Computational Optimization and Applications (2019) 74:517–545
https://doi.org/10.1007/s10589-019-00121-w

Minimizing the average searching time for an object within a graph

Ron Teller[1] · **Moshe Zofi**[1,2] · **Moshe Kaspi**[1]

Received: 31 July 2018 / Published online: 29 July 2019

Abstract

This paper presents a new graph search problem for which a searcher wishes to find an object that may be found at a set of locations. The searcher doesn't know the object's exact location, but does know the a-prior probability of finding the object at each location. He wishes to build a searching path for reaching the object that starts from a given location and ends when reaching the object (or after searching the entire set with a false result). The objective is to find a searching path which will minimize the average searching time. We consider two scenarios for this problem: one when there is an unknown number of objects on the set and another when there is exactly one object on the set (the sum of probabilities is equal to 1). We show that this problem is NP-Hard, and supply a branch and bound algorithm for finding an optimal solution for large scale problems. We also study greedy approaches and other heuristics and compare the performance of these algorithms in various situations.

Keywords Graph search algorithms · Object searching · Branch and bound · Dynamic programming · Greedy algorithm · Heuristic search · TSP · Ant colony optimization

✉ Moshe Zofi
zofi@bgu.ac.il

Ron Teller
rontel@bgu.ac.il

Moshe Kaspi
moshe@bgu.ac.il

[1] Department of Industrial Engineering and Management, Ben Gurion University of the Negev, Beer-Sheva, Israel

[2] Department of Industrial Management, Sapir College, Sderot, Israel

1 Introduction

We consider the following problem: a searcher wishes to find a specific object, which may be found at a number of locations. He doesn't know the object's exact location, but knows the a-prior probability of finding it at various locations. His goal is to construct a searching path for reaching the object (a prioritized list of locations he will visit in order to find the object). The search ends when the searcher finds the object or after he searched the entire set of locations with a false result.

The exact searching time on a given path is unknown (due to the probabilities) and therefore, the searcher's objective is to find the searching path which will minimize the average searching time (AST) - namely the path with the lowest expected time of finding the object. We call this problem the *Min AST*.

The *Min AST* problem has two scenarios: (a) when there is an unknown number of objects (the sum of probabilities of finding the object at the given set of locations ≥ 0) and (b) when there is exactly one object (the sum of probabilities of finding the object at the given set of locations $= 1$).

The *Min AST* problem can be reduced to the *minimum weight Hamiltonian path problem* and thus it is *NP-Hard*. This fact motivated us to look for approximation algorithms, meta-heuristics and special structures of the problem, where polynomial time algorithms may be applied. Some of these algorithms are inspired by the extensive research that was done on the Traveling Salesmen Problem (TSP) over the past decades.

In practice, searching objects at discrete locations is a scenario that may arise in various fields.

Consider the case in which we are interested in buying a single item, which may (or may not) be found in several stores—such as a unique clothing item in an unusual size or a rare replacement part for an electronic device. This item can be found in more than one store (the probability of finding the item in each store is given), and we want to determine the order in which to visit the stores such that we can find an item in as less time as possible. Consider another case in which we are trying to find a spot in a parking lot as quickly as possible. We know what the chance is of each parking lot to be full, but we still need to determine the order in which to visit the parking lots. All of these examples fit scenario (a) in which there can be more than one object, and the probability of finding it in one location does not necessarily affect the probability of finding it in another.

Consider another problem where a rescue team is sent to find a lost traveler and is given the locations he had planned to visit. The team has an a priori knowledge about former lost travelers at different locations, and therefore can assess the likeliness of the traveler to get lost at each of the locations he planned to visit. According to this information, the team tries to plan a course where the expected searching time for finding the lost traveler will be minimal. This example fits scenario (b), since there is exactly one lost traveler, and if the team doesn't find him at a specific location, then the chances of finding him at the other possible locations increases.

The problem described in this paper belongs to the field of search in which the prior knowledge of the location and behavior of the target is incomplete or

probabilistic. This field has been widely researched in the last half-decade and various problems have been formulated for scenarios where a searcher wishes to obtain a target or an object. As described in the survey by Benkoski et al. [1], a search problem can be categorized by the following features:

- The mobility of the target (moving or stationary)
- The motivation of the target (optimization problem or search game)
- The type of space and time (continuous or discrete)

Many search problems have been formulated as game theory problems. In these problems the searcher is trying to find a target that has its own motivation. In game theory, *Search games* are zero-sum games where the target is evading the searcher. Some of the well-known search games are *The Linear Search Problem* proposed by Beck [2] where the searcher tries to find an immobile evader on a straight line, and *The Princess and Monster Problem* proposed by Isaacs [3], where the searcher tries to find the target in a dark room and detect the target if he is close enough. Another known searching problem in game theory is *The Rendezvous Problem* [4], where the searcher and a mobile target wish to find each other as soon as possible. The *Min AST* problem described in this paper assumes the target has no motivation; therefore The *Min AST* problem is not a game, but an optimization problem.

Searching problems with a motiveless target (sometimes referred to as *Optimal Searcher Path problems*), deal with the computation complexity, the solution approaches and the algorithms for different searching scenarios. These problems differ in the type of space and time of the search environment (continuous or discrete). Most available research deal with scenarios where the searcher's searching effort is limited in time or in motion. Some studies deal with problems where the goal is to maximize the probability of detection within these limits. Stone [5] summarized a variety of scenarios where the goal is to optimally allocate searching resources. Brown [6] introduced the problem of searching a stationary target within a set of cells and with limited time effort, Stewart [7] dealt with the problem where the amount of resources spent is time dependent. Weber [8] introduced a problem where the space is discrete and limited (only two locations) but the time is continuous and the target moves with some random Markovian process. For other related problems in search theory see [1].

Many of the search problems in the literature (see [6–11]) take into consideration a detection function or a glimpse probability, that describes the chance of the searcher to find the target when both of them are at the same position or somewhat adjacent to each other. The problem described in this paper assumes that if the searcher and the object are at the same position, the searcher will surely find the object.

In recent years, some new optimization search problems were proposed, where the searching effort is unlimited, and the goal is to construct a path that covers the entire search space and has a minimal average searching time. These problems are similar to the *Min AST* problem. Jotshi and Batta [12] described a scenario where a searcher wants to locate an immobile entity that is uniformly distributed on a graph's edges. His objective is to minimize the expected searching time. The authors later

extended the problem for searching two immobile entities on an undirected network where the entity locations are probabilistically known and dependent [13]. Both problems are related to the Chinese Postman Problem in which we need to cover all edges and thus differ from the *Min AST*.

Another problem formulated by Berman et al. [14] deals with a person who wants to get service from stationary facilities as fast as possible. The facilities are prone to disruption (the facility has a constant probability of becoming inactive) and the person is given the a priori knowledge of the stochastic behavior of each facility. This problem can be seen as a search problem where the searcher wishes to find a functioning facility. Since the chance of the facility to fail is equal in all locations and is not conditioned by other factors, Berman's problem covers a specific case of many searching scenarios. The best known algorithm for obtaining an optimal result for Berman's problem is a forward dynamic programming, with a computation complexity of $\left(n^3 2^n\right)$. This algorithm assumes there are equal probabilities to all the vertices. Due to its computation complexity, Berman's algorithm struggles with obtaining optimal results in graphs with more than 20 vertices.

Scenario (a) of the *Min AST* problem is closely related to the problem of sequential testing of multi-component systems [15]. Each component is associated with a different test cost and a given probability that it is functioning correctly. The components functionality is independent of each other. In order to minimize the total cost of testing the system's functionality, we must determine the right order of testing all of the system's components. Scenario (a) of the *Min AST* is a variation of the sequential testing problem on complete graphs where the edge weights represent the test costs.

Another related problem is the minimum latency problem (or the traveling repairman problem) [16, 17], in which we want to find a tour for a single repairman passing through n vertices (representing machines or customers), which will minimize the average waiting time. The *Min AST* problem differs from this problem in two ways: (1) The *Min AST* objective is given from the searcher point of view (which is to minimize its total traveling time) and not from the machine's point of view (which is to minimize the accumulative waiting time until it is visited). (2) Each vertex is associated with an a-prior probability, which highly influences the order of visits, as shown in Example 1 in Sect. 2.3.

In this paper we extend the scope of the above possible searching scenarios and offer a more general formulation to discrete searching problems for a stationary object. We provide a mathematical formulation to this problem, and perform an analysis of several searching paths for general and for several special graphs and problem structures (on finite and infinite graphs).

Our paper proposes a dynamic programming algorithm and a branch and bound algorithm that unlike Berman's algorithm are not restricted to the special case in which all the probabilities are equal and that can also be applied to the scenario of dependent probabilities (exactly one object on the graph). We show that the ability to prune branches strongly depends on the probabilities on the vertices, and thus the runtime of this algorithm becomes shorter as the probabilities grow higher. In cases of high probabilities, the algorithm quickly yields optimal results for large scale problems (500 vertices and more). When the probabilities are relatively low, the *Min*

AST problem acts more and more like the TSP, and thus our bounding techniques become less efficient. We will explain why some well-known bounding techniques for the TSP (like the assignment problem relaxation and the cutting planes method) cannot be applied for the *Min AST* problem and propose other bounding techniques and test their cost-effectiveness.

We show that in many cases, good results for the *Min AST* problem may be obtained by using greedy approaches or other well-known meta-heuristics. The greedy algorithms result's quality gets better as the probability of finding the object at each vertex grows. We adapt the ant colony optimization algorithm to the *Min AST* problem and show that it produces very good results in all problem instances.

Performance analysis will be presented for each of the algorithms we suggest, after which we recommend using different solution approaches for different types of problem instances.

The paper is organized as follows: Sect. 2 provides mathematical definitions and formulations of the *Min AST problem*. Section 3 will introduce several solution approaches for the problem followed by Sect. 4 which will provide test results and a comparison between approaches. Section 5 will summarize the work and will provide ideas for future work.

2 Problem analysis

2.1 Problem definition and notation

Consider a complete graph $G = (V, E)$ comprised of a set of vertices V numbered $1, 2, \ldots, n$ and a set of connecting edges E, each with a weight $t_{i,j}$ that indicates the traveling time or distance between the vertices in its ends. The required object can be found at each vertex $j \in V$ with a given probability p_j. The graph may contain a single required object (i.e. $\sum_{j=1}^{n} p_j = 1$), or an unknown number of objects (i.e. $\sum_{j=1}^{n} p_j \geq 0$). The searcher's objective is to find the path that starts with a given vertex (indexed 0) and has the minimal average searching time (AST), which is the path with the least expected searching time. The searching process ends when the object is found, with no need to return to the starting vertex. Without loss of generality, we assume that the searcher's travelling speed is constant.

A valid search path *SP* is an ordered list of vertices that contains each of the graph's vertices exactly once.[1]

We note that even though the searcher may visit only a partial part of the path (if the object is found at early stages), all possible cases must be considered when calculating the AST.

We consider two scenarios: (a) where there is an unknown number of objects and (b) where there is exactly one object on the graph. In the first scenario, finding the object is not guaranteed by the end of the process, and the probability of finding the object at each vertex is not dependent on the state of the searching process. In the

[1] The case in which repeated visits on vertices (backtracking) is allowed is presented at Sect. 3.5

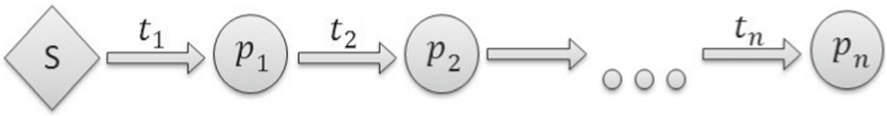

Fig. 1 A sample search path in a finite graph

second scenario, by the end of the searching process, the object is found with certainty. Moreover, if the object is not found at a certain vertex, the probability of finding it at each of the remaining vertices on the path increases (this paradigm will be elaborated in Sect. 2.3).

2.2 Search path analysis

In this section we show how to calculate the average searching time in general and also for some special cases for a given *SP*. Let v_i be the i th vertex in the *SP* and let t_i be the traveling time between v_{i-1} and v_i. We set p_0 to be 0 in all cases (the object is not located at the starting vertex). Our approach for calculating the AST for a given path is to add up all the traveling times (as indicated by the edges we chose) on the *SP* multiplied by the chance of actually traveling through them during the searching process. We analyze finite and infinite search paths (in finite and infinite graphs), for both scenarios (described above).

The probability of traversing the i th edge e_i (the edge between vertices v_{i-1} and v_i) on *SP* is calculated by multiplying all the probabilities of not finding the object at any of the $i-1$ vertices prior to the i th vertex. Thus, for an unknown number of objects, the probability of traversing the i th edge p_{e_i} is calculated as: $p_{e_i} = \prod_{j=1}^{i-1}\left(1-p_j\right)$.

Note that whether one node contains an object is independent of whether another node contains an object.

When we consider the scenario of a single object, the original probabilities on the vertices grow larger as we move along the *SP* (if the vertices visited in the *SP* do not contain the required item, then the probability of finding the item at the rest of the vertices increases). Updating the probabilities can be replaced with the simple sum: $p_{e_i} = 1 - \sum_{j=1}^{i-1} p_j$

The following section will demonstrate the calculation of the AST for several cases in finite and infinite graphs.

Search paths in finite graphs

Consider the *SP* presented at Fig. 1, containing n vertices on a finite graph.

In the most general case where t_i and p_i may vary for each i and when there is an unknown number of objects on the graph (scenario (a)), the AST will be calculated as follows:

$$AST = \sum_{i=1}^{n}\left[t_i \prod_{j=1}^{i-1}\left(1-p_j\right)\right]$$

Fig. 2 A sample search path in an infinite graph

When there is exactly one object (scenario (*b*)) the average searching time will be:

$$AST = \sum_{i=1}^{n} \left[t_i \left(1 - \sum_{j=1}^{i-1} p_j \right) \right]$$

Special cases

1. when $p_i = p_j = p \, \forall i,j$, then for scenario (*a*): $AST = \sum_{i=1}^{n} t_i (1-p)^{i-1}$ and for scenario (*b*): $AST = \sum_{i=1}^{n} t_i \left[1 - (i-1)p \right]$
2. when $t_i = t_j = t \, \forall i,j$, then for scenario (*a*): $AST = t \sum_{v}^{n} \left[\prod_{j=1}^{i-1} \left(1 - p_j \right) \right]$ and for scenario (*b*): $AST = t \sum_{i=1}^{n} \left[1 - \sum_{j=1}^{i-1} p_j \right]$
3. when $p_i = p_j = p \cap t_i = t_j = t \, \forall i,j$, then for scenario (*a*), we calculate *AST* using the sum of a finite geometric progression with $q = 1 - p$ and $a_1 = t$ such that:

$$AST = a_1 \frac{q^n - 1}{q - 1} = t \frac{(1-p)^n - 1}{(1-p) - 1} = t \frac{1 - (1-p)^n}{p}$$

And for scenario (*b*):

$$AST = t \sum_{i=1}^{n} 1 - (i-1)p = t \left[n - p \sum_{i=1}^{n-1} i \right] = nt \left[1 - \frac{(n-1)p}{2} \right]$$

Search paths in infinite graphs

Consider the *SP* presented at Fig. 2, containing an infinite number of vertices on an infinite graph

In the most general case (where t_i and p_i may vary for each i), the AST for scenario (*a*) is:

$$AST = \sum_{i=1}^{\infty} \left[t_i \prod_{j=1}^{i-1} \left(1 - p_j \right) \right]$$

In the case of scenario (*b*), if there is a coefficient c for which $p_i = 0 \, \forall i > c$, then the search path can be treated as a finite search path with c vertices and can be calculated as mentioned in the previous section. Otherwise: $AST = \sum_{i=1}^{\infty} \left[t_i \left(1 - \sum_{j=1}^{i-1} p_j \right) \right]$

Special cases

1. when $p_i = p_j = p \quad \forall i,j$, then for scenario (a): $AST = \sum_{i=1}^{\infty} t_i(1-p)^{i-1}$ and for scenario (b): $AST = \sum_{i=1}^{\infty} t_i\left[1-(i-1)p\right]$
2. when $t_i = t_j = t \quad \forall i,j$ then for scenario (a): $\text{AST} = t\sum_{i=1}^{\infty}\left[\prod_{j=1}^{i-1}\left(1-p_j\right)\right]$ and for scenario (b): $AST = t\sum_{i=1}^{\infty}\left[1-\sum_{j=1}^{i-1}p_j\right]$
3. when $p_i = p_j = p \cap t_i = t_j = t \quad \forall i,j$, then for scenario (a) we calculate AST using the sum of infinite geometric progression with $q = 1-p$ and $a_1 = t$, and get: $AST = \frac{a_1}{1-q} = \frac{t}{p}$. Note that in scenario (b) $p = \frac{1}{\infty} \sim 0$ since there is only one object: $\lim_{p\to 0} AST = \lim_{p\to 0}\frac{t}{p} = \infty$

We can see that in all the cases described above, the earlier an edge appears in the search path, the greater impact it has on the AST. This is due to a higher probability of being visited. When the probabilities of finding the object increase, the contribution of edges to the AST drops drastically as a function of their location in the search path.

2.3 Mathematical formulation

Consider a complete graph $G = \{V, E\}$ containing $|V| = n$ vertices (indexed $0, \ldots, n-1$) where all the vertices are connected by $|E| = n(n-1)$ edges.

Let T be a distance (or traveling time) matrix whose common element $t_{i,j}$ represents the traveling time along the edge connecting vertices i and j.

Let p_i represent the probability of finding the object at vertex i.

Let $x_{i,s} = \{1, 0\}$ be a binary decision variable which will receive the value of 1 if the searching path will visit vertex i in stage s and 0 otherwise.

Since the searching path starts at vertex 0 in stage 0, we can say that:

$$(1) \quad x_{0,0} = 1$$

The searching path will include all of the vertices, each in a different stage and therefore:

$$(2) \quad \sum_{i=1}^{n-1} x_{i,s} = 1 \quad \forall s$$

$$(3) \quad \sum_{s=0}^{n-1} x_{i,s} = 1 \quad \forall i$$

The distance between two consecutive vertices i and j visited at stages $s-1$ and s can be written as:

$$\sum_{i=0}^{n-1}\sum_{j=0}^{n-1} x_{i,s-1}x_{j,s}t_{i,j}$$

The probability of not finding the object until stage s in scenario (a) in which $\sum_{i=0}^{n-1} p_i > 0$ is:

$$\prod_{k=0}^{s-1}\sum_{i=0}^{n-1} x_{i,k}(1-p_i)$$

Therefore the objective of minimizing the average searching time (AST) can be written as:

$$MinAST = \sum_{s=1}^{n-1}\left[\left(\prod_{k=0}^{s-1}\sum_{i=0}^{n-1} x_{i,k}(1-p_i)\right)\left(\sum_{i=0}^{n-1}\sum_{j=0}^{n-1} x_{i,s-1}x_{j,s}t_{i,j}\right)\right]$$

Summarizing all the above we get the full complete formulation for scenario (a) as follows:

$$MinAST = \sum_{s=1}^{n-1}\left[\left(\prod_{k=0}^{s-1}\sum_{i=0}^{n-1} x_{i,k}(1-p_i)\right)\left(\sum_{i=0}^{n-1}\sum_{j=0}^{n-1} x_{i,s-1}x_{j,s}t_{i,j}\right)\right]$$

$$\begin{aligned}
&s.t.\\
&(1)\quad x_{0,0} = 1\\
&(2)\quad \sum_{i=1}^{n-1} x_{i,s} = 1 \quad \forall s\\
&(3)\quad \sum_{s=0}^{n-1} x_{i,s} = 1 \quad \forall i\\
&x_{i,j} = \{1,0\} \quad \forall i,j = 0,\ldots,n-1
\end{aligned}$$

For scenario (b) in which $\sum_{i=0}^{n-1} p_i = 1$, the probability of not finding the object until stage s is: $1 - \sum_{k=o}^{s-1}\sum_{i=0}^{n-1} p_i x_{i,k}$ and thus the objective changes to:

$$MinAST = \sum_{s=1}^{n-1}\left[\left(1 - \sum_{k=0}^{s-1}\sum_{i=0}^{n-1} p_i x_{i,k}\right)\left(\sum_{i=0}^{n-1}\sum_{j=0}^{n-1} x_{i,s-1}x_{j,s}t_{i,j}\right)\right]$$

The following example demonstrates the possible searching paths and their ASTs.

Example 1 Consider the graph presented at Fig. 3. The searcher is located at vertex 'S'. The probabilities of finding the object at vertices 'a', 'b' and 'c' are 0.1, 0.6 and 0.3 respectively. The traveling time between every two vertices is written on the connecting edge.

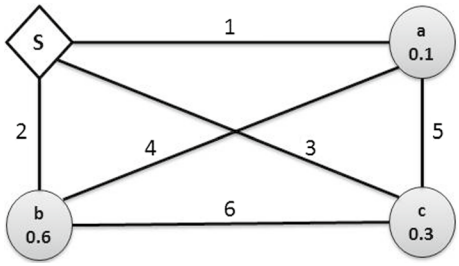

Fig. 3 A sample graph containing 4 vertices and 6 edges

This graph contains six possible searching paths. In the following, we calculate the AST for each possible *SP* for both scenarios (a) and (b) as described in Sect. 2.1. The best searching path in each scenario is emphasized using bold characters.

Scenario (a): there is an unknown number of objects in the graph.

$$[\text{SP 1}]\ S \to a \to b \to c:\ AST = 1 + (1-0.1) * 4 + (1-0.1) * (1-0.6) * 6 = 6.76$$
$$[\text{SP 2}]\ S \to a \to c \to b:\ AST = 1 + (1-0.1) * 5 + (1-0.1) * (1-0.3) * 6 = 9.28$$
$$[\textbf{SP 3}]\ \mathbf{S \to b \to a \to c}:\ \mathbf{AST = 2 + (1 - 0.6) * 4 + (1 - 0.6) * (1 - 0.1) * 5 = 5.4}$$
$$[\text{SP 4}]\ S \to b \to c \to a:\ AST = 2 + (1-0.6) * 6 + (1-0.6) * (1-0.3) * 5 = 5.8$$
$$[\text{SP 5}]\ S \to c \to a \to b:\ AST = 3 + (1-0.3) * 5 + (1-0.3) * (1-0.1) * 4 = 9.02$$
$$[\text{SP 6}]\ S \to c \to b \to a:\ AST = 3 + (1-0.3) * 6 + (1-0.3) * (1-0.6) * 4 = 8.32$$

Scenario (b): the graph contains exactly one object.

$$[\text{SP 1}]\ S \to a \to b \to c:\ AST = 1 + (1-0.1) * 4 + [1-(0.1+0.6)] * 6 = 6.4$$
$$[\text{SP 2}]\ S \to a \to c \to b:\ AST = 1 + (1-0.1) * 5 + [1-(0.1+0.3)] * 6 = 9.1$$
$$[\text{SP 3}]\ S \to b \to a \to c:\ AST = 2 + (1-0.6) * 4 + [1-(0.6+0.1)] * 5 = 5.1$$
$$[\text{SP }\mathbf{4}]\boldsymbol{S \to b \to c \to a}:\ \boldsymbol{AST = 2 + (1-0.6)*6 + [1-(0.6+0.3)] * 5 = 4.9}$$
$$[\text{SP 5}]\ S \to c \to a \to b:\ AST = 3 + (1-0.3) * 5 + [1-(0.3+0.1)] * 4 = 8.9$$
$$[\text{SP 6}]\ S \to c \to b \to a:\ AST = 3 + (1-0.3) * 6 + [1-(0.3+0.6)] * 4 = 7.6$$

Note that the optimal searching path for scenarios (a) and (b) in a given graph may differ.

2.4 Problem complexity

The min AST problem can be reduced to the well-known *NP-Complete* problem of determining whether a Hamiltonian path exists within a given graph [18].

Theorem 1 *Finding the minimal AST is NP-Hard for scenarios* (*a*) *and* (*b*).

Proof The problem of determining whether a Hamiltonian path exists in a graph $G = (V, E)$ starting from a specific vertex (v_s) is *NP-Complete*.□

Let G' be a weighted *complete* graph comprised out of the same vertices in G and Let $w_{\{u,v\}}$ represent the weight of the edge connecting vertices u and v.

Let $w_{\{u,v\}} = \begin{cases} 1, \{u, v\} \in E(G) \\ \infty, otherwise \end{cases}$, and let $p_i < 1 \quad \forall i \in V(G')$.

Let AST_{min} be the solution of finding the minimal AST for either scenario (*a*) or (*b*), with v_s as the starting vertex.

If $AST_{min} < \infty$, then a Hamiltonian path exists in the original G (the path with the minimal AST in G' by the algorithm is a Hamiltonian Path in G). If $AST_{min} = \infty$, then no Hamiltonian Path in G that starts from vertex v_s exists.

Thus, determining whether we can find an $AST_{min} < \infty$ is equivalent to determining whether a Hamiltonian path exists in the graph.

Obtaining an approximation with a constant ρ-factor to the TSP in its general case is *NP-Complete* [19]. This is also true for the *minimum weight Hamiltonian path* problem, as such an algorithm would solve the Hamiltonian path problem which is also *NP-Complete*.

Therefore; it is obvious that obtaining a constant ρ-factor to the *Min AST* problem is *NP-Complete* (Otherwise we could use any approximation algorithm for the *Min AST* problem to obtain an approximation to the *minimum weight Hamiltonian path* problem by setting all probabilities to 0).

3 Solution approaches

This section will present several solution approaches for the problem. We start with dynamic programming and branch and bound methods which obtain optimal solutions but may require high computations and memory usage. We then provide several heuristics that reach good solutions and require much less resources.

3.1 Dynamic programming

Berman et al. [14] introduced a forward dynamic programming algorithm that produces an exact solution for a problem when all probabilities are equal. We performed several adjustments in this algorithm so that it can reach the optimal solution when the probabilities are not equal [as required in scenarios (a) and (b)].

The algorithm has $|V| - 1$ stages (numbered 0 to $n - 1$). The number of stages is equal to the number of vertices in the complete search path (not including the starting vertex). In each stage, we generate a table in which the rows represent the group of vertices that have been added to the path until the previous stage (set A_i in the algorithm); and the columns represent the vertices that can be added to the path in the current stage (see Fig. 2).

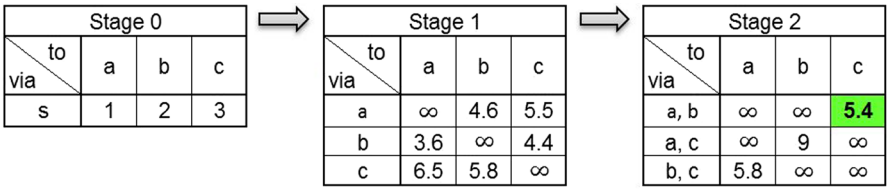

Fig. 4 Finding the minimal AST for scenario (a) on the graph presented at Fig. 3 using dynamic programming

This algorithm has a time complexity of $O\left(2^n n^3\right)$ (assuming a table of binomial coefficients is pre-processed); the i th stage of the algorithm requires $O\left(n\binom{n}{i}\right)$ memory as it keeps a table with $\binom{n}{i}$ rows and n columns. In order to retrieve the optimal path, all tables must be kept, and the total amount of memory required is $O\left(\sum_{i=1}^{n} n\binom{n}{i}\right) = O(2^n)$. This algorithm is not affected by the probabilities on the graph and runs in a constant amount of time for all graph instances of the same size.

Algorithm Forward-Dynamic-Programming (G, *LastStageTable*, *Stage*)

1. If *Stage* $= 0$, create a table *NewStageTable* where *NewStageTable*$[0, i] = t_{0,i}$

2. Else ($1 \le$ *Stage* $\le n - 1$), create a table *NewStageTable* with $\binom{n}{Stage}$ rows and n columns.

2.1. For each row i in *LastStageTable*:

2.1.1. Calculate the cumulative probability* for the row: $p_{r_i} = \prod_{l \in A_i} (1 - p_l)^{**}$

2.1.2. For each column j in *LastStageTable*:

2.1.2.1. If the vertex represented by column j belongs to the set A_i, then cell $C(i,j) = \infty$

2.1.2.2. Else $C(i,j) = \min_{k \in A_i} \left\{ LastStageTable\left[index\left(A_i - k\right)^{***}, k\right] + t_{k,j} p_{r_i} \right\}$

2.2. If *Stage* $= n - 1$, return the minimal value in *NewStageTable*,

Else, call *ForwardDynamicProgramming*(G, *NewStageTable*, *Stage* $+ 1$)

---

*At a given stage, the computation of the cumulative probability (which represents the chance of not finding the object in any of the vertices in the set A_i) is the same for all the cells in the same row, as they all share the same previously visited vertices.

**In the case of a single object in the graph, this expression is replaced with $1 - \sum_{l \in A} p_l$

***Function $index\left(A_i\right)$ returns the lexicographic index of the combination A_i (out of $\binom{n}{size\left(A_i\right)}$ combinations).

DP Example

Figure 4 presents the tables created using the algorithm in each stage for the sample graph given in Fig. 3. The example is suitable for scenario (*a*).

- Table S_0 (generated at stage 0) contains a single row representing the starting vertex s, and 3 columns representing all the other vertices (a, b, c), each cell contains the traveling time between s and the vertex in the column.

- Table S_1 (generated at stage 1) contains $\binom{3}{1} = 3$ rows (the number of all possible combinations of a single vertex), for each row we calculate its cumulative probability p_{r_i}, a sample calculation for row 1: $p_{r1} = 1 - p_a = 0.9$.

  The value of each cell is calculated by $\min_{k \in A_i} \left\{ S0\left[A_i - k, k\right] + t_{k,j} p_{r_i} \right\}$ where A_i depends on the cell row ($\{a\}$, $\{b\}$ or $\{c\}$) and j depends on the cell column (a, b or c).

  In stage 1, $A_i - k = \emptyset$, which means that $A_i - k$ is the starting vertex s.

  The calculation for the sample cell "to c via $\{a\}$" (row 1, column 3) is:

$$S_0\left[1, columnofa\right] + t_{a,c} p_{r1} = 1 + 0.9 * 5 = \mathbf{5.5}$$

  (note that there is no minimum function since A contains only one vertex).

- Table S_2 (generated in stage 2) contains $\binom{3}{2} = 3$ rows (the number of all possible combinations of two vertices), for each row we calculate its cumulative probability p_{r_i}, a sample calculation for row 1: $p_{r1} = \left(1 - p_a\right)\left(1 - p_b\right) = 0.36$.

  The value of each cell is calculated by $\min_{k \in A_i} \left\{ S_1\left[A_i - k, k\right] + t_{k,j} p_{r_i} \right\}$ where A_i depends on the cell row ($\{a, b\}$, $\{a, c\}$ or $\{b, c\}$) and j depends on the cell column (a, b or c).

  The calculation of the sample cell "to c via $\{a, b\}$" is:

$$\min \begin{cases} S_1\left[row\ of\ \{b\},\ column\ of\ a\right] + t_{a,c} * p_{r1} = 3.6 + 5 * 0.36 = 5.4 \\ S_1\left[row\ of\ \{a\},\ column\ of\ b\right] + t_{b,c} * p_{r1} = 4.6 + 6 * 0.36 = 6.76 \end{cases} = \mathbf{5.4}$$

- When stage 2 is complete we get $Stage = n - 1$, and return the minimal value found in S_2 (i.e., $MinAST = 5.4$).

  The optimal SP is reconstructed backwards.

  The last vertex added to the path is c. We have reached c from a (which provided the minimal value when we calculated "to c via $\{a, b\}$"). We have also reached vertex a from b, and we have reached b from s. Thus the optimal search path is: $s \rightarrow b \rightarrow a \rightarrow c$.

3.2 Branch and bound algorithm

In order to achieve optimal results in larger graphs, we propose a depth first branch and bound algorithm with two different bounding techniques which are very efficient when the probabilities are not low.

The main features of a branch and bound algorithm are:

1. A *branching technique* that determines how to split the search area into sub-areas.
2. A *bounding technique* that determines how to calculate a lower bound and an upper bound for each sub-area.
3. A *search strategy* that determines which sub-area will be explored at the next step.

Branching

The branching technique in our algorithm is to iteratively construct search paths. The root of the tree is a search path containing the starting vertex and all the other vertices as candidates. As the tree branches, we add more vertices to the search path. When the candidate list is empty (and the tree is at depth $|V|$), we have constructed a full search path.

Bounding

Many efficient lower bounding techniques for the Traveling Salesman Problem (such as *The Assignment Problem Relaxation*) are not suitable for the *Min AST* problem. Opposed to the TSP (in which the contribution of each edge is independent of its location in the tour), the contribution of each edge in the *Min AST* problem depends on the cumulative probability of the vertices along the path, prior to the traversed edge. The cutting plane method is also unsuitable since it is used to refine the feasible set of mixed integer linear programs, and the *Min AST* problem is not linear (see Sect. 2.4).

We offer the following two lower bounding techniques. The first provides a good bound with relatively minor computation effort, while the other provides a tighter bound but requires larger computation effort.

In Sect. 4 we will show that both bounds outperform each other depending on the probabilities.

Lower bound 1

Let v_c be the vertex added to the path in the considered node. Let v_f represent the vertex that was added to path in the predecessor (father) node. Let AST_f be the AST calculated in the predecessor node. Let $t_{f,c}$ represent the traveling time required between v_f and v_c. Let p_f and p_c represent the probabilities of finding the object in v_f and v_c

Let CP_f and CP_c denote the cumulative probability of not finding the object in the path constructed until the predecessor node and until the current node respectively.

Thus, the lower bound in a given node is $AST_c = AST_f + CP_f * t_{f,c}$ and the cumulative probability of not finding the object until reaching the current node is $CP_c = CP_f * (1 - p_c)$.

This lower bound is calculated from the average searching time of the partial path that was constructed until reaching the current node. This bound does not take into consideration the vertices which are not included in the partial path, and therefore is not very tight. When the probabilities are relatively high, the contribution of the edges added to the path in the late stages is negligible. Thus, the bound is close to the best possible solution in the current branch.

This bound is computed in $O(1)$ time.

Lower bound 2

This lower bound extends the previous by adding to AST_c the minimal value for any possible search path that may be constructed through the remaining unvisited vertices.

Let s be the vertex added to the path in the current node and V_r be the list of the remaining unvisited vertices which need to be added to the search path. Also let P_r be the list of the probabilities of the vertices in V_r

1. Generate a traveling time list T_r, for each vertex v in V_r.
   Add the minimal traveling time t_v that must be spent in order to visit vertex v using any of the available vertices $t_v = \min(t_{i,v}) \forall i \in V_r \cup \{s\}/\{v\}$
2. Order T_r in an ascending order and P_r in a descending order.
   Set $CP_t = CP$ where CP is the cumulative probability of not finding the object in the path constructed until the current node.
3. Go over the ordered lists T_r and P_r
   For each traveling time t_i and probability p_i, update the lower bound to $LB = LB + t_i * CP_i$, and set the temporary cumulative probability to $CP_t = CP_t * (1 - p_i)$.

Step 1 of the procedure requires a time complexity of $O(n^2)$ (in the worst case n vertices need to be examined, and finding the minimal traveling time t_v for each requires $O(n)$ calculations). Step 2 of the procedure requires $O(n \log n)$ calculations and Step 4 of the procedure requires $O(n)$ calculations.

In total, calculating this lower bound requires $O(n^2)$ calculations. Note that as the constructed path becomes larger, the list of the remaining vertices becomes shorter, thus this lower bound computes faster.

Upper bound

To obtain an upper bound we used the following three techniques:

1. Run any heuristic to obtain an initial upper bound.
   This algorithm may produce a tight upper bound, as shown in Sect. 4. This bound is mainly used for pruning branches in higher tree levels.
2. For each node, calculate the AST of the path constructed until the current node, and use a greedy algorithm to complete the path with the remaining vertices. This bound may be tighter than the previous bound in branches that start with different vertices than those found using the greedy algorithm.
   This upper bound requires $O(n^2)$ calculations, but computes faster in advanced stages (as the number of unvisited vertices decreases).
3. Each feasible solution (nodes at level $|V|$ in the tree) is tested against the best upper bound found. If it produces a better result, than the best upper bound is updated and the path that was constructed until that node is marked as the best path so far.

Search termination

When each of the nodes in the tree contains a full search path or is otherwise pruned, and when there are no more nodes to examine, the algorithm terminates. The upper bound obtained is the optimal AST, and the optimal search path is the path that was marked as the best path so far.

3.3 A greedy approach

As we saw earlier, the AST is affected by both (1) the traveling times and (2) the probabilities along the searching path. We propose the following three greedy

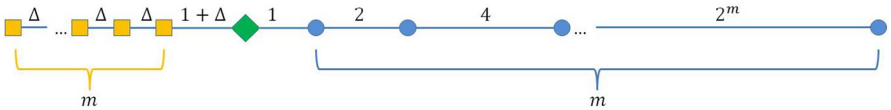

Fig. 5 A graph for Example 2

approaches for finding the minimal AST. The first approach prioritizes shorter traveling times. The second approach prioritizes higher probabilities, and the third approach is based on the ratio between the two.

1. **The closest unvisited vertex**
   At each new step of the search, turn to the closest unvisited vertex.
   Note that this vertex might have a very low probability.
2. **The unvisited vertex with the highest probability**
   At each new step of the search, turn to the vertex with the highest probability.
   Since we do not take the traveling time into consideration, this might lengthen the tour and thus increase the AST.
   Note that when all probabilities are equal, this method is essentially a random walk.
   Note that in the special case in which $t_{i,j} = t \, \forall i, j$ this method finds the optimal solution.
   This approach yields poor results and thus it was discarded.
3. **The unvisited vertex with the highest probability per traveling time ratio**
   At each new step of the search, we turn to the vertex with the best trade-off between the probability and the traveling time. (i.e., the vertex with the maximal ratio between the probability for finding the object and the traveling time needed to reach it).
   Note that when all probabilities are equal, this algorithm gives the same result as the *closest unvisited vertex* algorithm.

It should be noted that the *closest unvisited vertex* algorithm has been studied by Berman et al. [14] for scenarios in which all probabilities are equal, and obtained average results of 7% above the optimal solution.

Rosenkrantz et al. [20] proved that the greedy algorithm for the TSP (under the triangle inequality) yields an approximation of $\log |V|$. The following example will show that greedy algorithms (1) and (3) may reach solutions with a higher ratio than $log|V|$ of the optimal solution, when the probabilities are equal. This ratio (denoted as ρ) may increase when the probabilities are not equal.

Example 2 Consider the graph illustrated in Fig. 5. The search starts at the rhombus (green) vertex. The vertices in the graph are aligned on a straight line. The traveling time between each two adjacent vertices is presented on the edge connecting them. In the complete graph (containing a connecting edge between every pair of nodes), the traveling time between each two vertices is the sum of the traveling times along the straight line between them (and thus the triangle inequality holds). There are m

vertices on each side of the starting node. We assume that the probability of finding the object at each vertex is 0.5 (which implies that this example is relevant to the greedy ratio algorithm as well).

When $\Delta \to 0$, all the vertices on the left side of the stating vertex are essentially at the same position. In this case the optimal search path will visit all the squared (orange) vertices on the left side and then go back and visit the circle (blue) vertices on the right side, starting from the closest and moving towards the farthest.

A greedy algorithm will perform the opposite: at each step, the next circled (blue) vertex is closer than the closest squared (orange) vertex. At the i th step of the algorithm, the traveling time to the next circled vertex is 2^i, while the traveling time to the closest squared vertex is $2^i + \Delta$. Therefore, the greedy algorithm will choose to visit all the blue vertices first and then go back to visit all the orange vertices. In this case:

$$AST_{opt} = [1+\Delta] + \left[\sum_{i=1}^{m-1}(1-p)^i * \Delta\right] + \left[(1-p)^m * (2+\Delta)\right] + \left[(1-p)^{m+1}\sum_{i=1}^{m}(1-p)^i * 2^i\right]$$
$$\leq 1 + \left(\frac{1}{2}\right)^m * (1+2^m) \leq 3$$

$$AST_{greedy} = \left[\sum_{i=0}^{m-1}(1-p)^i * 2^i\right] + (1-p)^m * (2^m + 1 + \Delta) + (1-p)^{m+1}\left[\sum_{i=1}^{m-1}(1-p)^i * \Delta\right] \geq m-1$$

Therefore $\rho = \frac{Greedy}{OPT} \geq \frac{m-1}{3}$. Note that in this case ρ increases linearly with the number of vertices. This factor may be even worse when the probability is not constant for all the vertices.

Consider the same graph structure as in the previous example, but with $p \to 1$ for every squared vertex and $p \to 0$ for every circled vertex.

In this case:

$AST_{opt} \cong 1$ (the distance to the closest squared vertex).

The "closest unvisited vertex" method will yield $AST_{greedy} \cong 2^{m+1} + 1$ (twice the distance of the line connecting all circled vertices, and the distance to first squared vertex) and $\rho = \frac{Greedy}{OPT} \cong 2^{m+1}$.

Note that in this case the "unvisited vertex with maximal ratio" method will yield the optimal solution.

Semi greedy algorithms

As shown in Sect. 2.3, the impact of an edge on the AST becomes smaller as the edge is placed further in the search path. This implies that when trying to find a good solution, it is more important to find a good order of vertices in the earlier stages of the search path than in the later ones.

The following two heuristics emphasize the importance of choosing the first vertices along the path.

The first heuristic (1) finds a path *SPK* of length k with the minimal AST; (2) sets the starting vertex to the last vertex in *SPK*; and (3) repeats (1) and (2) after

subtracting the vertices in *SPK* from the graph, until all the vertices are included in the path.

Algorithm K-Steps-Forward $(s, k, vertex - list)$

1. If $vertex - list$ is empty then return the solution path.
2. If $|vertex - list| \rangle k$ then $k = |vertex - list|$.
3. Find the minimal AST path *SPK* of length k that starts in s and contains vertices from $vertex - list$.
4. Add *SPK* to the solution path.
5. Remove the vertices in *SPK* from $vertex - list$.
6. Set s to the be the last vertex in *SPK*.
7. Return to 1

We chose to find the *SPK* in step 3 using a simple brute force algorithm (which calculated the AST for all possible paths).

The second heuristic is similar, but with a slight difference: only the first j vertices in *SPK* are added to the solution path, and only they are subtracted from the graph. This somewhat reminds a chess game in which we look k steps forward but only perform j steps of what seems to be the best plan.

Algorithm K-Steps-Ahead-J-Steps-Forward $(s, j, k, vertex - list)$

1. If $vertex - list$ is empty then return the constructed path.
2. If $|vertex - list| \rangle k$ then $k = |vertex - list|$
3. Find the minimal AST path *SPK* of length k that starts in s from the vertices in $vertex - list$.
4. Add the first j vertices in *SPK* to the solution path.
5. Remove the first j vertices in *SPK* from $vertex - list$.
6. Set s to be the j^{th} vertex in *SPK*
7. Return to 1

At the i th step of the algorithm we calculate the AST for $\binom{n - i * j}{k}$ paths, each requiring $O(k)$ calculations. The algorithm will perform $\frac{n}{j}$ steps. Thus the runtime of this algorithm is

$$\sum_{i=1}^{\frac{n}{j}} \left(\binom{n - i * j}{k} * k \right) = O\left(k * n^{k+1}\right)$$

3.4 Ant colony optimization

Ant colony optimization (ACO) was chosen as the preferred meta-heuristic for the *Min AST* problem since it is suitable for solving path related problems, and because it was adjustable for the problem without weakening the algorithm's strength. In preliminary benchmarks, ACO outperformed other meta-heuristics and showed optimal or close to optimal results for a wide range of problem instances. The strength

of this method when considering the *Min AST* problem is in its ability to differentiate between search paths even with small AST margins. This is achieved by applying a ranking mechanism for the ant pheromone deposition (for further reading about ACO see Ref. [21]). This ranking mechanism is important since as the algorithm advances and reaches better solutions, the AST margins between the paths drop drastically. In this case in order to obtain optimal results, the algorithm has to be able to distinguish between these solutions and keep directing itself towards the optimal solution.

In this algorithm, the computation time needed for a single ant to construct an entire path is $O(n^2)$. The total number of ants that will be generated depends on the number of iterations (I) and number of ants in each iteration (A). Thus the required computation time for this algorithm is $O(n^2 * I * A)$.

3.5 Allowing repeated visits on the same vertices

The *Min AST* problem was formulated in Sect. 2 using a complete graph in which each pair of vertices is connected by a unique edge. This formulation refers to the case in which each vertex is visited exactly once. However, in many real-life searching scenarios (such as the ones presented in Sect. 1) revisiting of vertices is possible and in non-complete graphs is sometimes even necessary. Note that when repeated visits on the same vertices are allowed, valid search paths exist in any connected graph.

By performing the following adjustments, we can use the methods presented in Sects. 3.1, 3.2, 3.3 and 3.4 to solve the *Min AST* for any connected graph when repeated visits of vertices are allowed.

1. Calculate a shortest path matrix (whose common element indicates the shortest traveling time between each pair of vertices) and use it instead of the weighted adjacency matrix of the graph.
2. If the shortest path from vertex i to vertex j goes through another vertex k which was not yet visited, then the search path will not consider a direct move from vertex i to vertex j. This elimination will significantly shorten the list of candidate vertices for the next visit in any stage of the search.
3. After visiting a vertex for the first time, the probability of finding the object in it becomes 0, such that any new visit to that same vertex (performed on the way to a new vertex) will not increase the probability for finding the object.[2]

[2] When dealing with dynamic programming (Sect. 3.1), only the first adjustment is relevant.

4 Computational experiments and results analysis

This section presents several computational experiments. We tested all of the algorithms suggested in Sect. 3 on graphs with different sizes and different ranges of probabilities. The goal was to examine the running time and the size limits of the optimal solution methods (branch and bound and dynamic programming), and compare the heuristics' solution quality and running time to these results.

4.1 Experiment methodology

The test was divided into four probability groups:

A. $0.1 < p_i < 0.3$

B. $0.4 < p_i < 0.6$

C. $0.8 < p_i < 1.0$

D. $0.1 < p_i < 0.9$

100 different graphs for 8 different problem sizes were generated (10, 20, 30, 40, 50, 100, 200 and 500 vertices) for each probability group. The probability assigned to each vertex was selected uniformly between the group's probability boundaries. Each edge was assigned with a random traveling time between 0 and 10.

Not all size categories are available for all the probability groups: The DP algorithm cannot solve problems instances of size 30 and higher (in any group) due to running time and memory limits, and the branch and bound algorithm's running time is dependent on the probability distribution.

Groups B and D have the same mean probability, but group B's probability range is shorter than group D's. These probability distributions allow us to test the effect of the probability variance on the algorithms and not just on the mean.

All the experiments were tested considering scenario (a). The performance of the algorithms when considering independent or dependent probabilities is typically the same so that presenting the results for both scenarios is redundant.

We present the running time (in milliseconds) for both heuristics and optimal methods. The minimal time-span the computer was able to measure is 15.6 ms; therefore shorter running times are presented as ~0. Since in the branch and bound algorithm the running time is not deterministic (unlike all the other algorithms), we show the average, the median and the maximal running time.

The performance of all the heuristics are presented with respect to the optimal solution value for three comparison parameters:

(a) the percentage of runs in which the heuristics reached the optimal solution, (b) the average ratio between the heuristic's solution value to the optimal solution value, and (c) the worst ratio between the heuristic's solution value to the optimal solution value.

Tables 1, 2, 3 and 4 present the experiments for each of the probability groups.

Table 1 Case $0.1 < p_i < 0.3$ for all vertices (Group A)

Method	Comparison parameters	Problem size		
		n = 10	n = 20	n = 30
B&B with LB1	Running time (ms)	5.14	2658.72	664,525.9
	Median	~0	936	62,462.8
	Max	31.2	63,352	14,111,398.05
B&B with LB2	Running time (ms)	1.88	634.3	122,986.5
	Median	~0	265.2	19,983.72
	Max	15.6	6458.44	2,659,333.44
Dynamic programming	Running time (ms)	10.92	186,700.59	*
Greedy by distance	Running time (ms)	~0	~0	~0
	% Runs reached optimum	23%	4%	0%
	Average ratio optimum	111.15	117.68	130.04
	Worst ratio to optimum	152.89	174.73	198.18
Greedy by ratio	Running time	~0	~0	~0
	% Runs reached optimum	16%	3%	0%
	Average ratio optimum	115.54	120.62	128.62
	Worst ratio to optimum	210.24	183.33	190.47
Three steps ahead, one step fwd	Running time	~0	6.7	36.97
	% Runs reached optimum	27%	5%	4%
	Average ratio optimum	108.22	110.44	107.27
	Worst ratio to optimum	137.23	143.04	152.33
Ant colony optimization	Running time	24.96	94.38	207.01
	% Runs reached optimum	90%	48%	6%
	Average ratio optimum	100.01	102.57	105.48
	Worst ratio to optimum	112.38	123.35	128.07

*Denotes problem instances where the DP algorithm exceeded the running time limit (1 h for a single instance) and was out of memory

4.2 Branch and bound and dynamic programming results analysis

The experimental results show that the branch and bound algorithms (B&B) using LB1 or LB2 dominates the dynamic programming algorithm (DP). The running time of the DP is deterministic (meaning that it is constant for any problem instance of a given size) and exponential, while the running time of the B&B is non-deterministic (it may even vary between problems of the same size), and may be factorial in the worst case. In practice, the B&B algorithms were able to produce results much faster than the DP in all probability groups and for all problem sizes. Experiments of the DP algorithm for problems of size 30 and larger are not available since the expected running time of a problem instance of size 30 is more than a week. Moreover, the DP algorithm suffers from a major flaw: the memory consumption it requires grows very fast with the problem size, much like

Table 2 Case $0.4 < p_i < 0.6$ for all vertices (Group B)

Method	Comparison parameters	Problem size				
		n = 10	n = 20	n = 30	n = 40	n = 50
B&B with LB1	Running time (ms)	0.78	13.4	227.76	4068.81	63,520.33
	Median	~0	15.6	109.2	1653.61	20,014.92
	Max	15.6	109.2	3182.42	42,135.87	711,442.56
B&B with LB2	Running time (ms)	0.88	16.36	248.19	3937.15	54,621.87
	Median	~0	15.6	124.8	1747.21	19,281.72
	Max	15.6	124.8	2870.41	39,405.85	512,525.68
Dynamic programming	Running time (ms)	9.98	186,700.59	*	*	*
Greedy by distance	Running time (ms)	~0	~0	~0	~0	~0
	% runs reached optimum	35%	7%	0%	0%	0%
	Average ratio optimum	110.48	111.67	119.11	116.79	115.86
	Worst ratio to optimum	228.54	262.2	368.72	437.51	259.75
Greedy by ratio	Running time	~0	~0	~0	~0	~0
	% runs reached optimum	36%	8%	0%	0%	0%
	Average ratio optimum	110.79	111.09	120.36	115.2	115.01
	Worst ratio to optimum	202.34	262.2	368.71	238.16	245.02
Three steps ahead, one step fwd	Running time	0.93	6.39	36.97	123.86	313.56
	% Runs reached optimum	77%	40%	38%	25%	14%
	Average ratio optimum	100.55	100.41	100.44	100.4	100.24
	Worst ratio to optimum	110.05	112.56	110.89	110.35	114.93
Ant colony optimization	Running time	25.58	94.84	208.41	369.87	571.11
	% Runs reached optimum	99%	58%	11%	0%	0%
	Average ratio optimum	100.01	100.26	100.51	100.63	102.4
	Worst ratio to optimum	101.46	110.91	112.35	117.11	217.594

*Denotes problem instances where the DP algorithm exceeded the running time limit (1 h for a single instance) and was out of memory

its running time. For a problem of size 20, the memory consumption was about 0.25 GB and in problems of size 23 it exceeded the limit of 2 (Fig. 6).

The expected consumption for a problem of size 30 is 300 GB.

Table 3 Case $0.8 < p_i < 1.0$ for all vertices (Group C)

Method	Comparison parameters	Problem size							
		n = 10	n = 20	n = 30	n = 40	n = 50	n = 100	n = 200	n = 500
B&B with LB1	Running time (ms)	0.29	1.9	5.95	14.72	28.65	244.15	1771.7	47,147.5
	Median	~0	~0	~0	15.6	31.2	187.2	1497.6	39,951.8
	Max	15.6	15.6	31.2	46.8	78	873.6	10,717	168,871
B&B with LB2	Running time (ms)	0.53	3.26	11.18	30.17	58.76	518.32	3724.0	85,647
	Median	~0	~0	15.6	32.1	46.8	405.6	3088.8	72,524.8
	Max	15.6	15.6	46.8	109.8	187.2	1872	22,760	30,3359
Dynamic programming	Running time (ms)	11.23	186,874	*	*	*	*	*	*
Greedy by distance	Running time (ms)	~0	~0	~0	~0	~0	1.71	9.67	141.8
	% Runs reached optimum	69%	39%	43%	42%	51%	49%	46%	41%
	Average ratio optimum	100.87	101.23	100.95	101.38	100.62	101.61	100.9	101.66
	Worst ratio to optimum	119.56	138.7	138.2	150.04	139.43	140.67	133.6	153.72
Greedy by ratio	Running time	~0	~0	~0	~0	~0	1.71	9.6	142.11
	% Runs reached optimum	74%	48%	46%	46%	49%	48%	47%	47%
	Average ratio optimum	100.47	100.74	100.58	101.64	100.82	101.25	100.6	100.85
	Worst ratio to optimum	114.84	121.34	134.74	150.04	145.73	140.67	115.5	146.89
Three steps ahead, one step fwd	Running time	0.31	6.24	37.75	124.33	315.9	5693.4	106,841	**
	% Runs reached optimum	100%	100%	99%	100%	99%	99%	98%	**
	Average ratio optimum	100	100	~100	100	~100	~100	100	**
	Worst ratio to optimum	100	100	~100	100	100.2	~100	100	**
Ant colony optimization	Running time	24.96	94.53	208.26	369.41	572.05	2262.4	9031.6	60,399.6
	% Runs reached optimum	100%	94%	92%	88%	85%	79%	71%	54%
	Average ratio optimum	100	~100	~100	~100	100	100.02	~100	~100
	Worst ratio to optimum	100	~100	~100	~100	100	102.03	~100	100.23

*Denotes problem instances where the DP algorithm exceeded the running time limit (1 h for a single instance) and was out of memory

**Denotes problem instances where the *Three steps ahead, one step forward* algorithm's running time exceeded the time limit (1 h for a single instance)

Table 4 Case $0.1 < p_i < 0.9$ for all vertices (Group D)

Method	Comparison parameters	Problem size		
		n = 10	n = 20	n = 30
B&B with LB1	Running time (ms)	1.1	87.11	46,992.96
	Median	~0	15.6	780
	Max	15.1	577.2	2,937,483.22
B&B with LB2	Running time (ms)	0.936	71.44	21,168.56
	Median	~0	15.6	811.2
	Max	15.1	390	1,059,543.19
Dynamic programming	Running time (ms)	10.45	186,844.26	*
Greedy by distance	Running time (ms)	~0	~0	~0
	% Runs reached optimum	18%	2%	0%
	Average ratio optimum	127.99	124	121.38
	Worst ratio to optimum	411.2	486.58	270.5
Greedy by ratio	Running time	~0	~0	~0
	% Runs reached optimum	16%	1%	0%
	Average ratio optimum	125.4	127.31	118.64
	Worst ratio to optimum	411.2	402.25	226.3
Three steps ahead, one step fwd	Running time	~0	6.08	37.12
	% Runs reached optimum	73%	36%	19%
	Average ratio optimum	100.64	102.17	100.83
	Worst ratio to optimum	111.99	149.59	126.01
Ant colony optimization	Running time	25.89	95	207.32
	% Runs reached optimum	94%	51%	1%
	Average ratio optimum	100.04	100.15	100.46
	Worst ratio to optimum	103.35	111.62	127.67

*Denotes problem instances where the DP algorithm exceeded the running time limit (1 h for a single instance) and was out of memory

Our results show that the running time of the B&B algorithms varies between the probability groups: the algorithm produced results much faster for groups with higher probabilities. This finding was anticipated and is explained in Sect. 3. When the probabilities are high, the obtained bounds are tighter and allow pruning numerous branches in early stages. In the high probability group (0.8–1), LB1 performed about two times better than LB2. It seems that in these scenarios the calculation of the tighter bound is an "over-kill" and the pruning mechanism works well even without it. LB1 and LB2 performed in a similar manner when the probabilities spread was limited (0.4–0.6), and LB2 performed two times better when there was a large spread (0.1–0.9). In the low probability group (0.1–0.3), LB2 performed two times better than LB1. We can conclude that cost-effectiveness of calculating the tighter bound increases as the average probability decreases and its boundaries become wider. We note that the number of nodes LB2 generates is always smaller than LB1, but the generation time of each node

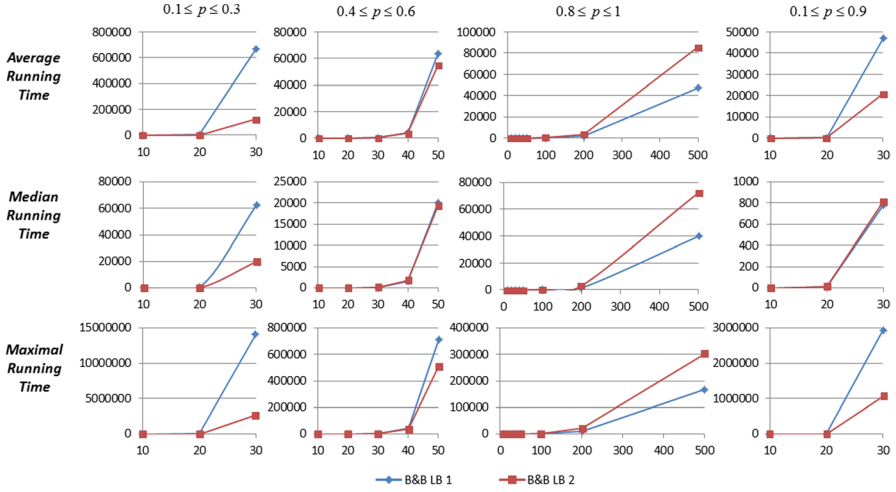

Fig. 6 Average, median and maximal running times for the branch and bound algorithms

is longer. LB2 becomes more effective than LB1 only if the number of nodes it generated was significantly smaller than LB1.

It can be seen that in both LB1 and LB2, the median running time is much lower than the average running time in all the probability groups. This is due to the probabilistic generation method of the graphs. In most cases, the running time will be close to the median, but occasionally an extremely hard instance will be generated (usually a problem instance where most of the probabilities are close to the lower limit of the probability group), and cause an unusually long running time. These instances push the average above the median, and in all cases the worst running time was many times the median (and the average).

We conclude that for any problem instance of this type, the branch and bound algorithms are preferable over the dynamic programming algorithm. LB1 should be used when the average probability is high and the variance of the probability is low, while LB2 should be used when the average probability is low and when the probability variance is high.

4.3 Heuristics results analysis

As expected, when the average probability grows, these heuristics yield better results. The contribution of edges to the AST is related to their location in the search path. This contribution becomes higher as the average probability increases (Fig. 7).

Solutions that were constructed by a greedy approach will generally have a good start, but this start may lead to very bad moves in later stages. As we described, these bad moves may have a very small impact on the AST when p is high. When the average probability is low these moves usually have a strong impact on the AST.

The *Greedy by distance* and *Greedy by ratio* algorithms produced similar results in all probability groups, even when the probability bounds were wide (0.1–0.9). These algorithms produce a result very quickly (a matter of milliseconds even in

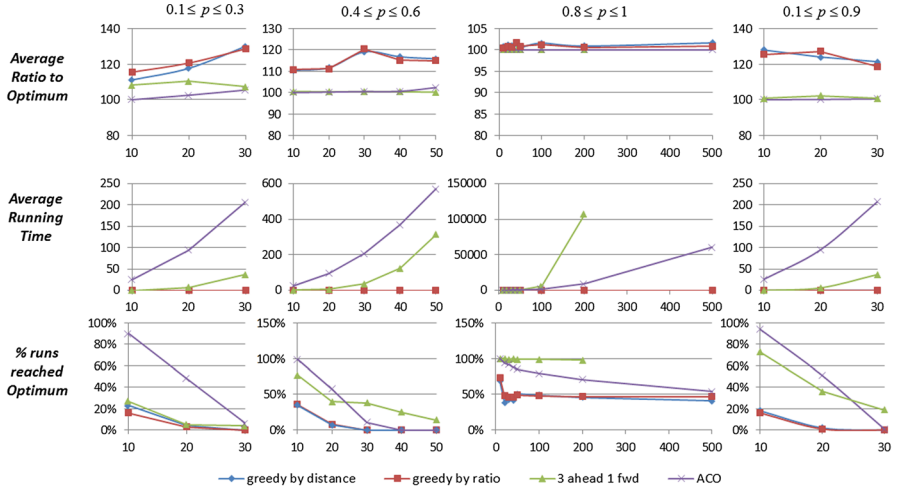

Fig. 7 The average ratio between the heuristics and the optimal solution, the average running time and the percentage of runs which reached the optimal solution

graphs of 500 vertices), but their average and worst ratio from the optimal solution is far inferior to the other heuristics. In some experiments, the solutions of these algorithms were four times the optimum.

The improved greedy algorithm: *Three steps ahead one step forward*, runs with $O(n^4)$ time while the simple greedy algorithms *Greedy by distance* and *Greedy by ratio* run with $O(n^2)$ time. This algorithm produced significantly better results than the simple greedy algorithms and similar to them, the quality of its results depends on the probability. In Group C (0.8–1) the algorithm has rarely produced a solution different from the optimal solution, and the average ratio from the optimal was negligible. In groups B (0.4–0.6) and D (0.1–0.9), the algorithm produced robust average results (100.2–102.2% from the optimum), but in some instances the solution was as high as 150% of the optimal solution. In group A the quality of the average ratio from the optimum dropped drastically (107–110%), and worst cases were as high as 150%. The *Three steps ahead one step forward* algorithm produced results in less than a second for problem instances of size 50, and a matter of seconds for problems of size 100. In instances with 200 vertices, the running time became long (~2 min in instances of size 200), and in instances with 500 vertices the running time was over an hour.

The *Ant colony optimization* algorithm showed a milder dependency on the average probability. In Group A (0.1–0.3) the average ratio between its result and the optimal solution was 105.5%, which is far superior to the other heuristics in this probability group. In other groups the algorithm produced robust results (100–102.4%).

The running time of the *ant colony optimization* algorithm strongly depends on the computational resources assigned to it (i.e. the number of iterations I and number of ants in an iteration A). Throughout the experiments, I and A were set as

constants. Thus, in small problem instances the running time of the algorithm was relatively long.

The running time and solution quality of the *Three steps ahead, one step forward* and the *ant colony optimization* algorithms depend on the amount of resources assigned to them: a wider look-ahead will yield better results but will severely increase the running time. Similarly, increasing the number of ants and iteration will generate better results but will increase the running time. The experiment showed that the resources assigned to the *Three steps ahead, one step forward* algorithm had better cost-effectiveness than those assigned to the *ant colony optimization* algorithm, when the probabilities are high. When the probabilities are small, the *ant colony optimization* algorithm made better use of the assigned resources. Thus we conclude that in problem instances where the average probability isn't low and the number of vertices is low (up to 100), the *Three steps ahead, one step forward* is preferred, while in instances where the average probability is low or the number of vertices is high (above 100 vertices), the *ant colony optimization* algorithm is preferred.

4.4 Running times when repeated visits are allowed

We examined the performance of the branch and bound algorithm when repeated visits on the same vertices are allowed and compared the running times to the ones needed when the vertices must be visited only once. Two cases were tested—the first deals with complete graphs and the second case deals with incomplete ones.

First case—multiple visits versus single visits in complete graphs

We compared the running time required for finding the *Min AST* in a complete graph using the branch and bound algorithm presented in Sect. 3.2 to the running time required when repeated visits are allowed (as presented in Sect. 3.5).

One would expect that allowing multiple visits and performing the adjustments presented in Sect. 3.5 will improve the efficiency of the algorithm and thus reduce the running time. However, after performing several tests this was proved to be wrong. On the one hand, when allowing multiple visits, the list of candidate vertices for the next visit in any stage of the search is much shorter. On the other hand, the lower bounds used in the algorithm are less efficient since the variance of the distances between each pair of vertices is smaller and thus fewer branches are bounded in earlier stages. After performing several test runs in different sizes of graphs we saw that there was no significant improvement in the running time when multiple visits are allowed compared to the case in which a vertex must be visited only once.

Second case—complete versus incomplete graphs

In many real-life searching scenarios we use a non-complete graph which usually represents a network of streets and roads. In such graphs, usually each vertex has degree 4.

We compared the running time required for finding the *Min AST* in a complete graph using the branch and bound algorithm presented in Sect. 3.2 to the running time required when the graph is incomplete and each vertex has degree 4. After performing several test runs in different sizes of graphs we saw that the running time required for incomplete graphs was significantly shorter. For a graph of 40 vertices

the running time was 15 times shorter and for a graph of 50 vertices the running time was 100 times shorter[3]. This improvement in the running time implies that the branch and bound algorithm can cope with larger graphs than the ones we tested.

5 Summary and future work

This paper presented a new searching problem for finding an object in a set of discrete locations with a known a priori probability of finding the object in each location. The objective was to find the search path that has the lowest expected searching time (*Min AST*). Two scenarios were considered for this problem: one in which the number of objects is unknown and another in which there is exactly one object. We modeled both scenarios as graph problems, and provided an analysis of the search paths and a mathematical formulation. A dynamic programming algorithm and a branch and bound algorithm with two different bounding techniques LB1 and LB2 were developed to find an optimal solution for the problem. LB1 requires less computation time but may be a loose bound, while LB2 requires more computation time but provides a tighter bound.

Our experiments showed that the branch and bound algorithm with both bounding techniques outperforms the dynamic programming algorithm. LB1 performed better than LB2 when the average probability was high and the probability variance was small. LB2 performed better than LB1 when the average probability was low.

Since the running time of the above algorithms increases exponentially with the size of the problem (and therefore, it is impractical to use them for large problems with low probability), we suggested several heuristics as an alternative.

We showed that the simple greedy algorithms yield reasonable results only when the average probability is very high. In other cases these algorithms returned poor results.

We developed an *ant colony optimization* algorithm and suggested the *K steps ahead J steps forward* algorithm which looks K steps ahead, but performs just the first J steps forward. The last two algorithms are computationally more complex, but reach significantly better results than the simple greedy algorithms.

We tested this heuristics for the case where $K = 3$ and $J = 1$ (which we called: *Three steps ahead, one step forward*) and showed that it had a better cost-effectiveness than the *ant colony optimization* algorithm in smaller problems (up to 100 vertices) and when the average probability was high. We also showed that the *ant colony optimization* algorithm had a better cost-effectiveness than the *Three steps ahead, one step forward* algorithm when the probabilities were low.

To the best of our knowledge, the branch and bound algorithm presented in this paper is the fastest algorithm for obtaining an optimal search path for discrete search problems where the goal is to minimize the expected searching time. It would be interesting to see if this holds for other discrete searching problems with different objectives and constraints.

[3] A comparison of running times in larger graphs was not performed since we were not able to use the branch and bound algorithm for complete graphs of more than 50 vertices.

An interesting extension of the *Min AST* problem for future study would be to divide the search between several searchers. Consider the example presented in Sect. 1, in which a search team is trying to find a lost traveler. The traveler might be in danger and thus the team may split into smaller teams (searching different places) in order to reach the traveler as soon as possible. One possible solution approach would be to first divide the searching area between the smaller teams, and then use the algorithms suggested in this paper by each team in its own area.

References

1. Benkoski, S.J., Monticino, M.G., Weisinger, J.R.: A survey of the search theory literature. Naval Res. Logist. **38**(4), 469–494 (1991)
2. Beck, A.: On the linear search problem. Isr. J. Math. **2**, 221–228 (1964)
3. Isaacs, R.: Differential Games: A Mathematical Theory With Applications to Warfare and Pursuit, Control and Optimization. Courier Dover Publications, Mineola (1999)
4. Alpern, S.: The rendezvous search problem. SIAM J. Control Optim. **33**(3), 673–683 (1995)
5. Stone, L.D.: Theory of Optimal Search, 2nd edn. Academic Press, New York (1975)
6. Brown, S.S.: Optimal search for a moving target in discrete time and space. Oper. Res. **28**(6), 1275–1289 (1980)
7. Stewart, T.J.: Search for a moving target when search motion is restricted. Comput. Oper. Res. **6**, 129–140 (1979)
8. Weber, R.R.: Optimal search for a randomly moving object. J. Appl. Probab. **23**(3), 708–717 (1986)
9. Morin, M. Abi-Zeid, I. Lang, P. Lamontagne, L., Maupin, P.: The optimal searcher path problem with a visibility criterion in discrete time and space. In: 12th International Conference on Information Fusion, 2009. FUSION'09, pp. 2217–2224. IEEE (2009)
10. Sato, H., Royset, J.O.: Path optimization for the resource-constrained searcher. Naval Res. Logist. **57**(5), 422–440 (2010)
11. Lau, H., Huang, S., Dissanayake, G.: Discounted MEAN bound for the optimal searcher path problem with non-uniform travel times. Eur. J. Oper. Res. **190**(2), 383–397 (2008)
12. Jotshi, A., Batta, R.: Search for an immobile entity on a network. Eur. J. Oper. Res. **191**(2), 347–359 (2008)
13. Jotshi, A., Batta, R.: Investigating the benefits of re-optimisation while searching for two immobile entities on a network. Int. J. Math. Oper. Res. **1**(1), 37–75 (2009)
14. Berman, O., Lanovsky, E., Krass, D.: Optimal search path for service in the presence of disruptions. Comput. Oper. Res. **38**, 1562–1571 (2011)
15. Ünlüyurt, T.: Sequential testing of complex systems: a review. Discrete Appl. Math. **142**(1–3), 189–205 (2004)
16. Bulhões, T., Sadykov, R., Uchoa, E.: A branch-and-price algorithm for the minimum latency problem. Comput. Oper. Res. **93**, 66–78 (2018)
17. Afrati, F., Cosmadakis, S., Papadimitriou, C.H., Papageorgiou, G., Papakostantinou, N.: The complexity of the travelling repairman problem. RAIRO Theor. Inform. Appl. **20**(1), 79–87 (1986)
18. Papadimitriou, C.H., Steiglitz, K.: Combinatorial Optimization: Algorithms and Complexity. Prentice-Hall, Upper Saddle River (1982)
19. Sahni, S., Gonzalez, T.: P-complete approximation problems. J. Assoc. Comput. Mach. **23**(3), 555–565 (1976)
20. Rosenkrantz, D.J., Stearns, R.E., Lewis II, P.M.: An analysis of several heuristics for the traveling salesman problem. SIAM J. Comput. **6**, 563–581 (1977)
21. Dorigo, M., Gambardella, L.M.: Ant colony system: a cooperative learning approach to the traveling salesman problem. IEEE Trans. Evolut. Comput. **1**(1), 53–66 (1997)