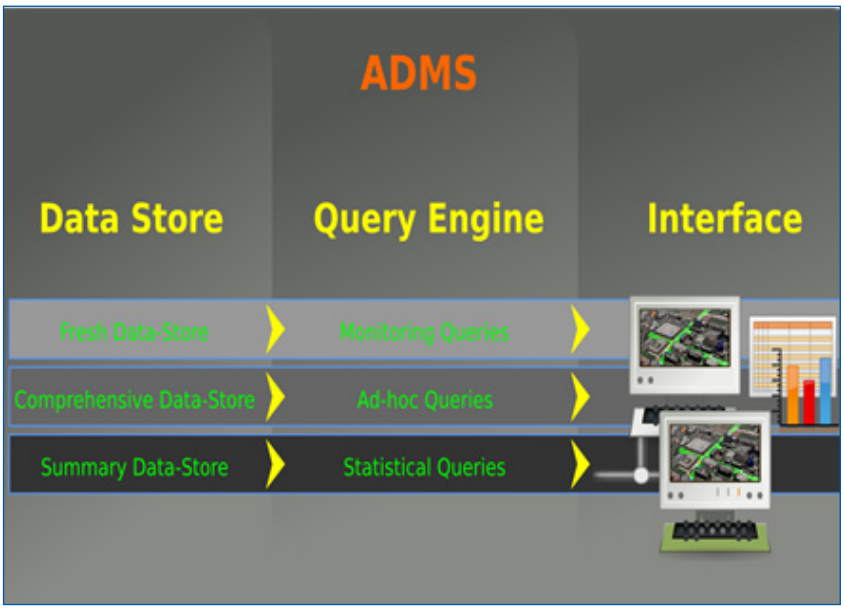
MOUNTAIN-PLAINS CONSORTIUM

MPC 19-407 | F. Banaei-Kashani and R. Fitzgerald

Route Planning for Enhanced Transportation Network Utilization: A System Optimization Approach for Route Planning in Advanced Traveler Information Systems



MOUNTAIN-PLAINS CONSORTIUM REGION 8

A University Transportation Center sponsored by the U.S. Department of Transportation serving the Mountain-Plains Region. Consortium members:

Colorado State University
North Dakota State University
South Dakota State University
University of Colorado Denver
University of Denver
University of Utah
Utah State University
University of Wyoming

Route Planning for Enhanced Transportation Network Utilization: A System Optimization Approach for Route Planning in Advanced Traveler Information Systems

Farnoush Banaei-Kashani, PhD
Assistant Professor

Robert Fitzgerald
PhD Candidate

University of Colorado Denver
Department of Computer Science and Engineering

December 2019

Acknowledgments

The authors extend their gratitude to the Mountain Plains Consortium, the U.S. Department of Transportation for funding this research.

Disclaimer

The contents of this report reflect the views of the authors, who are responsible for the facts and the accuracy of the information presented. This document is disseminated under the sponsorship of the Department of Transportation, University Transportation Centers Program, in the interest of information exchange. The U.S. Government assumes no liability for the contents or use thereof.

ABSTRACT

The existing online mapping systems process many user route queries simultaneously, yet solve each independently, using typical route guidance solutions. These route recommendations are presented as optimal, but often this is not truly the case, due to the effects of competition users experience over the resulting experienced routes, a phenomenon referred to in Game Theory as a Nash Equilibrium. Additionally, route plans of this nature can result in poor utilization of the road network from a system-optimizing perspective as well. In this project, we introduce an enhanced approach for route guidance, motivated by the relevance of a system optimal equilibrium strategy, while also maintaining fairness to the individual. With this approach, the objective is to optimize global road network utilization (as measured by mobility, global emissions etc.) by selecting from a set of generally fair user route alternatives in a batch setting.

For the first time, an approximate, anytime algorithm based on Monte Carlo Tree Search and Eppstein's Top-K Shortest Paths algorithm is presented to solve this complex dual optimization problem in real-time. This approach attempts to identify and avoid the potentially harmful network effects of sub-optimal route combinations. Experiments show that mobility optimization over the real road networks of Rye and Golden, Colorado in a microscopic traffic simulation with a network congestion-minimizing objective can lead to considerable improvement in mobility for users, as observed by a shorter travel time, with an improvement up to 12% with some consideration of route fairness.

As part of this research the following four objectives have been achieved as presented in the Main Body of the report:

1. Introduction of a transportation network utility function that captures utilization of the network based on throughput-of/mobility-through of the transportation network (other network utilization criteria such as overall travel quality, safety, environmental impact, etc., can be studied as part of future work).

2. Designed a multi-criteria route planning solution that uses the introduced utility function as the primary criterion, and an exemplary traveler interest (e.g., fastest route) as the secondary criterion to generate optimal routes for travelers in a transportation network.

3. Developed a data-driven simulation testbed (based on realistic road network and traffic data) to evaluate the designed route planning solution and compare its performance versus state-of- the-art route planning solutions.

4. Advanced knowledge by carrying out comparative analyses to answer the proposed research questions.

In addition to the aforementioned research tasks, we pursued and achieved three other objectives in this project as follows:

1. Advanced policy and practice with respect to transportation network utilization: Toward this end, in multiple occasions we presented our results a Colorado Department of Transportation (CDOT) and National Renewable Energy Laboratory (NREL) as well as MS2 user group.

2. Advanced education through the training of students: To pursue this objective, numerous assignments and course projects were included in both undergraduate level and graduate level courses offered over a period of three years at the Department of Computer Science and Engineering, University of Colorado Denver. Sample assignments as well as sample student work are included in the Appendix.

3. Built an evidence base by disseminating findings through publications and presentations: Results have been published from the studies in the 20th International Conference on Mobile Data Management (MDM 2019), which is a premier venue for presentation of data-driven methodologies for transportation management.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

I. INTRODUCTION

Online mapping systems offer highly granular route guidance, but have also increased the problems of congestion, as they do not address the contributions made by each route to the emergent network congestion [1]. Without a strategy to negotiate those effects, these route guidance providers escalate competition over the vital network corridors, which results in a common result referred to in Game Theory as a Nash Equilibrium [2], or in the transportation literature, as a User Equilibrium (UE) [3]. Needless to say, it is desirable to find assignments for these route plans which could instead enable a System-Optimal (SO) equilibrium, where overall network congestion has been minimized.

In particular, as more technologies are introduced into the road network, it becomes more relevant to consider how such plans might impact the network. Due to the proliferation of mobile devices, the network is now flooded with different forms of real-time communication. This has created the platform for Transportation Network Companies (TNC) to emerge. Stakeholders are now motivated to consider the impact of TNCs, and an opportunity exists to provide incentives to TNCs and delivery companies such as Lyft, Uber, and Grubhub to assist in the optimization of the road network utilization. Similarly, Connected and Autonomous Vehicles (CAV) bring the promise of great optimization opportunities to road networks, as they are unopinionated route-followers. Their complicit nature can help bring the road network traffic to a state of SO equilibrium.

The possibility of system-level route optimization is enticing, as transportation planning studies have long shown the benefits of SO approaches in small, abstract problem sets [4]. However, producing a SO route assignment online and in real time requires solving very challenging optimization problems within a limited computational budget. To address this problem, in this project an approach toward the goal of real-time SO route guidance is introduced and solved using a promising two-phase approximation technique, which balances out the objectives of both the driver agent and the network. To achieve this, a meta-heuristic, anytime algorithm, namely, Monte Carlo Tree Search (MCTS) is employed to identify a set of approximately SO routes from a multiset of options, which are each generated with respect to the driver agent's utility. Experimental results show a 10% improvement of average network travel time over selfish routing, where a meaningful majority of agents were receiving a fair assignment of an equal or faster route.

This project provides a mathematical formulation of static and dynamic SO routing problems is presented and discussed. It also presents a novel algorithmic solution, which is described and evaluated with respect to synthetic populations over real-world road networks. To explore the process of this research, first, related work in the areas of selfish and SO route guidance is presented in Section 2. Next, the generic problem of SO route guidance is discussed in Section 3. An overview of the presented solution appears in Section 4, and then in Section 5 the proposed approach, a two-phase algorithm based on Monte Carlo Tree Search is explained. Thereafter, an experimental comparative analysis is shown in Section 6, followed by Section 7, where the paper is concluded with future directions. Section 9 includes our code base for core algorithms presented in this report.

2. RELATED WORK

In order to review work related to route guidance techniques, the literature is categorized into two groups. First, route guidance techniques which optimize individual routes are considered. Second, research which is broadly associated with simulating or producing SO equilibria in transportation network flow problems is discussed.

2.1 User-Optimal Routing

The classic techniques for conducting an optimal path search for a route are Dijkstra's algorithm and the Bellman-Ford algorithm. Both techniques take advantage of the triangle inequality to find a minimum spanning tree rooted at some origin vertex. These are expanded into path search techniques by traversing the resulting tree back from the destination vertex. The idea is generalized to the all-pairs shortest paths scenario via the Floyd-Warshall algorithm, which exploits the recursive nature of shortest path trees. A* Search can then be used to guide the path building via a search heuristic. These techniques require no pre-computation time, but require algorithmic operations at query time.

Techniques which minimize query time and make the path search solvable in an online context, have the added costs of increased precompute time and memory requirements. The most straight-forward optimization of this sort is to create a lookup table of all possible route queries. This can be achieved by running the Floyd-Warshall algorithm, which produces a massive memory allocation. Parallel optimizations, such as PHAST [5], exist to address this constraint. Smaller sets of data can also be stored effectively on the vertices with linear performance, which nears lookup table queries, such as in Hub Labeling (HL) [6].

Some query time optimizations require a small sweep of the graph, but provide optimizations which do not have large tradeoffs in terms of memory footprint and pre-computation time, and are reasonably competitive in query times to PHAST and HL. Arc Flags [7] provides a bit-sized label identifying useful out-edges at each search step. Contraction Hierarchies (CH) [8] produces a hierarchy of hypergraphs, limiting the search space as the search ascends the hierarchy. Customizable Route Planning (CRP) [9] is more responsive to changes in network flows because it separates the precomputation process into a bootstrapping step and an updating step.

For more details on these techniques, refer to the recent comprehensive survey as discussed by Bast et al. 2016 [10].

2.2 System-Optimal Equilibrium

The study of road network utility comes from the field of traffic assignment [11], which estimates the network effects caused by the interaction of network flows. Those effects result in one of two steady state behaviors first identified by John Glen Wardrop in 1952 [3]: UE from selfish routing behavior, or SO as a result of network-optimizing behavior. Solution methods differ based on the scope of the problem, varying between the macroscopic, which is executed over aggregate flows (vehicles per unit time), and the microscopic, which is executed with respected to solving routes for individual driver agents. Macroscopic scale solutions, such as the Frank-Wolfe algorithm [12], only contain the expected network effects, while the individual route information is lost within the aggregate flow values.

In contrast to macroscopic scale solutions, a microscopic solution to traffic assignment requires running a playout with the interaction of the supply and demand. This extends the problem into the temporal setting, which is referred to as a Dynamic Traffic Assignment (DTA) problem [13]. Iterative agent-based simulators such as MATSim [14] and its successor BEAM [15] solve DTA with a UE objective by running successive "days" of simulation, modifying some or all of the agent routes in response to the effects observed between days. Recent work has extended the MATSim platform with SO route guidance approaches to multi-modal routing [16]. While the authors have proposed a new technique for producing SO equilibria, it is not presented as a solution for online route guidance. To the best of the authors' knowledge, the technique proposed here presents the first solution to SO route guidance in an online and microscopic DTA setting, suitable for real-world route guidance applications.

3. PROBLEM DEFINITION

In the following section, SO route guidance is presented. The objective is to assign routing to a set of agents in such a way that the aggregate effect of their experienced routes is optimal with respect to road network utilization.

Let G represent a road network, as a directed, connected, finite graph with vertices $V(G)$ and edges $E(G) \subseteq V(G) \times V(G)$. Each vertex $v \in V(G)$ represents a location. Each edge $(u, v) \in E(G)$ represents a road segment traversing the road network, with a positively valued and monotonically increasing link cost function $C(F_e, e)$, a function of the link flows F_e and any link attributes stored in e. Let R be a finite set of requests, where each request $r \in R$ is a tuple (o_r, d_r) associated with the r-th agent. A request captures the intent to seek point-to-point routing from an origin o_r to a destination d_r where $o_r, d_r \in V(G)$, for an agent seeking optimized routing. Therefore, each request be served a path of the form $p_r = (v_0, v_1, .., v_{n-1}, v_n)$ such that $(v_k, v_{k+1}) \in E(G) \forall 0 \leq v \leq n$, $v_0 = o_r$, and $v_n = d_r$. The complete set of path options for agent r is represented by the set P_r.

In this hypothetical setting, the desired outcome is a path assignment for each agent $r \in R$ which seeks to optimize both agent r and the unloaded road network G. The optimization of an agent's route is straightforward with a technique such as Dijkstra's algorithm for optimal path finding. However, optimizing the network through path selection implies that these paths are modifiable for the full set R of requests. The SO algorithm must decide a path selection for each agent $r \in R$ from all such alternatives P_r. The global measure of these path selections is viewed through the optimization objective C, as the intersection of path choices occurs on the set of network links $E(G)$.

More formally, one can conceptualize SO routing as a solution to the general optimization problem in (1a), which seeks to find P^* the best paths to assign for each agent, a byproduct of decision vector X. To evaluate X, the sum of edge costs is computed. Each edge cost is dependent on counting the agents which are routed on each edge (1b). This is computed by testing whether an edge has membership in a path (2) or does not (3). The optimization is constrained by solving for exactly one path per request (1c). Each decision variable is ensured to be exactly 0 or 1 (1d). The side effect of this optimization is the set of paths to assign, and the minima C^* is the estimated cost of this assignment.

$$C^* = \arg\min_X \sum_{e \in E(G)} C(F_e, e) \tag{1a}$$

$$\text{s.t} \quad F_e = \sum_{r \in R} \sum_{p \in P_r} In(e,p)\, x_{rp} \tag{1b}$$

$$\sum_{p \in P_r} = 1 \qquad \forall r \in R \tag{1c}$$

$$x_{rp} = \{0, 1\} \qquad \forall r \in R, p \in P_r \tag{1d}$$

$$In(e,p) = \begin{cases} 1, & \text{for } e \in p \quad (2) \\ 0, & \text{otherwise} \quad (3) \end{cases}$$

While this is sufficient to describe an abstract SO problem with a static set of inputs, it does not consider the dynamic nature of the arrival of requests. In order to consider dynamic arrival, one needs to modify the above minimization objective such that it routes a temporally diverse set of agents and maintains a picture of network effects from previous solutions. One approach to dynamic assignment is to allow agent route replanning, and implies that agent routes can be changed frequently. While re-planning is not the intention of this work, it is a problem of interest for future work in SO routing, in particular in how it relates to CAV routing.

A second formulation exists to address dynamic arrival, in which agents are considered as successive batches of temporally located collaborative routing problems. In this batch-based setting, one is concerned with a batch of agents whose departure time falls on a timeline between a simulation start time t_0 and end time T. All requests are then associated with a departure time $d(r) \in [t_0, T)$. Each batch of duration b begins at a start time t_s and ends at time t and forms a time window $[t_s, t)$. For each request r_i, if the departure time $d(r)$ is within the established time window, an optimizing assignment is attempted.

In this setting, it is no longer sufficient to assume the road network is unloaded, as a solution must capture the effects of the current network state at time t as an effect of previous batches. Instead, the revised optimization problem considers both the current batch path assignments as well as the currently observed edge flows F_{e_t}. The final batch optimization problem is shown in 4a. It is interesting to note that, as b gets smaller, the problem approaches the selfish routing scenario. In fact, if one limits the number of agents per batch to 1 and set $b = 1$, then it is exactly the selfish case.

$$C^* = \arg\min_X \sum_{e \in E(G)} C(F_e, e) \tag{4a}$$

$$\text{s.t} \quad F_e = \sum_{r \in R_t} \sum_{p \in P_r} In(e,p)\ x_{rp} \tag{4b}$$

$$\sum_{p \in P_r} = 1 \quad \forall r \in R_t \tag{4c}$$

$$x_{rp} = \{0,1\} \quad \forall r \in R_t, p \in P_r \tag{4d}$$

$$R_t = \{r \mid r \in R, d(r) \in [t_0, T)\} \tag{4e}$$

The cost function C can represent any single or multi-criteria function of link attributes. The most straight-forward function is the evaluation of the link cost/flow for a congestion minimization objective. It is simple to reason that multicriteria functions can also be used. For example, consider the scenario where land use data exists by which to query a link for its available services, such as the availability gas or charging stations. A cost function could then be introduced in which a linear combination of the utility of cost/flows as well as the refueling capacity is computed. As a second scenario, if the average fuel and emissions cost of link traversal is known, these values could be incorporated into an eco-routing objective. For the remainder of this report, a congestion minimizing objective is assumed. As a running example, consider an unloaded road network in Figure 4.1 and six agents in Table 4.1. This example is concerned with a batch $b = 5$ where $t_s = 10$ and $t = 1$

4. SOLUTION OVERVIEW

In the section above, SO route guidance was presented as a multi-choice knapsack problem (MCKP) [17]. In a MCKP, an optimal solution is a set of choices (*batch of path assignments*) found within a multi-set of choices (*agents*), where one item (*path*) must be chosen for each agent. A typical knapsack problem assumes that our link cost/flow functions are *independent*, in that the cost/flow effects for agent one do not interact with the cost/flow effects of agent two. However, this is only the case when no agent paths overlap. As soon as two paths overlap on a single edge, the problem requires a solution which is capable of solving these *interdependent* cost/flow functions. In the case of observing two agents interacting, the problem becomes a quadratic knapsack problem (QKP). As the count of agents grows within a batch, the size of the cost/flow function space grows in $O(kr)$ space as the full combination of possibilities becomes a product of each agent's alternate paths, denoted by k. This renders SO route guidance problems unsuitable for conventional linear solvers and their assumption of linear independent cost functions.

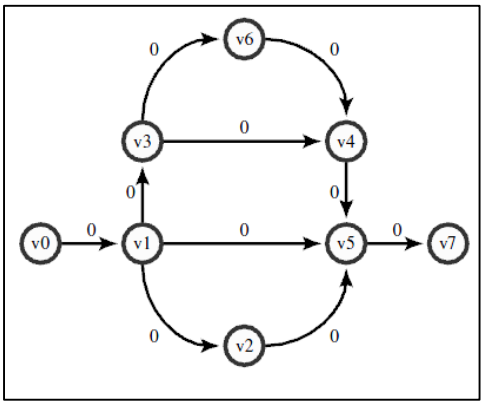


Figure 4.1 Uploaded Road Network

Table 4.1 A Set of 6 Routing Requests

	R		
r	$d(r)$	o_r	o_r
1	9	1	7
2	10	0	7
3	11	0	7
4	13	0	7
5	14	0	7
6	15	6	7

This nonlinear link interdependence caused by the intersection of paths is a challenge, but also an opportunity, as the discovery and reduction of this interdependence is synonymous with the minimization of estimated travel time congestion effects. One possible model for these relationships is that of a probabilistic graph, identifying estimated optimal cost instead of a fixed value. In this way, the structure of this search problem resembles that of a contextual bandit [18]. In contextual bandits, each choice (selection of paths for one agent) is modeled as a probabilistic reward function. The search *explores* by placing an artificially high reward on subspaces with zero or few observations. It balances exploration by eventually *exploiting* the more certain subspaces with high-valued rewards. These problems are solved by algorithms which employ a multi-armed bandit function.

Novel techniques which address this essential trade-off between exploitation and exploration is what has led to a resurgence in the field of reinforcement learning (RL), from which contextual bandits originate. A relatively new technique in RL is the Monte Carlo Tree Search (MCTS) [19] algorithm. It builds a tree which can capture the nested bandit problems described above. Each branch in the tree attempts to find the expected value of its subspace. At each step, as the search continues to sample the tree, it balances out these running expected values with an exploration term, which represents preference to explore previously unexplored subtree spaces.

In the following solution, MCTS is employed in an attempt to learn the link interdependences, which occur between supply and demand in SO route guidance. In order to solve system optimal route guidance for an arbitrary batch size, the search for an optimal combination of paths is solved by way of a two-phase algorithm. In phase one, sets of Top-K alternate paths are constructed for each request, which produces a set of alternate paths. In phase two, a meta-heuristic search for an optimal selection finds an approximately optimal solution by way of MCTS.

5. TWO-PHASED BATCH-BASED SYSTEM OPTIMAL ROUTE GUIDANCE WITH MCTS

To solve for phase one, Eppstein's top-k shortest paths algorithm [20] is executed. The top-k shortest paths for each agent is computed based on an unloaded road network; note that this can be computed offline. In Table 5.1, a scenario for the running example is presented, where agents 2 to 5 each have the same set of alternate paths shown. It is not required that agents share origins or destinations, but it greatly simplifies the example here. Each agent has a set of $k = 3$ alternate paths which are successively minimal, and because of this, the path visiting Vertex 6 is omitted. Note that path 1 is the true shortest path for each agent, and equivalent to the selfish routing solution.

To solve for phase two, MCTS [19] is executed. MCTS is referred to as an *anytime* algorithm in that it can return a result at any iteration. As applied to combinatorial search, MCTS is transformed from its traditional setting of a Markov Decision Process (MDP) into a meta-heuristic algorithm. This is done by placing an empty solution at the root, partial solutions at each branch, and complete solutions at the leaves. At each step through this tree, an alternate path is selected from the next ordered agent. To support this traversal, each branch stores locally-observed optimal values, along with a *reward*, which captures the desire to *exploit* this choice in future traversals. The local reward is typically a running mean of reward values of all associated sub-trees. The initial tree is a root node with no children and user-supplied initial mean reward (typically zero, or a higher "optimistic value").

MCTS is an iterative algorithm which, at each step, performs a **traversal**, an **estimation**, and a tree **update**. During an iteration, first a **traversal** seeks to find an unexplored subspace of the tree by way of a mutli-armed bandit selection. At each step in this traversal, a multi-armed bandit function is used to select a child as the next agent's alternate path to extend this partial solution. If it is an unexplored child, an allocation is made for a new branch at this subspace; if it is explored, the traversal continues. In the case that a new subspace has been instantiated, MCTS will then **estimate** its value. The true cost is only known with a complete solution, due to the interaction of agent paths. To estimate the cost of this subspace, choices are randomly added to this partial solution to construct a complete solution. The cost function C is then evaluated on this solution, producing, what is in effect, a sampling of the distribution of costs under the subspace associated with this estimate. A back-propagation of this estimate is returned as an update to this and all parent tree nodes, updating the local optimal values and rewards.

Traditional MCTS algorithms employ a multi-armed bandit function which is designed for a normalized cost function. For example, the popular UCT algorithm [21] in (5) is tuned to the range of [0, 1], where for each node n, $\bar{X}$ is its running mean reward, n_c is the number of observations of this node, n_p is the number of observations of its parent, and C_p is a tuning parameter. The second term is assumed to be infinite when $n = 0$. The second term represents the desire to explore when insufficient information on a subspace is known.

$$UCT = \bar{X} + 2C_p\sqrt{\frac{2\ln n_p}{n_c}} \qquad (5)$$

Table 5.1 Set of Alternate Paths for Agents 2, 3, 4, and 5

Path	Edges
1	(0,1),(1,5),(5,7)
2	(0,1),(1,2),(2,5),(5,7)
3	(0,1),(1,3),(3,4),(4,5),(5,7)

The structure of UCT is an intuitive and elegant solution to bandit problems, but its requirement of a normalized cost function is problematic for combinatorial optimization. In particular, the range of costs over a set of solutions to a given SO routing problem are not known until all combinations have been explored.

To address this problem, a multi-armed bandit function was proposed in [22] for combinatorial optimization, as shown in (6). This function extends the structure of UCT with a pair of global values capturing the minimum and maximum estimation observed ($\hat{z}^*$ and $\hat{w}^*$ respectively), which are updated at each iteration. Each node n tracks a locally optimal estimation $\hat{z}^*$ and locally average estimation $\bar{z}_n$. Exploitation is captured in (6b) and an *average-weighted exploration* in (6d).

$$U(n) = X(n) + E'(n) \tag{6a}$$
$$X(n) = \frac{e^a - 1}{e - 1} \tag{6b}$$
$$a = \frac{\hat{w}^* - \hat{z}_n^*}{\hat{w}^* - \hat{z}^*} \tag{6c}$$
$$E'(n) = \bar{X}(n)E(n) \tag{6d}$$
$$\bar{X}(n) = \frac{e^b - 1}{e - 1} \tag{6e}$$
$$b = \frac{\hat{w}^* - \hat{z}_n^*}{\hat{w}^* - \hat{z}^*} \tag{6f}$$
$$E(n) = C_p\sqrt{\frac{2\ln n_p}{n}} \tag{6g}$$

One possible solution to this problem is the set of optimal assignments P^* shown in Column 5 of Table 5.2, based on the cost function $C(x) = 4^x$, chosen for simplicity. Figure 5.1 shows the resulting search space of the running example where the solution was found. Leaves show possible solutions, along with their corresponding evaluations (C) below. The solution from the running example (C^*) is shown in **bold**.

Table 5.2 The Solution to the Running Example

		R		
r	$d(r)$	o_r	d_r	P^*
1	9	1	7	( )
2	10	0	7	(0,1,5,7)
3	11	0	7	(0,1,3,4,5,7)
4	13	0	7	(0,1,5,7)
5	14	0	7	(0,1,2,5,7)
6	15	6	7	( )

By solving (4a), the cost of this assignment is $C^* = 4^4 + 4^1 + 4^1 + 4^2 + 4^1 + 4^1 + 4^1 + 4^4 = 548$. As a result and side-effect of this optimization, the algorithm returns the set of paths associated with the minimal-cost combination. The resulting assignment produces flows as shown in Figure 6.1. Note that, as agents 1 and 6 have time values which fall outside of the time range [10-15] they were not considered in this batch.

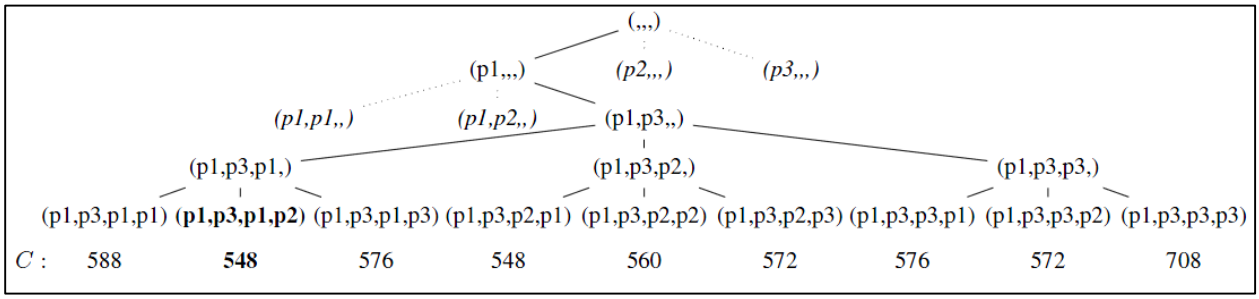


Figure 5.1 A Monte Carlo Tree Search of 4 Requests and 3 Alternate Paths per Request

6. EXPERIMENTS

To evaluate the promise of MCTS as a technique for solving SO route guidance, a simulation-based test bed was created.

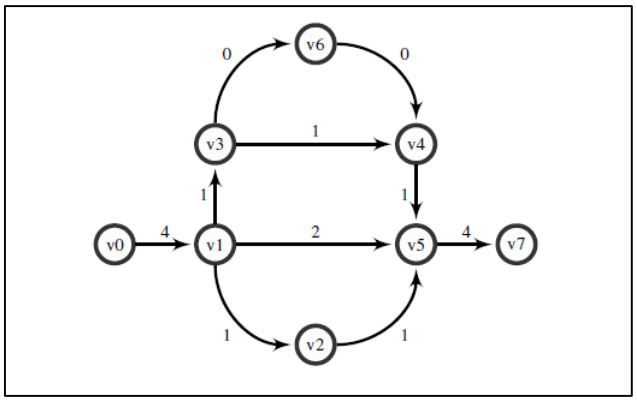


Figure 6.1 After Loading Assignment from Table 4.1

A behavioral study of this approach was performed over two real-world road networks. The following sections describe the methodology and results of these experiments.

6.1 Experimental Methodology

A test bed application was developed for interoperation with a transportation simulator to support study over city-sized inputs. The simulator would need to support this input data as well as allow for user extension to its route guidance system. Based on these requirements, the MATSim [14] transportation simulator was selected. The internal MATSim routing module is scoped to solve routes for isolated agents, so in order to support batch routing, an integration was made directly into the queue simulator QSim with the ability to modify routes for a batch of agents at a time. MATSim is run out-of-the-box, with no features (intersection signals, turn costs) added.

Three route guidance algorithms were implemented to support a comparative behavioral study. A selfish routing algorithm was provided, which finds the shortest path by way of Dijkstra's algorithm. While many behavior models exist for selfish routing, this implementation was chosen due to its simplicity. Two variations of the proposed solution were implemented, namely, SO MCTS and SO Rand, which are different in the way that they implement the technique's second phase. In particular, SO Rand selects a random path among top-k choices for each agent, and SO MCTS runs a search for the optimal selection for the duration of one additional batch duration. SO Rand is a "fair" random algorithm due to selecting from a set of near optimal alternatives, and it is used to illustrate the performance and usefulness of the meta-heuristic technique in SO MCTS.

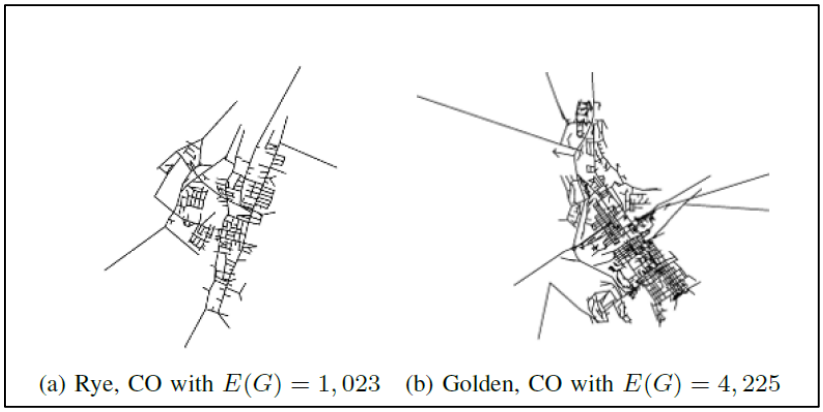
All algorithms were evaluated using a Dell Optiplex 790 desktop computer, with quad-core i7 processors, 16GB of RAM, a 7200rpm 500GB hard drive, and 8M L3 cache, running Ubuntu Xenial 16.04.3. The test bed and all algorithms were implemented in Scala, leveraging the Java interoperability for programmatic interaction with MATSim.

Road network data of Rye and Golden, Colorado were collected from OpenStreetMap [23], and pruned to a single, fully connected directed graph using the JOSM MATSim plugin [24] as pictured in Figure 6.2. These two road networks were chosen as they produce congestion effects on populations which are small enough to run on a single desktop. OpenStreetMap data provides capacity *q* (veh/hr), free-speed *f* (km/hr), and length *l* (m) values, which allowed for a realistic cost function to be adopted from the transportation literature, namely, the Bureau of Public Roads cost function [25]. This is shown, as modified in order to scale capacity values to the batch duration, and free-speed to meters, in (7).

$$C(x) = \frac{l}{f(b/3600)}[1 + 0.15(\frac{x}{1000q})^4] \qquad (7)$$

Each population was generated in a uniformly random distribution over origins, destinations, and departure times. For each trial, the same generated population file was used for each algorithm. Experiments were run for 30 minutes of simulation time in an attempt to reduce the impact of congestion ramp-up and ramp-down. The algorithm was expected to show useful results between some ranges of population sizes, but the bounds on this range were unknown. A reasonable lower bound was assumed at the point where the increase of a population size began to reflect in changes to experienced average travel time. The upper bound was identified by a threshold on experienced average travel time. In particular, population sizes with average speeds of 3 mph (walking speed) were treated as an upper bound on the utility of driving, as agents would be expected to use other modes of transportation beyond this point. For example, in the case of Rye, CO, the observed lower bound was found at 2,500 agents, and the upper bound was found to be 10,000 agents.

Each experiment was parameterized by a population size *p*, a batch duration *b*, the adoption rate of system optimal agents *r* (where *1- r* percent of agents are routed using the selfish route algorithm), and the requested number of alternate paths per agent, *k*. For any configuration of these inputs, fifteen trials of all three algorithms were performed. For each trial, a population of size *p* was generated, with *r*% requesting optimal routing. As the simulator advanced a time step, a representation of link vehicle counts was updated. For any agents requesting any form of routing, these link vehicle counts were used and treated as a network flow estimation model. Selfish route requests were solved using Dijkstra's algorithm based on the network flow snapshot. If the current time coincided with the end of a batch, the two-phase SO route guidance algorithm was run with that batch of agents and a computing budget of *b* seconds. After receiving a result from either SO MCTS or SO Rand, routes were assigned and playout continued until the completion of all simulated agent routes. Upon completion, the experienced average travel time was calculated for each experiment using Equation (8). Mobility was then measured by converting that value into a metric which captured the gains from optimal routing. The gains can be captured as a ratio of average travel time values from both an optimal routing experiment Avg(S^o) and selfish routing Avg(S^s). This ratio is offset by a realistic lower bound value Avg*.



(a) Rye, CO with $E(G) = 1,023$ (b) Golden, CO with $E(G) = 4,225$

Figure 6.2 Road Network Inputs

To find Avg*, a Monte Carlo sampling was performed with 15,000 independent shortest path calculations, each with a uniformly-random single-agent population. This approximated the average free-flow travel time in each network. These values were used to compute travel time improvement I, in (9).

$$Avg(S) = \frac{\sum_{s \in S} T(Path(s))}{|S|} \tag{8}$$

$$I = 1 - \left(\frac{Avg(S^o) - Avg^*}{Avg(S^s) - Avg^*} \right) \tag{9}$$

In order to measure the fairness of an outcome, the experienced travel times were stored for each agent as they were simulated in both the Selfish and SO playout. A histogram was generated from the table of experienced travel time differences.

The full list of data and algorithm parameters under test, along with test ranges, are listed in table IV. Unless otherwise noted, experiments have default parameters p = 7500, b = 5 seconds, r = 20%, and k = 15.

6.2 Experimental Results

In each the following figures, a plot of travel time improvement I is shown with respect to an experimental parameter. Results are distinguished by their road network, using the suffix “Rye” for Rye, CO and “Gld” for Golden, CO.

In Figure 6.3, the effect of population size p is shown. Both SO algorithms strictly improve I for all population sizes. Note that SO MCTS Rye outperforms SO MCTS Gld here, achieving up to 11% improvement. Both SO MCTS algorithms outperform their SO Rand equivalents; in fact, this is the case in all figures. We reason that the problem space size of the road network explains the better performance of MCTS Rye, and that higher parameter values exhaust our computational resource.

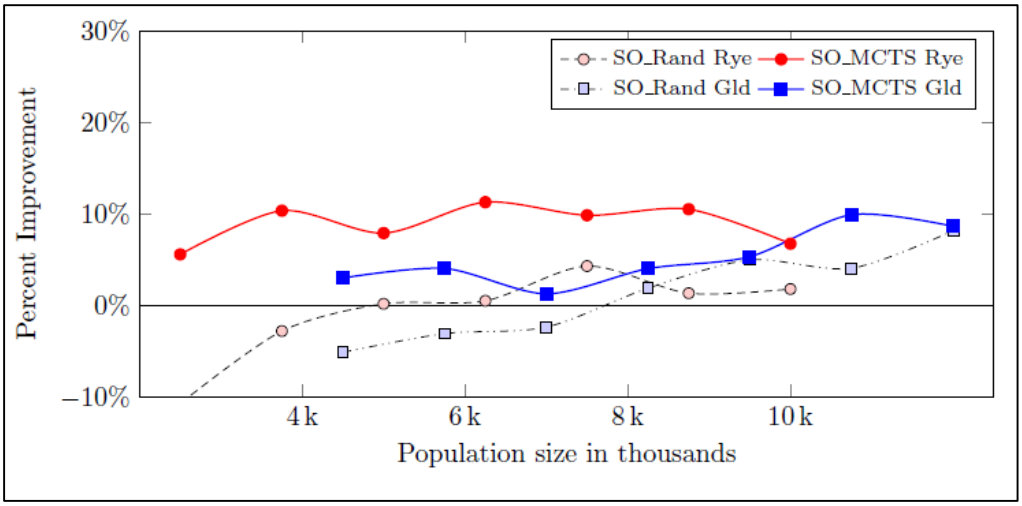


Figure 6.3 Population Size *p*

It is interesting to note that the SO Rand algorithms demonstrate higher improvement when compared to Selfish routing as population sizes exceed 6,000. In seeking an explanation, first we noted that, at these higher levels of congestion, the marginal cost/flow effect of additional agents is higher. We reasoned that the greater inconvenience due to SO Rand routing (which would usually lead to de-improvement) was dominated by the random allocation of alternate paths, which alone had a congestion-minimizing effect. From this, it is our perspective that a SO objective may no longer be useful for inputs which produce such a small difference in experienced travel times between SO Rand and SO MCTS. This is the case for most of the results related to SO MCTS Gld.

In Figure 6.4, the batch duration *b* is shown to have up to 12% travel time improvement for SO MCTS Rye. Both SO MCTS algorithms show consistent travel time improvement here, though as the distance between SO MCTS and SO Rand results grows smaller, the results are less consistent. Even low *b* values show moderate gains over selfish routing. This suggests that even a small number of agents with an optimal objective can positively influence system performance. Improvements eventually diminish for SO MCTS Rye; we attribute this to the search space, which had an average size of 1514 combinations per batch when $b = 16$. In future work, we intend to explore parallelization of the MCTS-based technique to address this.

The performance of SO MCTS Gld shows a similar drop at an earlier value for b, which is expected, but the gradual improvement beyond $b = 4$ for both Gld algorithms warrants further study.

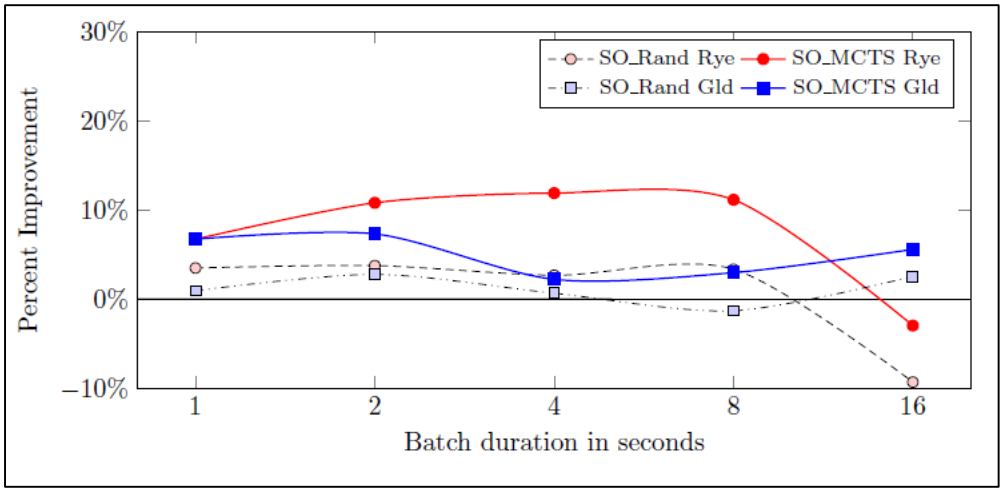


Figure 6.4 Batch Duration *b*

In Figure 6.5, route percentage *r* is shown. The plot for SO MCTS Gld is what was expected for the effect of increasing the adoption rate. However, SO MCTS Rye presents a consistent improvement over selfish routing and SO Rand Rye, with SO MCTS Rye reaching up to 9% travel time improvement.
Our interpretation is that, as SO routing becomes the objective of the majority of agents, their playout becomes more consistent with the expected cost of their routes. In contrast, when in the minority, there are many more agents with a selfish route who might jeopardize the experienced playout of each SO route.

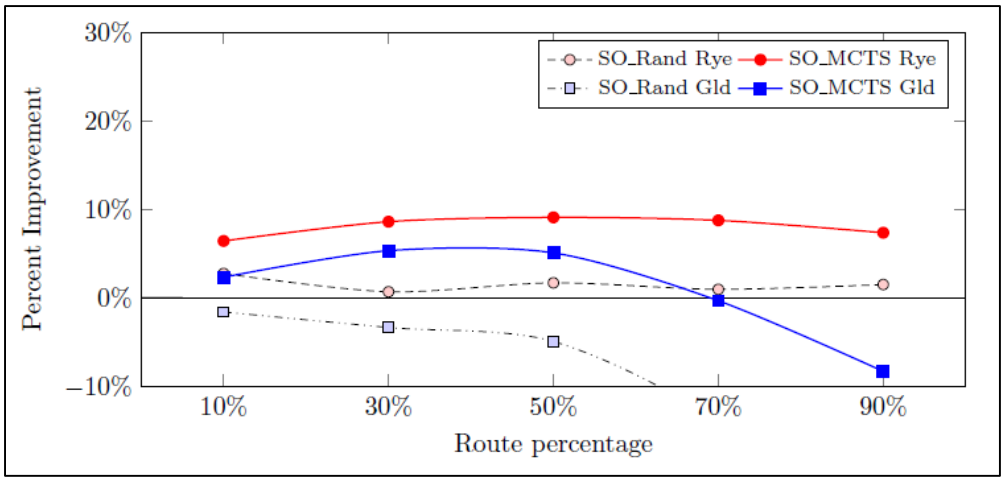


Figure 6.5 Route Percentage *r*

In Figure 6.6, the number of alternate paths per agent *k* is explored. Again, both SO MCTS algorithms show improvement over Selfish route guidance, with SO MCTS Rye reaching up to 11.5% improvement at $k = 2$. Given that our search space size grows in O (kn), we expected to find evidence of a threshold for performance with respect to *k*. However, all algorithms have an uncertain relationship to *k*. We interpret that *k* may be sensitive instead to the topology of each road network and warrants further study.

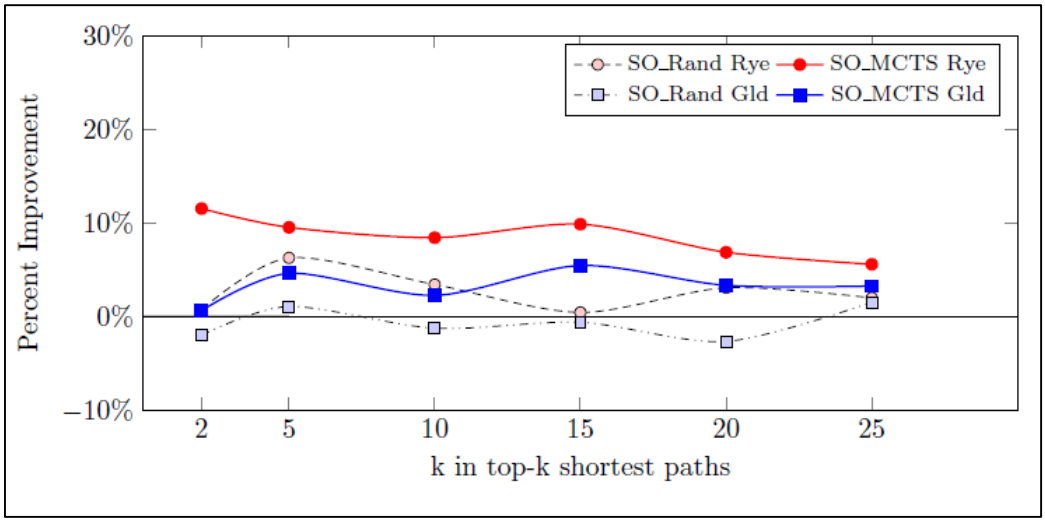


Figure 6.6 Alternate Path Requested *k*

In Figure 6.7, an example of the fairness of this routing is presented as a histogram of experienced travel time difference in seconds, where a negative value equates to a shorter trip due to SO routing. This data was taken from 15 trials with default parameters over the Rye, CO map. SO MCTS Rye favors better in all cases in aggregate. In particular, for each bucket below 0, there were more counts for SO MCTS than SO Rand, indicating that more agents were provided a shorter route. However, the overall performance with regards to fairness leaves much to be desired, with some agents experiencing nearly an hour increase in travel time due to SO routing. Our interpretation is that quality of route depends largely on the set of alternate paths that were produced. We intend to explore improving the quality of these alternative routes in our future work.

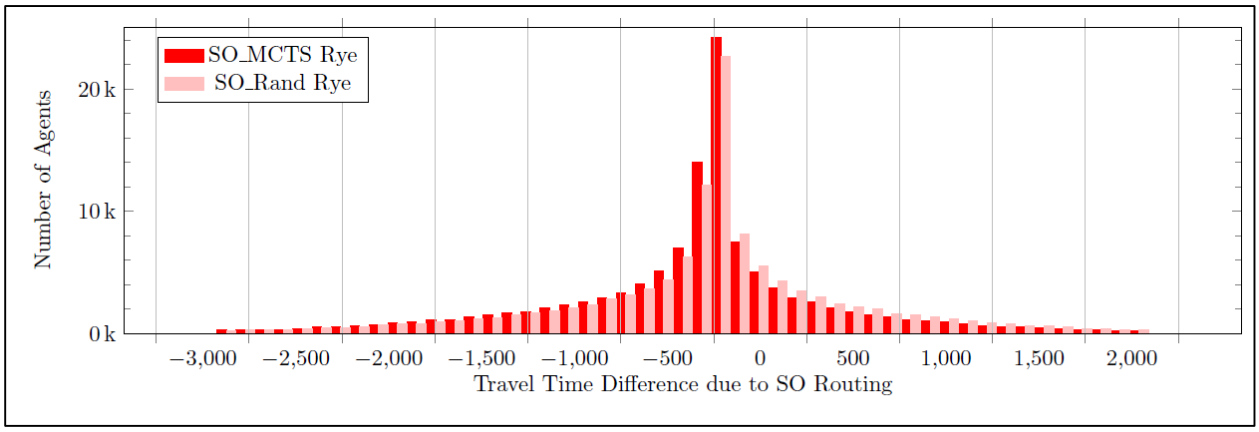


Figure 6.7 Fairness of SO MCTS Rye vs. SO Rand Rye

7. CONCLUSIONS AND FUTURE WORK

A new technique for route guidance was investigated in an attempt to solve for an approximate system optimal plan for a batch of agents. The problem corresponded with a Multiple Choice Knapsack Problem with value dependencies, which made it infeasible to solve with the state-of-the-art approaches. For the first time, the MCTS technique was introduced as a promising solution to SO route guidance, and was shown to produce consistent improvement to selfish routing over a meaningful set of inputs.

In the short term, some extensions to the work are immediately apparent. To address fairness, we will evaluate methods for producing higher-quality route alternatives, in an attempt to reduce the many unfair routes that were produced. Possibilities exist, such as producing a more diverse set of options or integrating a dynamic ranking to precomputed alternatives. Exploring objectives other than travel time as well as composing multi-criteria objectives are a natural extension of this work. In particular, we wish to explore eco-routing as an extension, as well as multi-modal trip guidance, including fleet, ride-hail and rideshare modes.

One feature of MCTS is that it is natural to parallelize the search. We are exploring cluster-based implementations to address horizontal scaling of the search technique in order to solve route guidance for larger road networks. Solving for a SO route guidance policy as a reinforcement learning problem is a promising alternative to the path-based solution described here. While this work assumed that each route received exactly one plan, we are interested in exploring the dynamic route plan scenario, either where we produce route plans in stages or allow for routes to be updated after they are set.

8. REFERENCES

[1] A. C. Madrigal. (2018) The perfect selfishness of mapping apps. [Online]. Available:

https://www.theatlantic.com/technology/archive/2018/03/mappingapps-

and-the-price-of-anarchy/555551/

[2] J. Nash, "Non-cooperative games," Annals of Mathematics, vol. 54, no. 2, pp. 286–295, 1951.

[3] J. G. Wardrop, "Some theoretical aspects of road traffic research," in Inst Civil Engineers Proc London/UK/, 1952.

[4] M. H. S and S. Peeta, Network performance under system optimal and user equilibrium dynamic assignments: implications for ATIS. Transportation Research Board, 1993.

[5] D. Delling, A. V. Goldberg, A. Nowatzyk, and R. F. Werneck, "Phast: Hardware-accelerated shortest path trees," Journal of Parallel and Distributed Computing, vol. 73, no. 7, pp. 940–952, 2013.

[6] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick, "Reachability and distance queries via 2-hop labels," SIAM Journal on Computing, vol. 32, no. 5, pp. 1338–1355, 2003.

[7] M. Hilger, E. Kohler, R. H. Mohring, and H. Schilling, "Fast point-to-point shortest path computations with arc-flags," The Shortest Path Problem: Ninth DIMACS Implementation Challenge, vol. 74, pp. 41–72, 2009.

[8] R. Geisberger, P. Sanders, D. Schultes, and C. Vetter, "Exact routing in large road networks using contraction hierarchies," Transportation Science, vol. 46, no. 3, pp. 388–404, 2012.

[9] D. Delling, A. V. Goldberg, T. Pajor, and R. F. Werneck, "Customizable route planning," in International Symposium on Experimental Algorithms. Springer, 2011, pp. 376–387.

[10] H. Bast, D. Delling, A. Goldberg, M. M¨uller-Hannemann, T. Pajor, P. Sanders, D. Wagner, and R. F. Werneck, "Route planning in transportation networks," in Algorithm engineering. Springer, 2016, pp. 19–80.

[11] J. de Dios Ort´uzar and L. G. Willumsen, Modelling transport, 4th ed. John Wiley & Sons, 2011.

[12] M. Fukushima, "A modified Frank-Wolfe algorithm for solving the traffic assignment problem," Transportation Research Part B: Methodological, vol. 18, no. 2, pp. 169–177, 1984.

[13] S. Peeta and A. K. Ziliaskopoulos, "Foundations of dynamic traffic assignment: The past, the present and the future," Networks and spatial economics, vol. 1, no. 3-4, pp. 233–265, 2001.

[14] A. Horni, K. Nagel, and K. W. Axhausen, The multi-agent transport simulation MATSim. Ubiquity Press London: 2016.

[15] BEAM Developers. Beam. [Online]. Available: http://beam.lbl.gov/

[16] C. Samal, L. Zheng, F. Sun, L. J. Ratliff, and A. Dubey, "Towards a socially optimal multi-modal routing platform," arXiv preprint arXiv:1802.10140, 2018.

[17] C. Wilbaut, S. Hanafi, and S. Salhi, “A survey of effective heuristics and their application to a variety of knapsack problems,” IMA Journal of Management Mathematics, vol. 19, no. 3, pp. 227–244, 2008.

[18] R. S. Sutton and A. G. Barto, Reinforcement learning: An introduction. MIT press, 2018.

[19] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, “A survey of monte carlo tree search methods,” IEEE Transactions on Computational Intelligence and AI in games, vol. 4, no. 1, pp. 1–43, 2012.

[20] D. Eppstein, “Finding the k shortest paths,” SIAM Journal on computing, vol. 28, no. 2, pp. 652–673, 1998.

[21] L. Kocsis and C. Szepesv´ari, “Bandit based monte-carlo planning,” in European conference on machine learning. Springer, 2006, pp. 282–293.

[22] J. P. Pedroso and R. Rei, “Tree search and simulation,” in Applied Simulation and Optimization. Springer, 2015, pp. 109–131.

[23] OpenStreetMap contributors, “Planet dump retrieved from https://planet.osm.org” https://www.openstreetmap.org , 2018.

[24] N. K¨uhnel and M. Zilske, “JOSM Editor MATSim Plugin,” https://github.com/matsim-org/josm-matsim-plugin, 2018.

[25] “Traffic assignment manual,” U.S. Department of Commerce, Washington, DC, 1964.

9. CODE: CORE ALGORITHMS

KSPLocalDijkstrasAlgorithm.scala

```
1  package cse.bdlab.fitzgero.sorouting.algorithm.local.ksp 2
3  import java.time.Instant 4
5  import scala.annotation.tailrec
6  import scala.collection.GenSeq 7
8  import cse.bdlab.fitzgero.graph.algorithm.GraphRoutingAlgorithm
9  import cse.bdlab.fitzgero.graph.config.KSPBounds
10 import cse.bdlab.fitzgero.sorouting.common.algorithm.local.sssp.SSSPLocalDijkstrasAlgorithm
11 import cse.bdlab.fitzgero.sorouting.common.model.roadnetwork._
12 import cse.bdlab.fitzgero.sorouting.common.model.roadnetwork.local.{LocalEdge, LocalGraph} 13
14 object KSPLocalDijkstrasAlgorithm extends GraphRoutingAlgorithm {
15   override type VertexId = SSSPLocalDijkstrasAlgorithm.VertexId
16   override type EdgeId = SSSPLocalDijkstrasAlgorithm.EdgeId
17   override type Graph = SSSPLocalDijkstrasAlgorithm.Graph
18   override type Path = List[PathSegment]
19   override type AlgorithmRequest = LocalODPair
20   override type PathSegment = cse.bdlab.fitzgero.sorouting.common.model.roadnetwork.PathSegment 21
22   type SSSPAlgorithmResult = SSSPLocalDijkstrasAlgorithm.AlgorithmResult 23
24   override type AlgorithmConfig = {
25     def k: Int
26     def kspBounds: Option[KSPBounds]
27   def overlapThreshold: Double 28 }
29
30   case class AlgorithmResult(od: AlgorithmRequest, paths: GenSeq[Path], iterations: Int) 31
32
33
34
35   implicit val simpleKSPOrdering: Ordering[Path] =
36     Ordering.by {
37       (odPath: Path) =>
38         odPath.map(_.cost match {
39           case Some(seqOfCosts) => seqOfCosts.sum
40           case None => 0D
41         }).sum
42     }.rever se 43
44
45
46       @param
47       @param
48       @param
49       @return
50
51   override def runAlgorithm(inputGraph: LocalGraph, request: LocalODPair, config: Option[AlgorithmConfig] = Some(KSPLocalDijkstrasConfig( 52
53     val startTime = Instant.now.toEpochMilli 54
55     val k: Int = config match {
```

```
56          case Some(conf) => conf.k
57          case None => 1
58        }
59
60
61        val graph: LocalGraph =
inputGraph.edges.foldLeft(inputGraph) { 62      (g, e) =>
63            g.updateEdge(e._1,
LocalEdge.setFlow(e._2, 0)) 64                  }
65
66        val kspBounds: KSPBounds = config match {
67          case Some(conf) =>
68            conf.kspBounds match {
69              case Some(ksp) => ksp
70              case None => KSPBounds.Iteration(1)
71            }
72        case None =>
KSPBounds.Iteration(1)  73                      }
74
75        val overlapThreshold: Double = config match {
76          case Some(conf) => conf.overlapThreshold
77          case None => 1.0D
78        }
79
80        SSSPLocalDijkstrasAlgorithm.runAlgorithm(graph, request) match {
81          case None =>
82            println(s"[KSP-ALG] #${request.id} could not find an initial shortest path. Halting
KSP with None")  83
84              None
85          case
Some(trueShortestPath) => 86
87
88            val solution = scala.collection.mutable.PriorityQueue[Path]()
89            solution.enqueue(trueShortestPath.path)
90            val reversedPath: Path =
trueShortestPath.path.reverse 91
92
93            @tailrec
96            def kShortestPaths(walkback: Path, previousGraph: Graph, iteration: Int = 1): Option[AlgorithmResult] = {val
                failedBoundsTest: Boolean =
97                kspBounds match {
98                  case KSPBounds.Iteration(i) => iteration >= i
99                  case KSPBounds.PathsFound(p) => solution.size > p
100                 case KSPBounds.Time(t) => Instant.now.toEpochMilli - startTime > t
101                 case KSPBounds.IterationOrTime(i, t) => iteration >= i || Instant.now.toEpochMilli - startTime > t
102               }
103
104
105           if (failedBoundsTest || walkback.isEmpty) {
106             if
(solution.isEmpty) { 107
108               None
109           } else {
110             val paths: Seq[Path] = solution.dequeueAll.take(k)
```

```
111               Some(AlgorithmResult(request, paths, iteration))
112             }
113           } else {
114
115
116
117             val thisEdgeId: EdgeId =
walkback.head.edgeId 118
119             graph.edgeById(thisEdgeId) match {
120               case Some(edge) =>
121                 val spurSourceVertex: VertexId = edge.src
122                 val spurPrefix: Path = if (walkback.tail.nonEmpty) walkback.tail.reverse else Nil
123                 val blockedGraph: Graph = previousGraph.removeEdge(thisEdgeId)
124                 val spurAlternatives = blockedGraph.outEdges(spurSourceVertex)
125                 if (spurAlternatives.isEmpty) {
126                   kShortestPaths(walkback.tail, blockedGraph, iteration + 1)
127                 } else {
128                   SSSPLocalDijkstrasAlgorithm.runAlgorithm(blockedGraph, LocalODPair(request.id, spurSourceVertex,
                  request.dst)) match {
129                     case None =>
130                       kShortestPaths(walkback.tail, blockedGraph, iteration + 1)
131                     case
Some(pathSpur) => 132
133
134                       val alternativePath: Path = spurPrefix ++
pathSpur.path 135
136
137                       val reasonablyDissimilar = true
138
139
140                       val (graphToRecurse, nextWalkback) = if (reasonablyDissimilar) {
141                         solution.enqueue(alternativePath)
142                         (blockedGraph, walkback.tail)
143                       } else {
144
145                         if (pathSpur.path.nonEmpty)
146                           (blockedGraph.removeEdge(pathSpur.path.head.edgeId), walkback)
147                         else
148                           (blockedGraph, walkback)
149                       }
150                       kShortestPaths(nextWalkback, graphToRecurse, iteration + 1)
151                   }
152                 }
153               case None =>
154                 println(s"[KSP-ALG] #${request.id} spur edge not found in graph: $thisEdgeId")
155                 kShortestPaths(walkback.tail, previousGraph, iteration + 1)
156             }
157           }
158         }
159         try {
160           kShortestPaths(reversedPath, graph)
161         } catch {
162           case e: Throwable =>
```

```
163             println(s"[KSP-ALG] #${request.id} error thrown")
164             println(e)
165             None
166         }
167     }
168 }
169 }
170
```

SSSPLocalDijkstrasAlgorithm.scala

```
1   package
cse.bdlab.fitzgero.sorouting.common.algorithm.local.ss
sp 2
3   import
scala.annotation.tailrec 4
5   import cse.bdlab.fitzgero.graph.algorithm.GraphRoutingAlgorithm
6   import cse.bdlab.fitzgero.sorouting.common.model.roadnetwork.{LocalODPair, PathSegment}
7   import
cse.bdlab.fitzgero.sorouting.common.model.roadnetwork.l
ocal._ 8
9   object SSSPLocalDijkstrasAlgorithm extends GraphRoutingAlgorithm {
10    type VertexId = String
11    type EdgeId = String
12    type Graph = LocalGraph
13    override type PathSegment = cse.bdlab.fitzgero.sorouting.common.model.roadnetwork.PathSegment
14    override type AlgorithmRequest = LocalODPair
15    override type AlgorithmConfig = Nothing
16    case class AlgorithmResult(od: AlgorithmRequest, path:
List[PathSegment]) 17
18

19      @param
20      @param
21      @param
22      @return
23

24    override def runAlgorithm(graph: Graph, odPair: AlgorithmRequest, config: Option[Nothing] = None):
Option[AlgorithmResult] = {
25      val requestName = s"REQ-${odPair.src}#${odPair.dst}"
26      if (odPair.src == odPair.dst) {
27        println(s"[SSSP] $requestName src equals dst, returning None for path")
28        None
29      } else {
30        for {
31          spanningTree <- minSpanningDijkstras(graph, odPair.src, Some(odPair.dst))
32          path <- backPropagate(graph, spanningTree, odPair.dst)
33        }
yield { 34
35          AlgorithmResult(odPair, path)
36        }
37      }
38    }
39

40

41

42      @param
43      @param
44

45    case class SearchData(edge: LocalEdge, cost:
Double = 0D) 46
47

48

49      @param
50      @param
51
```

```
52    case    class    BackPropagateData(π:
Option[EdgeId], d: Double) 53
54
55
56        @param
57        @param
58        @param
59        @return
60
61    def minSpanningDijkstras (graph: Graph, origin: VertexId, destination: Option[VertexId] = None):
Option[Map[VertexId, BackPropagateData 62
63
64      val frontierOrdering: Ordering[SearchData] = Ordering.by (_.cost)
65      val frontier: collection.mutable.PriorityQueue[SearchData] =
        collection.mutable.PriorityQueue()(frontierOrdering.reverse)
66      graph
67        .outEdges(origin)
68        .flatMap(graph.edgeById)
69        .foreach(e => {
70          val cost = e.attribute.linkCostFlow match {
71            case Some(linkCost) => linkCost
72            case None => Double.MaxValue
73          }
74          frontier.enqueue(SearchData(e, cost))
75        })
76
77
78      @tailrec def _dijkstras (
79        solution: Map[VertexId, BackPropagateData] = Map.empty[VertexId, BackPropagateData],
80        enqueued: Set[EdgeId] = Set.empty[EdgeId]
81      ): Option[Map[VertexId, BackPropagateData]] = {
82        if (frontier.isEmpty) {
83          if (destination.isDefined) {
84            None
85          }
86          else {
87            Some(solution)
88          }
89        }
90        else {
91          val (edge, cost) = {
92            val shortestFrontier: SearchData = frontier.dequeue
93            (shortestFrontier.edge, shortestFrontier.cost)
94          }96
97          val solutionUpdate: Map[VertexId, BackPropagateData] =
98            if (!solution.isDefinedAt(edge.dst)) {
99              solution.updated(edge.dst, BackPropagateData(Some(edge.id), cost))
100           } else solution
101
102         if (destination.isDefined && edge.dst == destination.get) {
103           Some(solutionUpdate)
104         } else {
105
106
```

```
107           val addToEnqueued = for {
108             e: String <- graph.outEdges(edge.dst).filter{!enqueued(_)}.toSet
109             localEdge <- graph.edgeById(e)
110             linkCost <- localEdge.attribute.linkCostFlow
111           } yield {
112             val sourceCost: Double = solutionUpdate(localEdge.src).d
113             frontier.enqueue(SearchData(localEdge, linkCost + sourceCost))
114             e
115           }
116
117           val enqueuedUpdate = enqueued ++ addToEnqueued
118           _dijkstras(solutionUpdate, enqueuedUpdate)
119         }
120       }
121     }
122     if (frontier.isEmpty) {
123       println("[DIJ] dijkstras initial state with empty frontier, returning None for solution")
124       None
125     } else {
126
127       _dijkstras(solution = Map(origin -> BackPropagateData(None, 0D)))
128     }
129
130 }
131
132
133     @param
134     @param
135     @param
136     @return
137
138   def backPropagate(g: Graph, spanningTree: Map[VertexId, BackPropagateData], destination: VertexId):
      Option[List[PathSegment]] = {
139     @tailrec def _backPropagate (
140       currentVertex: VertexId,
141       result: List[PathSegment] = List()
142     ): Option[List[PathSegment]] = {
143       if (spanningTree.isDefinedAt(currentVertex)) {
144         val currentNode: BackPropagateData = spanningTree(currentVertex)
145         currentNode.π match {
146           case None =>
147             Some(result.reverse)
148           case Some(edgeId) =>
149             val edge = g.edgeById(edgeId).get
150             val cost = edge.attribute.linkCostFlow.get
151             _backPropagate(edge.src, result :+ PathSegment(edge.id, Some(Seq(cost))))
152         }
153       } else None
154     }
155     _backPropagate(destination)
156   }
157 }
158
```

MonteCarloTreeSearch.scala

```
1   package cse.bdlab.fitzgero.mcts
2
3   import java.time.Instant
4
5   import scala.annotation.tailrec
6   import scala.collection.GenSeq
7
8   import cse.bdlab.fitzgero.mcts.core._
9   import cse.bdlab.fitzgero.mcts.core.terminationcriterion.TerminationCriterion
10  import
cse.bdlab.fitzgero.mcts.tree._ 11
12  trait
MonteCarloTreeSearch[S,A] { 13
14
15
16
17
18
19          @param
20          @param
21          @param
22          @return
23
24      def updateMetaData(simulationResult: Update, node: Tree, leafState: S):
Coefficients 25
26
27
28          @param
29          @return
30
31      def      getSearchCoefficients(tree:      Tree):
Coefficients 32
33
34
35          @param
36          @return
37
38      def      getDecisionCoefficients(tree:      Tree):
Coefficients 39
40
41
42          @param
43          @param
44          @return
45
46      def      createNewNode(state:      S,      action:
Option[A]): Tree 47
48
49
50          @param
51          @return
52
```

```
53      def generatePossibleActions(state: S):
Seq[A] 54
55
56
57          @param
58          @param
59          @return
60
61      def applyAction(state: S, action:
A): S 62
63
64
65          @param
66          @return
67
68      def evaluateTerminal(state: S):
Update 69
70
71
72          @param
73          @return
74
75      def stateIsNonTerminal(state: S):
Boolean 76
77
78
79          @param
80          @return
81
82      def selectAction(actions: Seq[A]):
Option[A] 83
84
85
86          @return
87
88      def startState:
S 89
90
91
92          @param
93          @return
94
```

```
95      def startNode(state: S):
Tree 96
97
98
99
100
 101    protected val terminationCriterion: TerminationCriterion[S,A,Tree]
 102    protected def actionSelection: ActionSelection[S,A]
 103    protected def random: RandomGenerator
 104
105
106
107
108     type
Coefficients 109
110
111
112
113
114     type                    Tree                    <:
MonteCarloTree[S,A,Reward,Update,Coefficients,Tree] 115
116
117
118
 119    type Reward
 120
121
122
123
 124    type Update
 125
126
127
 128        @return
129
130     def             rewardOrdering:
Ordering[Reward] 131
132
133
134
135
 136        @param
 137        @param
 138        @return
139
140     protected   def   treePolicy(node:   Tree,   coefficients:   Coefficients)(implicit   ordering:
Ordering[Reward]): Tree 141
142
143
```

```
144
145         @param
146         @return
147
148     protected def defaultPolicy(node: Tree): (Update, S)
149
150
151
152         @param
153         @param
154         @return
155
156     protected def backup(node: Tree, coefficients: Coefficients, delta:
Update): Tree 157
158
159
160         @param
161         @return
162
163     protected def expand(node: Tree): Option[Tree]
164
165
166
167         @param
168         @param
169         @return
170
171     protected def bestChild(node: Tree, coefficients: Coefficients)(implicit ordering:
Ordering[Reward]): Option[Tree] 172
173
174
175
176
177     def evaluateBranch(tree: Tree, coefficients: Coefficients): Reward =
tree.reward(coefficients) 178
179
180
181         @return
182
183     final def run(root: Tree = startNode(startState)): Tree = {
184       terminationCriterion.init()
185       while (terminationCriterion.withinComputationalBudget(root)) {
186         val v_t = treePolicy(root,getSearchCoefficients(root))(rewardOrdering)
187         val (delta,leaf) = defaultPolicy(v_t)
188         val c = updateMetaData(delta,v_t,leaf)
189         backup(v_t,c,delta)
190       }
191       root
192     }
```

```
193
194
195
196
197         @param
198         @param
199         @return
200
201     final private def _hasUnexploredActions(generatePossibleActions: (S) => Seq[A])(node: Tree): Boolean
        = {
202       val explored: GenSeq[A] = node.children match {
203         case None => Seq[A]()
204         case Some(c) => c.keys.toSeq
205       }
206       generatePossibleActions(node.state).diff(explored).nonEmpty
207
208 }
209
210
211         @return
212
213     final protected def hasUnexploredActions: (Tree) => Boolean =
    _hasUnexploredActions(generatePossibleActions) 214
215
216
217         @param
218         @return
219
220     final def bestGame(root: Tree): Seq[A] =
221       if (root.hasNoChildren) Seq()
222       else {
223         @tailrec
224         def _bestGame(node: Tree, solution: Seq[A] = Seq.empty[A]): Seq[A] = {
225           if (node.hasNoChildren) solution
226           else {
227             bestChild(node, getDecisionCoefficients(root))(rewardOrdering) match {
228               case None => solution
229               case Some(child) =>
230                 child.action match {
231                   case None => solution
232                   case Some(action) =>
233                     _bestGame(child, solution :+ action)
234                 }
235             }
236           }
237         }
238         _bestGame(root)
239       }
240
241
242
```

```
243         @param
244         @param
245         @return
246
247     final def bestMove(decisionCoefficients: Coefficients, root: Tree): Option[A] =
248       for {
249         child <- bestChild(root, decisionCoefficients)(rewardOrdering)
250         action <- child.action
251       } yield action
252
253 }
```

PedrosoReiMCTS.scala

```
1  package cse.bdlab.fitzgero.mcts.variant
2
3    import cse.bdlab.fitzgero.mcts.MonteCarloTreeSearch
4    import cse.bdlab.fitzgero.mcts.algorithm.backup.StandardBackup
5    import cse.bdlab.fitzgero.mcts.algorithm.bestchild.StandardBestChild
6    import cse.bdlab.fitzgero.mcts.algorithm.defaultpolicy.StandardDefaultPolicy
7    import cse.bdlab.fitzgero.mcts.algorithm.expand.StandardExpand
8    import cse.bdlab.fitzgero.mcts.algorithm.samplingpolicy.banditfunction.UCT_PedrosoRei.Objective
9    import cse.bdlab.fitzgero.mcts.algorithm.samplingpolicy.scalar.UCTScalarPedrosoReiReward
10   import cse.bdlab.fitzgero.mcts.algorithm.treepolicy.StandardTreePolicy
11   import cse.bdlab.fitzgero.mcts.tree._
12
13   trait PedrosoReiMCTS[S,A] extends MonteCarloTreeSearch[S,A]
14                             with StandardBestChild[S,A]
15                             with StandardTreePolicy[S,A]
16                             with StandardDefaultPolicy[S,A]
17                             with StandardBackup[S,A]
18                             with
StandardExpand[S,A] { 19
20     def objective: Objective
21
22     final override type Reward = Double
23     final override type Update = BigDecimal
24     final override type Coefficients = UCTScalarPedrosoReiReward.Coefficients
25
26
27
28     var globalBestSimulation: Update
29     var globalWorstSimulation: Update
30
31
32
33
34     var bestSolution: S
35
36
37
38
39     var nodesCreated: Long = 0
40
41     final override def rewardOrdering: Ordering[Reward] = scala.math.Ordering.Double
42
43     final override type Tree = MCTreePedrosoReiReward[S,A]
44
45     final override def startNode(s: S): MCTreePedrosoReiReward[S, A] = MCTreePedrosoReiReward(s, None,
objective = objective) 46
47     final override def createNewNode(state: S, action: Option[A]): MCTreePedrosoReiReward[S, A] = {
48       nodesCreated += 1
49     MCTreePedrosoReiReward(state, action, objective)
50     }
51
52
53     final override def updateMetaData(simulationResult: Update, node: Tree, leafState: S): Coefficients = {
54       if (objective.isWorseThan(simulationResult, globalWorstSimulation)) globalWorstSimulation = simulationResult
55       if (objective.isBetterThanOrEqualTo(simulationResult, globalBestSimulation)) {
56         bestSolution = leafState
57       globalBestSimulation = simulationResult
58       }
59       UCTScalarPedrosoReiReward.Coefficients(getSearchCoefficients(node).Cp, globalBestSimulation, globalWorstSimulation)
60     }
61   }
62
63
```

PedrosoReiMCTSLightImpl.scala

```
1   package cse.bdlab.fitzgero.sorouting.common.mcts.light.pedrosoreimcts
2
3   import scalaz.effect.IO
4
5   import cse.bdlab.fitzgero.mcts.algorithm.samplingpolicy.banditfunction.UCT_PedrosoRei.{Minimize, Objective}
6   import cse.bdlab.fitzgero.mcts.algorithm.samplingpolicy.scalar.UCTScalarPedrosoReiReward.{Coefficients, SearchCoefficient}
7   import cse.bdlab.fitzgero.mcts.core.terminationcriterion.{TerminationCriterion, TimeTermination}
8   import cse.bdlab.fitzgero.mcts.tree.MCTreePedrosoReiReward
9   import cse.bdlab.fitzgero.sorouting.common.mcts.light.Tag
10  import cse.bdlab.fitzgero.sorouting.common.model.population.light.RequestMCTS
11  import cse.bdlab.fitzgero.sorouting.common.model.roadnetwork.light.fixednetworktable.FixedNetworkTable
12  import cse.bdlab.fitzgero.sorouting.common.model.roadnetwork.light.snapshottable.SnapshotTable
13
14  class PedrosoReiMCTSLightImplIO(
15
16    val iOFixedNetworkTable     : FixedNetworkTable[IO],
17    val iOSnapshotTable         : SnapshotTable[IO],
18    val request                 : RequestMCTS,
19    val seed                    : Long = 0L,
20    val Cp                      : Double,
21    override val terminationCriterion : TerminationCriterion[Array[Tag], Tag, MCTreePedrosoReiReward[Array[Tag], Tag]] = TimeTermination(50
22    val startState              : Array[Tag] = Array(),
23    val costFunction            : (FixedNetworkTable[IO], SnapshotTable[IO], Map[Int, Float]) => IO[BigDecimal]
24  ) extends PedrosoReiMCTSLight {
25
26    override val objective: Objective = Minimize()
27    override var globalBestSimulation: BigDecimal = objective.defaultBest
28    override var globalWorstSimulation: BigDecimal = objective.defaultWorst
29    override var bestSolution: Array[Tag] = Array()
30
31    override def getSearchCoefficients(tree: Tree): Coefficients = Coefficients(Cp, globalBestSimulation, globalWorstSimulation)
32    override def getDecisionCoefficients(tree: Tree): Coefficients = Coefficients(SearchCoefficient, globalBestSimulation, globalWorstSimul
33    override def evaluateTerminal(state: Array[Tag]): BigDecimal = PedrosoReiMCTSLightImpl.evaluateCostFlowDelta(iOFixedNetworkTable, iOSna
34  }
35
36  object PedrosoReiMCTSLightImpl {
37    def apply(
38      fnt : FixedNetworkTable[IO],
39      sn  : SnapshotTable[IO],
40      req : RequestMCTS,
41      seed: Long,
42      Cp  : Double,
43      term: TerminationCriterion[Array[Tag], Tag, MCTreePedrosoReiReward[Array[Tag], Tag]],
44      start: Array[Tag],
45      f: (FixedNetworkTable[IO], SnapshotTable[IO], Map[Int, Float]) => IO[BigDecimal]
46    ): PedrosoReiMCTSLightImplIO = new PedrosoReiMCTSLightImplIO(fnt, sn, req, seed, Cp, term, start, f)
47    def evaluateCostFlowDelta(
48      iOFixedNetworkTable: FixedNetworkTable[IO],
49      iOSnapshotTable                 : SnapshotTable[IO],
```

```
      request              : RequestMCTS,
      state                : Array[Tag],
      costFunction         : (FixedNetworkTable[IO], SnapshotTable[IO], Map[Int, Float]) => IO[BigDecimal]
    ): IO[BigDecimal] = {
      val flows = Tag.tagsToGroupedFlows(state, request)
    costFunction(iOFixedNetworkTable,
iOSnapshotTable, flows)                          }
  }
```

UCTScalarPedrosoReiReward.scala

```
1  package cse.bdlab.fitzgero.mcts.algorithm.samplingpolicy.scalar 2
3    import   cse.bdlab.fitzgero.mcts.MonteCarloTreeSearch
4    import    cse.bdlab.fitzgero.mcts.algorithm.samplingpolicy.banditfunction.UCT_PedrosoRei
5
6
7    trait UCTScalarPedrosoReiReward[S,A] extends MonteCarloTreeSearch[S,A] {
8      self: {
9        type Reward = Double
10     type Coefficients=
UCTScalarPedrosoReiReward.Coefficients 11          } =>
12
13   }
14
15   object UCTScalarPedrosoReiReward {
16
17
18
19         @param
20         @param
21         @param
22
23     case class Coefficients (Cp: Double, globalBestSimulation: BigDecimal, globalWorstSimulation: BigDecimal) extends Serializable
24
25     val ExplorationCoefficient: Double = 1D/0.707D
26   val SearchCoefficient: Double = 0D
27   }
```

UCT_PedrosoRei.scala

```
1   package cse.bdlab.fitzgero.mcts.algorithm.samplingpolicy.banditfunction 2
3
4
5
6
7   object UCT_PedrosoRei { 8
9
10
11
12    sealed trait Objective {
13      def defaultBest: BigDecimal
14      def defaultWorst: BigDecimal
15      def defaultSimulation: BigDecimal
16      def isBetterThanOrEqualTo(a: BigDecimal, b: BigDecimal): Boolean
17    def isWorseThan(a: BigDecimal, b: BigDecimal): Boolean 18          }
19    case class Minimize(lowerBounds: BigDecimal = BigDecimal.decimal(0), upperBounds: BigDecimal = BigDecimal("9" * 50)) extends Objective
20      override def defaultBest: BigDecimal = upperBounds
21      override def defaultWorst: BigDecimal = lowerBounds
22      override def defaultSimulation: BigDecimal = upperBounds
23      override def isBetterThanOrEqualTo(a: BigDecimal, b: BigDecimal): Boolean = a <= b
24    def isWorseThan(a: BigDecimal, b: BigDecimal): Boolean = a > b 25          }
26    case class Maximize(lowerBounds: BigDecimal = BigDecimal.decimal(0), upperBounds: BigDecimal = BigDecimal("9" * 50)) extends Objective
27      override def defaultBest: BigDecimal = lowerBounds
28      override def defaultWorst: BigDecimal = upperBounds
29      override def defaultSimulation: BigDecimal = lowerBounds
30      override def isBetterThanOrEqualTo(a: BigDecimal, b: BigDecimal): Boolean = a >= b
31    def isWorseThan(a: BigDecimal, b: BigDecimal): Boolean = a < b 32          }
33
34
35
36        @param
37        @param
38        @param
39        @param
40        @param
41        @param
42        @param
43        @return
44
45    def apply(globalBestSimulation: BigDecimal,
46              globalWorstSimulation: BigDecimal,
47              childBestSimulation: BigDecimal,
48              childAverageSimulation: BigDecimal,
49              childVisits: Long,
50              parentVisits: Long,
51              Cp: Double): Double = { 52
```

```
53        val X = pedrosoReiExploitationTerm(globalBestSimulation, globalWorstSimulation,
childBestSimulation)
54        val E = pedrosoReiExplorationTerm(globalBestSimulation, globalWorstSimulation,
          childAverageSimulation, Cp, parentVisits, childVisits)
55        X
+
E
5
6
57      }
58
59
60
61
62          @param
63          @param
64          @param
65          @return
66
67      def pedrosoReiExploitationTerm(globalBestSimulation: BigDecimal, globalWorstSimulation:
BigDecimal, childBestSimulation: BigDecimal): D
68        if (globalWorstSimulation == globalBestSimulation) 0D
69        else {
70          pedrosoReiXTerm(globalBestSimulation,globalWorst
Simulation,childBestSimulation) 71                }
72      }
73
74      def pedrosoReiXTerm(globalBestSimulation: BigDecimal, globalWorstSimulation: BigDecimal,
localSimulation: BigDecimal): Double = {
75        val a: Double = ((globalWorstSimulation - localSimulation) / (globalWorstSimulation -
          globalBestSimulation)).toDouble
76        val numer: Double = math.pow(math.E, a) - 1D
77        val denom: Double = math.E - 1D
78      if (denom != 0)
numer / denom else 0D
79      }
80
81
82
83          @param
84          @param
85          @param
86          @return
87
88      def      pedrosoReiExplorationTerm(globalBestSimulation:      BigDecimal,      globalWorstSimulation:
BigDecimal, childAverageSimulation: BigDecimal, 89                if (Cp == 0) 0D
90        else if (globalWorstSimulation == globalBestSimulation) Double.PositiveInfinity
91        else {
92          val XBar = pedrosoReiXTerm(globalBestSimulation, globalWorstSimulation,
          childAverageSimulation)
93          val E = uctExploration(Cp, parentVisits, childVisits)
94          XBar * E
95        }
96      }
```

```
97
98    def uctExploration(Cp: Double, parentVisits: Long, childVisits: Long): Double = {
99      if (parentVisits == 0L)
100       0D
101     else if (childVisits == 0L)
102       Double.PositiveInfinity
103     else
104       Cp * math.sqrt(math.log(parentVisits) / childVisits)
105   }
106   }
107
```

ActionSelection.scala

```
1   package cse.bdlab.fitzgero.mcts.core
2
3   trait ActionSelection[S,A] {
4   def selectAction(actions: Seq[A]): Option[A] 5
    }
6
7   class RandomSelection[S,A](
8     random: RandomGenerator,
9     generatePossibleActions: (S) => Seq[A]
10  ) extends ActionSelection[S,A] {
11    def selectAction(actions: Seq[A]): Option[A] = {
12      actions match {
13        case Nil => None
14      case xs => Some(actions(random.nextInt(actions.size)))
15      }
16    }
17  }
18
19  object RandomSelection {
20    def apply[S,A](random: RandomGenerator, generatePossibleActions: (S) => Seq[A]):
      RandomSelection[S,A] =
21  new RandomSelection(random, generatePossibleActions) 22
    }
```

StandardBackup.scala

```
1   package cse.bdlab.fitzgero.mcts.algorithm.backup 2
3   import scala.annotation.tailrec
4
5   import cse.bdlab.fitzgero.mcts.MonteCarloTreeSearch
6
7   trait StandardBackup[S,A] extends MonteCarloTreeSearch[S,A] {
8     @tailrec
9     override protected final def backup(node: Tree, coefficients: Coefficients, delta: Update): Tree = {
10      node.parent() match {
11        case None =>
12          node.update(delta, coefficients)
13          node
14        case Some(parent) =>
15
16          node.update(delta, coefficients)
17      backup(parent, coefficients, delta) 18
        }
19    }
20  }
21
```

StandardBestChild.scala

```
1  package cse.bdlab.fitzgero.mcts.algorithm.bestchild
2
3  import cse.bdlab.fitzgero.mcts.MonteCarloTreeSearch
4
5  trait StandardBestChild[S,A] extends MonteCarloTreeSearch[S,A] {
6    override protected final def bestChild(node: Tree, coefficients: Coefficients)(implicit ordering: Ordering[Reward]): Option[Tree] = {
7      if (node.hasNoChildren) { None }
8      else {
9        val children = node.childrenNodes.values map {
10       tree: Tree => (tree.reward(coefficients), tree)
11     }
12       val bestChild = children.maxBy{_._1}._2
13       Some(bestChild)
14     }
15   }
16 }
17
```

StandardDefaultPolicy.scala

```
1  package cse.bdlab.fitzgero.mcts.algorithm.defaultpolicy
2
3  import scala.annotation.tailrec
4
5  import cse.bdlab.fitzgero.mcts.MonteCarloTreeSearch
6
7  trait StandardDefaultPolicy[S,A] extends MonteCarloTreeSearch[S,A] {
8
9    override protected final def defaultPolicy(monteCarloTree: Tree): (Update, S)= {
10     if (stateIsNonTerminal(monteCarloTree.state)) {
11
12
13       @tailrec
14       def _defaultPolicy(state: S): (Update, S) = {
15         if (stateIsNonTerminal(state)) {
16           selectAction(generatePossibleActions(state)) map { applyAction(state,_) } match {
17             case None =>
18
19               throw new IllegalStateException(s"Applying action to state $state but it produced an empty state. your applyAction and gene
20             case Some(nextState) =>
21           _defaultPolicy(nextState)
22           }
23         } else {
24           (evaluateTerminal(state), state)
25         }
26       }
27
28       _defaultPolicy(monteCarloTree.state)
29     } else {
30       (evaluateTerminal(monteCarloTree.state), monteCarloTree.state)
31     }
32   }
33 }
34
```

StandardExpand.scala

```scala
package cse.bdlab.fitzgero.mcts.algorithm.expand
import cse.bdlab.fitzgero.mcts.MonteCarloTreeSearch

trait StandardExpand[S,A] extends MonteCarloTreeSearch[S,A] {
  override protected final def expand(node: Tree): Option[Tree] = {
    for {
      action <- actionSelection.selectAction(generatePossibleActions(node.state))
    } yield {
      val newState = applyAction(node.state, action)
      val newNode = createNewNode(newState, Some(action))
      node.addChild(newNode)
    newNode
    }
  }
}

```

StandardMCTS.scala

```
1  package cse.bdlab.fitzgero.mcts.variant
2
3  import cse.bdlab.fitzgero.mcts.MonteCarloTreeSearch
4  import cse.bdlab.fitzgero.mcts.algorithm.backup.StandardBackup
5  import cse.bdlab.fitzgero.mcts.algorithm.bestchild.StandardBestChild
6  import cse.bdlab.fitzgero.mcts.algorithm.defaultpolicy.StandardDefaultPolicy
7  import cse.bdlab.fitzgero.mcts.algorithm.expand.StandardExpand
8  import cse.bdlab.fitzgero.mcts.algorithm.samplingpolicy.scalar.UCTScalarStandardReward
9  import cse.bdlab.fitzgero.mcts.algorithm.treepolicy.StandardTreePolicy
10 import cse.bdlab.fitzgero.mcts.tree._
11
12 trait StandardMCTS[S,A] extends MonteCarloTreeSearch[S,A]
13                         with StandardBestChild[S,A]
14                         with StandardTreePolicy[S,A]
15                         with StandardDefaultPolicy[S,A]
16                         with StandardBackup[S,A]
17                         with StandardExpand[S,A] {
18   final override type Reward = Double
19   final override type Update = Double
20   final override type Coefficients = UCTScalarStandardReward.Coefficients
21
22   final override def rewardOrdering: Ordering[Double] = scala.math.Ordering.Double
23
24   final override type Tree = MCTreeStandardReward[S,A]
25
26   final override def startNode(s: S): MCTreeStandardReward[S, A] = MCTreeStandardReward(s)
27
28   final override def createNewNode(state: S, action: Option[A]): MCTreeStandardReward[S, A] =
29     MCTreeStandardReward(state, action)
30   override def updateMetaData(simulationResult: Double, node: Tree, state: S): Coefficients = getSearchCoefficients(node)
31 }
32
33
34
35
```

StandardTreePolicy.scala

```
1   package  cse.bdlab.fitzgero.mcts.algorithm.treepolicy
2
3   import scala.annotation.tailrec
4
5   import  cse.bdlab.fitzgero.mcts.MonteCarloTreeSearch
6
7   trait StandardTreePolicy[S,A] extends MonteCarloTreeSearch[S,A] {
8
9     @tailrec
10    override protected final def treePolicy(node: Tree, coefficients: Coefficients)(implicit ordering: Ordering[Reward]): Tree = {
11      if (stateIsNonTerminal(node.state)) {
12        if (hasUnexploredActions(node)) {
13          expand(node) match {
14            case None => node
15            case Some(newChild) => newChild
16          }
17        } else {
18          bestChild(node, coefficients) match {
19            case None => node
20            case Some(bestChild) =>
21          treePolicy(bestChild,  coefficients)
22          }
23        }
24      } else                          {
25        node
26      }
27    }
28  }
29
```

APPENDIX: TEACHING MATERIALS

This following first shows the two course projects from CSCI 4951/5951 Big Data Systems, followed by an example of student work.

I.A CSCI 4951/5951 BIG DATA SYSTEMS COURSE PROJECT: SAMPLE #1

Course Project - Path Planning in Road Networks

The goal of this project is to implement and (partially) verify **a path planning** algorithm for computing the single-source shortest paths to all vertices in a directed graph.

Part 1 Implementing and Verifying a Heap-Based Priority Queue

Dijkstra's algorithm uses a priority queue for efficient exploration of the vertices in the graph. In the first part of this project you will implement and verify a heap-based priority queue.

Your priority queue should have the following signature:

```
class PriorityQueue<T(==)>
{
  constructor Init(N: nat);
  method insert(t: T, k: int);
  function method min(): T
  method deleteMin();
  method decreaseKey(t: T, k: int);
  function method isEmpty(): bool;
}
```

The priority queue should maintain a set of T values that are ordered by integer keys. There should always be at most one key/value pair for any T value in the queue. The constructor takes the initial capacity as argument. The `decreaseKey` operations sets the key for the given value `t` to `k`. It may assume that `k` is not larger than the old key associated with `t`. The semantics of the remaining operations is as expected.

(a) Implement the priority queue operations. Make sure that all operations have the expected worst-case running times of a heap-based implementation. In particular, `insert`, `deleteMin`, and `decreaseKey` should all run in time $\mathcal{O}(\log(n))$, where n is the size of the queue. The operations `min` and `isEmpty` should run in constant time.

(b) Add a dynamic frame to your priority queue implementations and add an invariant that ties it to the representation of the queue. Use Dafny to check that all queue operations satisfy there modifies clauses and preserve the representation invariant.

(c) Add contracts to all operations to specify their functional behavior. You may introduce additional ghost fields as you see fit. Use Dafny to verify that all contracts are satisfied by your implementation. Add additional representation invariants to your implementation if necessary.

Hint: The Boogie source code distribution contains a partial heap-based priority queue implementation. See `Test/dafny1/PriorityQueue.dfy`. This implementation only

stores keys instead of key/value pairs but you can use it as a starting point for your own implementation.

Part 2 Implementing and Verifying Dijkstra's Algorithm

In the second part of this project you will implement Dijkstra's Algorithm in Dafny and verify it against your priority queue implementation.

(a) Use the class `Graph` provided on the course web site and your priority queue implementation from Part 1 to implement a method

```
method shortestPaths(G: Graph, source : int)
returns (prev: array<int>)
  requires G != null && G.Valid;
  requires G.hasVertex(source);
```

The method should use Dijkstra's algorithm to compute an array `prev`. The array `prev` encodes (in inverse order) the shortest paths from the vertex `source` in the graph `G` to all other vertices in the graph. More precisely, `prev` maps a vertex `i` to its immediate predecessor on the shortest path from `source` to `i` if the path exists, and `-1` otherwise.

An object `G` of class `Graph` represents a directed graph with vertices `0` to `G.size-1` and edges encoded as adjacency lists for each vertex (see field `neighbors`). In your implementation you may assume that all edges in `G` have weight 1, i.e., the shortest distance between two vertices u and v is the infimum of the lengths of all paths from u to v in `G`.

(b) Use Dafny to verify that your implementation always terminates and that it satisfies all the contracts of the priority queue operations. Add loop invariants and decrease expressions to your implementation as you see fit. You do not need to verify that your implementation satisfies the specification of Dijkstra's algorithm, i.e., that it actually computes the shortest paths.

Hint 1: To make your life easier, avoid adding additional state to the objects of class `Graph`, e.g., by extending the class with additional mutable fields such as arrays. Keep all additional state local to the method `shortestPaths`.

Hint 2: You can start with Part 2 before you have finished Part 1. To do this, you need to provide an axiomatic interface specification of the priority queue. For example, an axiomatic interface specification of a set data structure is shown in Figure 1. Note that in the interface specification, the method `isEmpty` itself is used to denote the return value in the postcondition that defines the behavior of `isEmpty`. To tie the knot, once you are done with Part 1 you need to make sure that your implementation of the priority queue actually satisfies the axiomatic specification that you used in Part 2.

```
class Set<T(==)>
{
  ghost var Repr: set<objects>;
  ghost var Contents: set<T>;

  predicate Valid { this in Repr };

  ...

  method insert(x: T)
    requires Valid;
    modifies Repr;
    ensures Valid;
    ensures fresh(Repr - old(Repr));
    ensures Contents = old(Contents) + {x};

  function method isEmpty(x: T): bool
    requires Valid;
    reads Repr;
    ensures isEmpty() <==> (Contents == {});
}
```

Figure 1: Axiomatic interface specification of a set ADT

Part 3 Stretch Goals

If you are up for a challenge you can try to achieve one or both of the following stretch goals.

(a) For better performance, Dijkstra's algorithm is usually implemented using a priority queue based on Fibonacci heaps. Implement and verify such a priority queue.

(b) Verify that your implementation of Dijkstra's algorithm really computes the shortest paths to all vertices in the given graph. You can look at the implementation of the Shorr-Waite graph marking algorithm provided with the Boogie distribution to get some inspiration how to do this. Though, verifying Dijkstra's algorithm is more difficult and you might hit the limitations of what you can prove with the automated theorem prover underlying Dafny.

I.B CSCI 4951/5951 BIG DATA SYSTEMS COURSE PROJECT: SAMPLE #2

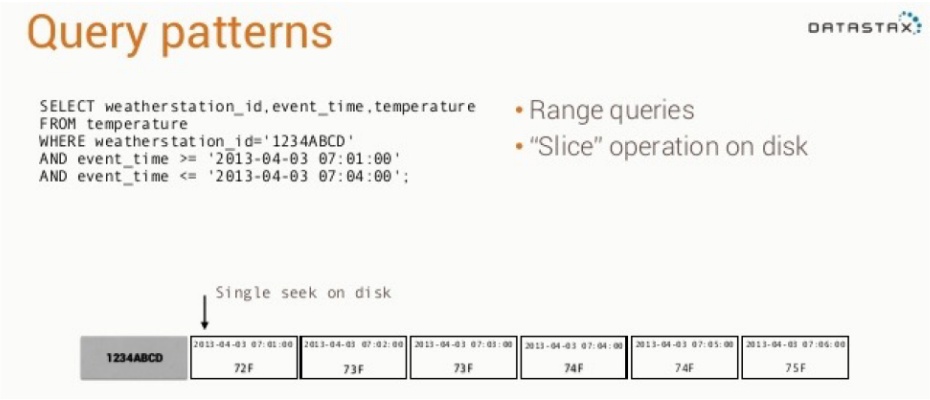
Network State Estimation

- Create a data storage solution for a route guidance system.
- The storage would require collecting raw data about road segments, captured as time series data, and be capable of generating additional fields for their load and speed-related statistics.

Customer	Time	Value1	Value2	Value3
1000111	2011-1-1 00:00:00.00000	0.092	1.30	45.19
1000111	2011-1-1 00:15:00.00000	0.082	1.42	47.82
1000111	2011-1-1 00:30:00.00000	0.090	1.19	46.37
1000111	2011-1-1 00:45:00.00000	0.085	1.23	42.78
1000112	2011-1-1 00:00:00.00000	0.089	1.29	47.23
1000112	2011-1-1 00:15:00.00000	0.094	1.53	49.08
...				

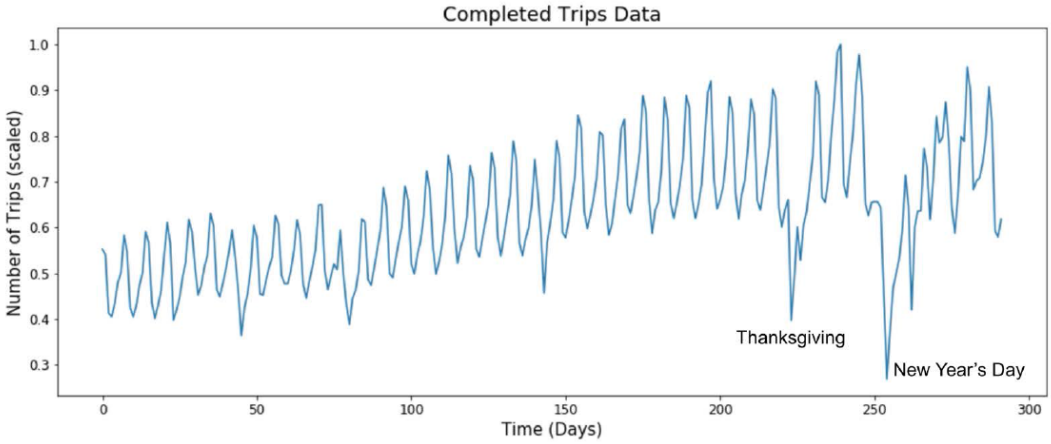
Network State Estimation

- This data would later be queried by an iterative machine learning algorithm to create predictions of future network state.
- The data storage solution would need to support query optimization for time ranges.



Network State Estimation

- You will be provided with a sample of simulated data.
- You will provide a linear regression algorithm that demonstrates the effectiveness of your data storage solution



I.C SAMPLE STUDENT COURSE PROJECT

Course Project Report

Fast Shortest Path Algorithm for Road Network and Implementation

Author:	XXX
Student #:	XXX
E-mail address:	XXX
Supervisor:	Dr. Farnoush Banaei-Kashani

Summary

In the road network application, the A* algorithm can achieve better running time than Dijkstra's. The restricted algorithm can find the optimal path within linear time but the restricted area has to be carefully selected. The selection actually depends on the graph itself. This algorithm can be used in a way that allowing search again by increasing the factor if the first search fails.

Acknowledgments

I would like to thank Professor Banaei-Kashani, who provides me the opportunity to study on this topic, directs and leads me on the right track.

Table of Contents

1. Introduction

Shortest Path problems are inevitable in road network applications such as city emergency handling and drive guiding system, in where the optimal routings have to be found. As the traffic condition among a city changes from time to time and there are usually a huge amounts of requests occur at any moment, it needs to quickly find the solution. Therefore, the efficiency of the algorithm is very important [1, 3 and 5].
Some approaches take advantage of preprocessing that compute results before demanding. These results are saved in memory and could be used directly when a new request comes up. This can be inapplicable if the devices have limited memory and external storage. This project aims only at investigate the single source shortest path problems and intends to obtain some general conclusions by examining three approaches, Dijkstra's shortest path algorithm, Restricted search algorithm and A* algorithm.
To verify the three algorithms, a program was developed under Microsoft Visual C++ environment. The three algorithms was implemented and visually demonstrated. The road network example is a graph data file containing partial transportation data of the Ottawa city.

2. Three Algorithms for Finding Path

2.1 Dijkstra's Shortest Path Algorithm

The Dijkstra's shortest path algorithm is the most commonly used to solve the single source shortest path problem today. For a graph G(V, E), where V is the set of vertices and E is the set of edges, the running time for finding a path between two vertices varies when different data structure are used. This project uses binary heap to implement Dijkstra's algorithm although there are some data structures that may slightly improve the time complexity, such as Fibonacci heap that can purchase time complexity of O(V*log(V)) [6].

```
Dijkstra's shortest path algorithm
for each u ∈ G:
        d[u] = infinity;
        parent[u] = NIL;
End for
        d[s] = 0;  // s is the start point
        H = {s};  // the heap
 while NotEmpty(H) and targetNotFound:
        u = Extract_Min(H);
        label u as examined;
        for each v adjacent to u:
                if d[v] > d[u] + w[u, v]:
                d[v] = d[u] + w[u, v];
                parent[v] = u;
                DecreaseKey[v, H];
```

Time Complexity:
The run time of first for loop is O(V). In each iteration of the while loop, Extract_Min of the heap is logV. The inner for loop iterates each adjacent node of the current node, the total run time is O(E). Therefore, the time complexity of this algorithm is O((V + E)*log(V) = O(E* log(V)). The correctness of this algorithm is well proved in [6].
As the number of nodes in a graph increases, the running time of the applied algorithm will become longer and longer. Usually, a road network of a city has more than 10^4 nodes. A fast shortest path algorithm becomes more desirable.

2.2 Restricted Search Algorithm

Basically, the structure of road networks is relative simple. They are large scaled, sparse and connected graph. When the Dijkstra algorithm is used to find the shortest path, it starts search from the start point and spreads as a circle until the radius arrives the destination. Most searches at the area opposite the direction of destination are useless. M. Fu et al. [4] described an optimal approach to find shortest path for Vehicle Navigation System by physically cutting off area within which the shortest path is not supposed to appear.

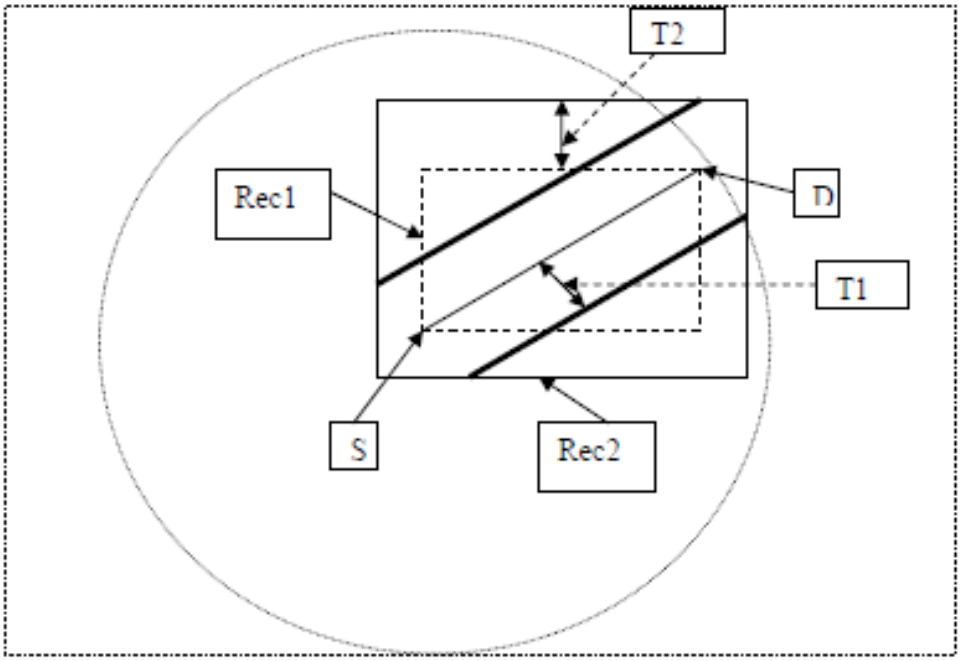


Figure 1, Search Area of Dijkstra and Restricted Search Algorithm
(S: Start point, D: Destination)

Instead of search the entire circle, the Restricted Search Method only search with the small area of the remaining part of rectangle Rec2 cutting off by the two bold straight lines. The rectangle Rec1 has the straight line of S and D as a diagonal and Rec2 is a rectangle extended from Rec1 by a threshold T2. The two straight bold lines parallel the

straight line of SD with a distance of T1. T1 and T2 are two variables that need to be decided to ensure that the optimal path is included within the restricted area. They usually range from 500m to 1500 m.

This project uses the following algorithm to achieve this goal.
Restricted Search Algorithm:

```
for each u ∈ G:
        d[u] = infinity;
        parent[u] = NIL;
End for
        d[s] = 0;
        H = {s};
while NotEmpty(H) and targetNotFound:
        u = Extract_Nin(H);
        label u as examined;
        for each v adjacent to u:
                if outOfRange(v), then continue;
                if d[v] > d[u] + w[u, v], then
                        d[v] = d[u] + w[u, v];
                        parent[v] = u;
                        DecreaseKey[v, H];

Procedure outOfRange(Constraint Area A, Vertex v):
        //A is a polygon given;
        //v is a Vertex being checked;
        Make a straight-line L from v to the right of v;
        Counter = 0;
        For each edge e of A
                if L intersects with e
                        increase Counter by one;
        if Counter is even
                return true;
        else
                return false;
```

The Dijkstra algorithm is used as the same way. However, instead of relaxing all adjacent nodes in each iteration, the algorithm filters out the nodes beyond the restricted area by checking if they are out of range. The checking procedure is implemented by counting the number of intersection of the horizontal line starting from the node to the right with the restricted area. The figure2 illustrates this method. If the number of intersection is odd, the node is within this area, otherwise it is not in the area.

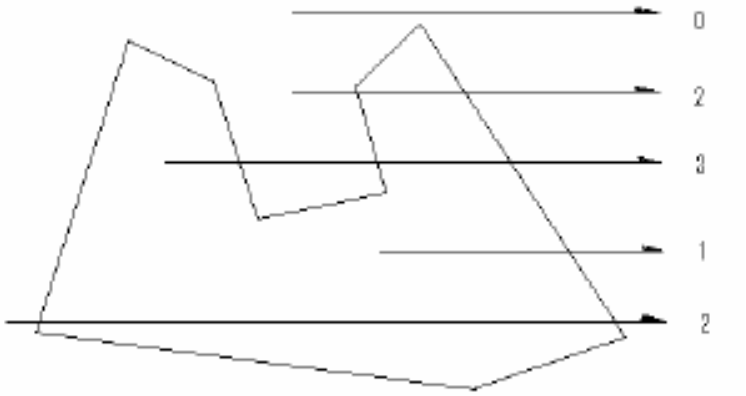


Figure 2, Check point in polygon

Time Complexity:

Suppose the nodes in the road network are distributed evenly.

Let density be C,

V*: the number of vertices examined by using Dijkstra algorithm.

V: the number of vertices examined by using this algorithm.

Soptimal: searched area by using this algorithm.

Sdij: searched area by using Dijkstra algorithm.

K1, K2: the factors of threshold.

Then, the ratio P can be used to describe the improvement of efficiency,

$P = V/V^*$

$= C^* Soptimal / C^* Sdij$

$<= 2 * T1 (R + T2 * \sqrt{2}) / \pi R^2$

$= 2 * K1 * R(R + K2 * R * \sqrt{2}) / \pi R^2$

$= 2 * K1(1 + \sqrt{2} * K2) / \pi$

Since K1 and K2 are small number, the great improvement can be achieved. For instance, if K2 = 0.4, $P \approx K1$.

Correctness:

There are two problems exist in this approach.

(1) The fixed threshold may not get the solution properly. As the distance from start point to the destination increases, the shortest path more likely spreads wider from the straight line of SD.

(2) The city traffic lines have different categories with different driving speed. The high way may be beyond the restricted area but in the shortest path.

To achieve a better solution, this project uses relative thresholds instead of the fixed ones, called factor K1, K2 (threshold / length of SD). To simplify the problem, they have same value in the implementation. The threshold proportionally increases with the distance from start point to the destination in a selected factor. The second problem could be solved by using logical position instead of physical location for each node. For example, upon a node being examined, its logical position will be the accumulation from the logical location of its parent by the cost of the edge between them. The cost of an edge is logical distance that relative to physical distance and road category (see section 3.1). The

logical location of start point is the same as its physical location. All nodes being examined use their logical position to decide if they are within the restricted area. This ensures that the nodes on different roads can be treated equally in searching. However, due to the time constraint, this project will not implement it in the program.
This algorithm actually uses the Dijkstra within the restricted area. It is obvious that a shortest path in this area will be found if the path exists. However, this shortest path may not be the shortest one in the whole area. Thus, it is optimal path with restricted area.

2.3 A* Search

The A* algorithm integrates a heuristic into a search procedure. Instead of choosing the next node with the least cost (as measured from the start node), the choice of node is based on the cost from the start node plus an estimate of proximity to the destination (a heuristic estimate). F. Engineer [2] described this approach to solve the problem of optimal path finding.
This project uses Euclidean distance as estimated distance to the destination. In the searching, the cost of a node V could be calculated as:

f(V) = distance from *S* to V + estimate of the distance to *D*.
= d(V) + h(V,D)
= d(V) + sqrt($(x(V) - x(D))^2 + (y(V) - y(D))^2$)

where x(V), y(D) and x(V), y(D) are the coordinates for node V and the destination node D.

The A* Search algorithm:

```
for each u ∈ G:
        d[u] = infinity;
        parent[u] = NIL;
End for
        d[s] = 0;
        f(V) = 0;
        H = {s};
while NotEmpty(H) and targetNotFound:
        u = Extract_Min(H);
        label u as examined;
        for each v adjacent to u:
                if d[v] > d[u] + w[u, v] , then
                        d[v] = d[u] + w[u, v];
                        p[v] = u;
                        f(v) = d[v] + h(v, D);
                        DecreaseKey[v, H];
```

Time Complexity:
This algorithm does not improve worst case time complexity, but it improves average time complexity. The shortest path search starts from start point and expands node that goes towards the destination. Therefore, the run time is much shorter than the Dijkstra's algorithm.

Correctness:
The algorithm uses the same approach as Dijkstra's except that it uses accumulated cost of edge plus the Euclidean distance from current node to the destination. This value is used to decide the position of a node in the min heap. The one with smallest value will be selected and removed from the heap. In the implementation, this value only affects the searching order. It doesn't modify the edge weights and accumulated distance. The accumulated distance is updated as the same way as Dijkstra when a node relax. Therefore, this algorithm is same as Dijkstra and it is correct.

3. Implementation

This program is developed under Microsoft Visual C++ environment. The three algorithms was implemented and visually demonstrated. The road network example is a graph data file containing partial transportation data of Ottawa city.

3.1 Brief Description

The map of Ottawa city is a directed graph with about 26000 vertices. There are four categories of road in the graph: minor road, regional road, major road and highway. Different type of road has different maximum driving speed. Therefore, physical distance of two points is not enough to describe the path. Instead, a logical distance is used to represent the real distance as well as shortest path. The logical distance is the physical distance times the type of the road segment. Assume:

Speed of highway / speed of minor = 3;
Speed of major / speed of minor = 1.5;
Speed of regional / speed of minor = 2.

Then, the type of minor is 6, regional 3, major 4 and highway 2. For example, if a segment of road is 10, the logical distance is 10 * 2 = 20 for highway, 10 * 6 = 60 for minor. Thus, if two paths have same length, the traveling times are the same. The shortest path is the path has shortest traveling time.

3.2 Main Feature

There are following main features in this program:
1, the window loads and displays the Ottawa city road network. This map is stored in the file "Ottawa_city.gph".
2, the user can drag and move the start point and destination on the window screen by using the mouse.
3, the window can display the shortest path as the user demands.
4, there are four categories of road in this graph: minor, regional, major and highway. They are represented in different colors.

3.3 Run program

1. Start up the program
2. From the menu bar, choose "File", "Open", "Open exist file". From the pop up dialog, select the file "ottawa_city.gph" in the "data" directory. Then, push "ok" button.
3. After the graph is loaded, use the mouse dragging the blue square and two squares show up, which represent start point and destination point respectively.
4. The graph can be zoomed out, zoomed in, moved left, moved right, moved up and moved down by using "page down", "page up", "left", "right", "up" and "down" keys respectively.
5. Press "p" key to toggle showing path between the two points.
6. From the "method" menu, choose either "Dijkstra", "astar", or "restricted" to apply the three algorithms to show the current path. The "restricted" menu has five selections of the factors respecting the absolute distance from start to destination, 0.2, 0.4, 0.6, 0.8 and 1.0.
7. Drag the point again to show the path on current setting.
8. Exit the program by selecting "exit" under "File" menu or closing the window.

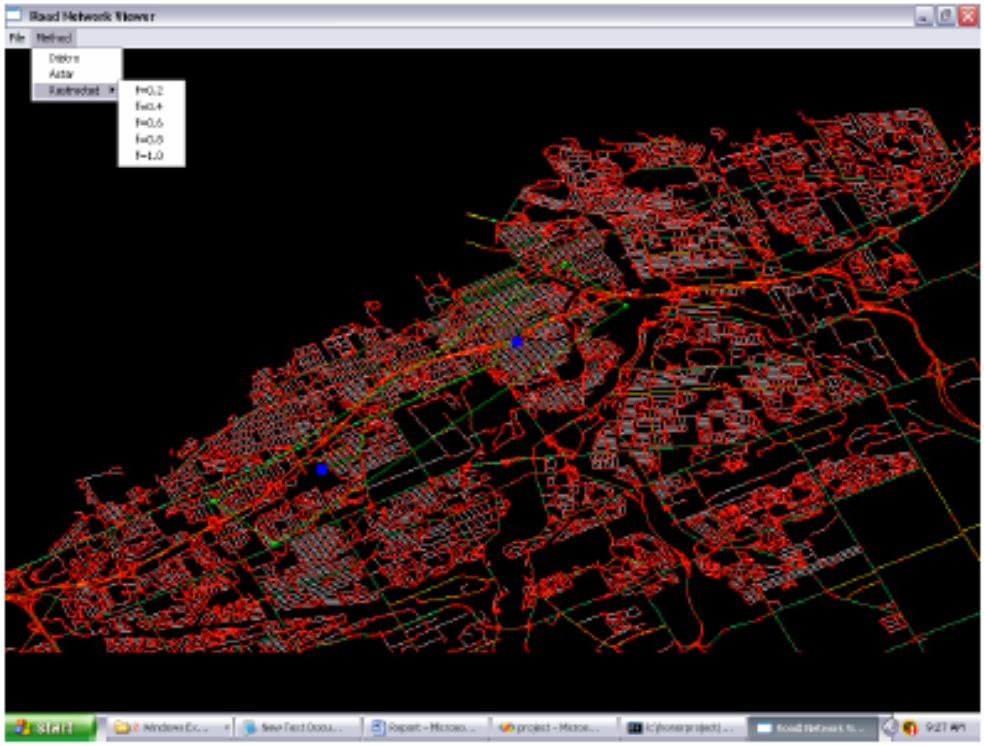


Figure 3, the snap shot of the screen

Figure 4, the view of zoom in

4. Experimentation

The first part compares the run time and accuracy of the three algorithms. The distance is the path length which is represented by logical distance as discussed in section 3.1. The accuracy of a algorithm for a particular distance is the path length found by this algorithm divided by the path length found by Dijkstra.

1). the running time and accuracy of the algorithms with respect to different distances

Distance	5009	7527	11262	13957	15164	19366	21498	25901	30657
Dijkstra	25/20	50/45	55/50	71/70	100/80	160/140	180/120	220/190	350/300
A*	24/19	40/38	45/40	50/45	70/60	110/100	130/100	140/120	250/212
Accuracy	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Restricted	12/10	15/10	20/15	20/18	40/30	60/54	68/40	60/40	60/40
Accuracy	1.28	1.26	1.02	1.25	1.26	1.1	1.2	1.1	1.06

Distance	34250	40903.	44752	50227
Dijkstra	401/300	410/380	410/350	410/300
A*	310/250	360/350	410/350	410/325
Accuracy	1.0	1.0	1.0	1.0
Restricted	100/70	120/50	120/90	170/150
Accuracy	1.2	1.26	1.04	1.05

Note: the restricted algorithm uses factor as 0.4. Ignore the data if it can not find a path.

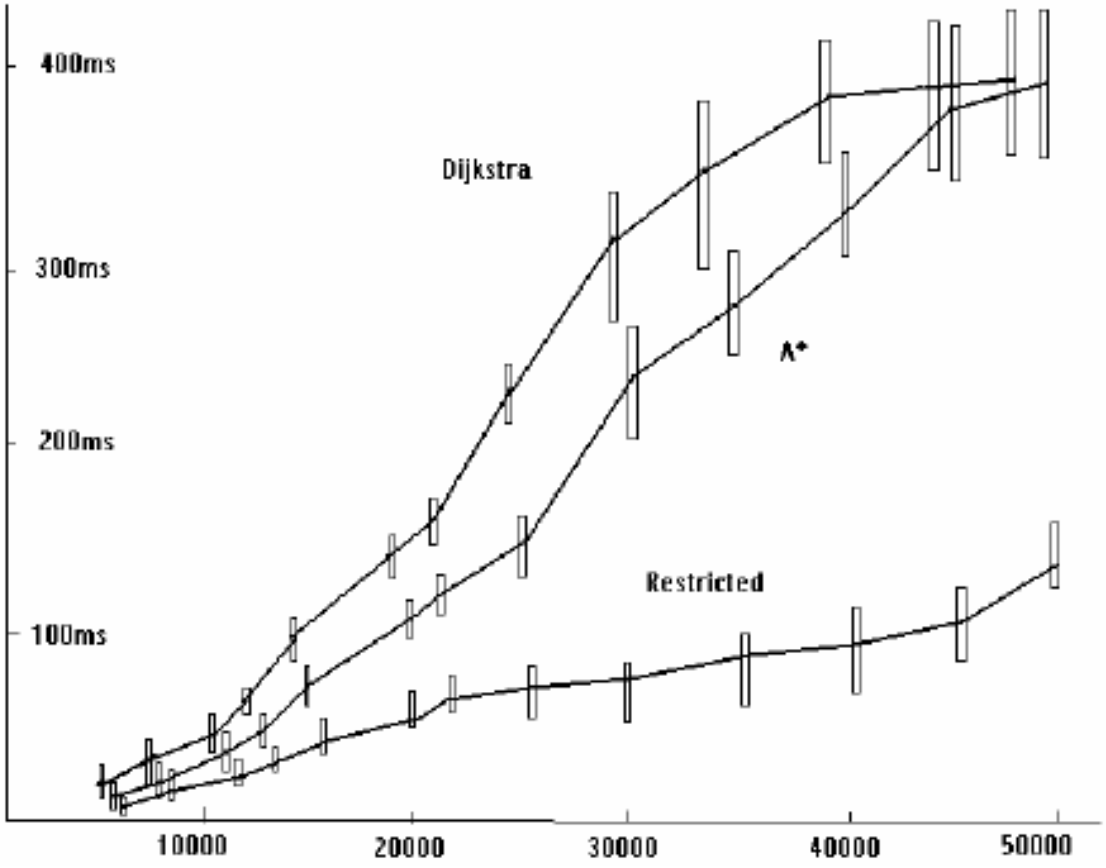


Figure 5, running time and accuracy of the three algorithms

The second part shows the effect of the value of factor on the search result. This is only for the restricted algorithm. Two measurements are used here: the search time of restricted algorithm divided by the search time of Dijkstra, and the path length of restricted algorithm divided by the path length of Dijkstra. The data was obtained by running the algorithm on different distance. So, it is a range from min to max. The "no" in the table means that there is no path can be found at most of time under a particular factor.

2). the effect of the factor from the restricted algorithm

factor	0.2	0.4	0.6	0.8	1.0
Time / Dijkstra time	no	0.23/0.64	0.33/0.69	0.6/1.0	0.8/1.15
Path / Dijkstra path	no	1.5/1.0	1.15/1.0	1.01/1.0	1.0

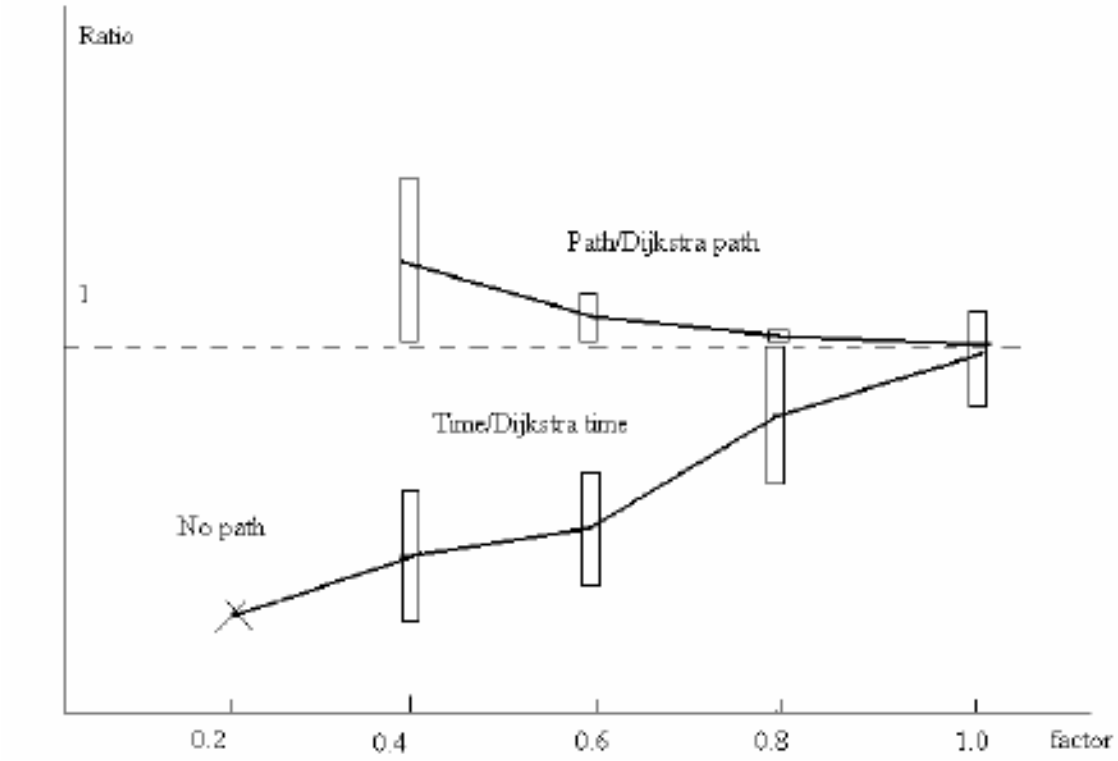


Figure 6, the effect of the factor

In the Figure 5, the running times of both Dijkstra and A* grow up fast until the distance reaching to 30000. This matches the theoretical time complexity. However, since the city map is limited, when the distance over 30000, the searching space dose not increase too much corresponding to the distance because there are no more unsearched points available beyond the searching area. So, the growing rate decreases.
The Figure 5 also shows that the A* algorithm is more efficient than Dijkstra algorithm. It roughly cuts running time to half because it restricts its search area towards destination by using Euclidean heuristic function. Again, as the distance increases, the cutting rate decreases due to the same reason discussed above.
The restricted algorithm shown in Figure 5 uses the factor of 0.4. This factor works in most of time. The result indicates that it running time is linear as long as it can find the path.

Actually, the factor selection is graph dependent. Different graph or different area of the graph have different smallest factor. In this graph, the factor of 0.2 doesn't works at almost all of the times. The Figure 6 gives the effect of the factor on the running time and accuracy. As long as the path can be found, the smaller factor gets faster running time. However, the tradeoff is the less accuracy. As an experiment result, when the factor is up to 1.0, the algorithm tends to find the shortest path but the running time is also the same as Dijkstra's.

5. Conclusion

The A* algorithm can achieve better running time by using Euclidean heuristic function although it theoretical time complexity is still the same as Dijkstra's. It can also guarantee to find the shortest path. The restricted algorithm can find the optimal path within linear time but the restricted area has to be carefully selected. The selection actually depends on the graph itself. The smaller selected area can get less search time but the tradeoff is that it may not find the shortest path or, it may not find any path. This algorithm can be used in a way that allowing search again by increasing the factor if the first search fails.

6. Reference

[1], Yu-Li Chouy H. Edwin Romeijnz Robert L. Smithx. Approximating Shortest Paths in Large-scale Networks with an Application to Intelligent. Transportation Systems. September 27, 1998

[2], Faramroze Engineer. Fast Shortest Path Algorithms for Large Road Networks. Department of Engineering Science. University of Auckland, New Zealand. http://www.esc.auckland.ac.nz/Organisations/ORSNZ/conf36/papers/Engineer.pdf

[3], Roozbeh Shad, Hamid Ebadi, Mohsen Ghods. Evaluation of Route Finding Methods in GIS Application. Dept of Geodesy and Geomatics Eng. K.N.Toosi University of Technology, IRAN. http://www.gisdevelopment.net/technology/gis/ma03202.htm

[4], Fu, Mengyin, Li, Jie, Deng, Zhihong. A practical route planning algorithm for vehicle navigation system. Proceedings of the World Congress on Intelligent Control and Automation (WCICA), v 6, WCICA 2004 - Fifth World Congress on Intelligent Control and Automation, Conference Proceedings, 2004, p 5326-5329

[5], Thomas Willhalm. Speed Up Shortest-Path Computations. January 27, 2005. http://www.mpi-sb.mpg.de/~sanders/courses/algen04/willhalm.pdf

[6] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest and Charles E. Leiserson. Introduction to Algorithms, 2nd edition, McGraw-Hill Higher Education, 2001, ISBN: 0070131511