Spatially-Dependent Reliable Shortest Path Problem (SD-RSPP)

By
Vishal Kanna Anand
Andrew Constantinescu
Sugumar Prabhakaran

AER1516: Robot Motion Planning
University of Toronto
Toronto, Canada

April 21, 2022

Contents

Section 1

Introduction

This project examines the use of two different approaches to solve the spatially-dependent reliable shortest path problem (SD-RSPP) for potential robot motion planning applications. The RSPP is a variation of the shortest path problem that adds a probability to link weights to represent link variability and is a more realistic representation of path planning in a network graph when applied to real world applications such as route planning in a road network. When applying the RSPP to a real world road network problem, the nodes represent intersections and the links with probabilistic weights represent roads with variable traffic and potential route closures. In these scenarios, using a shortest path algorithm such as Dijkstra or A* could potentially result in a sub-optimal route. A RSPP formulation is better suited for this type of problem and different approaches have been studied to solve the RSPP that are outlined further in our literature review section. In this project, we use two specific approaches to solve the RSPP: SDRSP-HA* algorithm outlined by B.Y. Chen et al. (2012) [1] and a mixed-integer programming model using the Gurobi industrial-grade mathematical optimization solver.

SDRSP-HA*

The SDRSP-HA* algorithm solves the spatially-dependent reliable shortest path problem (SD-RSPP), which is an extension of the reliable shortest path problem (RSPP). The paper by Chen expands the RSPP to define the SD-RSPP by modifying the problem such that the travel-time along a link is spatially correlated with neighbouring links. The algorithm constructs a two-level hierarchical approach to take into account the spatial correlation before solving the top hierarchy network using standard dynamic programming methods. The approach from the paper is implemented and evaluated on a custom graph network case study to determine the most reliable path at various confidence levels.

Mixed-Integer Programming Model

B.Y Chen et al. (2012) also provide a general optimization formulation for the RSPP that A. Chen and Ji (2005) [2] solve using a genetic algorithm. We modified this formulation as a mixed-integer programming model in order to solve our SD-RSPP case study and compare results. Mixed-integer programming (also known as mixed-integer linear programming) is a branch of mathematical optimization that originated from linear programming. Similar to linear programming, the problem is formulated in a standard form with a linear objective function that is minimized over a set of decision variables and subject to linear constraints. However, the addition of integer and binary variables increase the problem complexity to NP-complete and prevent the use of linear programming techniques such as the simplex algorithm. As a result, a specialized integer programming software (Gurobi) was used to solve our SD-RSPP mixed-integer programming model.

Robotics Applications

Solving the reliable shortest path problem with spatial dependencies is applicable to many real-world applications. For instance, applications include improving global route planning for self-driving cars, navigating in a busy city with traffic and route-planning for warehouse robots. Cities and distribution centers are constantly growing or becoming busier, thus further motivating a need for solving the reliable shortest path problem. Another potential non-robotic application for the RSPP is network routing for companies like Cisco pioneer to efficiently route data over complex and variable networks [3].

Section 2

Background and Relevance

Shortest Path Problem

The shortest path problem is formally defined in graph theory as finding the shortest-length path between two given nodes in a directed graph $G = (N, A)$, consisting of nodes N and arcs (or links) A between nodes with weights corresponding to their length [4]. A shortest path solution to this problem must satisfy the condition that no other path exists where the sum of the weights of the links along that path are lower than that of the shortest path. This problem has countless applications but particularly in robotics, it is used to represent motion planning problems such as finding the optimal path in a probabilistic road maps.

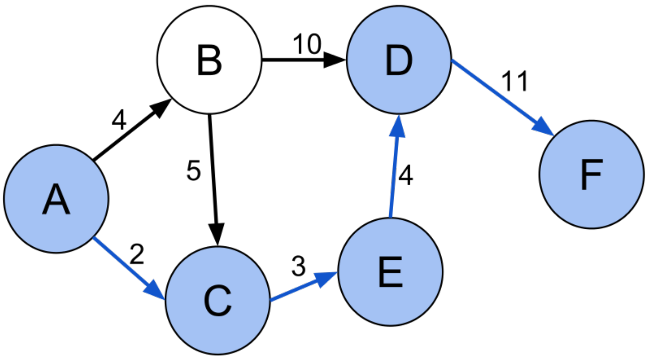

Figure 2.1: Example Directed Graph G=(N,A) with nodes N = {A, B, C, D, E, F} and weighted links A expressed as arrows [5].

Up until the 1950s, other than the exhaustive depth-first search technique, only heuristic approaches existed to solve the shortest path problem. During the 1950s, several optimal methods were found such as a matrix method by Shimbel in 1955 and a linear programming method by Dantzig in 1957. However, the first two major breakthroughs in solving the problem were the Bellman-Ford algorithm in 1958 and Dijkstra's algorithm in 1959. Collaboration between Richard Bellman and Lester Ford led to the dynamic programming approach of value iteration of the cost-to-go from the start node to the goal node. In 1959, Dijkstra published his well-known today algorithm that added a cost-to-come based priority queue to the dynamic programming approach in order to systematically explore nodes and obtain a solution in $O((N + A)log(N))$ time complexity. Finally, Hart, Nilsson and Raphael published the A* algorithm in 1968 to improve Dijkstra's algorithm with a heuristic cost-to-go as part of the priority queue, thus resulting in the most efficient, optimal algorithm for the shortest path problem today [4].

Stochastic Shortest Path Problem

Also in the 1950s, Markov decision process (MDP) was introduced and used by several individuals such as Bellman, Howard, and Shapley to represent problems that could be solved by dynamic programming. In 1962, Eaton and Zadeh first introduced the stochastic shortest path (SSP) problem, which is a variation of the (deterministic) shortest path problem that is represented with MDP. In the SSP problem, nodes represent states, links represent actions, weight representing cost, and a probability term associated with weights represents the transition probability of going from one state to another. This variation of the shortest path problem has particular applications in robotics motion planning and control such as moving a robot across unfamiliar terrain [6].

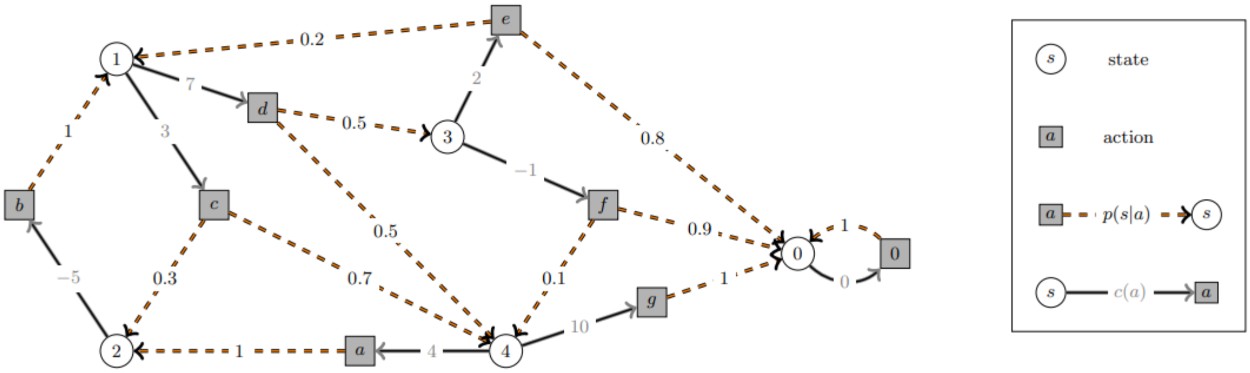

Figure 2.2: Example Markov Decision Process (MDP): weights on solid arrows represent cost and dotted arrows represent transition probabilities [6].

Reliable Shortest Path Problem

The reliable shortest path problem (RSPP) is a potentially more realistic variation of the shortest path problem since it takes probabilistic uncertainty of the weights into consideration, which can be more representative of certain real world applications. This uncertainty is represented by associating a probability distribution for the weights of each link. This type of problem is more realistic for scenarios such as route planning in a city where time between intersections (nodes) varies based on traffic. Another example is data routing through complex networks, where certain data pipelines reach capacity depending on the time-variable demand. Similar to the SSP, the RSPP has robot motion planning applications such as route planning given traffic data in a city for a self-driving car. This particular problem is currently being explored in more depth and is less understood in comparison to the shortest path problem or the SSP. A summary of most relevant research is presented in the next section.

Spatially-Dependent Reliable Shortest Path Problem

Spatial dependency is an extension of the reliable shortest path problem. It is defined as the dependency of the uncertainty of a link to the uncertainty of its neighbouring links. This dependency is mathematically represented for the entire graph by a covariance matrix of all the links in the graph and is a function of the size of the neighborhood, defined by a k-nearest neighbor value. If we define a neighborhood with $k = 1$, this signifies links that are immediately adjacent to the link in consideration. Gajewski and Rilett in 2003 [1] proved the validity of this approach and found that links directly adjacent to a link (lower values for k) have a strong dependence or correlation but links that are spatially distant even on the same path have low dependence. This concept is intuitive if we look at an example of a road network with a blockage at a certain road. In this case, the blockage will significantly affect the travel-time through the neighbouring roads but as you move further and further away, the road segments are less affected.

Section 3

Literature Review

The original solution to the reliable shortest path problem was presented by H. Frank (1969) [7] paper Shortest Paths in Probabilistic Graphs, where he used a Monte Carlo approach to simulate shortest path probability distributions for a network graph with random weighted lengths. These paths were then compared pairwise using hypothesis testing to determine the optimal path. The optimal path was defined by Frank as the path with the highest probability of achieving a travel-time below that of a "travel budget".

Mirchandani (1976) [8] proposed a recursive algorithm solution for a discretized version of Frank's problem. Probabilistic graph representation of problems became increasingly popular during this time period and led to variations. M. Roosta (1982) [3] presents an approach to solve a probabilistic network where the links may disappear. His approach found a reliable path to the goal by calculating the logarithm of maximum probability of the path to choose links, resulting in an optimal safest path.

In the coming years, probabilistic road graphs were solved by minimizing the expected travel cost. Sivakumar and Batta (1994) [9] were the first to solve this problem by minimizing the expected travel cost constrained by a maximum threshold value of variance of the travel cost. After linearizing this constrained shortest path, they solved the problem with Lagrangian relaxation. This paper led to an additional variation of the stochastic shortest path problem: the constrained shortest path (CSP) problem.

Sen et al.(2001) [10] tries to solve the CSP by taking a dual objective approach of minimizing both the mean and variance of the path to the goal. To do this, a series of relaxed quadratic programming models had to be solved. A unique approach to solving the Constrained Shortest Path problem was proposed by Lozano and Medaglia (2013) [11]. Their algorithm uses depth first search along with effective pruning strategies to traverse the road network and come up with a constrained path to the goal. The algorithm is known as the Pulse Algorithm.

In 2005, Chen and Ji [2] introduced the concept of an α-reliable path, where α refers to the confidence level associated with a path that meets the travel budget constraint. The approach used to solve this was a simulation-based genetic algorithm, which identifies the reliability score of each path so that a traveller can choose the right path given their level of risk tolerance. This paper led to a series of future work in the field of reliable path planning.

S. Lim (2008) [12] presented a parametric optimization approach to find the optimal path for the reliable shortest path problem. The solution uses a complex cost function that is then converted into parametric form and then optimized. Nikolova (2009) [13] used a similar parametric approach to transform the network into an optimization problem. However, these methods do not allow comparison of different routes based on the traveller's risk tolerance and is only useful in purely risk-averse situations.

More recently, Nie and Wu (2009) [14] used a dynamic programming approach with a label-correcting algorithm to solve for the reliable shortest path. However, this approach requires significant computation to generate link probabilities for a road network. In 2011, Ji et al. [15] extended Chen and Ji's work of α-reliable path to accommodate more than one confidence requirement for the path. For example, the path may need to satisfy a multi-objective confidence of on time arrival and a confidence for average time while considering spatially correlated travel-times of links. The proposed method to do this was a multi-objective simulation-based genetic algorithm.

Section 4

Implementation

Problem Formulation

The spatially-dependent reliable shortest path problem (SD-RSPP) for a road network G can be solved using a two-level hierarchical network. When the road links in a network are not spatially dependent, then the primal network can be solved trivially using different optimization techniques such as dynamic programming algorithms. However solving a spatially dependent road network becomes significantly more complex using traditional dynamic programming methods, so we equivalently express a primal network G using a ground hierarchy network (H_g) and a top hierarchy network (H_t). Using these constructed networks, the shortest path problem is once again simplified and can be solved using dynamic programming methods.

The primal network G consists of nodes representing intersections, directed edges representing roads that connect some of these nodes, and a covariance matrix representing the relationship between each edge. Each node consists of an (x, y) position in $\mathbb{R}^2$ and each edge consists of a weight representing the mean travel-time between the two nodes it connects.

We model travel-time uncertainty using a normal distribution, where the variance of each link travel-time is stored in the covariance matrix. For a path u originating at node r and terminating at node s, p_u^{rs} represents a set of nodes through which u must pass through. Equivalently, the path can be expressed in terms of edges $p_u^{rs} = [a^1, \dots, a^\lambda]$. The travel-time of a path can then be represented as a normal distribution with a mean travel-time t_u^{rs} and standard deviation σ_u^{rs} calculated as shown in equations 4.0.1 and 4.0.2. In the equations below, λ represents the number of edges in a path and k is the spatial-dependency factor, which dictates the neighbourhood of links that influence a path's standard deviation.

$$t_u^{rs} = \sum_{m=1}^{\lambda} (t_{ij})^m \tag{4.0.1}$$

$$\sigma_u^{rs} = \sqrt{\sum_{m=1}^{\lambda} (\sigma^m)^2 + \sum_{n=1}^{k} \sum_{m=1}^{\lambda-n} 2\mathrm{cov}(a^m, a^{m+n})} \tag{4.0.2}$$

For a path u from node r to node s, we can also calculate the inverse of the cumulative distribution function (CDF) at a confidence level α using equation 4.0.3. The confidence level α represents the traveler's desired on-time arrival probability. For risk-neutral travelers, $\alpha = 0.5$ because we consider only the mean travel-time of each road. Risk-seeking travelers ($\alpha < 0.5$) aim to take the shortest possible path but the probability of successfully completing the path in such a fast time is also lower. Finally, risk-averse travelers ($\alpha > 0.5$) seek really safe paths which care mostly about guaranteeing a certain arrival time.

$$\Phi_{rs,u}^{-1}(\alpha) = t_u^{rs} + z_\alpha \sigma_u^{rs} \tag{4.0.3}$$

It is important to note that the inverse CDF is non-additive: $\Phi_{rs}^{-1}(\alpha) \neq \Phi_{rj}^{-1}(\alpha) + \Phi_{js}^{-1}(\alpha)$, so we cannot use traditional dynamic programming methods such as Dijkstra's algorithm to solve for the most reliable path. This is the main motivation for developing a two-level hierarchical network to equivalently express the primal network.

Algorithms

SDRSP-HA*

From the primal road network G, we first construct the ground hierarchy network H_g, which consists of N directed-in trees (where N is the number of nodes in the primal network). For each directed-in tree, a node from G is used as the root and the remaining nodes all represent paths from this original node. Each subsequent level in each tree consists of a path containing one extra node than the previous level. For instance, the root of the tree consists of a single node, the next level in the tree consists of a path with two nodes, then a path with three nodes, and so on until we reach the level consisting of k nodes (k is the spatial dependency factor). Following this construction, we will have a directed-in tree where all the directed-in leaf nodes are paths with k nodes, which we will be referred to as border nodes. Once all the directed-in trees have been built, we can construct the top hierarchy network H_t which is built using all the border nodes from H_g. Unlike the ground hierarchy, there is only one network in the top hierarchy. Recall that each border node consists of a path, so H_t can be constructed by connecting edges if a path's last $N_p - 1$ nodes appear in another path's first $N_p - 1$ nodes (where N_p represents the number of nodes in the path).

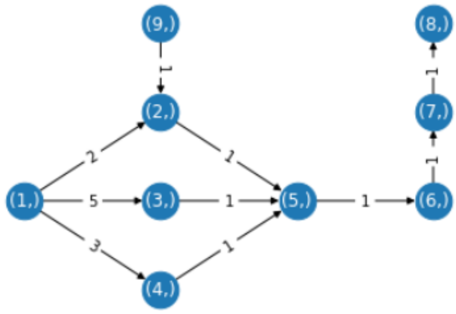

Figure 4.1: Primal network.

A simple example of a primal network and its corresponding ground and top hierarchies for a spatial-dependency factor of $k = 3$ are shown in figures 4.1, 4.2 and 4.3 respectively. Visually, we can identify the borders nodes from the ground hierarchy H_g to be paths with $k = 3$ nodes. These make up the nodes from the top hierarchy H_t. For each node in H_g or H_t that represents a path, we can calculate the path's travel-time distribution through the mean and standard deviation equations previously introduced in equations 4.0.1 and 4.0.2. We can also compute the inverse CDF of a path using equation 4.0.3, which can be used to compare different paths and help determine which one is the shortest, given our risk level.

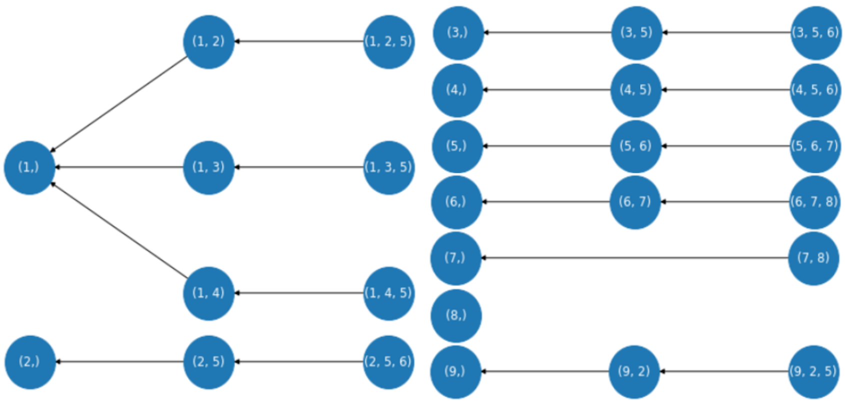

Figure 4.2: Ground hierarchy H_g network for $k = 3$.

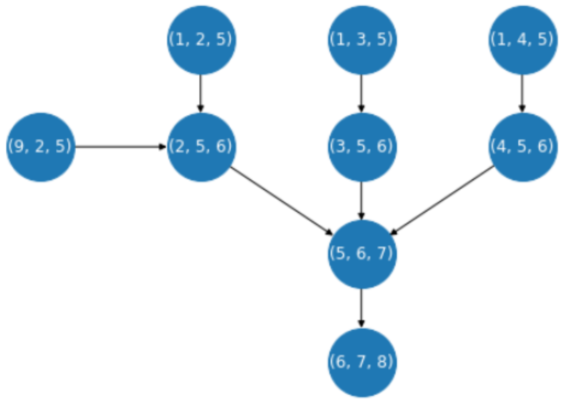

Figure 4.3: Top hierarchy H_t network for $k = 3$.

Using the inverse CDF equation defined in 4.0.3, we can compare different possible paths from the same start node and end node to determine which one is dominant. Figure 4.4 shows the inverse CDF of three paths as a function of arrival probabilities ($0 < \alpha < 1$). From inspecting the plot, it is clear that the green path is best suited for risk-seeking travellers, the blue path is best for risk-neutral travellers and the yellow path is optimal for extremely risk-averse travelers.

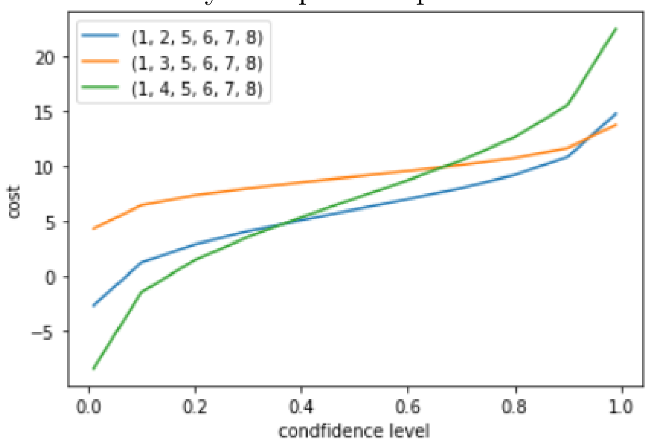

Figure 4.4: Inverse CDF for the three paths from origin (1,) to destination (8,) with $k = 3$.

Algorithm 1 `CHECK_DOMINANCE`

function CHECK_DOMINANCE($G, G_\Sigma, \hat{p}, \hat{P}, \alpha, k$) ▷ Check if $\hat{p}$ is dominated by a path in $\hat{P}$
    $\hat{P}_D$ `= []` ▷ List consisting of all paths dominated by $\hat{p}$
    for `path in` $\hat{P}$ **do**
        if $(\Phi^{-1}(G, G_\Sigma, \hat{p}, \alpha, k) > \Phi^{-1}(G, G_\Sigma, \texttt{path}, \alpha, k))$ **then**
            $\hat{p}$ `is a dominated path`
        else
            $\hat{P}_D$`.append(path)`
        end if
    end for
    $\hat{P}$`.remove(`$\hat{P}_D$`)`
    if $\hat{p}$ `is nondominated` **then**
        $\hat{P}$`.append(`$\hat{p}$`)`
    end if
    `Return` P_D
end function

A path $\hat{p}_u^{ij}$ dominates another path $\hat{p}_v^{ij}$ if and only if $\Phi_{ij,u}^{-1}(\alpha) < \Phi_{ij,v}^{-1}(\alpha)$ for all α. This statement holds true because the inverse CDF of a path represents the travel-time along that path given a probability α of on-time arrival. The function `check_dominance` shown in Algorithm 1 takes as input a path $\hat{p}_u^{ij}$ and a set of paths $\hat{P}^{ij}$ with the same start and goal nodes and returns a set of paths $\hat{P}_D^{ij}$ dominated by $\hat{p}_u^{ij}$. The function also returns whether the path $\hat{p}_u^{ij}$ is non-dominant (meaning $\hat{p}_u^{ij}$ dominates all the paths in $\hat{P}^{ij}$). Algorithm 2 shows the main implementation of SDRSP-HA* which solves the spatially-dependent reliable shortest path problem using a two-level hierarchical network. In comparison to the A* algorithm, this algorithm also uses a heuristic function to find the least-cost path. The SDRSP-HA* heuristic function is calculated for a path u from node i to node j as $F(\hat{p}_u^{ij}, \alpha) = \Phi_{ij,u}^{-1}(\alpha) + h(\hat{p}_u^{ij})$, where $h(\hat{p}_u^{ij})$ represents the euclidean distance from the path's last node j to the goal node s. The algorithm is initialized by calculating the heuristic F for each node in the top hierarchy network and adding each path into a scan eligible set in ascending order based on the heuristic. At all times, the scan eligible set consists of only non-dominated paths.

Algorithm 2 `SDRSP-HA*`

function SDRSP__HA__STAR($G, G_\Sigma, H_g, H_t, r, s, \alpha, k$)
    $\hat{P}^{rj}$ `= []` ▷ Ordered list consisting of all nondominated paths from r to j
    SE `= []` ▷ Ordered list consisting all scan eligible paths
    for `border_node in` H_g **do** ▷ Initialization
        for `child in` H_g`.border_node.children` **do**
            $\hat{p}^{rj} =$ `border_node` $\oplus$ `child`
            $\hat{p}^{rj}.F = \Phi^{-1}(G, G_\Sigma, \hat{p}^{rj}, \alpha, k) + h(\hat{p}^{rj})$
            $\hat{P}^{rj}$`.insort(`$\hat{p}^{rj}$`)`
            SE`.insort(`$\hat{p}^{rj}$`)`
        end for
    end for
    while `True` **do**
        if `(`SE`.empty() == True)` **then** ▷ Path Selection
            `Return -1` ▷ No path found
        end if
        $\hat{p}^{rj} = SE$`.pop`
        if `(j == s)` **then**
            `Return` $\hat{p}^{rj}$ ▷ RSP solution found
        end if
        $\hat{a}^{ij} =$ `get_last_link(`$\hat{p}^{rj}, k$`)` ▷ Get last k nodes in $\hat{p}^{rj}$
        for `child in` H_t`.`$\hat{a}^{ij}$`.children` **do** ▷ Path Extension
            $\hat{p}^{rw} = \hat{p}^{rj} \oplus$ `child`
            $\hat{p}^{rw}.F = \Phi^{-1}(G, G_\Sigma, \hat{p}^{rw}, \alpha, k) + h(\hat{p}^{rw})$
            $\hat{P}_D^{rw}$ `= CHECK_DOMINANCE(`$G, G_\Sigma, \hat{p}^{rw}, \hat{P}^{rw}, \alpha, k$`)`
            SE`.remove(`$\hat{P}_D^{rw}$`)` ▷ Remove dominated paths from scan eligibility
            if `(`$\hat{p}^{rw}$ `is nondominated)` **then**
                SE`.insort(`$\hat{p}^{rw}$`)` ▷ Add nondominated new path to scan eligibility
            end if
        end for
    end while
end function

Once a path is found to be dominated, it is removed from the scan eligible set. The algorithm continuously pops a path from the scan eligible set, extends it by one link and then checks dominance. When checking dominance, all dominated paths get removed and the extended path gets added to the scan eligible set only if it dominates all existing paths. If a valid path exists, this iterative procedure continues until the goal node is reached. Maintaining an order scan eligible set is critical to arriving at the lowest-cost reliable path.

Mixed-Integer Program Optimization

We altering the general formulation presented in Chen et al. (2012) [1] in order to develop a mixed-integer programming (MIP) model to solve the SD-RSPP. The problem must be formulated as a MIP model instead of a linear program due to the requirement to optimize over binary decision variables. For our model, a set of binary variables denoted x_{ij} (known as the path-link incidence variable) represent whether or not the corresponding link a_{ij} is in the most reliable (optimal) path.

Notation

A: set of all links; a_{ij}: link between node i and node j;

$SCS(i)$: set of all successor nodes of node i; $PDS(i)$: set of all predecessor nodes of node i;

r: start node; s: goal node; α: confidence level, where: $\alpha \in [0, 1]$;

z_α: inverse CDF of a standard normal distribution at α confidence level;

$\Phi^{-1}(\alpha)$: inverse CDF at α confidence level of path travel-time;

t_u: mean path travel-time; σ_u: standard deviation of path travel-time;

Objective Function

The objective function of the MIP model minimizes the inverse CDF of the path travel-times from equation 4.0.3 over all decision variables x_{ij}. In order to convert this to a suitable objective function for the model, we dropped the rs superscripts in equation 4.0.3, as the start and end nodes will be defined through linear constraints. The resulting objective function, equation 4.0.4, is a function of the mean path travel-time t_u, z-score for the user defined α confidence level, and the standard deviation of path travel-time σ_u.

$$\min_{x_{ij}} \Phi^{-1}(\alpha) = \min_{x_{ij}}(t_u + z_\alpha \sigma_u) \tag{4.0.4}$$

Since the optimal path u is unknown, a more useful objective function, equation 4.0.5, is obtained by substituting equations 4.0.1 and 4.0.2 in for t_u and σ_u. However, both functions need to be modified as a function of the binary path-link incidence variable x_{ij} instead of path u. The mean portion of the objective function t_u is obtained by summing all link mean travel-times t_{ij} over the set of all links, A. Since each link's mean travel-time is multiplied by the path-link incidence variable x_{ij}, only means that are on the optimal path will be summed. Similarly, the standard deviation portion σ_u is also modified by multiplying the link variance $(\sigma_a)^2$ with the path-link incidence variable x_{ij}, and by multiplying the covariance between two links σ_{ab} with the path-link incidence variables of both links x_{ij} and x_{kl}. This ensures that only variances and covariances that are on the optimal path are summed.

$$\min_{x_{ij}} \sum_{a_{ij} \in A} (t_{ij} \cdot x_{ij}) + z_\alpha \cdot \sqrt{\sum_{a_{ij} \in A} (\sigma_a)^2 \cdot x_{ij} + 2 \cdot \sum_{a_{ij} \in A} \sum_{b_{kl} \in A} (\sigma_{ab} \cdot x_{ij} \cdot x_{kl})} \tag{4.0.5}$$

Next, we need to define the constraints that the MIP model is subject to. The model will optimize the decision variables to minimize the objective function, while satisfying two types of constraints: boundary constraints for variables and general linear equality and inequality constraints.

subject to:

Decision Variable Boundary Constraints
A path-link incidence variable x_{ij} must be generated for each possible link a_{ij}, equation 4.0.6, and their domains must be constrained to fall into the set $\{0, 1\}$, since they are binary variables.

$$x_{ij} \in \{0, 1\}, \quad \forall a_{ij} \in A \tag{4.0.6}$$

Linear Equality Constraints
The last three constraints in equations 4.0.8, 4.0.9, and 4.0.10 ensure that the model enforces the directed nature of the links a_{ij} in the graph. Equation 4.0.8 ensures that there are no predecessors links before node r (start node) and equation 4.0.10 ensures that there are no successors after node s (goal node). Equation 4.0.9 ensures that all links between the start and goal node only follow one direction.

$$\sum_{j \in SCS(i)} x_{ij} - \sum_{k \in PDS(i)} x_{ki} = 1, \quad \forall i = r \tag{4.0.7}$$

$$\sum_{j \in SCS(i)} x_{ij} - \sum_{k \in PDS(i)} x_{ki} = 0, \quad \forall i \neq r; i \neq s \tag{4.0.8}$$

$$\sum_{j \in SCS(i)} x_{ij} - \sum_{k \in PDS(i)} x_{ki} = -1, \quad \forall i = s \tag{4.0.9}$$

Further Model Adjustments to Linearize Objective Function
The last adjustment required for the MIP model occurs in Gurobi-python code and involves 'partially' linearizing the objective function. Gurobi and other MIP solvers can optimize linear and quadratic objective functions but they cannot handle larger exponential powers and other non-linearities such as absolute value and square root functions. For example, our model cannot execute since the standard deviation σ_u is obtained by taking the square root of the variance $(\sigma_u)^2$ portion in equation 4.0.5. This issue is easily solved in Gurobi by adding another constraint using the `model.addGenConstrPow(x, y, N)` function, where `x` is the input variable path variance $(\sigma_u)^2$ and `y` is the output variable path standard deviation σ_u and `N`=0.5 is the exponential power to raise `x` to. By adding this constraint, our MIP model has a quadratic objective function because of the $x_{ij} \cdot x_{kl}$ term within the standard deviation term but this is acceptable by Gurobi. The full Gurobi-python implementation of the MIP model is attached in Appendix A.

Results

Validation of these algorithms is conducted on a more complex primal network, shown in Figure 4.5, with a spatial-influence factor of $k = 3$. This experimental network consists of only 14 nodes, but consists of 21 road edges thus creating significantly more possible paths that a traveler can take. For the experiment we set the start node as $r = 1$ and the goal node as $s = 14$. The edge weights correspond to the mean travel-time between the nodes that are connected. The covariance matrix for this primal network is expressed as G_Σ in 4.0.10. The diagonal elements correspond to the uncertainty in the travel-time for a specific road link, whereas the non-diagonal elements correspond to the covariance between two road links. Table 4.1 outlines the mapping between the primal network road links and the rows/column index in the covariance matrix. For reference, the top-left element of the covariance matrix is in the first row and first column.

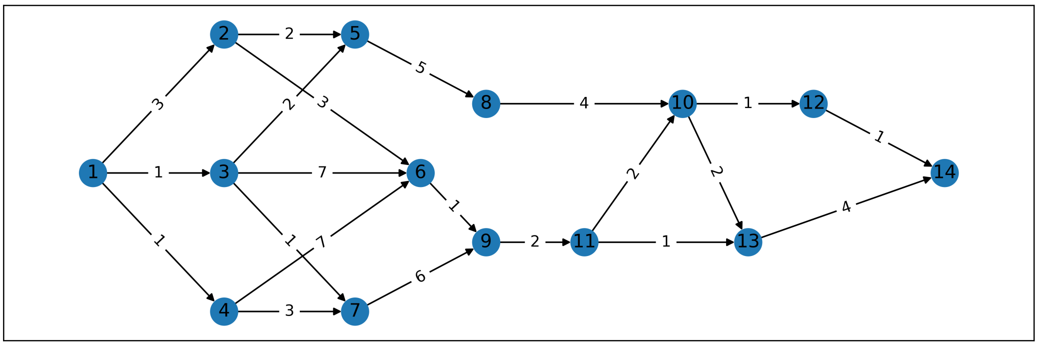

Figure 4.5: Primal network (G) used for algorithm experimentation.

$$
G_\Sigma = \begin{bmatrix}
2 & 1 & 2 & 0 & 0 & 0 & 0 & 0 & 2 & 0 & 9 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0.1 & 2 & 0 & 0 & 0 & 0 & 0 & 2 & 0 & 9 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 2 & 6 & 0 & 0 & 0 & 0 & 0 & 2 & 0 & 9 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0.5 & 0 & 0 & 0 & 0 & 2 & 0 & 9 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 2 & 0 & 0 & 0 & 2 & 0 & 9 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0.2 & 0 & 0 & 2 & 0 & 9 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 2 & 0 & 9 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.1 & 2 & 0 & 9 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 9 & 0 & 9 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 9 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
9 & 9 & 9 & 9 & 9 & 9 & 9 & 9 & 9 & 9 & 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.7 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.5 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.2 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.1
\end{bmatrix} \tag{4.0.10}
$$

Row/Column #	1	2	3	4	5	6	7	8
Road link	(1,2)	(1,3)	(1,4)	(2,5)	(2,6)	(3,5)	(3,6)	(3,7)
Row/Column #	9	10	11	12	13	14	15	16
Road link	(4,6)	(4,7)	(5,8)	(6,9)	(7,9)	(8,10)	(9,11)	(11,10)
Row/Column #	17	18	19	20	21			
Road link	(10,12)	(10,13)	(11,13)	(12,14)	(13,14)			

Table 4.1: Covariance matrix to road link indexing.

SDRSP-HA* Results

Table 4.3 outlines the solutions to the spatially-dependent reliable shortest path problem for varying risk-levels. We see that for travelers seeking a lot of risk and an on-time expected arrival chance of less than 37.4%, the most reliable path would be $\hat{p}_v^{rs} = (1, 3, 5, 8, 10, 12, 14)$. For relatively risk-neutral travelers and an on-time expected arrival probability between 37.4% and 78.0%, the most reliable path is $\hat{p}_u^{rs} = (1, 2, 6, 9, 11, 10, 12, 14)$. For mostly conservative travelers, with a desired arrival probability between 78.0% and 94.2%, the most reliable path is $\hat{p}_w^{rs} = (1, 3, 7, 9, 11, 10, 12, 14)$. Finally, for extremely risk-averse travelers, with a desired arrival probability greater than 94.2%, the most reliable path is $\hat{p}_x^{rs} = (1, 3, 7, 9, 11, 13, 14)$.

Reliable Shortest Path	Probability of on-time expected arrival
$\hat{p}_v^{rs} = (1, 3, 5, 8, 10, 12, 14)$	$0 < \alpha < 0.374$
$\hat{p}_u^{rs} = (1, 2, 6, 9, 11, 10, 12, 14)$	$0.374 \leq \alpha \leq 0.780$
$\hat{p}_w^{rs} = (1, 3, 7, 9, 11, 10, 12, 14)$	$0.780 \leq \alpha \leq 0.942$
$\hat{p}_x^{rs} = (1, 3, 7, 9, 11, 13, 14)$	$0.942 < \alpha < 1$

Table 4.2: SDRSP-HA* Summary - Spatially-dependent ($k = 3$) reliable shortest path for G.

The worst-case computational complexity of the SDRSP-HA* algorithm is $O(\hat{P}^2)$ where $\hat{P}$ represents the number of non-dominated paths in the top hierarchy. The worst-case for this algorithm does not make it feasible for use in large primal networks, such as road transportation city graphs. However it is possible that in practice, $\hat{P}$ is significantly smaller that the maximum possible size, but this was not explored within the scope of this report.

MIP Results

The MIP model achieved almost identical results to the SDRSP-HA* with one discrepancy occurring between $0.780 < \alpha < 0.850$. In the MIP model, the boundary for the most reliable path in row 2 of Table 4.3 extends until $\alpha = 0.850$, while in the SDRSP-HA* implementation, the boundary occurs at $\alpha = 0.780$. This is likely the result of the influence of the $k = 3$ value that was not factored into the calculation of link travel-time standard deviations in the MIP model. For the MIP model, only covariances between links on the path were included.

Reliable Shortest Path	Probability of on-time expected arrival
$\hat{p}_v^{rs} = (1, 3, 5, 8, 10, 12, 14)$	$0 < \alpha < 0.374$
$\hat{p}_u^{rs} = (1, 2, 6, 9, 11, 10, 12, 14)$	$0.374 \leq \alpha \leq 0.850$
$\hat{p}_w^{rs} = (1, 3, 7, 9, 11, 10, 12, 14)$	$0.850 \leq \alpha \leq 0.942$
$\hat{p}_x^{rs} = (1, 3, 7, 9, 11, 13, 14)$	$0.942 < \alpha < 1$

Table 4.3: MIP Model Results Summary - Spatially-dependent reliable shortest path for G.

Section 5

Conclusion and Future Work

The two-level hierarchical algorithm and the MIP model can both solve the reliable shortest path problem with spatial dependency between road links with very similar results. Effectively, risk-seeking travelers (lower value of α) will follow paths with larger variances because of the small probability that their travel time is shortest. In contrast, risk-averse travelers (larger value of α) will follow paths with minimal variance because the travel-time along these paths are more certain. The slight difference in the case study results between the two algorithms can be attributed to how spatial-dependency is taken into consideration. For the SDRSP-HA* algorithm, all k neighbouring links are considered, regardless of whether the neighbours are also on the path. However in the MIP model, for problem formulation specification, we consider all neighbours but only for links along the considered path.

Future Work

There are several points of interest that were not explored within the scope of this project but would provide for some interesting future research work:

In particular, how these two algorithms extend to large road networks would determine the impact of their respective time complexities and their overall usefulness in route-planning. The case study that was analyzed is too small for practical applications but due to challenges obtaining route probability data, it was suitable to prove both algorithms.

Secondly, how the spatial-influence factor k affect the computational complexity of the algorithms is an interesting question. Incorporating spatial-dependency in road networks provides more accurate solutions as it better replicates the real world. This is due to the fact that typically with road congestion, we observe that the traffic on neighbouring streets is typically also affected. However, increasing the value of k also increases our algorithm run-time because we have to consider more neighbours. It appears that there is a trade-off between solution accuracy and performance, which would be interesting to investigate further. For example, the MIP model had a simple method to incorporate link covariances and produced very fast results (<0.01s). However, since MIP problems are NP-complete, significantly large road networks may be intractable.

Finally, examining the scenario of travel-time distributions that vary with time could be useful. In our problem formulation, each road link consisted of a constant travel time distribution. However in real-world applications, the travel-time distribution for a road will vary depending on the time of day. For instance, during rush-hour, the travel-time distribution of a road will have a larger mean and larger variance. However, typically in the middle of the night, this distribution will have a smaller mean with more certainty. During planning problems with longer paths that span multiple hours, it may not be valid to assume that the travel-time distributions remain constant throughout the entire time.

Bibliography

[1] B. Y. Chen , W.H.K. Lam , A. Sumalee, and Z. Li, "Reliable shortest path finding in stochastic networks with spatial correlated link travel times", *International Journal of Geographical Information Science*, vol. 26, no. 2, pp. 365-386, 2012. Available: https://doi.org/10.1080/13658816.2011.598133. [Accessed: February 14, 2022].

[2] A. Chen and Z. Ji, "Path finding under uncertainty," *Journal of Advanced Transportation*, vol. 39, no. 1, pp. 19-37, 2005. Available: https://doi.org/10.1002/atr.5670390104. [Accessed: April 14, 2022].

[3] M. Roosta, "Routing through a network with maximum reliability," *Journal of Mathematical Analysis and Applications*, vol. 88, no. 2, pp. 341-347, 1982. Available: https://doi.org/10.1016/0022-247X(82)90197-4. [Accessed: April 18, 2022].

[4] A. Schrijver, "On the History of the Shortest Path Problem," *Documenta Mathematica*, Extra Volume ISMP, pp. 155-167, 2012. Available: CiteSeerx, http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.398.9778. [Accessed: April 14, 2022].

[5] A. Kalinin, "Shortest path with directed weights," *Wikipedia*, December 3, 2012. [Image file]. Available: https://en.wikipedia.org/wiki/Shortest_path_problem/media/File: Shortest_path_with_direct_weights.svg. [Accessed: April 14, 2022].

[6] M. Guillot and G. Stauffer, "The Stochastic Shortest Path Problem: A polyhedral combinatorics perspective," *HAL Open Science*, 2017. Available: https://hal.archives-ouvertes.fr/hal-01591475. [Accessed: April 14, 2022].

[7] H. Frank, "Shortest Paths in Probabilistic Graphs," *Operations Research*, vol. 17, no. 4, pp. 583-599, 1969. Available: https://www.jstor.org/stable/168536. [Accessed: April 17, 2022].

[8] P.B. Mirchandani, "Shortest distance and reliability of probabilistic networks," *Computers & Operational Research*, vol. 3, no. 4, pp. 347–355, 1976. Available: https://doi.org/10.1016/0305-0548(76)90017-4. [Accessed: April 17, 2022].

[9] R.A. Sivakumar and R. Batta, "The variance-constrained shortest path problem," *Transportation Science*, vol. 28, no. 4, pp. 309-316, 1994. Available: https://doi.org/10.1287/trsc.28.4.309. [Accessed: April 17, 2022].

[10] S. Sen, R. Pillai, S. Joshi, and A.K. Rathi, "A mean-variance model for route guidance in advanced traveler information systems," *Transportation Science*, vol. 35, no. 1, pp. 37–49, 2001. Available: https://doi.org/10.1287/trsc.35.1.37.10141. [Accessed: April 17, 2022].

[11] L. Lozano and A.L. Medaglia, "On an exact method for the constrained shortest path problem," *Computers & Operational Research*, vol. 40, no. 1, pp. 378–384, 2013. Available: https://doi.org/10.1016/j.cor.2012.07.008. [Accessed: April 17, 2022].

[12] S. Lim, Traffic Prediction and Navigation Using Historical and Current Information, M.S [Thesis], Massachusetts Institute of Technology, Cambridge, MA, United States of America, 2008. [Online]. Available: MIT Libraries DSpace@MIT.

[13] E.V. Nikolova, Strategic Algorithms, PhD [Dissertation], Massachusetts Institute of Technology, Cambridge, MA, United States of America, 2009. [Online]. Available: MIT Libraries DSpace@MIT.

[14] Y.M. Nie and X. Wu, "Reliable a Priori Shortest Path Problem with Limited Spatial and Temporal Dependencies," in *Transportation and Traffic Theory 2009: Golden Jubilee*, W.H.K. Lam, S.C. Wong, and H.K. Lo, Ed. New York: Springer, 2009, pp. 169-195. [Online]. Available: SpringerLink.

[15] Z. Ji, Y.S. Kim, and A. Chen, "Multi-objective α-reliable path finding in stochastic networks with correlated link costs: a simulation-based multi-objective genetic algorithm approach (SMOGA)," *Expert Systems Applications*, vol. 38, no. 3, pp. 1515–1528, 2011. Available: https://doi.org/10.1016/j.eswa.2010.07.064. [Accessed: April 17, 2022].

[16] Y. Zhang and A. Khani, "An algorithm for reliable shortest path problem with travel time correlations," *Tranportation Research Part B: Methodological*, vol. 121, pp. 92-113, 2019. Available: https://doi.org/10.1016/j.trb.2018.12.011. [Accessed: April 14, 2022].

[17] L. Santos, J. Coutinho-Rodrigues, and J.R. Current, "An improved solution algorithm for the constrained shortest path problem," Transportation Research Part B: Methodological, vol. 41, no. 7, pp. 756-771, 2007. Available: https://doi.org/10.1016/j.trb.2006.12.001. [Accessed: April 18, 2022].

Appendix A

Python scripts

aer1516_project_sdrsp_ha_star.py (Two-Level Hierarchy).

```python
# -*- coding: utf-8 -*-
"""aer1516_project_sdrsp_ha_star.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/159pvPPX6q6dCtC_H8PA9pWmvpl7W3Jdz

## **Appendix A - Spatially-Dependent Reliable Shortest Path Problem Hierarchial A* algorithm**

**Course:** AER1516 - Motion Planning for Robotics

**Due:** 22 April 2022

**Team:** Vishal Kanna Annand, Andrew Constantinescu, Sugumar Prabhakaran

### **Introduction**

This algorithm implementation solves the spatially-dependent reliable shortest path problem. This is accomplished by constructing a two-level hierarchy network used to compare the dominance between all paths.  Please see Section 4 of our paper for details on the implementation.
"""

# Import necessary modules
import numpy as np
import scipy.stats as st
import matplotlib.pyplot as plt
import networkx as nx
import bisect

# Build a primal network and draw it for visualization
def construct_road_network(index):
  if index != 0 and index != 1 and index != 2:
    return -1

  if index == 0:
    road_network = nx.DiGraph()
    road_network.add_node((1,), x=0, y=0)
    road_network.add_node((2,), x=1, y=1)
    road_network.add_node((3,), x=1, y=0)
    road_network.add_node((4,), x=1, y=-1)
    road_network.add_node((5,), x=2, y=0)
    road_network.add_node((6,), x=3, y=0)
    road_network.add_node((7,), x=3, y=1)
    road_network.add_node((8,), x=3, y=2)
    road_network.add_node((9,), x=1, y=2)
    road_network.add_edge((1,), (2,), cov_key=0, weight=2)
    road_network.add_edge((1,), (3,), cov_key=1, weight=5)
    road_network.add_edge((1,), (4,), cov_key=2, weight=3)
```

```
    road_network.add_edge((2,), (5,), cov_key=3, weight=1)
    road_network.add_edge((3,), (5,), cov_key=4, weight=1)
    road_network.add_edge((4,), (5,), cov_key=5, weight=1)
    road_network.add_edge((5,), (6,), cov_key=6, weight=1)
    road_network.add_edge((6,), (7,), cov_key=7, weight=1)
    road_network.add_edge((7,), (8,), cov_key=8, weight=1)
    road_network.add_edge((9,), (2,), cov_key=9, weight=1)
    road_network_cov = np.asarray([[10, 0.5, -0.5, 0, 0, 0, 0, 0, 0,
       0],
                                   [0.5, 0.1, 1.2, 0, 0, 0, 0, 0, 0, 0],
                                   [-0.5, 1.2, 40, 0, 0, 0, 0, 0, 0, 0],
                                   [0, 0, 0, 1, 0, 0, 0, 0, 0, 0],
                                   [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
                                   [0, 0, 0, 0, 0, 1, 0, 0, 0, 0],
                                   [0, 0, 0, 0, 0, 0, 1, 0, 0, 0],
                                   [0, 0, 0, 0, 0, 0, 0, 1, 0, 0],
                                   [0, 0, 0, 0, 0, 0, 0, 0, 1, 0],
                                   [0, 0, 0, 0, 0, 0, 0, 0, 0, 1]])
  if index == 1:
    road_network = nx.DiGraph()
    road_network.add_node((1,), x=0, y=0)
    road_network.add_node((2,), x=1, y=1)
    road_network.add_node((3,), x=1, y=0)
    road_network.add_node((4,), x=1, y=-1)
    road_network.add_node((5,), x=2, y=0)
    road_network.add_edge((1,), (2,), weight=2, cov_key=0)
    road_network.add_edge((1,), (3,), weight=3, cov_key=1)
    road_network.add_edge((1,), (4,), weight=4, cov_key=2)
    road_network.add_edge((2,), (3,), weight=2, cov_key=3)
    road_network.add_edge((3,), (5,), weight=4, cov_key=4)
    road_network.add_edge((4,), (5,), weight=4, cov_key=5)
    road_network_cov = np.asarray([[2, -1, -1, 2, 0.3, -0.2],
                                    [-1, 1, -0.5, -1, 1.5, -0.6],
                                    [-1, -0.5, 1, -0.3, -0.4, 0.5],
                                    [2, -1, -0.3, 2, 2, -0.4],
                                    [0.3, 1.5, -0.4, 2, 6, -1.5],
                                    [-0.2, -0.6, 0.5, -0.4, -1.5, 1]])

  if index == 2:
    road_network = nx.DiGraph()
    road_network.add_node((1,), x=1, y=3)
    road_network.add_node((2,), x=3, y=4)
    road_network.add_node((3,), x=3, y=3)
    road_network.add_node((4,), x=3, y=2)
    road_network.add_node((5,), x=5, y=4.5)
    road_network.add_node((6,), x=5, y=3.5)
    road_network.add_node((7,), x=5, y=2)
    road_network.add_node((8,), x=7, y=4)
    road_network.add_node((9,), x=7, y=2)
    road_network.add_node((10,), x=9, y=4)
    road_network.add_node((11,), x=9, y=2)
    road_network.add_node((12,), x=11, y=4)
    road_network.add_node((13,), x=11, y=2)
    road_network.add_node((14,), x=13, y=3)
    road_network.add_edge((1,), (2,), cov_key=0, weight=3)
    road_network.add_edge((1,), (3,), cov_key=1, weight=1)
    road_network.add_edge((1,), (4,), cov_key=2, weight=1)
```

```
    road_network.add_edge((2,), (5,), cov_key=3, weight=2)
    road_network.add_edge((2,), (6,), cov_key=4, weight=3)
    road_network.add_edge((3,), (5,), cov_key=5, weight=2)
    road_network.add_edge((3,), (6,), cov_key=6, weight=7)
    road_network.add_edge((3,), (7,), cov_key=7, weight=1)
    road_network.add_edge((4,), (6,), cov_key=8, weight=7)
    road_network.add_edge((4,), (7,), cov_key=9, weight=3)
    road_network.add_edge((5,), (8,), cov_key=10, weight=5)
    road_network.add_edge((6,), (9,), cov_key=11, weight=1)
    road_network.add_edge((7,), (9,), cov_key=12, weight=6)
    road_network.add_edge((8,), (10,), cov_key=13, weight=4)
    road_network.add_edge((9,), (11,), cov_key=14, weight=2)
    road_network.add_edge((11,), (10,), cov_key=15, weight=2)
    road_network.add_edge((10,), (12,), cov_key=16, weight=1)
    road_network.add_edge((10,), (13,), cov_key=17, weight=2)
    road_network.add_edge((11,), (13,), cov_key=18, weight=1)
    road_network.add_edge((12,), (14,), cov_key=19, weight=1)
    road_network.add_edge((13,), (14,), cov_key=20, weight=4)
    road_network_cov = np.asarray([[  2,   1,   2,   0,   0,   0,   0,   0,   2,   0,   9,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
                                   [  1, 0.1,   2,   0,   0,   0,   0,   0,   2,   0,   9,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
                                   [  1,   2,   6,   0,   0,   0,   0,   0,   2,   0,   9,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
                                   [  0,   0,   0, 0.5,   0,   0,   0,   0,   2,   0,   9,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
                                   [  0,   0,   0,   0,   2,   0,   0,   0,   2,   0,   9,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
                                   [  0,   0,   0,   0,   0, 0.2,   0,   0,   2,   0,   9,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
                                   [  0,   0,   0,   0,   0,   0,   1,   0,   2,   0,   9,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
                                   [  0,   0,   0,   0,   0,   0,   0, 0.1,   2,   0,   9,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
                                   [  2,   2,   2,   2,   2,   2,   2,   2,   9,   0,   9,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
                                   [  0,   0,   0,   0,   0,   0,   0,   0,   0,   1,   9,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
```

```
                                             [  9,   9,   9,   9,   9,   9,   9,   9,   9,   9,   2,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
                                             [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   3,   0,   0,   0,   0,   0,   0,   0,   0,   0],
                                             [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0, 0.7,   0,   0,   0,   0,   0,   0,   0,   0],
                                             [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   1,   0,   0,   0,   0,   0,   0,   0],
                                             [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   2,   0,   0,   0,   0,   0,   0],
                                             [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   1,   0,   0,   0,   0,   0],
                                             [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   1,   0,   0,   0,   0],
                                             [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0, 0.5,   0,   0,   0],
                                             [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0, 0.2,   0,   0],
                                             [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   1,   0],
                                             [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0, 0.1]])
    # draw network
    positions = {i:[road_network.nodes[i]['x'],road_network.nodes[i]['y']] for i in list(road_network.nodes)}
    t_ij = {(i, j):road_network.edges[(i,j)]['weight'] for (i,j) in list(road_network.edges)}
    nx.draw_networkx(road_network, positions, with_labels=True, font_color='white', node_size=600)
    nx.draw_networkx_edge_labels(road_network, pos=positions, edge_labels=t_ij)
    plt.show()
    return road_network, road_network_cov

# Helper function for debugging
```

```
def print_ground_hierarchy(Hg):
  for tree in Hg:
    print(tree.adj)

# Construct the ground hierarchy network
def build_Hg(G, k=3, draw_Hg=False):
  Hg = []
  for n in G.nodes:
    Hg_n = nx.DiGraph()
    Hg_n.add_node(n)

    parents = [n]
    for i in range(1, k):
      next_parents = []
      for p in parents:
        p_number = (p[-1],)
        children = list(G.successors(p_number))
        for c in children:
          n2 = p + c
          Hg_n.add_edge(n2, p)
          next_parents.append(n2)
      parents = next_parents

    Hg.append(Hg_n)
    if draw_Hg == True:
      nx.draw(Hg[0], with_labels=True, font_color='white', node_size
        =3000)
      # print_ground_hierarchy(Hg[0])
  return Hg

# Function that takes as input a directed-in-tree from the ground
   hierarchy Hg and returns all the border nodes, which are to be used
   as the nodes in the top hierarchy Ht
def get_border_nodes(tree, k=3):
  border_nodes = []
  for in_deg_node in tree.in_degree:
    node = in_deg_node[0]
    num_in_nodes = in_deg_node[1]
    if num_in_nodes == 0 and len(nx.descendants_at_distance(tree, node,
        k-1)) == 1:
      border_nodes.append(node)
  return border_nodes

# Construct the top hierarchy network
def build_Ht(G, Hg, k=3, draw_Ht=False):
  if k == 1:
    return G
  Ht = nx.DiGraph()
  nodes = []
  for tree in Hg:
    nodes = nodes + get_border_nodes(tree, k)   # create list of all
        the border nodes from all the trees in the ground hierarchy Hg
  Ht.add_nodes_from(nodes)
  for node_i in Ht.nodes:
    for node_j in Ht.nodes:
      if (node_i[1:] == node_j[0:-1]):
        Ht.add_edge(node_i, node_j)
```

```python
  if draw_Ht == True:
    positions = {(1,2,5):[1,8], (1,3,5):[3,8], (1,2,6):[5,8], (1,3,6):[7,8], (1,4,6):[9,8], (1,3,7):[11,8], (1,4,7):[13,8], (2,5,8):[1,6], (3,5,8):[3,6],
                 (2,6,9):[5,6], (3,6,9):[7,6], (4,6,9):[9,6], (3,7,9):[11,6], (4,7,9):[13,6], (6,9,11):[5,4], (7,9,11):[12,4], (5,8,10):[2,2], (9,11,10):[5,2],
                 (9,11,13):[12,2], (8,10,12):[1,0], (11,10,12):[3,0], (8,10,13):[5,0], (11,10,13):[9,0], (10,12,14):[2,-2], (10,13,14):[7,-2], (11,13,14):[12,-2]}
    nx.draw(Ht, positions, with_labels=True, font_color='white', node_size=2500)
    print(Ht.adj)
  return Ht

# Calculate the euclidean distance between the last node in the path and the goal node
def calc_euclidean_distance(G, goal_node):
  d = {}
  for node in G.nodes:
    e_dist = np.sqrt((G.nodes[node]['x'] - G.nodes[goal_node]['x'])**2 + (G.nodes[node]['y'] - G.nodes[goal_node]['y'])**2)
    d[node] = np.round(e_dist,2)
  return d

# Calculate the mean travel time for a given path
def travel_time_mean(G, path):
    mean = 0
    for i in range(0, len(path)-1):
      edge = ((path[i],),(path[i+1],))
      mean += G.edges[edge]['weight']
    return mean

# Calculate the travel time covariance for a given path
def travel_time_stdv(G, Sigma, path, k=3):
    var = 0
    cov = 0
    lamb = len(path)-1
    for i in range(0, lamb):
      edge = ((path[i],),(path[i+1],))
      var += Sigma[G.edges[edge]['cov_key']][G.edges[edge]['cov_key']]

    for n in range(1, k+1):
      for m in range(1, lamb - n + 1):
        edge_1 = ((path[m-1],),(path[m],))
        edge_2 = ((path[m+n-1],),(path[m+n],))
        cov += 2*Sigma[G.edges[edge_1]['cov_key']][G.edges[edge_2]['cov_key']]

    stdv = np.round(np.sqrt(var + cov), 3)
    return stdv

# Calculate the inverse of the cumulative distribution function (CDF) of path travel time at a confidence level of "alpha"
def inv_cdf(G, Sigma, path, alpha, k=3):
  weight = travel_time_mean(G, path) + st.norm.ppf(alpha)*travel_time_stdv(G, Sigma, path, k)
```

```
  return np.round(weight, 2)

# Some helper functions
def print_Pkey(nd_P, key):
  for i in range(0, len(nd_P[key])):
    print("P ", i, ", nodes:", nd_P[key][i].nodes, ", h:", nd_P[key][i].h, ", F:", nd_P[key][i].F, ",tt_mu:", nd_P[key][i].tt_mu)

def print_SE(scan_eligible_set):
  for i in range(0, len(scan_eligible_set)):
    print("SE ", i, ", nodes:", scan_eligible_set[i].nodes, ", h:", scan_eligible_set[i].h, ", F:", scan_eligible_set[i].F, ",tt_mu:", scan_eligible_set[i].tt_mu)

# Define path class to consist of:
# nodes: a list of nodes in the path
# h: the euclidean distance between the last node in the path and the goal node
# F: the heuristic function
# tt_mu: the mean travel time of the path
class Path:
  def __init__(self, nodes, h, F, tt_mu):
    self.nodes = nodes
    self.h = h
    self.F = F
    self.tt_mu = tt_mu

# Returns the last "k" nodes from a path
def get_last_link(path, k):
  link = ()
  for i in range(0,k):
    link = (path.nodes[-1-i],) + link
  return link

# Check dominance of a path
def check_dominance(G, G_cov, p_hat, P_hat_k, alpha, k):
  P_D = []
  nondominated = True
  for i in range(0, len(P_hat_k)):
    if (inv_cdf(G, G_cov, p_hat.nodes, alpha, k) > inv_cdf(G, G_cov, P_hat_k[i].nodes, alpha, k)):
      nondominated = False
    else:
      P_D.append(P_hat_k[i])
  for j in range(0, len(P_D)):
    P_hat_k.remove(P_D[j])
  if len(P_hat_k) == 0:
    P_hat_k.append(p_hat)

  return nondominated, P_hat_k, P_D

# Main algorithm
# G: a graph representing a road network
# G_cov: the covariance matrix between the links in the road network
# r: the origin node
# s:the destination node
# alpha: the level of confidence/risk
```

```
# k: the spatial influence factor
def sdrsp_ha_star(G, G_cov, r, s, alpha=0.5, k=3):
  # Step 1: Initialization
  Hg = build_Hg(G, k)   # construct the ground hierarchy
  Ht = build_Ht(G, Hg, k)  # construct the top hierarchy
  P = {}    # initialize P_hat_rj (dict)
  tt_mean_sorted = []
  SE = []   # initialize scan eligible set (list)
  F_sorted = []
  hs = calc_euclidean_distance(G, s)  # calculate the heuristic for each node

  origin_border_nodes = get_border_nodes(Hg[list(G.nodes).index(r)], k)  # obtain the border nodes from the tree rooted at the origin
  for border_node in origin_border_nodes:
    for child in list(Ht.successors(border_node)):
      new_path = border_node + tuple(set(child) - set(border_node))
      h = hs[(new_path[-1],)]
      F = inv_cdf(G, G_cov, new_path, alpha, k)
      tt_mean = travel_time_mean(G, new_path)
      new_path_obj = Path(new_path, h, F, tt_mean)
      key = (new_path_obj.nodes[0], new_path_obj.nodes[-1])
      bisect.insort(tt_mean_sorted, tt_mean)
      idx_tt = tt_mean_sorted.index(tt_mean)
      if key in P:
        P[key].insert(idx_tt, new_path_obj)
      else:
        P[key] = [new_path_obj]
      bisect.insort(F_sorted, F)
      idx = F_sorted.index(F)
      SE.insert(idx, new_path_obj)

  while True:
    # Step 2: Path selection
    if (len(SE) == 0):
      print("Scan eligible set is empty! No solution found.")
      break
    curr_path = SE.pop(0)
    if ((curr_path.nodes[-1],) == s):
      print("Path from SE contains goal node!")
      print("Solution found:", curr_path.nodes)
      break

    # Step 3: Path extension
    curr_path_last_link = get_last_link(curr_path,k)
    for child in list(Ht.successors(curr_path_last_link)):
      new_path = curr_path.nodes + tuple(set(child) - set(curr_path.nodes))
      h = hs[(new_path[-1],)]
      F = inv_cdf(G, G_cov, new_path, alpha, k)
      tt_mean = travel_time_mean(G, new_path)
      new_path_obj = Path(new_path, h, F, tt_mean)
      key = (new_path_obj.nodes[0], new_path_obj.nodes[-1])

      # Check dominance
      if key in P:
        nondominated, P[key], P_D = check_dominance(G, G_cov,
```

```
            new_path_obj, P[key], alpha, k)
    else:
      nondominated = True
      P[key] = [new_path_obj]
      P_D = []

    for path_obj in P_D:
      # print_SE(SE)
      if path_obj in SE:
        SE.remove(path_obj) # remove P_D from SE
        F_sorted.remove(path_obj.F)
    if nondominated == True:
      bisect.insort(F_sorted, new_path_obj.F)
      idx = F_sorted.index(new_path_obj.F)
      SE.insert(idx, new_path_obj) # add new_path_obj to SE if nondominated

road_network, road_network_cov = construct_road_network(index=2)
sdrsp_ha_star(road_network, road_network_cov, r=(1,), s=(14,), alpha=0.374, k=3)
```

aer1516_project_sdrsp_gurobi_mip_model.py (Mixed Integer Programming Model).

```
# -*- coding: utf-8 -*-
"""aer1516_project_RSPP_MIP_gurobi_model.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1ze8eZy3S4aXp4D-0FbTirl2n_MrxnX1z

## **Appendix B - Reliable Shortest Path using Mixed-Integer Programming**

**Course:** AER1516 - Motion Planning for Robotics

**Due:** 22 April 2022

**Team:** Vishal Kanna Annand, Andrew Constantinescu, Sugumar Prabhakaran

### **Introduction**

This code here is to implent a mixed integer program (MIP) or a mixed integer linear program (MILP) to solve the spatially dependent reliable shortest path problem.  Please see Section 4 of our paper for details on the implementation.
"""

# import necessary modules
import numpy as np
import matplotlib.pyplot as plt
import networkx as nx
from scipy.stats import norm
from math import sqrt

# install gurobi industrial MIP solver
!pip install gurobipy
import gurobipy as gp

# ****INPUT YOUR OWN ACADEMIC LICENSE grbgetkey****

# Create environment with WLS license
e = gp.Env(empty=True)
# e.setParam('WLSACCESSID', 'XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX')
# e.setParam('WLSSECRET', 'XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX')
# e.setParam('LICENSEID', XXXXXX)

e.start()

# Create the model within the Gurobi environment
model = gp.Model(env=e)

"""### **Mathematical Formulation**

**User Defined Parameters**
* $N$: set of nodes, with $N = \{1, 2, ..., n\}$
* $A$: set of arcs, with $A = \{a_{ij},...\}$, where $a_{ij}$: link
```

```
    from node $i$ to node $j$
* $T_{ij} = (t_{ij}, \sigma_{ij})$:  the normal distribution (mean, std deviation) of travel time for link $a_{ij}$
* $r, s$: start and end nodes respectively
* $\alpha$: User defined confidence level ($\alpha> 0.5$: risk-averse, $\alpha=0.5$: risk-neutral, $\alpha < 0.5$: risk-seeking)

**Decision Variables**
* $x_{ij} \in \{0,1\}$: binary decision variable signifying link-path incidence - i.e. if link is on path = 1
* $y_{ab}$: binary decision variable that means link $a \in A$ connects to link $b \in A$

**Objective Function**
* $\Phi^{-1}(\alpha) = \sum_{a_{ij} \in A}t_{ij}\cdot x_{ij} + z_{\alpha}\cdot \sqrt{\sum_{a_{ij} \in A}(\sigma_a)^2 \cdot x_{ij} + 2\cdot \sum_{a_{ij} \in A}\sum_{b_{kl} \in A}(\sigma_{ab}\cdot y_{ab})}$, where:
* $\Phi^{-1}(\alpha)$: is the inverse cumulative density function (cdf) of the overall path travel time that we want to minimize as a function of $\alpha$
* $z_{\alpha}$: Is the inverse cdf of a standard normal distribution at a $\alpha$ confidence level
* $(\sigma_a)^2$: variance of link $a_{ij}$
* $\sigma_{ab} = $: covariance between link $a_{ij}$ and link $b_{kl}$

"""

# USER DEFINED PARAMETERS
scenario = 2

if scenario == 1:
    n = 5               # number of nodes
    r = 1               # start node
    s = 5               # finish node

    N = [i for i in range(1, n+1)]                          # Set of nodes
    A = [(1,2), (1,3), (1,4), (2,3), (3,5), (4,5)]          # Set of arcs
    A_w = [(1,2, 2), (1,3, 3), (1,4, 4), (2,3, 2), (3,5, 4), (4,5, 4)]   # incl wts
    t_ij = {(i,j):k for (i,j,k) in A_w}                     # dict of mean time for arc
    idx = {(j,k):i for i, (j,k) in enumerate(A)}    # index needed for dict later

    cov_matrix = np.array([[ 2.0, 0.0, 0.0, 0.0, 0.0, 0.0],
                           [-1.0, 1.0, 0.0, 0.0, 0.0, 0.0],
                           [-1.0,-0.5, 1.0, 0.0, 0.0, 0.0],
                           [ 2.0,-1.0,-0.3, 2.0, 0.0, 0.0],
                           [ 0.3, 1.5,-0.4, 2.0, 6.0, 0.0],
                           [-0.2,-0.6, 0.5,-0.4,-1.5, 1.0]])

    positions = {1:[1,3], 2:[3,4], 3:[3,3], 4:[3,2], 5:[5,3]} #for visualization

elif scenario == 2:
    n = 14              # number of nodes
```

```python
    r = 1               # start node
    s = 14              # finish node

    N = [i for i in range(1, n+1)]                              # Set of nodes
    A = [(1, 2), (1, 3), (1, 4), (2, 5), (2, 6), (3, 5), (3, 6),# Set of arcs
         (3, 7), (4, 6), (4, 7), (5, 8), (6, 9), (7, 9), (8, 10),
         (9, 11), (11, 10), (10, 12), (10, 13), (11, 13), (12, 14), (13, 14)]
    A_w = [(1, 2, 3), (1, 3, 1), (1, 4, 1), (2, 5, 2), (2, 6, 3), # incl wts
           (3, 5, 2),(3, 6, 7), (3, 7, 1), (4, 6, 7), (4, 7, 3), (5, 8, 5),
           (6, 9, 1), (7, 9, 6), (8, 10, 4), (9, 11, 2), (11, 10, 2),
           (10, 12, 1), (10, 13, 2), (11, 13, 1),(12, 14, 1), (13, 14, 4)]
    t_ij = {(i,j):k for (i,j,k) in A_w}                         # dict of mean time for arc
    idx = {(j,k):i for i, (j,k) in enumerate(A)}    # index needed for dict later

    cov_matrix = np.array([[  2,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,    0,   0,   0,   0,   0,   0,   0,   0,   0],
                           [  1, 0.1,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,    0,   0,   0,   0,   0,   0,   0,   0,   0],
                           [  1,   2,   6,   0,   0,   0,   0,   0,   0,   0,   0,   0,    0,   0,   0,   0,   0,   0,   0,   0,   0],
                           [  0,   0,   0, 0.5,   0,   0,   0,   0,   0,   0,   0,   0,    0,   0,   0,   0,   0,   0,   0,   0,   0],
                           [  0,   0,   0,   0,   2,   0,   0,   0,   0,   0,   0,   0,    0,   0,   0,   0,   0,   0,   0,   0,   0],
                           [  0,   0,   0,   0,   0, 0.2,   0,   0,   0,   0,   0,   0,    0,   0,   0,   0,   0,   0,   0,   0,   0],
                           [  0,   0,   0,   0,   0,   0,   1,   0,   0,   0,   0,   0,    0,   0,   0,   0,   0,   0,   0,   0,   0],
                           [  0,   0,   0,   0,   0,   0,   0, 0.1,   0,   0,   0,   0,    0,   0,   0,   0,   0,   0,   0,   0,   0],
                           [  2,   2,   2,   2,   2,   2,   2,   2,   9,   0,   0,   0,    0,   0,   0,   0,   0,   0,   0,   0,   0],
                           [  0,   0,   0,   0,   0,   0,   0,   0,   0,   1,   0,   0,    0,   0,   0,   0,   0,   0,   0,   0,   0],
                           [  9,   9,   9,   9,   9,   9,   9,   9,   9,   9,   2,   0,    0,   0,   0,   0,   0,   0,   0,   0,   0],
                           [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   3,    0,   0,   0,   0,   0,   0,   0,   0,   0],
```

```
                                [ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0, 0.7,  0,  0,  0,  0,  0,  0,  0,  0],
                                [ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  1,  0,  0,  0,  0,  0,  0,  0],
                                [ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  2,  0,  0,  0,  0,  0,  0],
                                [ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  1,  0,  0,  0,  0,  0],
                                [ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  1,  0,  0,  0,  0],
                                [ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0, 0.5,  0,  0,  0],
                                [ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0, 0.2,  0,  0],
                                [ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  1,  0],
                                [ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0, 0.1]])

    positions = {1:[1,3], 2:[3,4], 3:[3,3], 4:[3,2],     # for visualization
                 5:[5,4], 6:[6,3], 7:[5,2], 8:[7,3.5], 9:[7,2.5],
                 10:[10, 3.5], 11:[8.5,2.5], 12:[12, 3.5], 13:[11,2.5],
                 14:[14, 3]}

# VISUALIZE DIRECTED GRAPH

# create graph object and add nodes and edges with weights
plt.figure(figsize=(12,4))
G = nx.DiGraph()
G.add_nodes_from(N)
G.add_weighted_edges_from(A_w)

# draw network
nx.draw_networkx(G, positions, with_labels=True)
nx.draw_networkx_edge_labels(G, pos=positions, edge_labels=t_ij)
plt.savefig('primal_network.png', dpi=600)
plt.show()

# INITIALIZE GUROBI MIP MODEL
model = gp.Model("reliable_shortest_path")
model.setParam('TimeLimit', 60) # seconds
model.Params.LogToConsole = 0   # suppress outputs

def optimize_model(alpha, N, A, r, s, t_ij, idx, cov_matrix):
    '''
    FUNCTION: given a network graph (N, A) problem and alpha confidence level,
```

```
    construct a mixed-integer programming model in gurobi and return the most
    reliable path and the objective function value (inverse cdf of the path
    travel-time)

    INPUTS:
    alpha (float)       - confidence between [0, 1], where: 0.5 is risk-neutral,
                          <0.5 is risk-seeking and >0.5 is risk-averse
    N (list)            - list of nodes
    A (list)            - list of links (Ex. a_ij is link from node i to j)
    r (int)             - start node
    s (int)             - goal node
    t_ij (dict)         - dict of mean travel-time for each link a_ij
    idx (dict)          - dict of covariance matrix index for each link a_ij
    cov_matrix (array)  - numpy array containing var, covariance of all links

    OUTPUTS:
    opt_reliable_path (list) - sequence of links from start node to goal node
    obj_val (float)          - objective function value
    '''

    z = norm.ppf(alpha) # inv. cdf of std norm. distr at alpha confidence

    # DECISION VARIABLES
    x = model.addVars(A, vtype=gp.GRB.BINARY)   # link-path incidence variables

    # additional variables to linearize objective function
    var = model.addVar(name='var')
    covariance = model.addVar(name='covariance')
    path_var = model.addVar(name='path_var')
    path_std = model.addVar(name='path_std')

    # CONSTRAINTS
    #enforce one-way direction of links from node i to node j for all links
    model.addConstrs(gp.quicksum(x[i,j] for j in G.successors(i))-
                     gp.quicksum(x[k,i] for k in G.predecessors(i)) == 1
                     for i in N if i == r)
    model.addConstrs(gp.quicksum(x[i,j] for j in G.successors(i))-
                     gp.quicksum(x[k,i] for k in G.predecessors(i)) == 0
                     for i in N if (i != r and i != s))
    model.addConstrs(gp.quicksum(x[i,j] for j in G.successors(i))-
                     gp.quicksum(x[k,i] for k in G.predecessors(i)) == -1
                     for i in N if i == s)

    # OBJECTIVE FUNCTION
    # objective function:  path mean + z*sqrt(var + covar)
    path_mean = gp.quicksum(t_ij[a]*x[a] for a in A)
```

```python
    var = gp.quicksum(cov_matrix[idx[a],idx[a]]*x[a] for a in A)
    covariance = gp.quicksum(cov_matrix[idx[a], idx[b]]*x[a]*x[b]
                             for a in A for b in A if a !=b)

    #linearize sqrt in objective function
    model.addConstr(path_var == var + 2*covariance)
    model.addGenConstrPow(path_var, path_std, 0.5)  # path_std = sqrt(path_var)

    model.modelSense = gp.GRB.MINIMIZE              # minimization
    model.setObjective(path_mean+z*path_std)        # objective function
    model.optimize()                                # execute

    # RESULTS
    opt_reliable_path = []                          # initialize list to store path

    # iterate through each link that is on the path
    for a in A:
        if x[a].x != 0:
            opt_reliable_path.append(a)         # store link in optimal path list

    obj_val = model.getObjective().getValue() # optimal obj. function value

    return opt_reliable_path, obj_val

# OUTPUT RESULTS

for alpha in list(np.around(np.linspace(0.05, 0.95, 19), 2)):

    opt_path, obj_val = optimize_model(alpha, N, A, r, s, t_ij, idx, cov_matrix)
    print("alpha =", alpha, ":", opt_path, obj_val)
```