtList){
                if(district.getCfsProb() == districtProbList.get(randomIndex)){
                    minProbDistrict = district;
                }
            }
            return minProbDistrict;
        }

        private static ArrayList <Integer> getLongBorderDistrictList(int
            longBorderDistrictSize, double longDistrictBorderRatio){
            ArrayList <Integer> longBorderDistrictList = new ArrayList <Integer>();
            for(District district: districtsArrayList){
//              System.err.println("\tDistrict " + district.getID() + " size: " +
    district.getAtomNum() +" borderRatio: " + district.getBorderRatio());
                if(district.getAtomNum() > longBorderDistrictSize &&
                    district.getBorderRatio() > longDistrictBorderRatio){
                    longBorderDistrictList.add(district.getID());
                }
        }
        return longBorderDistrictList;
    }

    public static ArrayList<District>
        getPorbDistrictListSortedByProb(ArrayList<District> districtList){
          ArrayList<Double> districtProbList = new ArrayList<Double>();
          ArrayList<District> sorteddistrictProbList = new ArrayList<District>();
          for(District district: districtList){
                districtProbList.add(district.getCfsProb());
          }
          Collections.sort(districtProbList);
          for (int i = 0; i < districtProbList.size(); i++) {
                for(District district: districtList){
                      if(districtProbList.get(i) == district.getCfsProb()){
                            sorteddistrictProbList.add(district);
                      }
                }
          }
          return sorteddistrictProbList;
    }

    public static ArrayList<District> getPorbDistrictListRevOrd(ArrayList<District>
        districtList){
          ArrayList<District> sorteddistrictProbList = new ArrayList<District>();
          ArrayList<District> revOrderList = new ArrayList<District>();
          sorteddistrictProbList = getPorbDistrictListSortedByProb(districtList);
          while(sorteddistrictProbList.size() != 0){
                revOrderList.add(sorteddistrictProbList.remove(sorteddistrictProbList.size()-1));
          }
          return revOrderList;
    }

    public static District getMaxProbDistrict(ArrayList<District> districtList){
          ArrayList<Double> districtProbList = new ArrayList<Double>();
          District maxProbDistrict = null;
          for(District district: districtList){
                districtProbList.add(district.getCfsProb());
          }
          Collections.sort(districtProbList);
          int randIndex = districtProbList.size()-2 + new Random().nextInt(2);
          for(District district: districtList){
                if(district.getCfsProb() == districtProbList.get(randIndex)){
                      maxProbDistrict = district;
                }
          }
          return maxProbDistrict;
    }

    public static District getWorstCmptDistrict(ArrayList<District> districtList){
          ArrayList<Double> districtCmptList = new ArrayList<Double>();
          District worstCmptDistrict = null;
          for(District district: districtList){
                districtCmptList.add(district.getMaxDistanceAreaRatio());
          }
          Collections.sort(districtCmptList);
          int randIndex = districtCmptList.size()-2 + new Random().nextInt(2);
          for(District district: districtList){
                if(district.getMaxDistanceAreaRatio() ==
                    districtCmptList.get(randIndex)){
                      worstCmptDistrict = district;
                }
          }
          return worstCmptDistrict;
    }

    private static ArrayList<Atom> getIsolatedAtomsList() {
          ArrayList<Atom> isolatedAtomsList = new ArrayList<Atom>();
```

```
            for(Atom bdAtom: getBorderAtomsSet()){
                  if(bdAtom.getMostFreNbDisrict() != bdAtom.getDistrictNum()){
                        isolatedAtomsList.add(bdAtom);
                  }
            }
            return isolatedAtomsList;
      }

      private static double currentMaxDistanceAreaRatio() {
            double max = -1;
            for(District district: districtsArrayList){
                  if(district.getMaxDistanceAreaRatio() > max){
                        max = district.getMaxDistanceAreaRatio();
                  }
            }
            return max;
      }

      private static double currentCfsProbStdev() {
            double cfsProbStdev;
            ArrayList<Double> cfsProbList = new ArrayList<Double>();
            for(District district: districtsArrayList){
                  cfsProbList.add(district.getCfsProb());
            }
            cfsProbStdev = MyMath.stdev(cfsProbList);
            return cfsProbStdev;
      }

      private static void changeOneAtom(DistrictPlan newPlan, Atom selectedAtom,
                  int givDistrictID, int acptDistrictID) {
            newPlan.getPlanArr()[selectedAtom.getId()] = acptDistrictID;
            //adjust atom
            atomsArrayList.get(selectedAtom.getId()).setDistrictNum(acptDistrictID);
            //adjust district
            districtsArrayList.get(givDistrictID).removeAtom(atomsArrayList.get(selectedAtom.getId()));
            districtsArrayList.get(acptDistrictID).receiveAtom(atomsArrayList.get(selectedAtom.getId()));
//          System.out.println("Atom " + selectedAtom.getId()+ " changes from district "
    + givDistrictID + " to "+ acptDistrictID);
//          System.out.println("givDistrictID"+districtsArrayList.get(givDistrictID));
//          System.out.println("acptDistrictID"+districtsArrayList.get(acptDistrictID));
      }


      private static boolean checkProb() {
//          System.out.println("chekcing prob");
            boolean probCondition = true;
            for(District district: districtsArrayList){
//                System.out.println("district.getCfsProb(): " + district.getCfsProb());
                  if(district.getCfsProb() < PROB_LB || district.getCfsProb() > RPOB_UB){
                        probCondition = false;
//                      System.out.println("prob not proper");
                  }
            }
            return probCondition;
      }

      private static ArrayList<Integer> getConvexDistrictIdList(){
//          System.err.println("\ncheckDistrictGrmd");
            ArrayList<Integer> convexDistrictIdList = new ArrayList<Integer>();
            for(District district: districtsArrayList){
                  Coordinate massCenter = district.getMassCenter();
                  boolean isMassCenterInDistrict = false;
                  double mcX = massCenter.getX();
                  double mcY = massCenter.getY();
                  for(Atom atom: district.getDistrictAtomsArrayList()){
                        double lbX = atom.getCoor().getX() - ADJ_ATOM_DISTANCE/2;
                        double ubX = atom.getCoor().getX() + ADJ_ATOM_DISTANCE/2;
                        double lbY = atom.getCoor().getY() - ADJ_ATOM_DISTANCE/2;
                        double ubY = atom.getCoor().getY() + ADJ_ATOM_DISTANCE/2;
                        if(mcX > lbX && mcX < ubX && mcY > lbY && mcY < ubY){
                              isMassCenterInDistrict = true;
//                            System.err.println("MassCenter: " + massCenter +
    "atomCenter: " + atom.getCoor());
                        }

                  }//end of for
//                System.err.println("District: " + district.getID() + "
    isMassCenterInDistrict:" + isMassCenterInDistrict);
                  if(isMassCenterInDistrict == false){
                        convexDistrictIdList.add(district.getID());
                  }
            }
            return convexDistrictIdList;
      }

      private static boolean checkSize() {
            boolean sizeCondition = true;
//          System.out.println("checking size");
            for(District district: districtsArrayList){
//                System.out.println("district.getAtomNum(): " + district.getAtomNum());
                  if(district.getAtomNum() < SIZE_LB || district.getAtomNum() > SIZE_UB){
                        sizeCondition = false;
                        if(district.getAtomNum() < SIZE_LB){
//                            System.out.println("size too small");
                        }else{
//                            System.out.println("size too large");
                        }
                  }
            }
            return sizeCondition;
      }

      private static boolean checkCompact() {
            boolean cmptCondition = true;

//          System.out.println("Checking compactness");
            ArrayList<Integer> notCmptDistrictList = new ArrayList<Integer>();
            for (District district : districtsArrayList) {
//                System.out.println("districtID: " + district.getID() + "
    district.getMaxDistanceAreaRatio(): " + district.getMaxDistanceAreaRatio());
                  if (district.getMaxDistanceAreaRatio() > COMPACT_EACH_DISTRICT) {
                        notCmptDistrictList.add(district.getID());
//                      System.out.println("District " + district.getID() + " cmpt: "+
    district.getMaxDistanceAreaRatio());
                        cmptCondition = false;
//                      System.out.println("compact failed");
                  }
            }
//          System.out.println("notCmptDistrictList: " + notCmptDistrictList);
            return cmptCondition;
      }



      private static boolean checkBorderNbDistance() {
//          System.out.println("checking NbBorderDistance");
            boolean distanceCondition = true;
            for(Atom borderAtom: getBorderAtomsSet()){
                  if(!checkNbAtomDistance(borderAtom)){
                        distanceCondition = false;
//                      System.out.println("Atom: " + borderAtom.getId() + " failed in nb
    distance check");
                  }
            }
            return distanceCondition;
      }
      private static boolean checkNbAtomDistance(Atom selectedAtom) {

            boolean distanceCondition = true;
            ArrayList<Double> sortedDistanceInDistrict = new ArrayList<Double>();
            for(Atom atom:
                districtsArrayList.get(selectedAtom.getDistrictNum()).getDistrictAtomsArrayList()){
                  double distance = Coordinate.calDistTwoPoints(selectedAtom.getCoor(),
                      atom.getCoor());
                  sortedDistanceInDistrict.add(distance);
                  Collections.sort(sortedDistanceInDistrict);
            }
            if(sortedDistanceInDistrict.size() >= 3 && sortedDistanceInDistrict.get(2) >
                NB_DISTANCE_2ND){
                  distanceCondition = false;
//                System.out.println("2nd distance "+sortedDistanceInDistrict.get(2)+"
    too long!!!!failed");
            }
            return distanceCondition;
      }

      private static boolean checkPlanConectivity() {
            boolean planConectivity = true;
            for(District district: districtsArrayList){
                  if(checkDistrictConectivity(district.getID())==false){
                        planConectivity = false;
                  }
            }
            return planConectivity;
      }

      private static boolean checkDistrictConectivity(int districtID) {
//          System.out.println("chekcing connectivity");
            District district = districtsArrayList.get(districtID);
//          for(Atom atom: district.getDistrictAtomsArrayList()){
//                System.out.println("\nfor atom id: " + atom.getId());
            Atom atom = district.getDistrictAtomsArrayList().get(0);
            HashSet<Atom> conectSet = new HashSet<Atom>();
            conectSet.add(atom);
            int oriSize = 0;
            int currSize = conectSet.size();

            while (oriSize < currSize) {
                  oriSize = conectSet.size();
                  HashSet<Atom> tempSet = new HashSet<Atom>();
                  for (Atom ele : conectSet) {
                        for (Atom nb : ele.getNbAtomsListSameDistrict()) {
                              tempSet.add(nb);
                        }
                  }
                  for (Atom tempSetAtom : tempSet) {
                        conectSet.add(tempSetAtom);
                  }
                  currSize = conectSet.size();
            }
            if (conectSet.size() != district.getAtomIndexList().size()) {
                  return false;
                  // System.out.println("failed in conectivity check");
            }else{
                  return true;
            }
//          }//end of for loop
      }

      private static boolean getConvexCondition2(Atom selectedAtom) {
            System.out.println("convexCondition2");
            boolean convexCondition2 = true;
            ArrayList<Integer> nbsNBnumList = new ArrayList<Integer>();
```

```
        for(Atom neighbour: selectedAtom.getNbAtomsList()){
            if(neighbour.getDistrictNum() == selectedAtom.getDistrictNum()){
                ArrayList<Integer> nbList = new ArrayList<Integer>();
                for(Atom nbsnb: neighbour.getNbAtomsList()){
                    if(nbsnb.getDistrictNum() == neighbour.getDistrictNum() &&
                        nbsnb.getId() != selectedAtom.getId()){
                         nbList.add(nbsnb.getId());
                    }
                }
                nbsNBnumList.add(nbList.size());
            }
        }
        System.out.println("nbsNBnumList: " + nbsNBnumList);
        for(Integer i: nbsNBnumList){
            if(i < 2){
                convexCondition2 = false;
            }
        }
        return convexCondition2;
    }

    private static boolean getConvexCondition1(Atom selectedAtom) {
        // convexCondition1:an exchange which causes an atom to be adjacent to
        // only one other atom in its district;
        boolean convexCondition1;
        ArrayList<Integer> nbAtomsSameDistrictList = new ArrayList<Integer>();
        for(Atom atom: selectedAtom.getNbAtomsList()){
            if(atom.getDistrictNum() == selectedAtom.getDistrictNum()){
                nbAtomsSameDistrictList.add(atom.getId());
            }
        }
        System.out.println("convexCondition1");
        System.out.println("selectedAtom: " + selectedAtom.getId());
        System.out.println("neighbour: " + selectedAtom.getNbAtomsList().size());
        System.out.println("neighbour same districts: " + nbAtomsSameDistrictList);
        convexCondition1 = (nbAtomsSameDistrictList.size() == 1)?false:true;
        return convexCondition1;
    }

    private static int selectAcptDistrictID(HashSet<Integer> nbDistrictList) {
        int acptDistrictID = 0;
        int randInt = new Random().nextInt(nbDistrictList.size());
        int x = 0;
        for(Integer districtNum: nbDistrictList){
            if(x == randInt){
                acptDistrictID = districtNum;
                break;
            }
            x++;
        }
        return acptDistrictID;
    }

    private static HashSet<Integer> getNbDistrictList(Atom selectedAtom) {
        HashSet<Integer> nbDistrictList = new HashSet<Integer>();
        for(Atom nbAtom: selectedAtom.getNbAtomsList()){
            if(nbAtom.getDistrictNum() != selectedAtom.getDistrictNum()){
                nbDistrictList.add(nbAtom.getDistrictNum());
            }
        }
        return nbDistrictList;
    }

    private static ArrayList<Atom> getBorderAtomsSet() {
        ArrayList<Atom> borderAtoms = new ArrayList<Atom>();
        for(Atom atom:atomsArrayList){
            boolean isBorder = false;
            for(Atom neighbor: atom.getNbAtomsList()){
                if(atom.getDistrictNum() != neighbor.getDistrictNum()){
                    isBorder = true;
                }
            }
            if(isBorder){
                borderAtoms.add(atom);
            }
        }
        return borderAtoms;
    }

    private static void setSimConditions() {
        //set conditions
        CFS_INTER_ARRIVAL_TIME_MEAN = 684;
        TIME_ON_SCENE_MEAN = 1507;
        CAR_SPEED_MEAN = 0.000045;
        PATROL_DISPATCH_RULE = 2;
        CFS_QUEUE_CAPACITY = 1;
        //initiate the CFSqueue
        cfsQueue = new CFSqueue(CFS_QUEUE_CAPACITY);
        currentPlanIndex = 0;
    }

    private static int[] getRowFromMatrix(int matrix_row_num) {
        Input.readFileIntoIntMatrix(DISTRICT_PLAN_MATRIX_PATH, DISTRICT_PLAN_MATRIX);
        int [] arr = new int[ATOM_NUM];
        for (int i = 0; i < arr.length; i++) {
            arr[i] = DISTRICT_PLAN_MATRIX[matrix_row_num][i];
        }
        return arr;
    }

    private static void evaluateDistrictPlan(DistrictPlan plan) {
        if(init(plan.getPlanArr()))
            run();
        plan.setMeasure(strategyAveRespTimeList.get(strategyAveRespTimeList.size() -
            1),
                strategyWorkLoadPropStdList.get(strategyWorkLoadPropStdList.size()
                    - 1));
    }

    private static boolean init(int [] currentPlan) {
        clearAllVariables();
        if(readDataSet()){
            initDistrictPlan(currentPlan);
            initAtoms();
            initDistricts();
            initCars();
            setRespCarPreferListForEachAtom();
            initEventList();
            return true;
        }else{
            return false;
        }
    }

    private static void clearAllVariables() {
        currentTick = 0;
        MyMath.zeroDoubleArray(CFS_PROB_EACH_ATOM);
        MyMath.zeroDoubleArray(PATROL_FREQUENCY_EACH_ATOM);
        MyMath.zeroIntArray(DISTRICT_PLAN_ARRAY);
        MyMath.zeroDoubleMatrix(COOR_EACH_ATOM);
        District.setCountZero();
        Atom.setCountZero();
        Car.setCountZero();
        atomsArrayList.clear();
        districtsArrayList.clear();
        carsArrayList.clear();
```

```
            cfsQueue.clearQue();
            responseTimeArrayList.clear();
            eventList.clear();
      }

      private static void initDistrictPlan(int[] currentPlan) {
            for (int i = 0; i < currentPlan.length; i++) {
                  DISTRICT_PLAN_ARRAY[i] = currentPlan[i];
            }
      }

      private static boolean readDataSet() {

            if(!Input.readFileIntoDoubleArray(CFS_PROB_EACH_ATOM_PATH,
                CFS_PROB_EACH_ATOM)){
                  System.err.println("Error when reading CFS_PROB_EACH_ATOM");
                  return false;
            }else if(!Input.readFileIntoDoubleArray(PATROL_FREQUENCY_EACH_ATOM_PATH,
                PATROL_FREQUENCY_EACH_ATOM)){
                  System.err.println("Error when reading PATROL_FREQUENCY_EACH_ATOM");
                  return false;
            }else if(!Input.readFileIntoIntMatrix(DISTRICT_PLAN_MATRIX_PATH,
                DISTRICT_PLAN_MATRIX)){
                  System.err.println("Error when reading DISTRICT_PLAN_MATRIX");
                  return false;
            }else if(!Input.readFileIntoDoubleMatrix(COOR_EACH_ATOM_PATH,
                COOR_EACH_ATOM)){
                  System.err.println("Error when reading COOR_EACH_ATOM");
                  return false;
            }else if(!Input.readFileIntoIntMatrix(ADJACENCY_MATRIX_PATH,
                ADJACENCY_MATRIX)){
                  System.err.println("Error when reading ADJACENCY_MATRIX");
                  return false;
            }else
                  return true;
      }

      private static void initEventList() {
            eventList.add(new CFS_ArrivalEvent(CFS_INTER_ARRIVAL_TIME_MEAN));
      }

      private static void run() {
            //record one strategy statistics

            while (currentTick <= SIM_RUN_LENGH) {
                  // responds to the event
                  while (currentTick == eventList.nextEvent().getEventOccurTime()) {
                        eventList.nextEvent().actions();
                        eventList.delete();
                        if(DISPLAY_CONSOLE_INFO) System.out.println("\t" +
                            eventList.toString());
                  }
                  updateCarsAvailabilityStatis();
                  currentTick++;
            }

            displayTemporalStats();
            displayCarsTasksNumStats();
      }

      private static void displayCarsTasksNumStats() {
            DecimalFormat df = new DecimalFormat();
            df.setMaximumFractionDigits(3);
            df.setMinimumFractionDigits(3);
//          System.out.println("\nThe Number of Incidents Responded by Cars");
//          System.out.print("CarID\t\t");
            for (int i = 0; i < carsArrayList.size(); i++) {
//                System.out.print(i + "\t");
```

```
            }
//          System.out.println();
//          System.out.print("TasksNum\t");
            ArrayList<Double> arrListNum = new ArrayList<Double>();

            double sum = 0;
            for (Car car : carsArrayList) {
//                System.out.print(car.getTaskNum() + "\t");
                  arrListNum.add((double) car.getTaskNum());
                  sum += car.getTaskNum();
            }
            ArrayList<Double> arrListProb = new ArrayList<Double>();
//          System.out.print("\nProportion:\t");
            for (int i = 0; i < arrListNum.size(); i++) {
//                System.out.print(df.format(arrListNum.get(i) / sum) + "\t");
                  arrListProb.add(arrListNum.get(i) / sum);
            }
//          System.out.println();
//          System.out.println("Workload Stdev: "+ df.format(MyMath.stdev(arrListProb)));
            strategyWorkLoadPropStdList.add(MyMath.stdev(arrListProb));

      }

      private static void displayTemporalStats() {
            DecimalFormat df = new DecimalFormat();
            df.setMaximumFractionDigits(3);
            df.setMinimumFractionDigits(3);
//          System.out.println("\nTemporal Stats\t");
//          System.out.print("\tOld Printout: AveRespTime: " +
    df.format(MyMath.average(responseTimeArrayList)) + " ");
            //update strategyAveRespTimeList
            strategyAveRespTimeList.add(MyMath.average(responseTimeArrayList));
      }


      private static void updateCarsAvailabilityStatis() {
            for (Car car : carsArrayList) {
                  car.updateIdleBusyTime();
            }
      }


      private static void setRespCarPreferListForEachAtom() {
            // set respCarPreferList for each atom
            for (Atom atom : atomsArrayList) {
                  atom.setRespCarPreferList(districtsArrayList);
            }
      }

      private static void initCars() {
            // init cars
//          System.out.println("--initCars--");
            for (int i = 0; i < CAR_NUM; i++) {
                  carsArrayList.add(new Car(districtsArrayList));
//                System.out.println(carsArrayList.get(i));
            }
      }

      private static void initDistricts() {
            // init districts
//          System.out.println("\n--initDistricts--");
            for (int i = 0; i < DISTRICT_NUM; i++) {
                  districtsArrayList.add(new District(atomsArrayList));
//                System.out.println(districtsArrayList.get(i));
            }
      }

      private static void initAtoms() {
            for (int i = 0; i < ATOM_NUM; i++) {
                  Atom atom = new Atom(COOR_EACH_ATOM[i][0], COOR_EACH_ATOM[i][1]);
                  atomsArrayList.add(atom);
            }

            // cfs prob for each atom, district plan, patrol frequency
            for (int i = 0; i < atomsArrayList.size(); i++) {
                  Atom atom = atomsArrayList.get(i);
                  atom.setCFSprob(CFS_PROB_EACH_ATOM[i]);
                  atom.setDistrictNum(DISTRICT_PLAN_ARRAY[i]);
                  atom.setPatrolFrequency(PATROL_FREQUENCY_EACH_ATOM[i]);
            }

            //init adjacency
            for (int i = 0; i < ADJACENCY_MATRIX.length; i++) {
                  for (int j = 0; j < ADJACENCY_MATRIX[0].length; j++) {
                        if(ADJACENCY_MATRIX[i][j] != -1){
                              atomsArrayList.get(i).addNbAtom(atomsArrayList.get(ADJACENCY_MATRIX[i][j]));
                        }
                  }
            }
      }
}
```

B.3.5 District Fitness Function

```
package RunSim;

import java.util.ArrayList;

import org.jgap.*;

import Entities.CFSqueue;
import Entities.Coordinate;
import Entities.DistrictPlan;

public class DistrictFitnessFunction extends FitnessFunction implements
            SimParameters {
      private final static String CVS_REVISION = "$Revision: 1.18 $";
      public static final int MAX_BOUND = 4000;
      public static int count = 0;
      public static double currBestScore = -999;
      // sim conditions
      static int currentTick;
      static int currentCondiIndex;
      static int currentPlanIndex;
      public static int CFS_INTER_ARRIVAL_TIME_MEAN;
      public static double CAR_SPEED_MEAN;
      public static int TIME_ON_SCENE_MEAN;
      public static int PATROL_DISPATCH_RULE;
      public static int CFS_QUEUE_CAPACITY;
      public static CFSqueue cfsQueue;

      // record information variables
      StringBuilder currPlanArrLogSb;
      StringBuilder newPlanArrLogSb;
      StringBuilder planMeasureLogSb;
      StringBuilder newPlanEvaluationSb;

      public DistrictFitnessFunction(StringBuilder newPlanArrLogSb,
                  StringBuilder newPlanEvaluationSb) {
            this.newPlanArrLogSb = newPlanArrLogSb;
            this.newPlanEvaluationSb = newPlanEvaluationSb;
      }
```

```
    public double evaluate(IChromosome a_subject) {
        count++;
        SimRun.setSimConditions();
        ArrayList<Coordinate> seedsCoorsList = new ArrayList<Coordinate>();
        for (int i = 0; i < 8; i++) {
            double x = getNumberOfCoinsAtGene(a_subject, i * 2);
            double y = getNumberOfCoinsAtGene(a_subject, i * 2 + 1);
            Coordinate coor = new Coordinate(x, y);
            seedsCoorsList.add(coor);
        }
        System.out.println(seedsCoorsList);
        int[] dPlanArr = new int[ATOM_NUM];
        dPlanArr = DistrAlgProc.genOnePlanBySeeds(seedsCoorsList);
        System.out.println("after seeds process:");
        for (int i = 0; i < dPlanArr.length; i++) {
            System.out.print(dPlanArr[i] + "\t");
        }
        System.out.println("");
        DistrictPlan distPlan = new DistrictPlan(dPlanArr);
        // record information
        for (int j = 0; j < distPlan.getPlanArr().length; j++) {
            newPlanArrLogSb.append(distPlan.getPlanArr()[j] + "\t");
        }
        newPlanArrLogSb.append("\n");

        double fitness;
        if (dPlanArr[0] == -1) {
            fitness = 10.0;
            distPlan.setMeasure(0, 0, 0);
        } else {
            SimRun.evaluateDistrictPlan(distPlan);
            fitness = distPlan.getMeasure().getWeightedScore() - 40.0
                    * distPlan.getMeasure().getCompactness() + 100.0;
        }
        // record information
        newPlanEvaluationSb.append(distPlan.getMeasure().getAveRespTime()
                + "\t" + distPlan.getMeasure().getWorkloadStdev() + "\t"
                + distPlan.getMeasure().getCompactness() + "\t"
                + distPlan.getMeasure().getWeightedScore() + "\t" + fitness
                + "\n");
        if (distPlan.getMeasure().getWeightedScore() != 1.91
                && distPlan.getMeasure().getWeightedScore() > currBestScore) {
            currBestScore = distPlan.getMeasure().getWeightedScore();
        }

        System.out.println("PlanIndex: " + count + "\tCurrent score:"
                + distPlan.getMeasure().getWeightedScore() + "\tBest Score:"
                + currBestScore);
        System.out.println();
        // System.exit(1);
        return Math.max(1.0d, fitness);
    }

    public static double getNumberOfCoinsAtGene(
            IChromosome a_potentialSolution, int a_position) {
        Double numCoins = (Double) a_potentialSolution.getGene(a_position)
                .getAllele();
        return numCoins.doubleValue();
    }
}
```

B.3.6 Run GA

```java
package RunSim;

import java.io.*;

import org.jgap.*;
import org.jgap.audit.*;
import org.jgap.data.*;
import org.jgap.impl.*;
import org.jgap.xml.*;
import org.w3c.dom.*;

import Entities.Coordinate;

public class OptDistrictGA implements SimParameters {
      private final static String CVS_REVISION = "$Revision: 1.27 $";
      private static final int MAX_ALLOWED_EVOLUTIONS = 50;
      private static final int POPULATION_SIZE = 20;
      public static EvolutionMonitor m_monitor;

      public static void findDistrictDesign(boolean a_doMonitor,
                  StringBuilder newPlanArrLogSb, StringBuilder newPlanEvaluationSb)
                  throws Exception {
            Configuration conf = new DefaultConfiguration();
            conf.setPreservFittestIndividual(true);
            conf.setKeepPopulationSizeConstant(false);

            FitnessFunction myFunc = new DistrictFitnessFunction(newPlanArrLogSb,
                        newPlanEvaluationSb);
            conf.setFitnessFunction(myFunc);
            if (a_doMonitor) {
                  // Turn on monitoring/auditing of evolution progress.
                  // --------------------------------------------------
                  m_monitor = new EvolutionMonitor();
                  conf.setMonitor(m_monitor);
            }
            Gene[] sampleGenes = new Gene[16];

            for (int i = 0; i < 8; i++) {
                  sampleGenes[i * 2] = new DoubleGene(conf, -78.508, -78.462);
                  sampleGenes[i * 2 + 1] = new DoubleGene(conf, 38.015, 38.051);
            }

            IChromosome sampleChromosome = new Chromosome(conf, sampleGenes);
            conf.setSampleChromosome(sampleChromosome);
            conf.setPopulationSize(POPULATION_SIZE);
            Genotype population;
            try {
                  Document doc = XMLManager.readFile(new File("JGAPExample32.xml"));
                  population = XMLManager.getGenotypeFromDocument(conf, doc);
            } catch (UnsupportedRepresentationException uex) {
                  population = Genotype.randomInitialGenotype(conf);
            } catch (FileNotFoundException fex) {
                  population = Genotype.randomInitialGenotype(conf);
            }
            population = Genotype.randomInitialGenotype(conf);
            long startTime = System.currentTimeMillis();
            for (int i = 0; i < MAX_ALLOWED_EVOLUTIONS; i++) {
                  if (!uniqueChromosomes(population.getPopulation())) {
                        throw new RuntimeException("Invalid state in generation " + i);
                  }
                  if (m_monitor != null) {
                        population.evolve(m_monitor);
                  } else {
                        population.evolve();
                  }
            }
            long endTime = System.currentTimeMillis();
```

```
        System.out.println("Total evolution time: " + (endTime - startTime)
                    + " ms");
        IChromosome bestSolutionSoFar = population.getFittestChromosome();
        double v1 = bestSolutionSoFar.getFitnessValue();
        System.out.println("The best solution has a fitness value of "
                    + bestSolutionSoFar.getFitnessValue());
        bestSolutionSoFar.setFitnessValueDirectly(-1);
    }

    public static void main(String[] args) throws Exception {
        // record information
        StringBuilder newPlanArrLogSb = new StringBuilder();
        StringBuilder newPlanEvaluationSb = new StringBuilder();
        newPlanArrLogSb.append("");
        newPlanEvaluationSb.append("AveRespTime" + "\t" + "WorkloadStdev"
                    + "\t" + "Compactness" + "\t" + "TotalScore" + "\t" + "Fitness"
                    + "\n");

        findDistrictDesign(false, newPlanArrLogSb, newPlanEvaluationSb);
        IO.Output.writeToFile(NEW_PLAN_ARRAY_LOG_PATH,
                    newPlanArrLogSb.toString());
        IO.Output.writeToFile(NEW_PLAN_EVALUATION_LOG_PATH,
                    newPlanEvaluationSb.toString());
    }

    public static boolean uniqueChromosomes(Population a_pop) {
        // Check that all chromosomes are unique
        for (int i = 0; i < a_pop.size() - 1; i++) {
            IChromosome c = a_pop.getChromosome(i);
            for (int j = i + 1; j < a_pop.size(); j++) {
                IChromosome c2 = a_pop.getChromosome(j);
                if (c == c2) {
                    return false;
                }
            }
        }
        return true;
    }
}
```

B.4 SimEvent

B.4.1 Car Arriving At Scene Event

```
// CarArriveSceneEvent.java

package SimEvent;

import Entities.Car;
import org.apache.commons.math3.distribution.ExponentialDistribution;

public class CarArriveSceneEvent extends Event {

    public CarArriveSceneEvent(int eventOccurTime) {
        super(eventOccurTime);
    }

    public CarArriveSceneEvent(int eventOccurTime, Car car) {
        super(eventOccurTime, car);
    }

    @Override
```

```
	public void actions() {

		// schedule CarLeaveSceneEvent
//		int timeOnScene = (int) MyRandom.normal(TIME_ON_SCENE_MEAN,TIME_ON_SCENE_STD);
		int timeOnScene = (int) new
		    ExponentialDistribution(RunSim.SimRun.TIME_ON_SCENE_MEAN).sample();
		eventList.add(new CarLeaveSceneEvent(this.getEventOccurTime()+ timeOnScene,
		    this.getCar()));
		//set closeTime for incident carried by the car
		Car currCar = this.getCar();
		currCar.getCfsIncident().setLeaveScnTime(this.getEventOccurTime()+
		    timeOnScene);
		//display
		if (DISPLAY_CONSOLE_INFO)displayInConsole(timeOnScene);
		if(DISPLAY_CONSOLE_INFO)
		    System.out.println("###########"+currCar.getCfsIncident());
		//update oneServerServiceTimeList
		int tempDispatchTime = currCar.getCfsIncident().getDispatchTime() -
		    currCar.getCfsIncident().getCreateTime();
		int tempTravelTime = currCar.getCfsIncident().getCarArrTime() -
		    currCar.getCfsIncident().getDispatchTime();
	}

	private void displayInConsole(int timeOnScene) {
		System.out.println("At time:" + this.getEventOccurTime()
				+ ", CarArriveSceneEvent occurs, carID:"
				+ this.getCar().getID());
		System.out.println("\tSchedule a CarLeaveSceneEvent for car "
				+ this.getCar().getID() + " at time: "
				+ (this.getEventOccurTime() + timeOnScene));
	}

	@Override
	public String toString() {
		return super.toString() + " CarArriveSceneEvent";
	}

}
```

B.4.2 Car Leaving Scene Event

```
// CarLeaveSceneEvent.java
package SimEvent;

import java.text.DecimalFormat;

import Entities.Car;
import Entities.Coordinate;

public class CarLeaveSceneEvent extends Event {

	public CarLeaveSceneEvent(int eventOccurTime) {
		super(eventOccurTime);
	}

	public CarLeaveSceneEvent(int eventOccurTime, Car car) {
		super(eventOccurTime, car);
	}

	@Override
	public void actions() {

		// schedule a CarRetrunToBaseEvent
		Coordinate taskCoor = this.getCar().getTaskLocation();
```

```
        Coordinate baseCoor = this.getCar().getBaseLocation();
        double distance = Coordinate.calDistTwoPoints(taskCoor, baseCoor);
        double carSpeed = RunSim.SimRun.CAR_SPEED_MEAN;
        int travelTime = (int) (distance / carSpeed);
        eventList.add(new CarRetrunToBaseEvent(this.getEventOccurTime()+ travelTime,
            this.getCar()));
        if (DISPLAY_CONSOLE_INFO) displayInConsole(taskCoor, baseCoor, distance,
            carSpeed, travelTime);
    }

    private void displayInConsole(Coordinate taskCoor, Coordinate baseCoor,
            double distance, double carSpeed, int travelTime) {
        System.out.println("At time:" + this.getEventOccurTime()+ ",
            CarLeaveSceneEvent occurs, carID:"+ this.getCar().getID());
        displayTravelPlanInformation(taskCoor, baseCoor, distance, travelTime,
                carSpeed);
        System.out.println("\tSchedule a CarRetrunToBaseEvent for car "
                + this.getCar().getID() + " at time: "
                + (this.getEventOccurTime() + travelTime));
    }

    private void displayTravelPlanInformation(Coordinate taskCoor,
            Coordinate baseCoor, double distance, int travelTime,
            double carSpeed) {
        DecimalFormat df = new DecimalFormat();
        df.setMaximumFractionDigits(3);
        df.setMinimumFractionDigits(3);
        System.out.println("\tCurrentLocation: " + taskCoor);
        System.out.println("\tBaseLocation: " + baseCoor);
        System.out.print("\tTravelDistance = " + df.format(distance));
        System.out.print(" TravelSpeed = " + df.format(carSpeed));
        System.out.println(" TravelTime = " + travelTime);
    }

    @Override
    public String toString() {
        return super.toString() + " CarLeaveSceneEvent";
    }

}
```

B.4.3 Car Returning to Base Event

```
// CarRetrunToBaseEvent.java
package SimEvent;

import static SimEvent.Utils.*;

import java.util.ArrayList;
import Entities.Car;


public class CarRetrunToBaseEvent extends Event {

    public CarRetrunToBaseEvent(int eventOccurTime) {
        super(eventOccurTime);
    }

    public CarRetrunToBaseEvent(int eventOccurTime, Car car) {
        super(eventOccurTime, car);
    }

    @Override
    public void actions() {
        // change the status of the car
        if(DISPLAY_CONSOLE_INFO) displayBeforeReturn();
        this.getCar().setStatus(0);
        if(DISPLAY_CONSOLE_INFO) displayCarStatusArray("After returning to base: ");

        //since there is idle car, try dispatch
        tryDispatchQueueIncidentsToCars(this);
//          writeOneStrategyStatsLineToFile();

    }

    private void displayBeforeReturn() {
        System.out.println("At time:" + this.getEventOccurTime()
                    + ", CarRetrunToBaseEvent occurs, carID:"
                    + this.getCar().getID());
        displayCarStatusArray("Before returning to base: ");
    }

    @Override
    public String toString() {
        return super.toString() + " CarRetrunToBaseEvent";
    }

    private void displayCarStatusArray(String str) {
        System.out.print("\t" + str + "Car availability: CarID");
        for (int i = 0; i < carsArrayList.size(); i++) {
              System.out.print("[" + i + "]");
        }
        System.out.print(" :");
        for (Car car : carsArrayList) {
              String statusStr = (car.getStatus() == 1) ? "1 " : "0 ";
              System.out.print(statusStr);
        }
        System.out.println("");
    }
}
```

B.4.4 CFS Incident Arrival Event

```
// CFS_ArrivalEvent.java

package SimEvent;

import Entities.CFSincident;
import static SimEvent.Utils.*;
import org.apache.commons.math3.distribution.ExponentialDistribution;


public class CFS_ArrivalEvent extends Event implements RunSim.SimParameters {

      public CFS_ArrivalEvent(int eventOccurTime) {
            super(eventOccurTime);
      }

      //use queue to store CFS incidents
      @Override
      public void actions() {
            //generate CFS incident
            CFSincident newIncident = generateNewIncident();
            //try to put the incident into the queue, if the queue is full, drop the
                incident, make a record
            tryPutIncidentIntoQueue(newIncident);
            //dispatch incidents in queue to cars{all cars are busy; dispatch one car;
                dispatch multiple cars}
            tryDispatchQueueIncidentsToCars(this); //code is in Utils.java
            // schedule the next arrival event
            scheduleNextCfsArrivalEvent();
      }

      private CFSincident generateNewIncident() {
            CFSincident newIncident = new CFSincident();
            newIncident.setCreateTime(this.getEventOccurTime());
            // randomly generate the atom where CFS_ArrivalEvent occurs based on
                CFS_PROB_EACH_ATOM
            int atomNum = MyRandom.selectOneEleBasedOnProbList(CFS_PROB_EACH_ATOM);
            newIncident.setAtomNum(atomNum);
            newIncident.setDistrictNum(atomsArrayList.get(atomNum).getDistrictNum());
            newIncident.setCoor(atomsArrayList.get(atomNum).getCoor());
            if(DISPLAY_CONSOLE_INFO) System.out.println("\nAt time:" +
                this.getEventOccurTime()+ ", CFS event generated at Atom:" +
                newIncident.getAtomNum());
            return newIncident;
      }

      private void tryPutIncidentIntoQueue(CFSincident newIncident) {
            boolean enQueSuccessFlag = RunSim.SimRun.cfsQueue.enQue(newIncident);
            if(enQueSuccessFlag == false){//queue is full
                  newIncident.setIfDropped(1);
                  if(DISPLAY_CONSOLE_INFO) System.out.println("**********incident dropped
                      because CFS queue is full!");
                  if(DISPLAY_CONSOLE_INFO) System.out.println("@@@@@@@@@@"+newIncident);
            }else{
                  if(DISPLAY_CONSOLE_INFO) System.out.println("**********incident is
                      stored in the queue.");
            }
      }

      private void scheduleNextCfsArrivalEvent() {
            //exponential distribution
            int cfsInterArrivalTime = (int) new
                ExponentialDistribution(RunSim.SimRun.CFS_INTER_ARRIVAL_TIME_MEAN).sample();
            eventList.add(new CFS_ArrivalEvent(this.getEventOccurTime()+
                cfsInterArrivalTime));
            //display in console
            if (DISPLAY_CONSOLE_INFO)
                  System.out.println("\tSchedule the CFS_ArrivalEvent at time: " +
                      (this.getEventOccurTime() + cfsInterArrivalTime));
      }

      @Override
      public String toString() {
            return super.toString() + " CFS_ArrivalEvent";
      }
}
```

B.4.5 Event

```
// Event.java
package SimEvent;

import java.util.ArrayList;

import Entities.Car;

public abstract class Event implements Comparable, RunSim.SimParameters {
      private int eventOccurTime;
```

```
	private Car car;

	public Event(int eventOccurTime) {
		super();
		this.eventOccurTime = eventOccurTime;
	}

	public Event(int eventOccurTime, Car car) {
		super();
		this.eventOccurTime = eventOccurTime;
		this.car = car;
	}

	public int getEventOccurTime() {
		return eventOccurTime;
	}

	public void setEventOccurTime(int eventOccurTime) {
		this.eventOccurTime = eventOccurTime;
	}

	public Car getCar() {
		return car;
	}

	public void setCar(Car car) {
		this.car = car;
	}

	public abstract void actions();

	@Override
	public int compareTo(Object o) {
		Event e = (Event) o;
		if (this.getEventOccurTime() < e.getEventOccurTime())
			return -1;
		else if (this.getEventOccurTime() > e.getEventOccurTime())
			return 1;
		else
			return 0;
	}

	@Override
	public String toString() {
		return "Time=" + eventOccurTime;
	}
}
```

B.4.6 Event List

```
// EventList.java

package SimEvent;

import java.util.ArrayList;
import java.util.Collections;

public class EventList implements RunSim.SimParameters {

	private ArrayList<Event> eventArrayList = new ArrayList<Event>();

	public EventList() {
	}
```

```
    public void add(Event event) {
        eventArrayList.add(event);
        Collections.sort(eventArrayList);
    }

    public Event nextEvent() {
        return eventArrayList.get(0);
    }

    public void delete() {
        eventArrayList.remove(0);
    }

    public void clear(){
        eventArrayList.clear();
    }

    @Override
    public String toString() {
        String str = "\n\tCurrent EventList\n";
        for (Event e : eventArrayList) {
            str += "\t\t" + e.toString() + "\n";
        }
        return str;
    }
}
```

B.4.7 My Random Functions

```
// MyRandom.java
package SimEvent;

import java.util.ArrayList;
import java.util.Random;

public class MyRandom {
    public static int selectOneEleBasedOnProbList(double[] probList) {
        ArrayList<Interval> intervalArrList = new ArrayList<Interval>();

        intervalArrList.add(new Interval(0, probList[0]));
        double lb = probList[0];
        for (int i = 1; i < probList.length - 1; i++) {
            double up = lb + probList[i];
            intervalArrList.add(new Interval(lb, up));
            lb = up;
        }
        intervalArrList.add(new Interval(lb, 1));
        // System.out.println(intervalArrList);
        Double randomNum = Math.random();
        // System.out.println("randomNum = " + randomNum);
        int index = 0;
        for (int i = 0; i < intervalArrList.size(); i++) {
            if (randomNum > intervalArrList.get(i).getLowerBound()
                    && randomNum < intervalArrList.get(i).getUpperBound()) {
                index = i;
            }
        }
        // System.out.println("index = " + index);
        return index;
    }

    public static double normal(double a, double b) {
        Random randOjb = new Random();
        return a + b * randOjb.nextGaussian();
```

```
	}

	public static int randomInt(int n){ //from 0 to n -1
		Random generator = new Random();
		return generator.nextInt( n );
	}
}

class Interval {
	double lowerBound;
	double upperBound;

	public Interval(double lowerBound, double upperBound) {
		super();
		this.lowerBound = lowerBound;
		this.upperBound = upperBound;
	}

	public double getLowerBound() {
		return lowerBound;
	}

	public void setLowerBound(double lowerBound) {
		this.lowerBound = lowerBound;
	}

	public double getUpperBound() {
		return upperBound;
	}

	public void setUpperBound(double upperBound) {
		this.upperBound = upperBound;
	}

	@Override
	public String toString() {
		return "[" + lowerBound + ", " + upperBound + "]";
	}
}
```

B.4.8 Utility

```
// Utils.java

package SimEvent;

import static SimEvent.Utils.displayCarStatusArray;

import java.text.DecimalFormat;
import java.util.ArrayList;

import Entities.Atom;
import Entities.CFSincident;
import Entities.Car;
import Entities.Coordinate;
import Entities.District;

public class Utils implements RunSim.SimParameters{

	public static void tryDispatchQueueIncidentsToCars(Event event) {
		if(DISPLAY_CONSOLE_INFO) System.out.print("**********Before dispatch, current
			queueLen: " + RunSim.SimRun.cfsQueue.getQueLen() + ", IdleCarNum: " +
			getIdleCarNum()+ ", ");
```

```
        if(DISPLAY_CONSOLE_INFO) System.out.println(" Queue: " +
            RunSim.SimRun.cfsQueue.toString());
        if(RunSim.SimRun.PATROL_DISPATCH_RULE == 3){ // no-cross boundary cases, case
            3
              //idea: loop through all incidents in queue, first deque, if
                  corresponding car is busy, enque the incident, if idle, dispatch
              int currQueLen = RunSim.SimRun.cfsQueue.getQueLen(); // in case
                  cfsQueue.getQueLen() changes inside the loop
              for (int i = 0; i < currQueLen; i++) {
                    CFSincident incident = RunSim.SimRun.cfsQueue.deQue();
                    Car responsibleCar = carsArrayList.get(incident.getDistrictNum());
                    if(responsibleCar.getStatus() == 1){ // busy
                          //put incident back to queue, enque
                          RunSim.SimRun.cfsQueue.enQue(incident);
                          if(DISPLAY_CONSOLE_INFO) System.out.println("**********car
                              " + responsibleCar.getID()+ " is busy at this time");
                    }else{ // idle
                          dispatchOneIncidentToOneCar3(event, incident);
                    }
              }
        }else{// case 1, 2
              while(!RunSim.SimRun.cfsQueue.isQueEmpty() && getIdleCarNum()!= 0){
                    if(DISPLAY_CONSOLE_INFO) System.out.println("**********start of
                        while iteration");
                    //get incident from queue
                    CFSincident incident = RunSim.SimRun.cfsQueue.deQue();
                    // dispatch a car to incident based on selected rules

                    // dispatch a car to incident
                    if(RunSim.SimRun.PATROL_DISPATCH_RULE == 1){
                          dispatchOneIncidentToOneCar1(event, incident);
                    }else if (RunSim.SimRun.PATROL_DISPATCH_RULE == 2){
                          dispatchOneIncidentToOneCar2(event, incident);
                    }
                    if(DISPLAY_CONSOLE_INFO) System.out.println("**********end of
                        while iteration");
              }//end of while
        }
        if(DISPLAY_CONSOLE_INFO) System.out.print("**********After dispatch, current
            queueLen: " + RunSim.SimRun.cfsQueue.getQueLen() + ", IdleCarNum: " +
            getIdleCarNum() + ", ");
        if(DISPLAY_CONSOLE_INFO) System.out.println(" Queue: " +
            RunSim.SimRun.cfsQueue.toString());
}

private static void dispatchOneIncidentToOneCar3(Event event, CFSincident incident)
     {
        int atomNum = incident.getAtomNum();
        if(DISPLAY_CONSOLE_INFO) displayInfoBeforeDispatch(atomNum);
        // change car status
        Car preferCar = carsArrayList.get(incident.getDistrictNum());
        preferCar.setStatus(1);
        preferCar.addOneTaskNum();
        event.setCar(preferCar);
        // schedule a CarArriveSceneEvent

        Coordinate atomCoor = atomsArrayList.get(atomNum).getCoor();
        preferCar.setTaskLocation(atomCoor);// store the task location in car, for
            future use

        double distance = Coordinate.calDistTwoPoints(preferCar.getBaseLocation(),
            atomCoor);
        double carSpeed = RunSim.SimRun.CAR_SPEED_MEAN;
        int travelTime = (int) (distance / carSpeed);
        //record information
        responseTimeArrayList.add((double)(event.getEventOccurTime()+ travelTime -
            incident.getCreateTime()));
```

```
      eventList.add(new CarArriveSceneEvent(event.getEventOccurTime()+ travelTime,
          preferCar));

      //update CFSincident information
      incident.setCarNum(preferCar.getID());
      incident.setDispatchTime(event.getEventOccurTime());
      incident.setCarArrTime(event.getEventOccurTime()+ travelTime);

      // display info in console
      if (DISPLAY_CONSOLE_INFO)displayInfoAfterDispatch(event, atomNum, preferCar,
          distance, carSpeed, travelTime);

      //record cross-boundary information
      if(preferCar.getID() == atomsArrayList.get(atomNum).getDistrictNum()){
            incident.setCrsBdrResp(0);
      }else{
            incident.setCrsBdrResp(1);
      }
      if(DISPLAY_CONSOLE_INFO) System.out.println("~~~~~~~~~~"+incident);

      //let car carry the incident information
      preferCar.setCfsIncident(incident);
}

public static void dispatchOneIncidentToOneCar1(Event event, CFSincident incident) {
      //assumption: cars stay at district centers
      int atomNum = incident.getAtomNum();
      if(DISPLAY_CONSOLE_INFO) displayInfoBeforeDispatch(atomNum);
      ArrayList<Integer> respCarPreferList =
          atomsArrayList.get(atomNum).getRespCarPreferList();
      for (int preferCarIndex : respCarPreferList) {
            Car preferCar = carsArrayList.get(preferCarIndex);
            if (preferCar.getStatus() == 0) { //find the first available car in list
                  // change car status
                  preferCar.setStatus(1);
                  preferCar.addOneTaskNum();
                  event.setCar(preferCar);
                  // schedule a CarArriveSceneEvent
                  Coordinate carCoor = preferCar.getBaseLocation();
                  Coordinate atomCoor = atomsArrayList.get(atomNum).getCoor();
                  preferCar.setTaskLocation(atomCoor);// store the task location in
                      car, for future use
                  double distance = Coordinate.calDistTwoPoints(carCoor, atomCoor);
                  double carSpeed = RunSim.SimRun.CAR_SPEED_MEAN;
                  int travelTime = (int) (distance / carSpeed);
                  //record information
                  responseTimeArrayList.add((double)(event.getEventOccurTime()+
                      travelTime - incident.getCreateTime()));

                  eventList.add(new CarArriveSceneEvent(event.getEventOccurTime()+
                      travelTime, preferCar));

                  //update CFSincident information
                  incident.setCarNum(preferCar.getID());
                  incident.setDispatchTime(event.getEventOccurTime());
                  incident.setCarArrTime(event.getEventOccurTime()+ travelTime);

                  //display info in console
                  if(DISPLAY_CONSOLE_INFO) displayInfoAfterDispatch(event, atomNum,
                      preferCar, distance, carSpeed, travelTime);

                  //record cross-boundary information
                  if(preferCar.getID() ==
                      atomsArrayList.get(atomNum).getDistrictNum()){
                        incident.setCrsBdrResp(0);
                  }else{
                        incident.setCrsBdrResp(1);
                  }
```

```
                    if(DISPLAY_CONSOLE_INFO)
                        System.out.println("~~~~~~~~~~~"+incident);
                    //let car carry the incident information
                    preferCar.setCfsIncident(incident);

                    break;
                }
            }
        }

        public static void dispatchOneIncidentToOneCar2(Event event, CFSincident incident) {
            //assumption: randomly generate cars location when incident comes
            ArrayList<Car> availableCarsList = new ArrayList<Car>();
            for (Car car : carsArrayList) {
                if (car.getStatus() == 0)
                    availableCarsList.add(car);
            }
            // randomize a temp location in its district for car in availableCarsList
            // find carID which is nearest to the incidentAtom
            double shortestDistance = 99999999;
            int nearestCarID = -1;
            for (Car car : availableCarsList) {
                int distNum = car.getDistrictNum();
                District district = districtsArrayList.get(distNum);
                ArrayList<Atom> districtAtomsArrayList =
                    district.getDistrictAtomsArrayList();
                int len = districtAtomsArrayList.size();
                int randomIndex = MyRandom.randomInt(len);
                Atom carAtAtom = districtAtomsArrayList.get(randomIndex);
                // calculate distance
                Coordinate carCoor = carAtAtom.getCoor();
                Coordinate incidCoor = incident.getCoor();
                Double distance = Coordinate.calDistTwoPoints(carCoor, incidCoor);
                if (distance < shortestDistance) {
                    shortestDistance = distance;
                    nearestCarID = car.getID();
                }
            }

            // change car status
            Car preferCar = carsArrayList.get(nearestCarID);
            preferCar.setStatus(1);
            preferCar.addOneTaskNum();
            event.setCar(preferCar);
            // schedule a CarArriveSceneEvent
            int atomNum = incident.getAtomNum();
            Coordinate atomCoor = atomsArrayList.get(atomNum).getCoor();
            preferCar.setTaskLocation(atomCoor);// store the task location in car, for
                future use
            double distance = shortestDistance;
            double carSpeed = RunSim.SimRun.CAR_SPEED_MEAN;
            int travelTime = (int) (distance / carSpeed);
            //record information
            responseTimeArrayList.add((double)(event.getEventOccurTime()+ travelTime -
                incident.getCreateTime()));

            eventList.add(new CarArriveSceneEvent(event.getEventOccurTime()+ travelTime,
                preferCar));

            //update CFSincident information
            incident.setCarNum(preferCar.getID());
            incident.setDispatchTime(event.getEventOccurTime());
            incident.setCarArrTime(event.getEventOccurTime()+ travelTime);

            // display info in console
            if (DISPLAY_CONSOLE_INFO)displayInfoAfterDispatch(event, atomNum, preferCar,
                distance, carSpeed, travelTime);
```

```
            //record cross-boundary information
            if(preferCar.getID() == atomsArrayList.get(atomNum).getDistrictNum()){
                  incident.setCrsBdrResp(0);
            }else{
                  incident.setCrsBdrResp(1);
            }
            if(DISPLAY_CONSOLE_INFO) System.out.println("~~~~~~~~~~~"+incident);

            //let car carry the incident information
            preferCar.setCfsIncident(incident);
      }


      public static int getIdleCarNum(){
            int idleCarNum = 0;
            for(Car car: carsArrayList){
                  if(car.getStatus() == 0){
                        idleCarNum++;
                  }
            }
            return idleCarNum;
      }

      public static void displayInfoBeforeDispatch(int atomNum) {
            ArrayList<Integer> respCarPreferList =
                atomsArrayList.get(atomNum).getRespCarPreferList();
            System.out.println("\trespCarPreferList (carID)" + respCarPreferList);
            displayCarStatusArray("Before Dispatch: ");
      }

      public static void displayCarStatusArray(String str) {
            System.out.print("\t" + str + "Car availability: CarID");
            for (int i = 0; i < carsArrayList.size(); i++) {
                  System.out.print("[" + i + "]");
            }
            System.out.print(" :");
            for (Car car : carsArrayList) {
                  String statusStr = (car.getStatus() == 1) ? "1 " : "0 ";
                  System.out.print(statusStr);
            }
            System.out.println("");
      }

      public static void displayInfoAfterDispatch(Event event, int atomNum, Car preferCar,
                  double distance, double carSpeed, int travelTime) {
            System.out.println("\tDispatch car " + preferCar.getID()
                        + " to atom " + atomNum);
            displayCarStatusArray("After Dispatch: ");
            displayTravelPlanInformation(preferCar, distance, travelTime,
                        carSpeed);
            System.out.println("\tSchedule a CarArriveSceneEvent for car "
                        + preferCar.getID() + " at time: "
                        + (event.getEventOccurTime() + travelTime));
      }

      public static void displayTravelPlanInformation(Car preferCar, double distance,
                  int travelTime, double carSpeed) {
            DecimalFormat df = new DecimalFormat();
            df.setMaximumFractionDigits(3);
            df.setMinimumFractionDigits(3);
            System.out.println("\tBaseLocation: " + preferCar.getBaseLocation());
            System.out.println("\tTaskLocation: " + preferCar.getTaskLocation());
            System.out.print("\tTravelDistance = " + df.format(distance));
            System.out.print(" TravelSpeed = " + df.format(carSpeed));
            System.out.println(" TravelTime = " + travelTime);
      }
}
```

Bibliography

[1] Federal emergency management agency. http://www.fema.gov/about.

[2] National highway traffic safety administration. http://www.nhtsa.gov/.

[3] Official website of e-handbook of statistical methods, nist/sematech. http://www.itl.nist.gov/div898/handbook/.

[4] Official website of java repast. http://repast.sourceforge.net/.

[5] Official website of state of rhode island, department of environmental management. http://www.dem.ri.gov/index.htm.

[6] *JMP 9 Design of Experiments Guide*. SAS Publishing, 2010.

[7] A.A.Aly and D.W.Litwhiler. Police briefing stations: a location problem. *AIIE Transactions (0569-5554)*, 11(1):12–22, 1979.

[8] Emile Aarts and Jan Korst. Simulated annealing and boltzmann machines: a stochastic approach to combinatorial optimization and neural computing. 1988.

[9] M. Altman. The computational complexity of automated redistricting: Is automation the answer? *Rutgers Computer and Technology Law Journal*, 23(1):81–142, 01/01 1997.

[10] Sigrún Andradóttir. A method for discrete stochastic optimization. *Management Science*, 41(12):1946–1961, 1995.

[11] Sigrún Andradóttir. A global search method for discrete stochastic optimization. *SIAM Journal on Optimization*, 6(2):513–530, 1996.

[12] Sigrún Andradóttir. A review of simulation optimization techniques. In *Proceedings of the 30th conference on Winter simulation*, pages 151–158. IEEE Computer Society Press, 1998.

[13] Jay April, Fred Glover, James P Kelly, and Manuel Laguna. Practical introduction to simulation optimization. In *Simulation Conference, 2003. Proceedings of the 2003 Winter*, volume 1, pages 71–78. IEEE, 2003.

[14] Farhad Azadivar. A tutorial on simulation optimization. In *Proceedings of the 24th conference on Winter simulation*, pages 198–204. ACM, 1992.

[15] Farhad Azadivar. Simulation optimization methodologies. In *Proceedings of the 31st conference on Winter simulation: Simulation—a bridge to the future-Volume 1*, pages 93–100. ACM, 1999.

[16] Fernando Bação, Victor Lobo, and Marco Painho. Applying genetic algorithms to zone design. *Soft Computing*, 9(5):341–348, 2005.

[17] M. A. Badri, A. K. Mortagy, and C. A. Alsayed. A multi-objective model for locating fire stations. *European Journal of Operational Research*, 110:243–260, 1998.

[18] M. O. Ball and L. F. Lin. A reliability model applied to emergency service vehicle location. *Operations Research*, 41:18–36, 1993.

[19] William Ernest Biles. A gradientregression search procedure for simulation experimentation. In *Proceedings of the 7th conference on Winter simulation-Volume 2*, pages 491–497. Winter Simulation Conference, 1974.

[20] Marko Blais, Sophie D Lapierre, and Gilbert Laporte. Solving a home-care districting problem in an urban setting. *Journal of the Operational Research Society*, 54(11):1141–1147, 2003.

[21] George E. P. Box, J. S. Hunter, and William Gordon Hunter. *Statistics for experimenters : design, innovation, and discovery.* Wiley-Interscience, 2005.

[22] Burak Boyaci and Nikolas Geroliminis. Extended hypercube models for large scale spatial queuing systems. Technical report, Ecole Polytechnique Federale de Lausanne, 2011.

[23] Burcin Bozkaya, Erhan Erkut, and Gilbert Laporte. A tabu search heuristic and adaptive memory procedure for political districting. *European Journal of Operational Research*, 144(1):12–26, 2003.

[24] L. Brotcorne, G. Laporte, and F. Semet. Ambulance location and relocation models. *European Journal of Operational Research*, 147:451–463, 2003.

[25] Felipe Caro, Takeshi Shirabe, M Guignard, and A Weintraub. School redistricting: embedding gis tools with integer programming. *Journal of the Operational Research Society*, 55(8):836–849, 2004.

[26] Y. Carson and A. Maria. Simulation optimization: methods and applications. In *Proceedings of the 1997 Winter Simulation Conference*, pages 118–126, 1997.

[27] VLADIMÍR Černỳ. Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm. *Journal of optimization theory and applications*, 45(1):41–51, 1985.

[28] J. M. Chaiken and W. E. Walker. *Patrol car allocation model: users manual.* Rand Corporation Publication, 1989.

[29] Chung-I Chou. A knowledge-based evolution algorithm approach to political districting problem. *Computer Physics Communications*, 182(1):209–212, 2011.

[30] Chung-I Chou, You-ling Chu, and Sai-Ping Li. Evolutionary strategy for political districting problem using genetic algorithm. In *Computational Science–ICCS 2007*, pages 1163–1166. Springer, 2007.

[31] G. W. Cordner and Kathryn E. Scarborough. *Police Administration*. Anderson; 7 edition, 02/26 2010.

[32] Kevin M. Curtin, Karen Hayslett-McCall, and Fang Qiu. Determining optimal police patrol areas with maximal covering and backup covering location models. *Networks and Spatial Economics*, 10(1):125–145, 2010.

[33] Steven J. D'Amico, Shoou-Jiun Wang, Rajan Batta, and Christopher M. Rump. A simulated annealing approach to police district design. *Computers and Operations Research*, 29(6):667–684, 05/01 2002.

[34] Andrew F Daughety and Mark A Turnquist. Simulation optimization using response surfaces based on spline approximations. In *Proceedings of the 10th conference on Winter simulation-Volume 1*, pages 183–193. IEEE Press, 1978.

[35] Lawrence Davis et al. *Handbook of genetic algorithms*, volume 115. Van Nostrand Reinhold New York, 1991.

[36] Joan M Donohue, Ernest C Houck, and Raymond H Myers. Some optimal simulation designs for estimating quadratic response surface functions. In *Proceedings of the 22nd conference on Winter simulation*, pages 337–343. IEEE Press, 1990.

[37] Andreas Drexl and Knut Haase. Fast approximation methods for sales force deployment. *Management Science*, 45(10):1307–1323, 1999.

[38] R. A. Fisher. The arrangement of field experiments. *Journal of the Ministry of Agriculture of Great Britain*, 33:503–513, 1926.

[39] Mark Fleischer. Simulated annealing: past, present, and future. In *Proceedings of the 27th conference on Winter simulation*, pages 155–161. IEEE Computer Society, 1995.

[40] Bernhard Fleischmann and Jannis N Paraschis. Solving a large scale districting problem: a case report. *Computers & Operations Research*, 15(6):521–533, 1988.

[41] Edward Forrest. Apportionment by computer. *American behavioral scientist*, 8(4):23–23, 1964.

[42] Michael C Fu, Fred W Glover, and Jay April. Simulation optimization: a review, new developments, and applications. In *Proceedings of the 37th conference on Winter simulation*, pages 83–95. Winter Simulation Conference, 2005.

[43] Saul Gass and Thomas Saaty. The computation algorithm for the parametric objective function. *Naval Research Logistics Quarterly*, 2(1), 1955.

[44] M. Gendreau, G. Laporte, and F. Semet. Solving an ambulance location model by tabu search. *Location Science*, 5(2):75–88, 1997.

[45] M. Gendreau, G. Laporte, and F. Semet. A dynamic model and parallel tabu search heuristic for real-time ambulance relocation. *Parallel Computing*, 27:1641–1653, 2001.

[46] M. Gendreau, G. Laporte, and F. Semet. The maximal expected coverage relocation problem for emergency vehicles. *The Journal of the Operational Research Society*, 57(1):22–28, 2006.

[47] John A George, Bruce W Lamar, and Chris A Wallace. Political district determination using large-scale network optimization. *Socio-Economic Planning Sciences*, 31(1):11–28, 1997.

[48] Peter W Glynn. Likelihood ratio derivative estimators for stochastic systems. Technical report, DTIC Document, 1989.

[49] William L Goffe, Gary D Ferrier, and John Rogers. Global optimization of statistical functions with simulated annealing. *Journal of Econometrics*, 60(1):65–99, 1994.

[50] David Edward Goldberg et al. *Genetic algorithms in search, optimization, and machine learning*, volume 412. Addison-wesley Reading Menlo Park, 1989.

[51] RG González-Ramírez, NR Smith, RG Askin, Pablo A Miranda, and JM Sánchez. A hybrid metaheuristic approach to optimize the districting design of a parcel company. *Journal of applied research and technology*, 9(1):19–35, 2011.

[52] Shanti S Gupta and Subramanian Panchapakesan. *Multiple decision procedures: theory and methodology of selecting and ranking populations*, volume 44. Siam, 1979.

[53] Bruce Hajek. A tutorial survey of theory and applications of simulated annealing. In *Decision and Control, 1985 24th IEEE Conference on*, volume 24, pages 755–760. IEEE, 1985.

[54] Charles D. Hale. *Police patrol, operations and management.* Prentice Hall, Upper Saddle River, New Jersey 07458, 01/01 1980.

[55] P. G. Hancock and N. C. Simpson. Fifty years of operational research and emergency response. *Journal of the Operational Research Society*, 60(51):126–139, 2009.

[56] Keith Harris. Mapping crime: Principle and practice. Technical report, National Institute of Justice, 1999.

[57] Sidney Wayne Hess, JB Weaver, HJ Siegfeldt, JN Whelan, and PA Zitlau. Nonpartisan political redistricting by computer. *Operations Research*, 13(6):998–1006, 1965.

[58] J. Hogg. The sitting of fire stations. *Operational Research Quarterly*, 19(1):275–287, 1968.

[59] Jason C Hsu and Barry L Nelson. Optimization over a finite number of system designs with one-stage sampling and multiple comparisons with the best. In *Proceedings of the 20th conference on Winter simulation*, pages 451–457. ACM, 1988.

[60] Samuel H. Huddleston and Donald E. Brown. A statistical threat assessment. *IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans*, 39(6), 2009.

[61] Samuel H. Huddleston and Donald E. Brown. Using discrete event simulation to evaluate time series forecasting methods for security applications. In *Proceedings of the 2013 Winter Simulation Conference*, 2013.

[62] D. S. Johnson. The np-completeness column: an ongoing guide. *Journal of Algorithms*, 3(2):182–195, 09/01 1982.

[63] D. Joshi, Leen-Kiat Soh, and A. Samal. Redistricting using heuristic-based polygonal clustering. In *Data Mining, 2009.ICDM '09.Ninth IEEE International Conference*, pages 830–835, 2009.

[64] J Kalcsics, S Nickel, and M Schröder. *A generic geometric approach to territory design and districting.* Fraunhofer-Institut für Techno-und Wirtschaftsmathematik, Fraunhofer (ITWM), 2009.

[65] Jörg Kalcsics, Stefan Nickel, and Michael Schröder. Towards a unified territorial design approachapplications, algorithms and gis integration. *Top*, 13(1):1–56, 2005.

[66] Scott Kirkpatrick. Optimization by simulated annealing: Quantitative studies. *Journal of statistical physics*, 34(5-6):975–986, 1984.

[67] J. P. Kleijnen. Kriging for interpolation in random simulation. *Journal of the Operational Research Society*, 54(3):255–262, 2003.

[68] J. P. Kleijnen. Kriging metamodeling in simulation: A review. *European Journal of Operational Research*, 192(3):707–716, 2009.

[69] Jack P. C. Kleijnen. *Design and analysis of simulation experiments.* Springer, 12/25 2010.

[70] B. A. Knoppers and H. F. Miller. Computer simulation of police dispatching and patrol functions. In *International symposium on criminal justice information and statistics systems proceedings*, 1972.

[71] Richard C. Larson. Measuring the response patterns of new york city police patrol cars. *Rand Corporation*, page 73, 07/01 1971.

[72] Richard C. Larson. A hypercube queuing model for facility location and redistricting in urban emergency services. *Computers & Operations Research*, 1(1), 1974.

[73] A. Law. *Simulation Modeling and Analysis.* Mcgraw Hill Higher Education, 08/01 2006.

[74] Zhenping Li, Rui-Sheng Wang, and Yong Wang. A quadratic programming model for political districting problem. In *Proceedings of the firsst international symposium on optimization and system biology (OSB). Bejing, China*, 2007.

[75] Nick Malleson. *Agent-Based Modelling of Burglary.* PhD thesis, School of Geography, University of Leeds, 2010.

[76] G. Matheron. Principles of geostatistics. *Economic Geology*, 58(8):1246C1266, 1963.

[77] Klaus Meffert, N Rotstan, C Knowles, and U Sangiorgi. Jgap-java genetic algorithms and genetic programming package. *URL: http://jgap. sf. net*, 2012.

[78] Anuj Mehrotra, Ellis L Johnson, and George L Nemhauser. An optimization based heuristic for political districting. *Management Science*, 44(8):1100–1114, 1998.

[79] R Minciardi, PP Puliafito, and R Zoppoli. A districting procedure for social organizations. *European Journal of Operational Research*, 8(1):47–57, 1981.

[80] Phillip S. Mitchell. Optimal selection of police patrol beats. *The Journal of Criminal Law*, 63(4):577–584, 1972.

[81] D. C. Montgomery. *Design and Analysis of Experiments.* Wiley; 7 edition, 07/28 2008.

[82] Douglas J Morrice and Lee W Schruben. Simulation sensitivity analysis using frequency domain experiments. In *Proceedings of the 21st conference on Winter simulation*, pages 367–373. ACM, 1989.

[83] Antonio GN Novaes, JE Souza de Cursi, Arinei CL da Silva, and João C Souza. Solving continuous location–districting problems with voronoi diagrams. *Computers & Operations Research*, 36(1):40–59, 2009.

[84] Ibrahim Hassan Osman. Metastrategy simulated annealing and tabu search algorithms for the vehicle routing problem. *Annals of Operations Research*, 41(4):421–451, 1993.

[85] Federica Ricca, Andrea Scozzari, and Bruno Simeone. Weighted voronoi region algorithms for political districting. *Mathematical and Computer Modelling*, 48(9):1468–1477, 2008.

[86] Roger Z Ríos-Mercado and Elena Fernández. A reactive grasp for a commercial territory design problem with multiple balancing requirements. *Computers & Operations Research*, 36(3):755–776, 2009.

[87] Roger Z Ríos-Mercado and Juan C Salazar-Acosta. A grasp with strategic oscillation for a commercial territory design problem with a routing budget constraint. In *Advances in Soft Computing*, pages 307–318. Springer, 2011.

[88] Herbert Robbins and Sutton Monro. A stochastic approximation method. *The annals of mathematical statistics*, pages 400–407, 1951.

[89] Matthew Rosenshine. Contributions to a theory of patrol scheduling. *Operational Research Quarterly (1970-1977)*, 21(1):99–106, 1970.

[90] Reuven Y Rubinstein and Alexander Shapiro. *Discrete event systems: Sensitivity analysis and stochastic optimization by the score function method*, volume 346. Wiley New York, 1993.

[91] J. Sacks, W.J. Welch, T.J. Mitchell, and H.P. Wynn. Design and analysis of computer experiments. *Statistical Science*, 4(4):409C435, 1989.

[92] Stephen Sacks. Evaluation of police patrol patterns. Technical report, Economics Working Papers, University of Connecticut, 2003.

[93] M Angélica Salazar-Aguilar, Roger Z Ríos-Mercado, José L González-Velarde, and Julián Molina. Multiobjective scatter search for a commercial territory design problem. *Annals of Operations Research*, 199(1):343–360, 2012.

[94] M Angélica Salazar-Aguilar, Roger Z Ríos-Mercado, and José Luis González-Velarde. A bi-objective programming model for designing compact and balanced territories in commercial districting. *Transportation Research Part C: Emerging Technologies*, 19(5):885–895, 2011.

[95] Thomas J. Santner, Brian J. Williams, and William I. Notz. *The design and analysis of computer experiments*. Springer, 2003.

[96] Y. Sawaragi, H. Nakayama, and T. Tanino. *Theory of Multiobjective Optimization (vol. 176 of Mathematics in Science and Engineering)*. Academic Press Inc, Orlando, FL, 1985.

[97] Hans-Paul Paul Schwefel. *Evolution and optimum seeking: the sixth generation*. John Wiley & Sons, Inc., 1993.

[98] Perwez Shahabuddin. Rare event simulation in stochastic models. In *Proceedings of the 27th conference on Winter simulation*, pages 178–185. IEEE Computer Society, 1995.

[99] Michael A. Smith and Donald E. Brown. Application of discrete choice analysis to attack point patterns. *Information Systems and e-Business Management*, 5(3), June 2007.

[100] R. E. Steuer. *Multiple Criteria Optimization: Theory, Computation, and Application.* Krieger Pub Co, 1989.

[101] Rajan Suri. Infinitesimal perturbation analysis of discrete event dynamic systems: A general theory. In *Decision and Control, 1983. The 22nd IEEE Conference on*, volume 22, pages 1030–1038. IEEE, 1983.

[102] James R Swisher, Paul D Hyden, Sheldon H Jacobson, and Lee W Schruben. A survey of simulation optimization techniques and procedures. In *Simulation Conference, 2000. Proceedings. Winter*, volume 1, pages 119–128. IEEE, 2000.

[103] Harold Szu and Ralph Hartley. Fast simulated annealing. *Physics letters A*, 122(3):157–162, 1987.

[104] P. E. Taylor and S. J. Huxley. A break from tradition for the san francisco police: patrol officer scheduling using an optimization-based decision support system. *Interfaces*, 19(1):4–24, 1989.

[105] G. H. Tzeng and Y. W. Chen. The optimal location of airport fire stations: A fuzzy multi-objective programming and revised genetic algorithm approach. *Transportation Planning and Technology*, 23:37–55, 1999.

[106] Peter JM Van Laarhoven and Emile HL Aarts. *Simulated annealing.* Springer, 1987.

[107] Darrell Whitley. A genetic algorithm tutorial. *Statistics and computing*, 4(2):65–85, 1994.

[108] T. Wisborg, A. B. Guttmorsen, M. Sorensen, and H. K. Flaatten. The potential of anaesthesiologist-manned ambulance service in rural or urban district. *Acta Anaesthesiologica Scandinavica*, 38:657–661, 1994.

[109] L. Yang, B. F. Jones, and S. H. Yang. A fuzzy multi-objective programming for optimization of fire station locations through genetic algorithms. *European Journal of Operational Research*, 181:903–915, 2007.

[110] Yue Zhang and Donald Brown. Simulation optimization of police patrol districting plans using response surfaces. *Simulation*, 90(6):687–705, 2014.

[111] Yue Zhang and Donald E Brown. Police patrol district design using agent-based simulation and GIS. In *Intelligence and Security Informatics (ISI), 2012 IEEE International Conference on*, pages 54–59. IEEE, 2012.

[112] Yue Zhang and Donald E Brown. Police patrol districting method and simulation evaluation using agent-based model & GIS. *Security Informatics*, 2(7):1–13, 2013.

[113] Yue Zhang and Donald E Brown. Simulation optimization of police patrol district design using an adjusted simulated annealing approach. In *Symposium on Theory of Modeling and Simulation, 2014 Spring Simulation Multi-Conference*, 2014.

[114] Yue Zhang, Samuel H. Huddleston, Donald E Brown, and Gerard P. Learmonth. A comparison of evaluation methods for police patrol district designs. In *Proceedings of the 2013 Winter Simulation Conference*, pages 2532–2543, 2013.