2025 NDIA MICHIGAN CHAPTER
GROUND VEHICLE SYSTEMS ENGINEERING
AND TECHNOLOGY SYMPOSIUM
<AUTONOMY, ARTIFICIAL INTELLIGENCE, & ROBOTICS>(AAIR) TECHNICAL SESSION
AUGUST 12-14, 2025 - NOVI, MICHIGAN

OPTIMIZING FIRING POSITION USAGE FOR SURVIVABILITY AND EFFECTIVENESS IN ARTILLERY SHOOT-AND-SCOOT TACTICS

Thomas Jonsson Damgaard[1], Mikael Rittri, PhD[1]

[1]Carmenta Geospatial Technologies

ABSTRACT

In shoot-and-scoot tactics, a common rule is that artillery units should not reuse firing positions; a more cautious rule is that they should not even pass near an old firing position when relocating. We use the cautious rule to define a variant of the traveling salesman problem, where an artillery unit shall use as many firing positions as possible with minimal travel time and never reuse or pass near an old firing position. We develop greedy and randomized heuristic algorithms and test them on some examples, and an auxiliary algorithm that finds a lower bound of the travel time. We also use "independent sets" of graph theory to reduce a problem instance to one or several smaller instances. We find that one can get good solutions reasonably fast by running a randomized algorithm repeatedly and that problem reduction via independent sets can improve performance.

Citation: T. Jonsson Damgaard, M. Rittri, "Optimizing Firing Position Usage for Survivability and Effectiveness in Artillery Shoot-and-Scoot Tactics" In *Proceedings of the Ground Vehicle Systems Engineering and Technology Symposium* (GVSETS), NDIA, Novi, MI, Aug. 12-14, 2025.

1 INTRODUCTION

Artillery has always played an important role in warfare, including modern warfare. "UAVs haven't made artillery irrelevant, but in many cases they have made artillery systems more effective and precise", a quote from Harry Lye [1]. Lye highlights the importance of artillery systems in suppressing hostile forces, such as SEAD missions (Suppression of Enemy Air Defenses) [2]. However, Lye stresses that they are also vulnerable on the battlefield. We intend to build algorithms that improve the survivability of artillery systems in active combat by choosing which firing positions to use and in what order.

1.1 Shoot-And-Scoot Tactics

The advantages of artillery systems are their long range and their ability to fire beyond visual line of sight, allowing them to surprise the enemy. However, after the first rounds have been fired, the enemy can use counter-battery radar systems to analyze projectile trajectories to determine their source and respond with their own artillery, and the response may take only 5 to 12 minutes [3]. For this reason, artillerymen use *shoot-and-scoot tactics:* First they find a good position to fire from. Then after firing a number of rounds, they quickly relocate about 500 meters to a different area to hide from incoming returned fire. Finally, they will move to a new position and repeat the process.

By applying shoot-and-scoot tactics, Ukrainians can operate their howitzer fleet in a safer way [4]. There are modern artillery systems, such as the Archer system [5], that are designed for speed by using automatic reload systems and rapid emplacement and displacement of guns, and by being mounted on flexible terrain vehicles. These are the key elements of effective shoot-and-scoot efforts.

1.2 Background

In his master's thesis, Temiz [3] constructed an elaborate 3D agent-based simulation of artillery tactics. The basic concepts are that artillery units like howitzers are usually combined into a group of two, three or six units. Such a group is deployed in a *position area* that needs to be 1.5 $\times$ 3 km for a group of three units or 3 $\times$ 3 km for a group of six units. The position area should contain several useful *firing positions*.

Temiz studied several aspects of shoot-and-scoot tactics, one of them being how artillery units should move between different firing positions within a position area. Of the experts that Temiz interviewed, a majority said that one should avoid reusing a firing position. And one expert went further:

"One SME [subject matter expert] mentioned that using a path that goes through an already fired region was unwise because it is easier for the enemy to shoot old targets in a shorter time since they already have the required calculations for that area." [3, page 19].

The experts also stressed that one should avoid following predictable patterns:

"They suggested randomizing decisions when choosing the next firing position or the path to the next position." [3, page 20].

Temiz's simulation is based on the game engine Unity 3D, and good routes between firing positions are calculated from terrain conditions by a Unity asset for pathfinding by Aron Granberg. The simulation uses a sophisticated algorithm to give the firing positions scores that are used when choosing the next position to scoot to. A position will get a lower score after it has been used, which decreases its probability of being reused later. But there is no strict prohibition of reuse, since the firing positions are a finite resource. And the pathfinder will not avoid passing through or near a used position when planning a path to another position.

Although Temiz focused on movements within one position area, the entire position area may eventually become too threatened so that the entire group needs to relocate to another position area. So, one could distinguish between

- "micro-scooting", where individual artillery units relocate to a different firing position within the same position area, and
- "macro-scooting", where an entire artillery group relocates to a different position area.

Other related work is discussed in Section 9.

1.3 Purpose

We have focused on the expert advice that Temiz quoted, that an artillery unit should not pass through or near a previously used firing position when relocating to another position. To our knowledge, the algorithmic implications of this advice have not been studied before. The reason why Temiz did not implement such a rule in his simulation could be that counter-battery radar will not detect an artillery unit that is just relocating. However, we think the rule has become more important in recent years due to modern military drones. For example, after the enemy has located a firing position by counter-battery radar and attacked it, they may send a surveillance drone to the position to inspect the damage and to loiter there in the hope of finding the artillery unit.

Since we want to define a computational problem in a way that is simple enough to formalize, we will not distinguish between "micro-scooting" and "macro-scooting". Instead, we just assume that a set of predetermined *positions* are known, which we consider to be geographical points. We also assume that the terrain around the positions is known and that the terrain capabilities of an artillery vehicle are known. Then, the computational problem can be described informally as follows:

1. Which of the positions shall be visited and in what order,
2. to maximize the number of positions used and minimize the travel time between them,
3. while avoiding the risk of either reusing or passing near old positions?

The definition of "near" is based on a *risk radius* chosen by a human expert. In the special case of a zero risk radius, our problem is essentially the well-known *traveling salesman problem* (TSP), so we call our problem the *traveling artilleryman problem* (TAP). The traveling salesman problem is NP-hard, meaning that optimal solutions cannot be found in polynomial time if P $\neq$ NP as is widely believed, and the traveling artilleryman problem must also be NP-hard since it is a generalization. Our ambition is to develop heuristic algorithms that can produce good solutions in reasonable time for problems of modest size, say about 20 positions.

According to point 2 above we will minimize the travel time, but travel time is just one example of travel cost. One could minimize fuel consumption instead, or some weighted sum of different travel costs.

1.4 Limitations

To keep the problem statement manageable, we assume the following:

1. There is only one artillery unit that is moving. The problem formulation should still be useful for a group of vehicles that can travel together, but not if the group needs to split up along parallel paths to the next position.
2. According to the problem statement, one should minimize the total travel time between all positions, which is not necessarily the same as minimizing the total time spent leaving the risk circle of a recently used position – which can be regarded as more important. But in practice, we believe these two measures are highly correlated.
3. The level of risk is binary. That is, the artillery unit must not travel at all within a risk circle around a previously used position, but full safety is assumed just outside the risk circle. This means that our algorithms might find an optimal solution for one risk radius, failing to see that a much better solution would be possible with a slightly shorter radius. The user could try several values for the radius, but it may be better to refine the problem statement with a level of risk that varies with distance; see Section 10.3.
4. Our problem statement assumes that all positions are equally useful, but they can differ in quality. If the quality can be represented by a single numeric weight, one could try to maximize the sum of the weights of the used positions, instead of just the number of positions used. But with several different position criteria – see Section 9.1 – one would get a multi-objective optimization problem where it can be hard to rank solutions or explore a large enough subset of the Pareto-optimal solutions.

1.5 How this Paper is Organized

Section 2 reviews the basics of graph theory. Using graph theory, Section 3 defines the traveling artilleryman problem (TAP) formally. Section 4 presents some simple heuristic algorithms for the traveling salesman problem and discusses whether they can be adapted to TAP. Two adaptations, Nearest Neighbor and Random Neighbor, are described in Section 5. Section 6 gives the results of trying the algorithms on some TAP instances. Section 7 describes a way to preprocess TAP instances whose risk radius is long enough to cause conflicts between alternative positions. Section 8 combines the conflict analysis and the two adaptations into one main algorithm. Section 9 describes related work, Section 10 suggests future work, and Section 11 concludes. Notation and acronyms are summarized on the last page.

2 GRAPH THEORY

This section describes the basic concepts of graph theory [6]; we will introduce more concepts as needed later in the paper.

In graph theory, a graph G consists of a pair (V, E) where V is a set of *nodes*, also known as *vertices*, and E is a set of *edges* between nodes, also known as *links* or *arcs*.

Nodes are just atomic objects; in a diagram they are usually shown as dots.

An edge connects two nodes. In a *directed* graph, an edge can be represented by an *ordered* pair of nodes (u, v), but in an *undirected* graph, an edge can be represented by an *unordered* pair of nodes $\{u, v\}$. Diagrams usually show edges as lines, with arrowheads if they are directed.

There is often a convention that an edge cannot connect a node to itself.

In a *weighted* graph, the nodes or the edges are assigned numeric weights. So, there are *edge-weighted* graphs and *node-weighted* graphs.

A *path* (or a *route*) in a graph is a sequence of nodes $[v_1, v_2, ..., v_n]$ such that (v_i, v_{i+1}) is an edge of the graph for all $i < n$; see Figure 1. In an edge-weighted graph, the weight of a path is defined as the sum of the weights of the edges in the path. In the rest of the paper, we shall use the term “cost” instead of “weight”, and we can write the cost of an edge between u and v as $c(u, v)$.

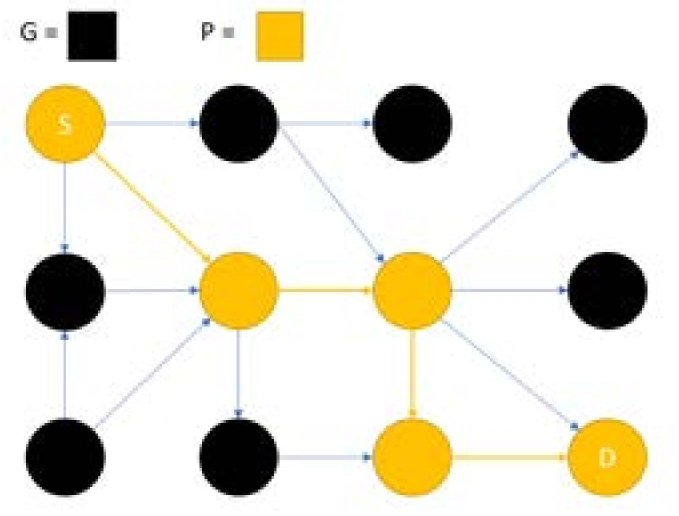

Figure 1: A black graph G and a yellow path P from node S (source) to node D (destination).

In the *single-pair shortest path problem* we seek a least-cost path from a given source node to a given destination node; this is often solved by the *A* search algorithm* [7]. In the *single-source shortest path problem* we seek least-cost paths from a given source node to all other nodes; this is often solved by *Dijkstra’s algorithm* [8, section 24.3]. The terms “shortest path” and “path length” are often used instead of “least-cost path” and “path cost”, but this is a metaphoric usage of “length” that would be confusing in our setting, since our graph nodes will represent geographic points and the literal distance between them differs from the travel cost, which will be the travel time in our examples.

In most applications, edge costs are nonnegative numbers and they satisfy the *triangle inequality*, which says that if three edges connect three nodes u, v and w, it is always the case that

$$c(u, w) \leq c(u, v) + c(v, w)$$

2.1 How Graphs will be Used in this Paper

In this paper we will use two kinds of graphs: a *terrain graph* and a *conflict graph*.

The *terrain graph* is a directed edge-weighted graph that represents the terrain for artillery operations and can be based on raster data for terrain elevations, soil strength, etc., with a node for each raster cell (pixel). Two raster cells that are adjacent (side by side or diagonally) can be connected by an edge, whose cost is the travel time along the edge for the artillery unit. The terrain graph should also include information about roads that can be used by the artillery unit. This can be done in two ways: one can rasterize the roads, treating asphalt as just another soil type, or one can convert road line segments to extra graph edges that connect two non-adjacent raster cells. Our software module for pathfinding uses the latter approach, which has the advantage that bridges and tunnels can get a correct representation.

The positions mentioned in the introduction will be represented as a small subset of the terrain nodes. However, the positions will also be used as the only nodes of a smaller undirected graph without weights, the *conflict graph*, where two positions are connected by a *conflict edge* if they are closer to each other than the risk radius.

3 TRAVELING ARTILLERYMAN PROBLEM

This section gives a formal definition of the traveling artilleryman problem (TAP) in terms of graph theory.

3.1 Input

A problem instance consists of five parts:

- A directed graph known as the *terrain graph* where each edge has a nonnegative cost.
- A subset of the graph nodes, the *positions*.
- One special position, the *start position*.
- A function R that maps each position p to a set of nodes $R(p)$, the *risk circle* for p, such that $p \in R(p)$.
- One special node, the *final safety node*, which must not be in any risk circle.

3.2 Output

A *solution* to the problem consists of three parts:

- An ordered sequence of positions $[p_1, ..., p_m]$ where p_1 is the given start position; these are the *used positions*.
- A path in the terrain graph, known as a *leg*, from p_i to p_{i+1} for each $i < m$.
- A final path in the terrain graph from p_m to the final safety node. This path is not a leg, but it can be called the *final escape path*.

3.3 The Risk-Circle Constraint

To be *valid*, a solution must satisfy the *risk-circle constraint:*

- For all $i < m$, the nodes in the leg from p_i to p_{i+1} must not be in any risk circle $R(p_k)$ for any $k < i$, and the nodes in the final escape path must not be in the risk circle of any used position except the last.

A consequence of the constraint is that a used position p must not be in the risk circle of any position that has been used earlier, because there must always be a path away from p whose nodes avoid all such risk circles, and p is the first node of any such path.

The term “feasible” is more common than “valid” in optimization literature, but we think “valid” is more clear. But in the rest of the paper we usually omit “valid” for brevity.

3.4 Ranking Valid Solutions

The *cardinality* of a solution is m, the number of used positions. The *cost* of a solution is the sum of the costs of all edges in all legs, but remember that the final escape path is not a leg. We ignore the cost of the final escape path due to a basic assumption that the artillery unit is safe when it is not in any risk circle of a position that has been used. The reason for having the final safety node in the problem statement is just to disallow solutions that arrive at a dead end without a safe escape path, as in Figure 18.

The solutions are ranked by cardinality and cost in a total lexicographic order:

- Of two solutions with unequal cardinality, the solution with the higher cardinality is better.
- Of two solutions with equal cardinality, the one with the lower cost is better.

We think this order makes sense in practice, although one could construct an extreme problem instance where a solution wins by using only one position more than another solution with much lower cost.

As usual in the optimization literature, we call an algorithm for TAP *exact* if it finds optimal solutions and *heuristic* if it finds suboptimal solutions.

3.5 Allowing any Start Position

In the basic version of TAP as described above, the start position is prescribed as part of the input.

We can get a more relaxed version of the problem statement by not prescribing a start position, instead allowing the algorithm to choose where to start. This version can be called *TAP with arbitrary start position*. However, we can solve the relaxed problem if we can solve the basic problem. That is, if we have implemented a good algorithm for the basic TAP, we could simply run it repeatedly for each position to find alternative solutions to the relaxed problem and then choose the best alternative.

3.6 About Risk Circles

In practice, we intend to study only problems where nodes are geometric points on a 2D surface and the function $R(p)$ returns the set of nodes closer to p than a fixed distance, the *risk radius*. In the formal problem statement, it was simpler to require R as part of the input, which is a way to say that there must be an unambiguous way to check whether a node is within a risk circle around a position.

4 THE TRAVELING SALESMAN PROBLEM

Our traveling artilleryman problem (TAP) is a generalization of the well-known traveling salesman problem (TSP), which is an NP-hard problem in graph theory [9, 10, 11]. Given a graph, directed or undirected, in which any pair of distinct nodes are connected by an edge with a nonnegative cost, we seek the path of minimum cost that passes through all nodes and finally returns to the start node. The canonical example is a salesman who wants to visit a set of towns with minimal travel cost.

In this section we will review TSP in the hope of finding ideas that can be applied to TAP. If there were no risk circles in TAP, one could use an approach we can call *legs as edges*. This approach interprets the positions of a TAP instance as the nodes of a TSP problem, and the least-cost legs between the positions as the edges of the TSP problem. Unfortunately, the risk circles in TAP make the least-cost edge between two positions dependent on which other positions have already been used in the solution, since the risk circles around such positions may invalidate some otherwise useful legs. Still, there are some TSP algorithms that *can* be used with the legs-as-edges approach.

4.1 Brute-force Search for TSP

There is a naive brute-force search algorithm to solve a TSP problem exactly: one can generate all permutations of the nodes and then pick the best. A set of n nodes has $1 \times 2 \times ... \times n = n!$ permutations (n factorial). But since TSP is usually formulated

Table 1: Naive brute-force search for TSP.

n	**time**
10	0.2 ms
12	0.02 s
14	3 s
16	11 min
18	49 h
20	704 days
22	810 years
24	409 600 years
26	0.25 billion years
28	173 billion years
30	140 000 billion years

for undirected graphs, each permutation has the same cost as its reversal, and since TSP requires a closed loop, each solution has the same cost as any cyclic permutation of it. This means that the number of permutations to be checked is only $n!/(2n)$, but that is still a rapidly growing function. Let us be optimistic and assume that one permutation can be evaluated in 1 nanosecond, then Table 1 shows the required execution time for graphs of different sizes. Since the sun will become a red giant in only 5 billion years, we cannot hope to find the optimal solution for a TSP instance of more than 27 nodes by naive brute-force search.

However, if we have somehow already found a good tentative solution, we can use a refined variant of brute-force search to search for a better solution. Let us say that we have 26 nodes named **`a`**, **`b`**, ..., **`z`** and that we generate permutations in alphabetical order. At some point we will start generating all permutations that begin with the 13-letter prefix **`thequickbrown`**, for example, and 13 more letters are needed so there are $13! = 6\,227\,020\,800$ such permutations. If the cost of the prefix alone is greater than the cost of our good solution, then we can skip all these permutations, since edge costs are assumed to be nonnegative. One can say that this trick uses zero as a lower bound of the cost of traversing the remaining nodes, and one could refine it further by using a tighter lower bound, for example the sum of the costs of the 14 cheapest edges that remain unused.

4.2 Sophisticated Algorithms for TSP

Since brute-force search can solve only small TSP instances, it is impressive that researchers using better algorithms have found a provably optimal solution for a TSP instance of 24 978 nodes (the cities of Sweden), even though they used more than 8 years of CPU time [12].

For more practical use, there are modern heuristic algorithms that in reasonable time can solve TSP instances with millions of nodes to near-optimality, with a high probability of producing solutions only 2 to 3 percent worse than the optimal [11]. This solution quality is higher than the usual performance of heuristic algorithms for NP-hard problems [10]. You may wonder: how can anyone know how close to optimality such a solution is, if the optimal solution is not known? The answer is that there are also reasonably fast algorithms that can produce a surprisingly tight *lower bound* for the cost of the optimal solution, the Held–Kamp lower bound [10, 13].

But since we do not know how to adapt such sophisticated algorithms for TAP, let us now look at the two simplest heuristic algorithms for TSP.

4.3 The Greedy Algorithm for TSP

An algorithm is called *greedy* if it makes choices in a shortsighted way and never backtracks to undo a previous choice. We shall describe an algorithm known as *the greedy algorithm for TSP* [10], even though other algorithms for TSP are also greedy. We write Greedy with a capital ‘G’ to indicate that we mean this particular algorithm.

1. Start by choosing the least-cost edge.
2. Then choose the least-cost edge of the remaining ones, even if it does not share any node with any previously chosen edge.

3. Continue in this way, but never choose an edge that would block the chosen edges from eventually becoming a complete solution. That is, never choose an edge that would make three chosen edges connect to the same node, or an edge that would create a closed loop that fails to include all nodes of the graph.

The Greedy algorithm cannot be used for TAP by treating legs as edges, because the risk-circle constraint makes the least-cost leg between two positions depend on which positions have been used before. And since the Greedy algorithm does not choose the edges in the order they will be traversed, the algorithm does not know which risk circles to avoid when constructing a leg.

But we can adapt the Greedy algorithm to find a lower bound of the cost of a TAP solution of a given cardinality m. We can do that via the legs-as-edges approach by making the algorithm *overgreedy:* that is, it shall ignore the prohibition of blocking edges in paragraph 3, and when calculating the least-cost leg between two positions it shall ignore risk circles of other positions. This variant is "overgreedy" in the sense that it is usually too greedy to produce a valid solution: it will just produce a set of legs that may be disconnected or form many small loops. But any valid solution of cardinality m must contain $m-1$ legs, so when the overgreedy algorithm has chosen $m-1$ legs, the sum of their costs must be a lower bound of the cost of any solution of cardinality m. This algorithm may not give a *tight* lower bound, but its simplicity makes it attractive. For examples and more details about the algorithm, see Section 6.4.

4.4 Nearest Neighbor for TSP

Another TSP algorithm known as the Nearest Neighbor algorithm (NN) is also greedy in the general sense but collects edges in a different order.

First, a terminology warning. The colloquial meaning of "nearest" is minimum *distance*, but in Nearest Neighbor for TSP this is just the metaphorical usage of graph theory where the "distance" along a graph edge should be understood as the *cost* of the edge (see Section 2).

The Nearest Neighbor algorithm is simple:

1. Start by choosing a random start node and put it in an ordered list of nodes.

2. Repeatedly choose the least-cost edge from the last node of the current list and append its destination node to the list. But don't choose an edge that returns to any previous node of the list, unless we are choosing the final edge to close the loop.

Instead of choosing a random start node, one could run the algorithm repeatedly starting from each node in turn, and then choose the best solution. This approach is known as Repeated Nearest Neighbor.

For graphs where the edge costs satisfy the triangle inequality, Nearest Neighbor for TSP will seldom produce a solution that is more than twice as costly as the optimal one [10]. And since the algorithm chooses the edges in the order they will be traversed, it can easily be adapted to solve TAP problems with the legs-as-edges approach.

5 BASIC TAP ALGORITHMS

This section describes a Nearest Neighbor algorithm for TAP and a randomized variant.

5.1 Nearest Neighbor for TAP

The Nearest Neighbor algorithm for TAP is similar to the one for TSP.

1. Put the prescribed start position in an ordered list of positions.

2. Repeatedly choose the least-cost *valid* leg from the last position to a *valid* destination position and append the destination position to the list, until no more valid positions can be reached.

3. If the final escape node can be reached by a *valid* path from the last position, terminate by success, otherwise by failure.

In step 2, a leg is *valid* if it avoids all risk circles of previous positions in the list. We cannot demand that a valid leg also avoids the risk circle of the current position since it starts from its center; instead we say that a destination position is *valid* only if it is outside the risk circle of the current position.

In step 3, the final escape path is *valid* if it avoids all previous risk circles. Again, we cannot demand that it also avoids the risk circle of the last position since it starts from its center.

5.2 Random Neighbor for TAP

Random Neighbor is a simple variation of Nearest Neighbor.

The difference is that for each choice made in step 2, we do not necessarily choose the *least-cost* valid leg from the last position to another valid position. Instead, we choose between the possible legs randomly, using probabilities that are constructed from the leg costs by a *bias function*. To keep some of the useful greed of Nearest Neighbor, the bias function should give higher probability to legs of lower cost. The average solution quality of Random Neighbor may be worse than Nearest Neighbor, but we can run Random Neighbor many times and hope to get some good solutions eventually.

This idea of balancing greed with randomness is one cornerstone of *GRASP*, Greedy Randomized Adaptive Search Procedures, which is a meta-heuristic – that is, a set of general design principles for heuristic algorithms. GRASP is described by Resende and Silva [14] who list eight main approaches to achieve a good balance, one of them being the bias functions of Bresina [15].

The other cornerstone of GRASP is using local search to improve the generated solutions. Local search is excellent for TSP [10, 11] but we have not tried to implement it for TAP; see Section 10.1.

6 EXPERIMENTS WITH TAP ALGORITHMS

Since TAP is a generalization of TSP and therefore NP-hard, we do not care about worst cases or asymptotic complexity. We just hope to develop algorithms that can handle typical problem instances of modest size by finding solutions that are good but not necessarily optimal. This section describes our practical experiments on some problem instances.

6.1 Problem Examples

We tried to find realistic problem examples in the literature. Although Temiz made some screenshots from his model [3, figures 4 and 5], he did not publish example coordinates of firing positions. Instead, we generated problem instances from the 40 case points recommended by Quinn and Kunkleman [16], of which the 20 westmost ones are intended for friendly artillery and the 20 eastmost ones for enemy artillery. In Figure 2 we have used our own software to generate a map of the same area as is shown in Quinn and Kunkleman's Figure 3, and we added black dots for the 40 case points in their Table 3. We labeled their 20 friendly case points by `a`, `b`, ..., `t` and their 20 enemy case points by `A`, `B`, ..., `T`. The 20 friendly case points are in the northwest of a large area without major roads; this is the *Hohenfels Training Area* in Germany. The area does contain some minor roads that we do not show in our screenshots, because we think they would not provide any benefit for a tracked vehicle. You can read more about Quinn and Kunkleman's work in Section 9.1.

The case points of Quinn and Kunkleman are not ideal for realistic examples of the traveling artilleryman problem since they represent the centers of position areas, whereas shoot-and-scoot usually refers to scooting between firing positions inside one position area. Still, we believe their 40 case points have an interesting geographic distribution and are suitably many, so we think they give examples that are realistic enough.

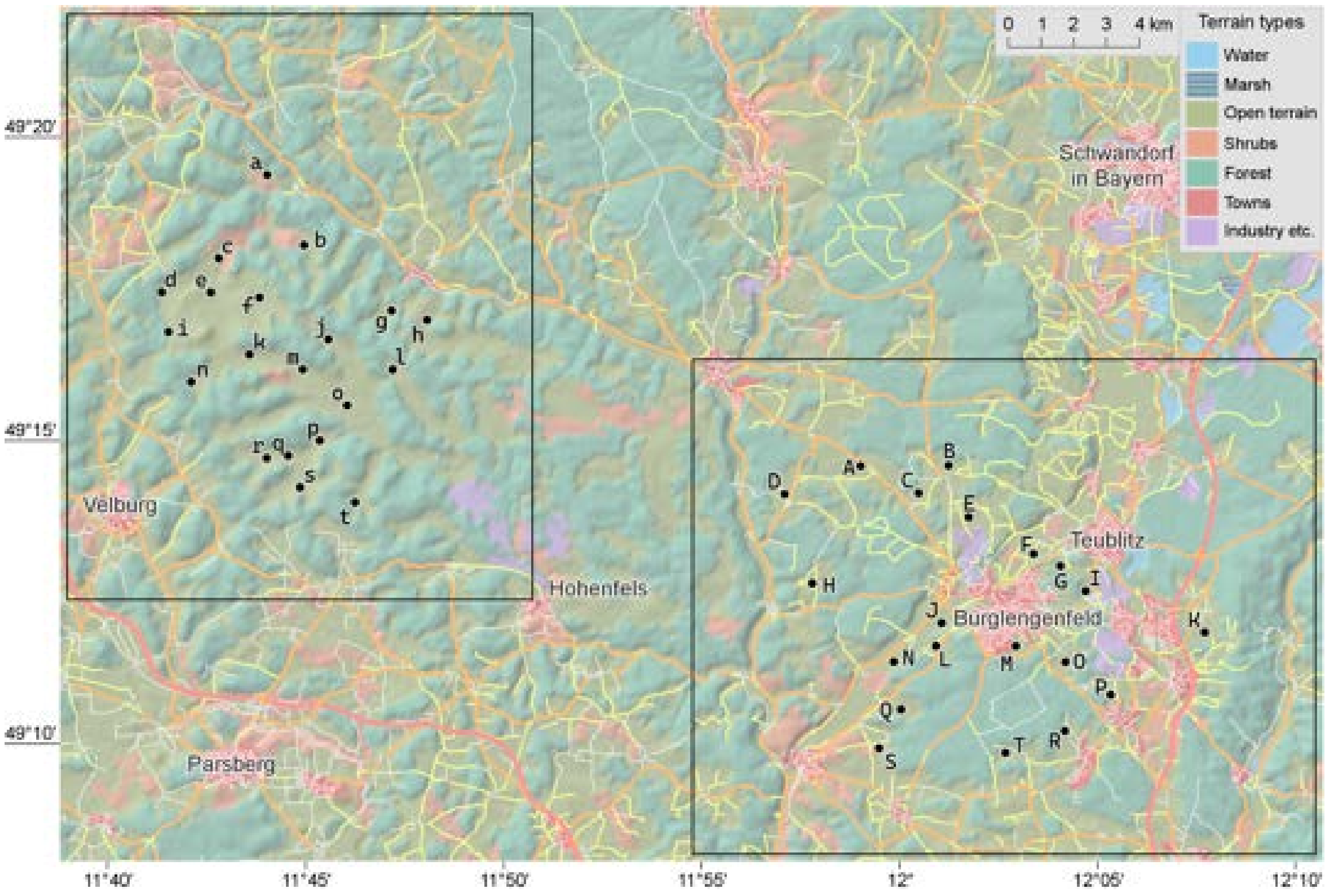

Figure 2: A scenario from Quinn and Kunkleman [16].

Our first problem instance, which we call W-20 (W for West), is based on the 20 friendly positions with a final safety node in the northwest, with **`a`** as the prescribed start position, and with a risk radius of 2 km, enough to make many risk circles contain positions other than their own center. See Figure 3.

Our second problem instance, K-20, is based on the 20 enemy positions with **`K`** as the prescribed start position, also with a risk radius of 2 km, and with a final safety node in the southwest. As a slight variation we defined a third instance, D-20, which differs only in starting from **`D`**. These instances are shown together in Figure 4. (Screenshots of the eastern examples are shown in a more zoomed-out scale in order to fit.)

We did not have strong preferences for particular start positions, but first we chose **`a`** and **`K`** for two reasons:

1. Neither of them contains another position in its risk circle, so using them does not exclude any other position.

2. Each of them is relatively far away from its two nearest neighbors, so the two legs to these neighbors are probably costly. Therefore, a solution that starts at one of these should benefit by having to use only one of these costly legs instead of both.

Later, we selected **D**, which is also somewhat isolated, as an alternative to **K**, because we discovered that the Nearest Neighbor algorithm gives an excellent solution when starting from **K**, which we suspected was an atypical result.

The precise location of a final safety node will not affect the cost of the solutions to a TAP instance, since the cost of the final escape path is not added to the solution cost. The idea is to place the final safety node somewhere in accessible terrain outside the convex hull of all possible risk circles, to disallow solutions that reach a dead end as in Figure 18.

To fully define our problem instances, we also need a terrain graph. We have chosen an approach that is convenient for our experiments, where the terrain graph is constructed internally by our software module for pathfinding in terrain, the TerrainRouteOperator [17]. More specifically, the structure of the terrain graph is based on elevation data, terrain type data and road data, and the cost for each edge is calculated from these geodata and from a description of the vehicle's terrain capabilities and road speeds. We have let the terrain graphs be bounded by the two black rectangles in Figure 2, which also appear as the black frames of most of our screenshots (some initial experiments had indicated that the pathfinder should never need to go beyond these rectangles). Letting our pathfinder software construct the terrain graph is convenient for us, since we can then use it to find the least-cost leg from one position to another.

However, this choice has the drawback that other researchers will not be able to reproduce the same terrain graph exactly even if they have the same geodata, since they would need to know an impractical amount of our implementation details. If we ever would like to publish our problem examples as public benchmarks, we would need to define a machine-readable export format for the internally generated terrain graph and the other parts of the problem input.

We have used several geodata sources which are

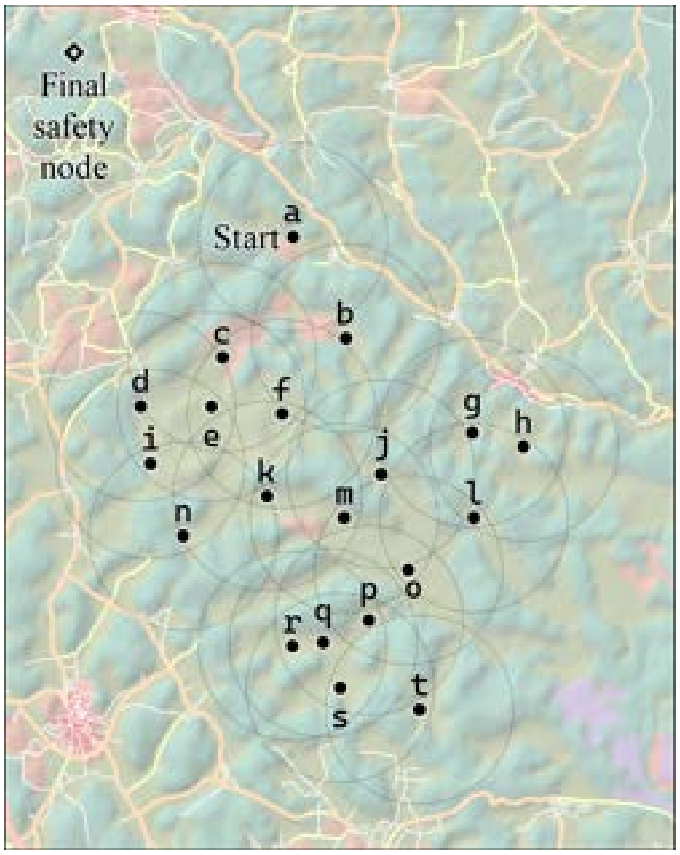

Figure 3: The TAP instance W-20.

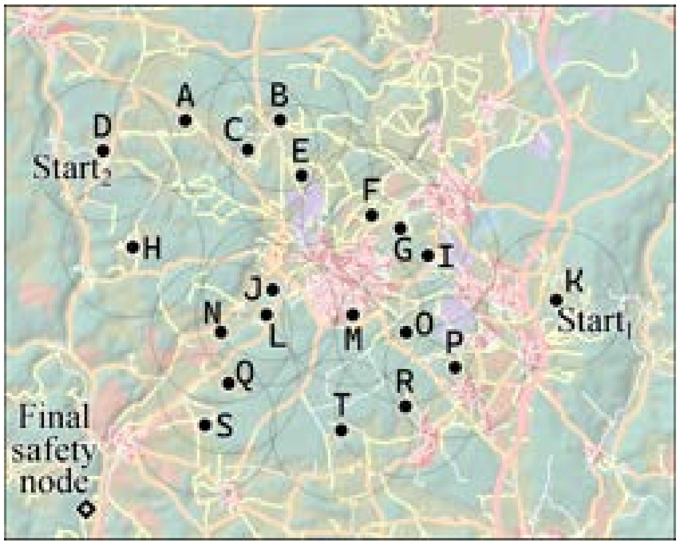

Figure 4: The TAP instances K-20 and D-20.

credited in Section 14. Of these, the CORINE land-cover dataset is somewhat problematic since it has many more categories than the number of colors that the human eye can distinguish in a hillshaded background map. So in our screenshots we have grouped the original categories into only seven simplified categories; see the legend in Figure 2. Some input to our pathfinding software is based on the original CORINE categories, but they must be converted to soil strength classes and roughness classes [17, 18] in a way that is just an estimate.

Furthermore, we do not know the precise terrain capabilities of the howitzer models studied by Quinn and Kunkleman, so we have used terrain capabilities that describe another kind of tracked vehicle that is much smaller. Our vehicle model may be too optimistic about trafficability in forest, and in the areas of interest there are only some small scattered parts that are considered untrafficable. But our pathfinder tends to avoid forests since they are assumed to slow the vehicle down.

6.2 Resolution and Computation Time

In our experiments, we have used a terrain data resolution of 3×3 arc seconds, corresponding to about 60×93 meters at the latitude of Hohenfels, which is coarser than what is recommended for our pathfinding software.

One effect of the coarse resolution is that the terrain graph gets fewer nodes and edges, which gives faster computations, and you should remember this when reading about our computation times (which were measured on an Intel® Core™ i7-10700K processor). Higher resolution would cause slower computations, or to be more precise: terrain graphs are *sparse* since the number of edges is proportional to the number of nodes, and for sparse graphs the time complexity of Dijkstra's algorithm (used by the pathfinder) is $\mathcal{O}((|E| + |V|)\log|V|)$, where $|E|$ is the number of edges and $|V|$ is the number of nodes.

Another effect of the coarse resolution is that the generated terrain paths will be less realistic, an effect that is similar to the uncertainties about terrain types and vehicle capabilities (mentioned in the previous subsection). Fortunately, for the purpose of experimenting with TAP algorithms it should not matter how well the terrain paths match the reality around Hohenfels: we just want the basic structure of the terrain graph to be typical for TAP instances.

For operational use, though, the resolution would probably be much too coarse, since the fastest path from one position to another can be very sensitive to resolution. For example, a terrain feature that is long and narrow can be present in high resolution but missing in low resolution. If the feature is an obstacle like a river, pathfinding in low resolution could generate an impossible path across the river, and if the feature is an opportunity like a piece of land between two lakes, then pathfinding in low resolution would overlook the opportunity.

Our pathfinding software for terrain vehicles [17] is based on a traditional Swedish terrain classification system [18] for which a resolution of 10 to 25 meters is recommended, and our implementation assumes that the terrain vehicle is smaller than the resolution. If one would like to use a very high terrain resolution, then different wheels of the vehicle could be on different slopes and soil types, which would make the analysis difficult: one would need more detailed terrain classification systems and better vehicle modeling as in the Next Generation NATO Reference Mobility Models [19].

6.3 Should we Precompute Leg Costs?

When implementing our algorithms for TAP, we faced a basic design decision: should leg costs be precomputed? Of course, doing so would be pointless for Nearest Neighbor since this algorithm is run only once and requests each leg cost at most once. But precomputed leg costs could make sense for Random Neighbor which is intended to be run many times.

If we do not precompute leg costs, the computation time of our TAP algorithms will be dominated by our pathfinder module. In this module, paths in a terrain graph are found either by A* search (when there is one goal node) or by Dijkstra's algorithm (when there are several alternative goal nodes). In both cases, most of the computation time is spent on managing a large priority queue of nodes, the frontier of the search. There is no doubt that the Random Neighbor algorithm for TAP would become much faster if it could look up leg costs in a precomputed table.

The difficulty is that the required size of the precomputed table grows exponentially with the number of positions in a TAP instance. To be more precise, let n be the number of positions, then we need to compute the least-cost leg for each of $n^2 - n$ ordered pairs of distinct positions: that is only a quadratic number of pairs. But for each pair, the least-cost leg can depend on the used/unused status of the other $n - 2$ positions, and we do not know that status when the table is calculated. So, for each pair we have to calculate several alternative least-cost legs, one leg for each of the 2^{n-2} states in which the other $n - 2$ positions can be. Thus, the table must contain $(n^2 - n)2^{n-2}$ leg costs.

We must also consider the time it takes to fill the table. Let us assume that for a particular used/unused state for all n positions, Dijkstra's algorithm will need 0.3 seconds to find the least-cost legs from one position to all others; this estimate is based on our implementation (but see the disclaimer in the previous section). For each of 2^n states we would need to run Dijkstra's algorithm n times, one for each position, so the total time would be $0.3n2^n$ seconds.

Table 2 gives examples of the required computation time and table size (assuming 2 bytes per leg cost). The main difficulty is the long computation times, which indicate that precomputed leg costs will not improve operational use of the Random Neighbor algorithm for TAP.

However, for problem instances that are small enough, precomputed leg costs may still be useful together with brute-force search (Section 4.1) adapted for TAP. This design is unlikely to be fast enough for operational use, but could be useful in an academic setting to find optimal solutions for benchmark instances, which would tell us how close to optimality the heuristic algorithms are. Although the problems must be "small enough" for this approach, we will demonstrate in Section 7 that some TAP instances can be reduced to smaller ones with the same optimal solutions.

Table 2: Estimates for a table of leg costs.

n	**size**	**time**
7	2.6 KiB	$4\frac{1}{2}$ min
8	7 KiB	10 min
9	18 KiB	23 min
10	45 KiB	51 min
11	110 KiB	1 h 52 min
12	264 KiB	4 h
14	1.4 MiB	19 h
16	7.5 MiB	87 h
18	38.2 MiB	16 days
20	190 MiB	72 days
22	924 MiB	320 days
24	4.3 GiB	3.8 years
26	20.3 GiB	16.6 years
28	94.5 GiB	71 years
30	435 GiB	306 years

6.4 Overgreedy Lower Bounds

It is hard to judge the solution quality of a heuristic algorithm when the optimal solution is unknown, but a lower bound of the cost of an optimal solution will give *some* information about the quality. So, let us begin by finding lower bounds for our TAP instances using the overgreedy algorithm of Section 4.3.

When used for TAP, the overgreedy algorithm can give a result only for a given cardinality (the number of positions used by a solution). For our first TAP instance W-20, we shall use cardinality 10 since this is the highest possible, as we will learn in Section 7, so the overgreedy algorithm needs to find the 9 least-cost legs. For W-20 the prescribed start node is `a`, so we can refine the algorithm description of Section 4.3 by starting with the least-cost leg from `a` and then add the 8 least-cost legs of the remaining ones. The overgreedy algorithm should never ask the pathfinder to find a leg between two positions closer to each other than the risk radius, since two such positions cannot be used in the same solution, but otherwise the pathfinder should ignore risk circles. Each pair of positions will have two different least-cost legs between them, one in each direction, since the terrain graph is a directed graph, but the overgreedy algorithm should select at most one of them since both cannot be used in a solution.

Figure 5 shows the result for W-20. Since our leg costs are travel times, they are displayed as *minutes*:*seconds* or as *hours*:*minutes*:*seconds*. The lower bound is the sum of the nine leg costs: 2 h 19 min 20 s or 8360 seconds.

For our second problem instance, K-20, the maximal cardinality is 11 as we shall learn in Section 7, and the overgreedy algorithm chooses the least-cost leg from `K` and the nine least-cost legs from the rest; Figure 6.

For our third problem instance, D-20, the maximal cardinality is also 11. The overgreedy algorithm chooses the same legs as for K-20, because the ten least-cost legs overall happen to include the least-cost leg both from `K` and from `D`. So, the lower bound is the same.

The best solutions we have found for W-20, K-20 and D-20 have costs that are 29%, 44% and 48% higher than the corresponding lower bound. We think that our best solutions are nearly optimal while the lowest bounds are not tight, but we do not have any proof of that.

6.5 Trying Nearest Neighbor

Now that we know lower bounds for some TAP instances, let us try the simplest heuristic algorithm

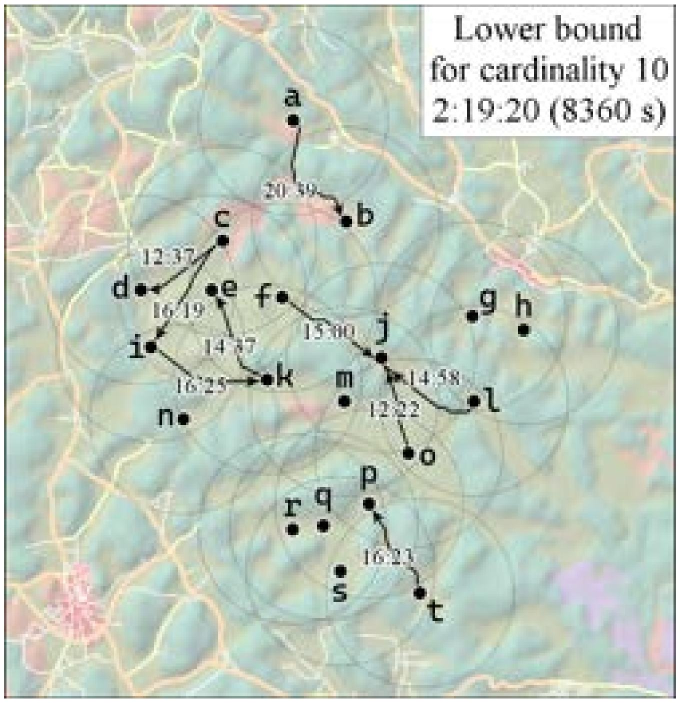

Figure 5: Overgreedy lower bound for W-20.

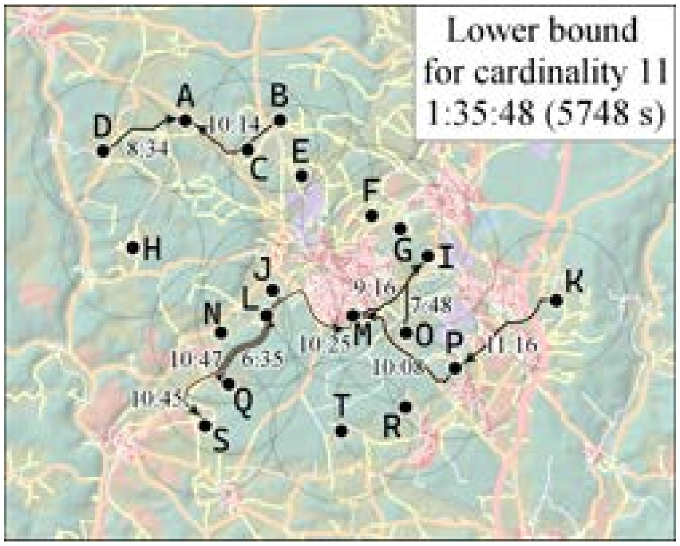

Figure 6: Overgreedy lower bound for K-20 and D-20.

to find solutions. In Section 7 we will describe how one can sometimes reduce problems by removing positions that cannot be part of an optimal solution, but for now we shall use all positions of the TAP instances.

First, we use Nearest Neighbor for TAP with legs as edges on our first problem instance, W-20

(Figure 3). As an example of how the algorithm proceeds, see Figure 7. It started from the prescribed start position `a` and is shown after choosing a sequence of five positions. For the next choice of leg, the pathfinder has generated all possible legs from the current position `o` to some other valid position, with their costs. (In this step, the pathfinder should use Dijkstra's algorithm once instead of A* search many times, for efficiency.) The grayed-out positions can no longer be chosen since they are inside a risk circle of a previously used position. The remaining valid goal positions are colored green, and of them it is `q` that is reachable with the least cost, 16 min 33 s, so it will be chosen as the next one. Note that the leg from `o` to `q` passes through or very near `p`, but this is allowed since `p` has not been used as a firing position.

Figure 8 shows the complete solution, where the final escape path is shown as a dashed line from the final position `g` to the safe node, although most of the dashes coincide with the last half of the leg from `d` to `g`. The top right label has a first line showing how many positions are used and in what order, and a second line showing the cost of the solution. After using 10 positions there are no more valid ones left, and we will prove in Section 7 that more than 10 cannot be used in this problem instance. The cost of this solution is its total travel time, 4 h 02 min 05 s or 14525 s, which is mediocre: about 35% worse than the best solution we have found for this instance. The computation time for each added leg was about 0.3 s and the entire solution required about 3 s (but see the disclaimer in Section 6.2).

What greedy choices made this a mediocre solution? For a human being, it is obvious that it was a mistake to leave position `g` unused when going south from `a` the first time, since the risk circles generated going south eventually force a long detour for the last leg from `d` when only `g` and `h` remain as valid choices. Another improvement is possible in the southeast by visiting `t` before `q`. Other local improvements may be possible, but we did not see any obvious ones. These two manual

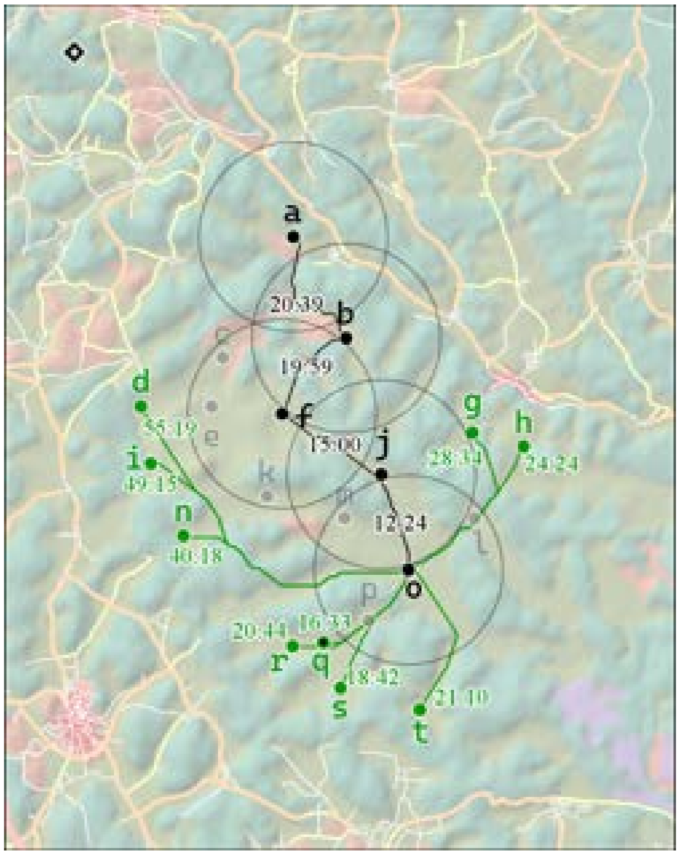

Figure 7: Nearest Neighbor in progress, W-20.

improvements give a solution (Figure 10) whose cost is only 4.8% higher than the best solution we know of. A human and a computer may be able to find good TAP solutions by cooperating: the computer uses the Nearest Neighbor algorithm and the human improves the solution manually.

For our second TAP instance, K-20, the Nearest Neighbor algorithm gave the solution shown in Figure 9, which is excellent. If the solution had used `F` instead of `I` its cost would have been lower but only by 1.8%, and the improved solution is the best we have found for this instance. In fact, all our attempts with Random Neighbor (see next subsection) found only four solutions that were better than the one from Nearest Neighbor. For our third TAP instance, D-20, where one shall start from `D` instead of `K`, the Nearest Neighbor algorithm gave the solution shown in Figure 11, which uses only ten positions. This is a poor solution since one can use eleven positions also

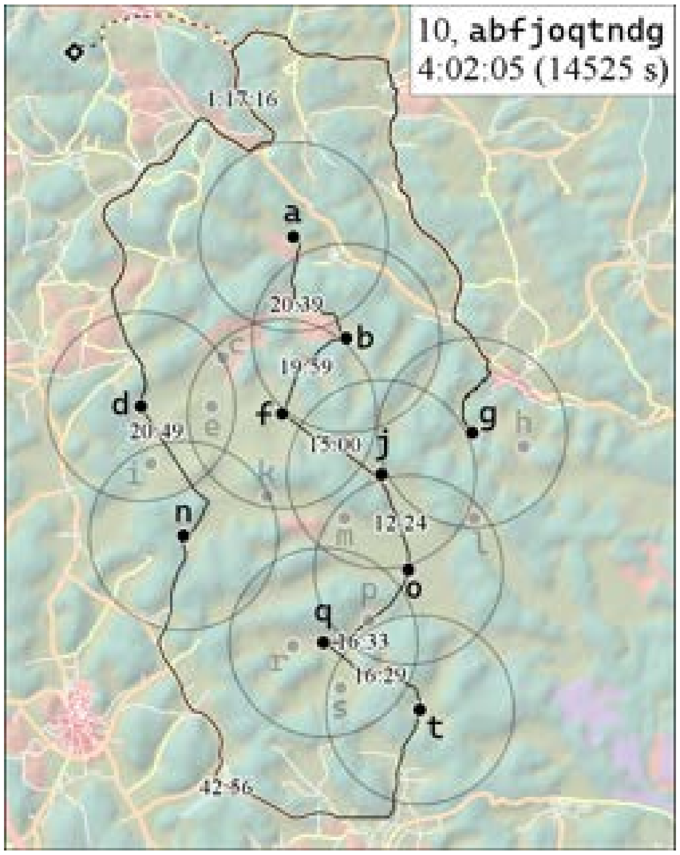

Figure 8: Result from Nearest Neighbor, W-20.

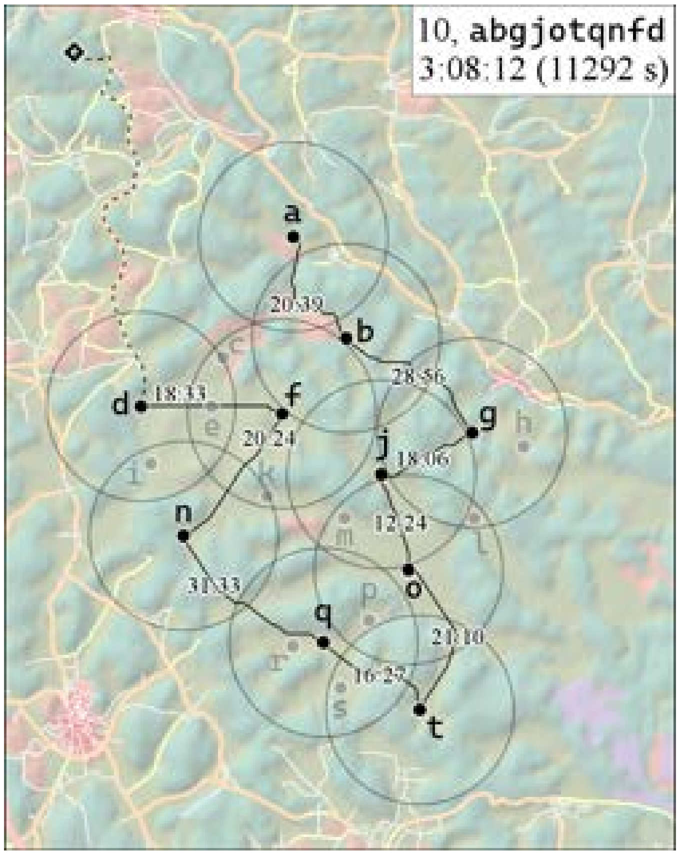

Figure 10: NN, W-20, with two manual improvements.

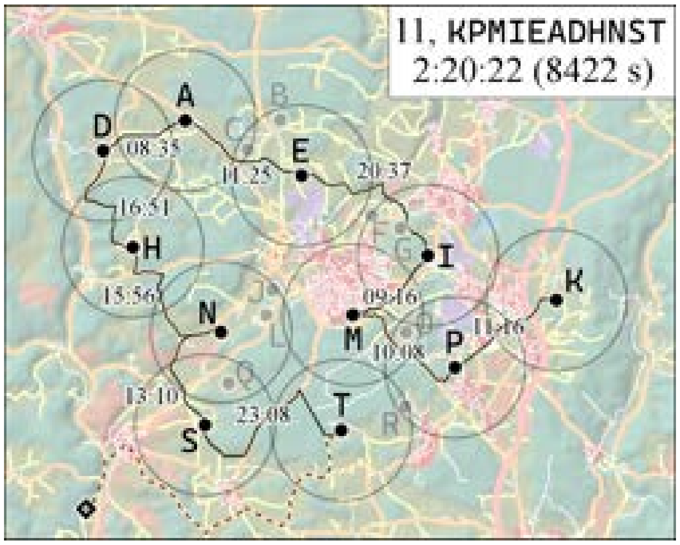

Figure 9: Result from Nearest Neighbor, K-20.

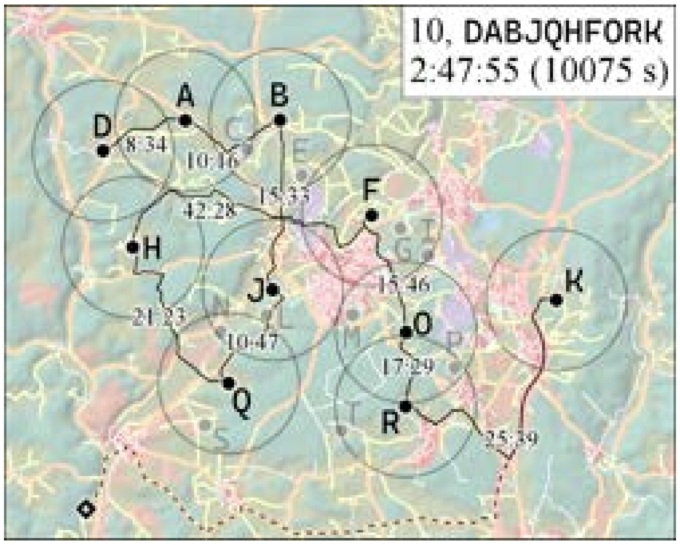

Figure 11: Result from Nearest Neighbor, D-20.

when starting from **D**, and by our lexicographic ranking, any solution with cardinality 11 is better than all with cardinality 10.

6.6 Trying Random Neighbor

As mentioned in Section 5.2, the Random Neighbor algorithm uses *bias functions* to balance greed with randomness. The term "bias function" was coined by Bresina [15], who lets the argument to his bias functions be the *rank* of the alternatives when sorted by cost; that is, the least-cost alternative gets rank 1, the next gets rank 2, etc.

If one demands that a bias function produces probabilities, it will be difficult to specify by a formula since the sum of the probabilities for an exhaustive set of disjoint events must sum to 1. So, like Bresina, we allow our bias functions to produce nonnegative *weights* that do not need to sum to 1, and then we say that alternatives shall be chosen with probabilities that are proportional to their weights. In other words, the normalization of weights to a sum of 1 is handled by the main algorithm rather than by the bias function.

On the other hand, when bias functions are plotted in diagrams, they are easier to compare if the function values have been normalized. Since a new normalization must be computed for each specific set of alternatives, we have plotted diagrams based on the set of alternatives in Figure 7 where the algorithm shall choose the next position from nine possible alternatives.

We have tried three families of bias functions, where each family has one or two parameters that can be instantiated to get a bias function.

Exponential bias functions

The first family is *exponential* bias functions with a base parameter b between 0 and 1. These functions are defined as follows, where r is the rank of an alternative:

$$\text{exp-bias}_b(r) = b^r \tag{1}$$

When Random Neighbor uses this kind of bias function, we can abbreviate it to ExpRankRN. In our experiments we let the base parameter b get the values 0.75, 0.50, 0.25 and 0.0625. Figure 12 shows that 0.75^r (the green curve) has a thicker tail that will make ExpRankRN choose a costly option more often than when the slim-tailed function 0.0625^r is used (the red curve).

Polynomial bias functions

The second family is *polynomial* bias functions with a nonnegative degree parameter d. For this family, we decided to let their argument be the cost c of an alternative rather than its rank, on a hunch that some useful information can be lost when converting cost to rank.

$$\text{poly-bias}_d(c) = c^{-d} \tag{2}$$

This formula is convenient since we do not need to worry about the unit of measure for our leg costs. The weights would become much smaller if we switched our leg cost unit from seconds to minutes, but there would be no difference after the weights are rescaled to probabilities that sum to 1. When Random Neighbor uses this kind of bias function, we can abbreviate it to PolyCostRN. See Figure 13.

Gaussian bias functions

In both the exponential and polynomial bias families, the functions are convex (by convention, that means that the area *above* the function curve is convex). In our experiments, which will be described in detail soon, we found that the tail of the functions needed to be kept slim to decrease the probability of choosing a costly alternative, but tweaking the parameters for a slimmer tail would also make the "neck" steeper (the beginning of the function curve). With a steeper neck, the bias function will choose more greedily among the best alternatives, and more and more of the generated solutions will just reproduce the solution from Nearest Neighbor.

For our third and final family we used bias functions that have a concave neck followed by a convex tail, in the hope that one could get a slim tail without an overly steep neck. The family can be called *Gaussian* bias functions since they are inspired by the bell-shaped Gaussian (normal) probability distribution. To avoid a dependency on the leg cost unit, we normalized the costs for each set of alternatives. That is, we found the minimal cost c_{min} of the set and for each alternative with a cost of c, we let its normalized cost c' be defined by

$$c' = c/c_{\text{min}} \tag{3}$$

Then we can define the Gaussian bias functions with two parameters μ and σ.

$$\text{gauss-bias}_{\mu,\sigma}(c') = \exp\left(-\frac{(c'-\mu)^2}{2\sigma^2}\right) \tag{4}$$

Since such a function is decreasing only where $c' \geq \mu$, we should ensure that μ is at most 1, the smallest value of c', otherwise the function will be anti-greedy for $c' < \mu$. We have only tried with $\mu = 1$. When Random Neighbor uses this kind of bias function, we can abbreviate it to GaussCostRN. See Figure 14.

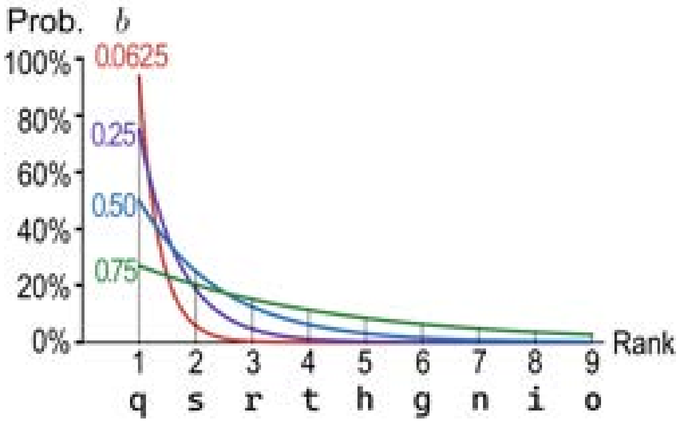

Figure 12: Probabilities from ExpRankRN for the nine alternatives in Fig. 7, using various values of b.

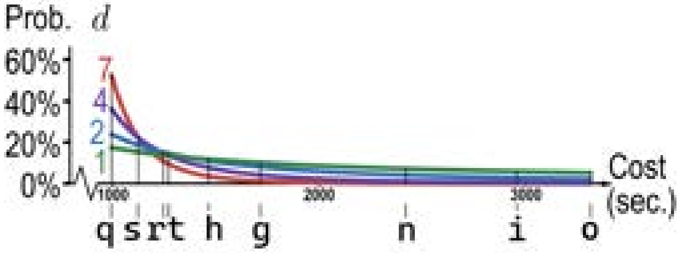

Figure 13: Probabilities from PolyCostRN for the nine alternatives in Fig. 7, using various values of d.

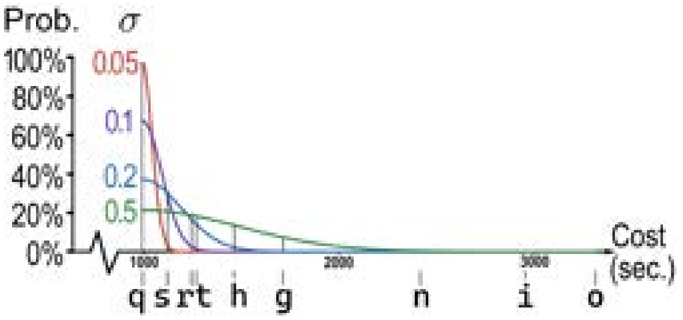

Figure 14: Probabilities from GaussCostRN for the nine alternatives in Fig. 7, using $\mu = 1$ and various values of σ.

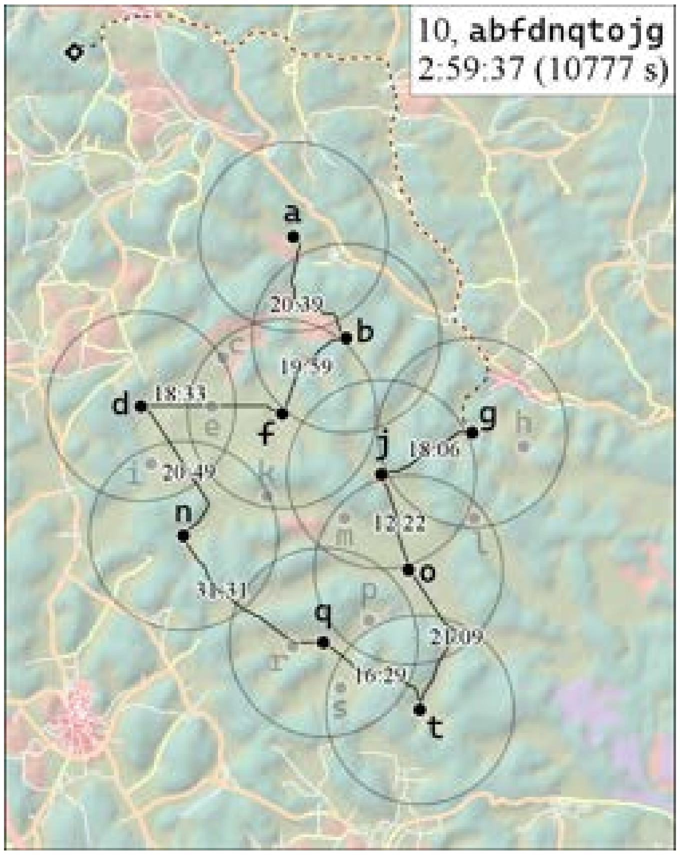

Figure 15: Winning solution of W-20.

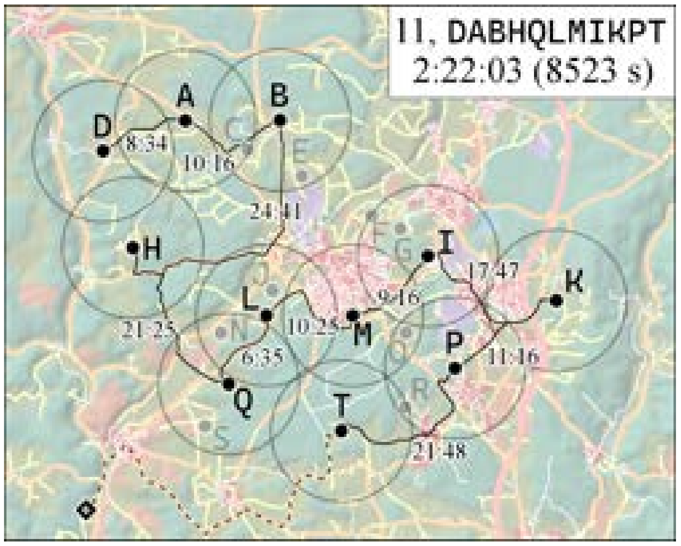

Figure 16: Winning solution of D-20.

Highlighted solutions

Before tabulating the results from using these bias functions, let us look at some highlights. The winning solutions for W-20 and D-20 are shown in Figures 15 and 16. The winning solution for K-20 used 11 positions with a cost of 8274 seconds; we do not show it in a separate figure since it differs from the Nearest Neighbor solution for K-20 (Figure 9) only by using `F` instead of `I`. Figure 17 shows a valid but poor solution for W-20 that uses only 7 positions: after the 7th has been used, there are no more positions outside the risk circles. Figure 18 shows how the search can reach a dead end: after using the last position `j` the vehicle cannot reach the final safety node, because it must not pass through the surrounding risk circles of `b`, `g`, `o`, and `k`. Since our algorithm never backtracks, it must give up in this situation (it does not see that using `h` instead of `g` would have allowed escape from `j`).

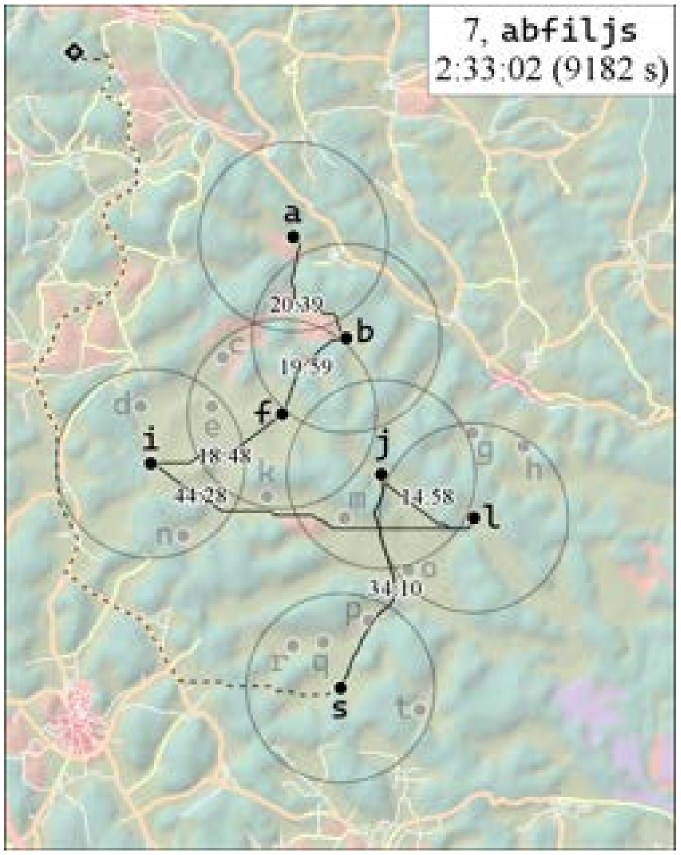

Figure 17: A valid solution of W-20 using only 7 positions.

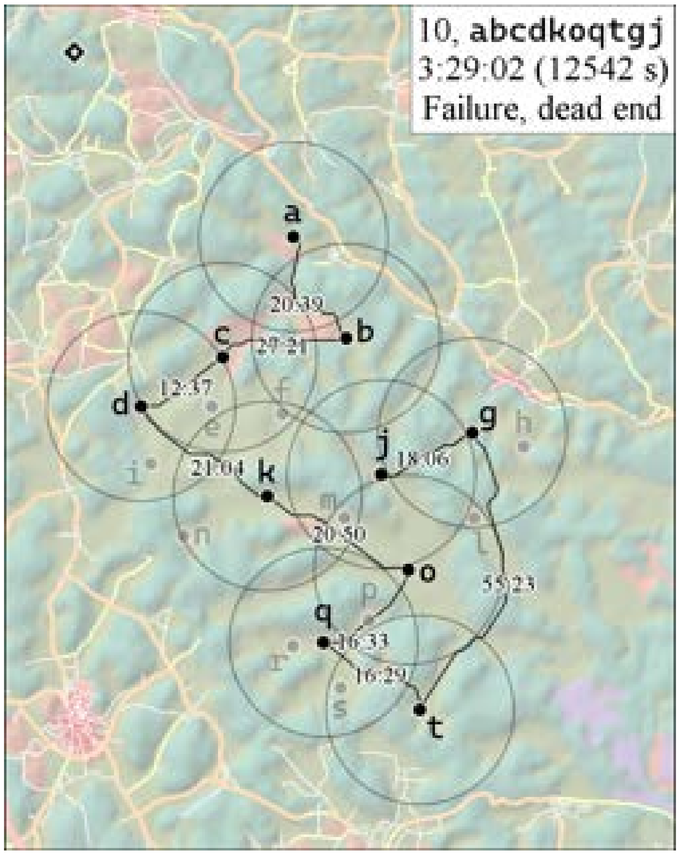

Figure 18: A failed attempt at W-20 that cannot reach safety.

Results

In Section 7 we will describe how one can sometimes reduce problems by removing positions that cannot be part of an optimal solution, but for now we shall use all positions of the TAP instances, as a baseline.

We have used ExpRankRN, PolyCostRN and GaussCostRN with various parameter settings on our three TAP instances. Each setting was tested 1000 times per TAP instance since the random choices may need many attempts to give good solutions.

To draw conclusions about which bias functions and which parameter settings are best, we tabulate three *metrics* for each batch of 1000.

The first metric is the quality of the best solution from each batch. Since solutions shall be ranked lexicographically, with high cardinality being more important than low cost (Section 3.4), the best solution is the one with least cost among those with maximal cardinality. The cost of that solution is tabulated in the rightmost columns with the caption **Winning cost**. But it may be unwise to deduce the best bias functions and parameters only from the winning cost since, by that metric, a batch of 1000 attempts could produce a single excellent solution just by a stroke of luck. So we also used two other metrics that may be more stable, statistically.

The second metric is the number of solutions with maximal cardinality in the batch, which is tabulated in the middle columns with the caption **# tens** for W-20 and the caption **# elevens** for K-20 and D-20.

Our third metric is the number of “top-tier” solutions in the batch, tabulated in columns with the caption **# top-tier**. The idea is that for operational use, it should not be essential to find the solution with the minimal possible cost, because the costs are calculated travel times that are only approximations of the true travel times. A more sensible ambition could be to just find a solution (of maximal cardinality) with a cost that is at most about 10% higher than the minimal cost, and we call that a *top-tier* solution. Instead of using 10% exactly, we used slightly different top-tier thresholds for the three problem instances:

- For W-20, a top-tier solution must have a cost less than 12000 seconds (3 h 20 min): less than 11.35% higher cost than the overall winning solution.

- For D-20, a top-tier solution must have cost less than 9600 seconds (2 h 40 min): less than 12.64% higher cost than the overall winning solution.

- For K-20 where Nearest Neighbor gave an excellent solution with a cost of 8422 seconds, a top-tier solution must have even lower cost: less than 1.79% higher cost than the overall winning solution.

Table 3: Summary of column meanings.

# top-tier	**# tens, # elevens**	**Winning cost**
Number of valid solutions with max cardinality and not much higher cost than the overall winner.	Number of valid solutions with max cardinality.	Least cost among the valid solutions of max cardinality.

Table 4: ExpRankRN used on W-20.

b	**# top-tier**	**# tens**	**Winning cost**
0.75	1	247	11538 s
0.50	11	324	**10787 s**
0.25	**34**	415	**10787 s**
0.0625	8	**757**	**10787 s**

Table 5: ExpRankRN used on K-20.

b	**# top-tier**	**# elevens**	**Winning cost**
0.75	0	406	9049 s
0.50	0	536	**8422 s**
0.25	0	859	**8422 s**
0.0625	0	**934**	**8422 s**

For W-20 and D-20, we wanted to use a round number as the threshold. For K-20, if we had defined “top-tier” in a way that included the Nearest Neighbor solution, the top-tier metric would favor bias functions with an extremely thin tail that nearly always generate the Nearest Neighbor solution – which would not be an interesting result.

In each column of the tables, the best result or results will be shown in boldface, except if all results in a column are identical.

When trying out ExpRankRN, we let the base parameter b get the values 0.75, 0.50, 0.25 and 0.0625, the same as in Figure 12. The results for W-20 and K-20 are shown in Tables 4 and 5; we have not tried ExpRankRN on D-20.

Table 6: PolyCostRN used on W-20.

d	**# top-tier**	**# tens**	**Winning cost**
0	0	204	14810 s
1	0	212	12752 s
2	1	206	11895 s
3	9	246	10932 s
4	8	292	11327 s
5	14	320	11397 s
6	27	310	10787 s
7	**35**	333	**10777 s**
8	28	**384**	**10777 s**
9	28	378	**10777 s**

Table 7: PolyCostRN used on K-20.

d	**# top-tier**	**# elevens**	**Winning cost**
0	0	379	12327 s
1	0	386	9344 s
2	0	436	9543 s
3	0	483	8582 s
4	3	581	**8274 s**
5	**4**	667	**8274 s**
6	2	724	**8274 s**
7	2	757	**8274 s**
8	4	835	**8274 s**
9	1	**864**	**8274 s**

Table 8: PolyCostRN on D-20.

d	**# top-tier**	**# elevens**	**Winning cost**
0	0	337	12086 s
1	0	383	9884 s
2	2	381	9433 s
3	15	461	8877 s
4	25	504	**8523 s**
5	50	544	8700 s
6	72	614	8945 s
7	80	596	8945 s
8	120	647	8945 s
9	**156**	**677**	8945 s

When trying out PolyCostRN, we let the degree parameter d get the values 0, 1, 2, ..., 9. The results are shown in Tables 6 – 8.

We have not tried GaussCostRN on these problem instances, but in Section 7 we will try it on a variant of D-20 in which three positions have been removed.

Takeaways

Not all differences between rows are statistically significant, and we have not found any single bias function that outperforms all others.

Based on the number of top-tier results, it seems that ExpRankRN works best when $b = 0.25$, and on this metric, it is about as good as PolyCostRN with $d = 7$.

For PolyCostRN, d should be at least 4, but there is no value that is always best in the range from 4 to 9. Overall, the bias functions with thick tails perform poorly because they choose expensive alternatives too often. Although slimming the tail tends to increase the number of top-tier results, our bias formulas will make the "neck" steeper when the tail is slimmed, and we have noticed (but not tabulated) that a side-effect of a steep neck is that one gets fewer *unique* results – which can make it harder to find an optimal or near-optimal solution.

For PolyCostRN used on D-20, the number of top-tier results increases with the degree d, more so than for W-20 and K-20. On the other hand, for D-20 the mid-range d values 4 and 5 find winning solutions with a lower cost than the values 6 to 9, which is strange since $d = 9$ gives 6.24 times as many top-tier results as $d = 4$. It is less strange if we count only unique top-tier solutions, because $d = 4$ produces 20 unique top-tier solutions while $d = 9$ produces 31 or only 1.55 times as many.

For both W-20 and K-20 we found the overall winning solution many times, which made us believe that these winners were optimal. But for D-20 we found the overall winning solution only three times, and only when using PolyCostRN with $d = 4$. Our bias functions seem to make the overall winner of D-20 very improbable, but we do not know why, and we have become less certain that the overall winners we have found are optimal.

To conclude, for TAP instance K-20 it is very hard for Random Neighbor to beat the Nearest Neighbor solution which was only 1.8% more costly than the winning solution. For the two other instances, we can reach about 3.5% and 15.6% probability of getting a top-tier result when running Random Neighbor once. In operational use, one would have to repeat Random Neighbor several times to increase the probability, but not 1000 times:

- For W-20, one can repeat Random Neighbor 30 times to get a 66% chance of getting a top-tier result.
- For D-20, one can repeat Random Neighbor 7 times to get a 69% chance of getting a top-tier result.

In the next section, we will see that one can sometimes identify positions that can never be part of an optimal solution, and by removing them before running Random Neighbor, one can improve the chances.

7 CONFLICTING POSITIONS

After running thousands of attempts with Random Neighbor on each problem instance, we had empirical evidence that highest possible cardinality of a valid solution was 10 for W-20 and 11 for K-20 and D-20. To find definitive proof that these cardinalities are optimal, we used a branch of graph theory that studies *independent sets*.

7.1 Independent Sets in Graphs

In an undirected graph, an *independent set* is defined as a set of nodes in which no two nodes are connected by an edge. Such a set can be *maximal* or *maximum*:

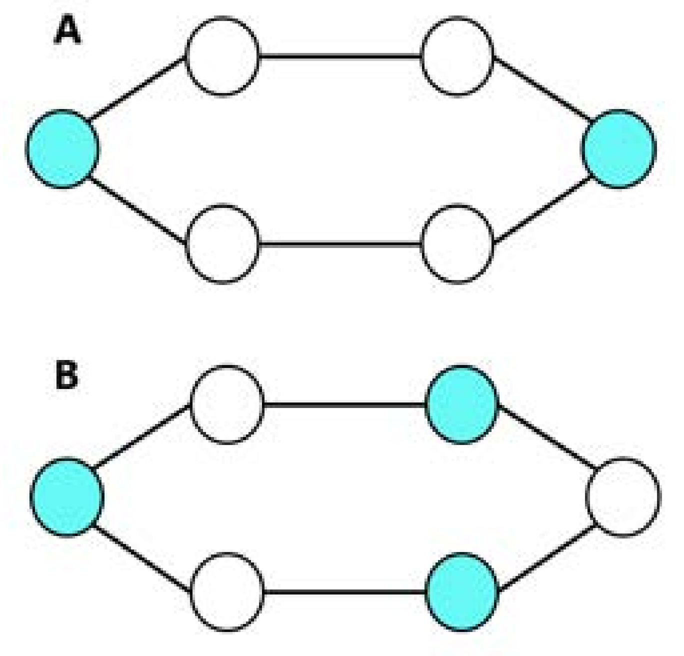

Figure 19: Examples of independent sets. A: the two blue nodes form a maximal independent set. B: the three blue nodes form a maximum independent set.

1. A *maximal* independent set is an independent set that is not a proper subset of any other independent set in the graph.
2. A *maximum* independent set is an independent set with the maximum number of nodes that an independent set can have in the graph; this is the *independence number* of the graph.

One can say that a *maximal* independent set is a *local* maximum in a search for a large independent set, while a *maximum* independent set is a *global* maximum (which is not necessarily unique). A maximum independent set must also be maximal, but the converse is false. For examples, see Figure 19.

7.2 The Conflict Graph of TAP

In the traveling artilleryman problem, let us say that position p *excludes* position q if $q \in R(p)$. Using just the formal problem statement, exclusion is not necessarily a symmetric relation, but it is symmetric in our practical instances of the problem where positions are embedded in a 2D surface and the risk-circle sets are defined via geometrical circles with the same radius. That is, in the practical instances, p excludes q if and only if q excludes p, and then we say that p and q are *in conflict*, otherwise they are *independent*. In a valid solution any pair of used positions must be independent, regardless of in which order they were used.

Let us define an undirected *conflict graph* that is much smaller than the full terrain graph. The nodes of the conflict graph are the positions of the TAP problem instance, and two positions are connected by an edge if they are in conflict. The independence number of the conflict graph is an upper bound of the cardinality of a valid solution of the TAP instance.

7.3 Finding the Independence Number

For our TAP instance W-20, the conflict graph is shown in Figure 20, and we hope to prove that its independence number is 10. We already know that it must be at least 10, since we have found solutions with cardinality 10.

Calculating the independence number of a graph is an NP-hard problem, but there are only 20 positions in W-20 so the conflict graph is small. We decided to implement an exhaustive search algorithm that loops through the unsigned 32-bit integers from 0 to $2^{20} - 1$, treating the bit-pattern of each integer as a coded subset of the positions. For each integer interpreted as a subset, the exhaustive search algorithm checks whether the subset is independent, in which case its number of elements is noted. Most of the code was straightforward, but for the task of counting the 1-bits of an integer we used a non-trivial method [20]; for alternatives, see [21]. Despite the naivety of the exhaustive search algorithm, it took only about 5 milliseconds of CPU time to find that the independence number for the conflict graph was indeed 10. The result confirms that no valid solution of W-20 can have more than 10 positions,

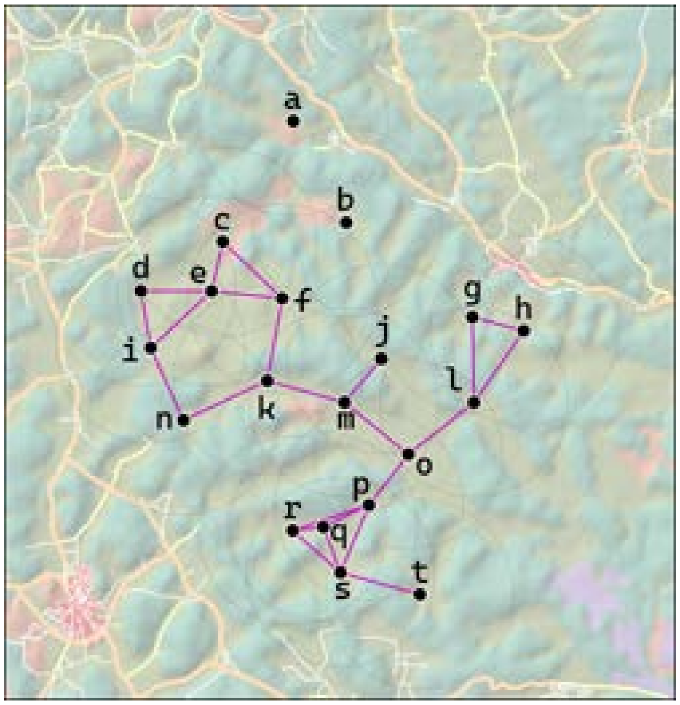

Figure 20: The conflict graph of W-20.

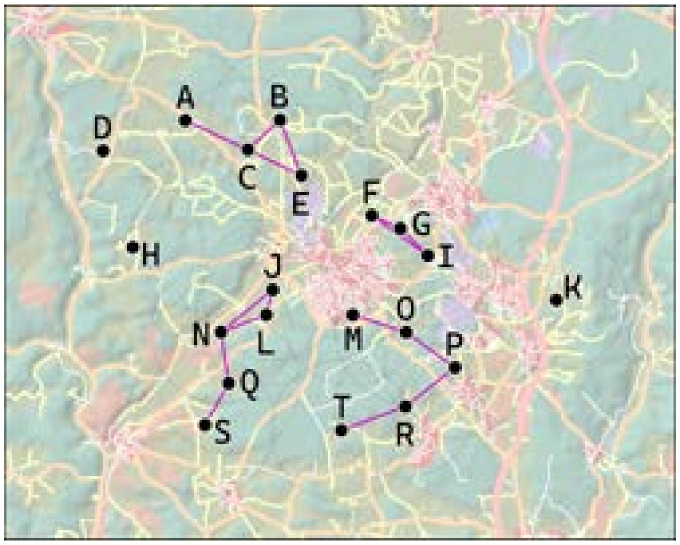

Figure 21: The conflict graph of K-20 and D-20.

although it does not give a short human-readable proof.

For our other TAP instance, K-20, the conflict graph (Figure 21) consists of the three isolated nodes **`D`**, **`H`** and **`K`** and four disconnected sub-graphs. This time we do not need any computer assistance to find the independence number, because every isolated node must be part of a maximum independent set, and the four sub-graphs are small and can be studied separately. From the top left sub-graph (with the positions **`A`**, **`B`**, **`C`**, **`E`**) we can choose at most two independent nodes, from the top right sub-graph (**`F`**, **`G`**, **`I`**) only one, from the bottom left sub-graph (**`J`**, **`L`**, **`N`**, **`Q`**, **`S`**) at most two, and from the bottom right sub-graph (**`M`**, **`O`**, **`P`**, **`R`**, **`T`**) at most three. Thus, the independence number is 3 + 2 + 1 + 2 + 3 = 11.

The time complexity of the exhaustive search algorithm is $\mathcal{O}(n^2 2^n)$ for graphs in general, where n is the number of nodes. But we think that conflict graphs for TAP tend to be *sparse* in the sense that the number of edges is proportional to n rather than to n^2, and then the complexity should be $\mathcal{O}(n 2^n)$. Under this assumption, our exhaustive search implementation should require about 8 seconds for $n = 30$, about 3 hours for $n = 40$, and about 155 days for $n = 50$.

There are better algorithms. The algorithm by Xiao and Nagamochi [22] can calculate the independence number using polynomial space and a time complexity of $\mathcal{O}(1.1996^n)$ which is still exponential but much better, but they do not give examples of computation times on benchmark graphs, and algorithms with the best time complexity are not always fastest in practice. However, many such examples are given by Akiba and Iwata [23] who have implemented and tested three algorithms on many benchmark graphs. Their results are hard to summarize since the difficulty of a graph cannot be predicted just from its number of nodes and edges, but for the fifteen benchmark graphs with fewer than 100 nodes (the largest of which had 88 nodes), even their slowest algorithm required less than 1 second of computation time. (The algorithms of Akiba and Iwata find a *minimum vertex cover* instead of a maximum independent set, but the two problems are equivalent [24].) Also, in many applications it is enough to find independent sets that are large but not necessarily of maximum size: see the KaMIS project of Dahlum et al. [25].

7.4 Remove Blocking Positions

Let us investigate whether the best solutions of W-20 favor some positions over others. When we run the Random Neighbor algorithm 1000 times with the polynomial bias function of degree 7, we found 333 valid solutions of cardinality 10, and Table 9 tells how often each position was used.

We can see a stark pattern: the five positions with blue names in the table, **a**, **b**, **j**, **o** and **t**, are used by all 10-cardinality solutions, while the five positions with red names in the table, **e**, **l**, **m**, **p** and **s**, are not used at all. (Admittedly, it is not surprising that the prescribed start position **a** is used by all solutions.)

The pattern can be explained by further analysis of the maximum independent sets. Let us define two new concepts:

- A node in an undirected graph is a *maximum-independent-set blocker* or *MIS-blocker* if it is not an element of any maximum independent set of the graph.
- A node in an undirected graph is a *maximum-independent-set requisite* or *MIS-requisite* if it is an element of every maximum independent set of the graph.

We made a minor modification of our exhaustive search algorithm, which confirmed that the five unused positions **e**, **l**, **m**, **p** and **s** are indeed MIS-blockers for the conflict graph for W-20 while the five favored positions **a**, **b**, **j**, **o** and **t** are indeed MIS-requisites, as illustrated in Figure 22. We do not know if the more sophisticated algorithms mentioned in Section 7.3 would be easy to adapt to find MIS-blockers. When we discovered that MIS-blockers and MIS-requisites were useful concepts in our setting, we contacted some experts – see Acknowledgments, Section 14 – to ask whether these concepts had established names and if there were algorithms to find such nodes, but the answer was negative (although they said that the concepts may have been used in the innards of some algorithm for finding the dependency number).

Table 9: Some positions are favored, others avoided.

position	times used
a	333
b	333
c	153
d	274
e	0
f	180
g	207
h	126
i	59
j	333
k	98
l	0
m	0
n	235
o	333
p	0
q	233
r	100
s	0
t	333

Knowing which nodes are MIS-blockers, we can remove them from W-20 and solve the reduced problem with 15 positions, call it W-15, since it must have the same optimal solution. We repeated our RN experiment on W-15 and got the results in Tables 10 and 11.

So, if we run ExpRankRN on W-15 just once with $b = 0.25$, we get a probability of about 86% of getting a solution of cardinality 10 and a probability of about 10.6% of getting a top-tier result. This is a significant improvement over ExpRankRN on W-20 where the corresponding probabilities were about 42% and 3.4%. Assuming that the probability of a top-tier result is exactly 10.6%, we will get a 66% chance of a top-tier result for W-15 by running Random Neighbor just 10 times, instead of the 30 times needed for W-20.

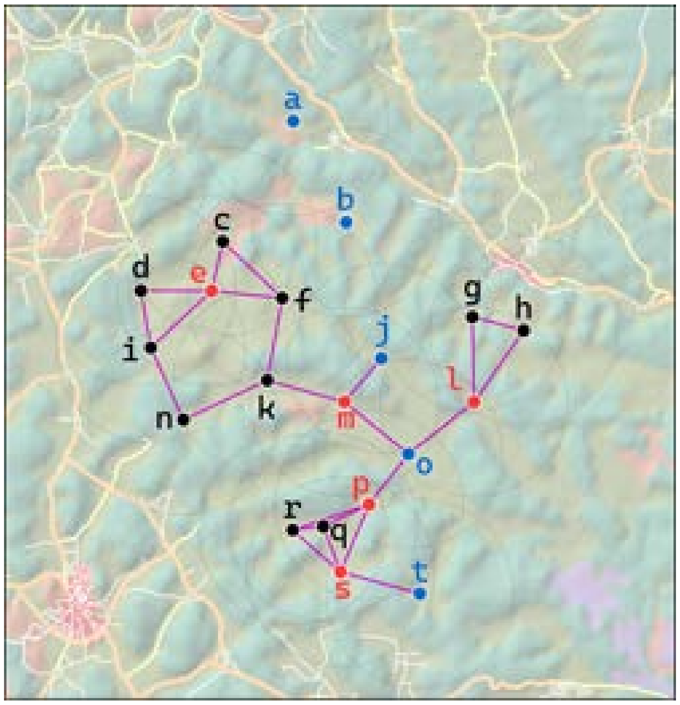

Figure 22: Analyzed conflicts of W-20.
Red: MIS-blockers; blue: MIS-requisites.

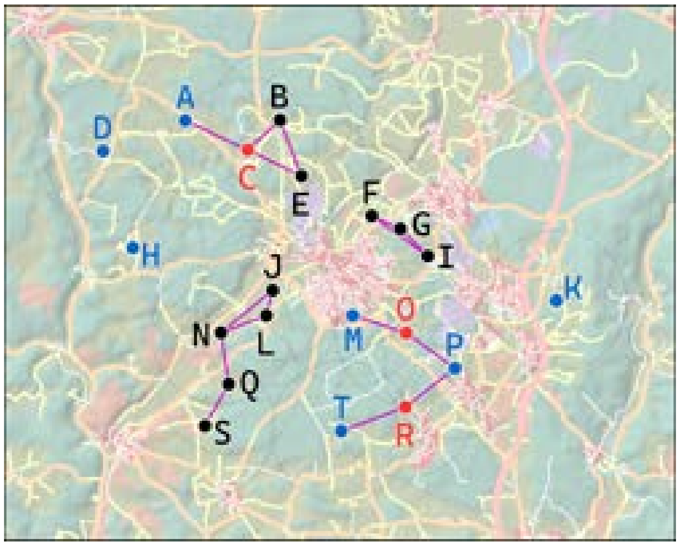

Figure 23: Analyzed conflicts of K-20 and D-20.
Red: MIS-blockers; blue: MIS-requisites.

The five MIS-blockers can also be removed when running the overgreedy algorithm to find a lower bound of an optimal solution. Three legs in Figure 5 are connected to the MIS-blockers `e`, `l` and `t`, so they are not relevant. Without the five MIS-blockers,

Table 10: PolyCostRN used on W-15.

d	# top-tier	# tens	Winning cost
0	0	763	14237 s
1	0	782	12101 s
2	6	804	11213 s
3	25	**819**	**10777 s**
4	46	805	**10777 s**
5	69	790	**10777 s**
6	71	794	**10777 s**
7	74	742	**10777 s**
8	**80**	778	**10777 s**
9	79	769	**10777 s**

Table 11: ExpRankRN used on W-15.

b	# top-tier	# tens	Winning cost
0.75	2	805	11310 s
0.50	34	821	10787 s
0.25	**106**	859	**10777 s**
0.0625	95	**943**	10787 s

the overgreedy algorithm would find three other legs: one from `q` to `o` with a cost of 16 min 26 s, one from `t` to `q` with a cost of 16 min 27 s, and one from `g` to `j` with a cost of 18 min 06 s. These changes raise the lower bound by 5 min 01 s or 3.6%.

For our second problem instance, K-20, the MIS-blockers and MIS-requisites can be found manually by studying each sub-graph separately in Figure 21. As shown in Figure 23 there are three MIS-blockers, `C`, `O` and `R`, so by removing them we get a reduced problem instance K-17 that has the same optimal solution as K-20.

We have repeated the Random Neighbor experiment also for K-17; see Table 12. This time every produced solution had cardinality 11, which is a significant improvement of the results for K-20 (Table 7). But the number of top-tier solutions of K-17 is not significantly greater than for K-20, nor is the best result consistently better.

Since `D` is among the 17 remaining positions, we can use them to define a reduced version D-17 also

Table 12: PolyCostRN used on K-17.

d	**# top-tier**	**# elevens**	**Winning cost**
0	0	1000	10212 s
1	0	1000	9999 s
2	0	1000	8750 s
3	0	1000	8476 s
4	0	1000	8422 s
5	**7**	1000	**8274 s**
6	3	1000	**8274 s**
7	4	1000	**8274 s**
8	2	1000	**8274 s**
9	1	1000	**8274 s**

Table 13: PolyCostRN used on D-17.

d	**# top-tier**	**# elevens**	**Winning cost**
0	0	1000	11036 s
1	0	1000	10440 s
2	3	1000	9008 s
3	11	1000	9056 s
4	30	1000	**8819 s**
5	65	1000	8945 s
6	94	1000	8945 s
7	115	1000	8945 s
8	131	1000	8945 s
9	**170**	1000	8945 s

Table 14: GaussCostRN used on D-17.

μ	σ	**# top-tier**	**# elevens**	**Winning cost**
1	0.10	202	1000	8945 s
1	0.07	300	1000	8945 s
1	0.05	**316**	1000	8945 s
1	0.03	197	1000	8945 s

of our third problem instance D-20 in which `D` is the prescribed start node. For this problem instance it makes sense to repeat both Nearest Neighbor and Random Neighbor with the 17 positions, since Nearest Neighbor with all 20 positions managed to use only 10 positions instead of 11 (Figure 11).

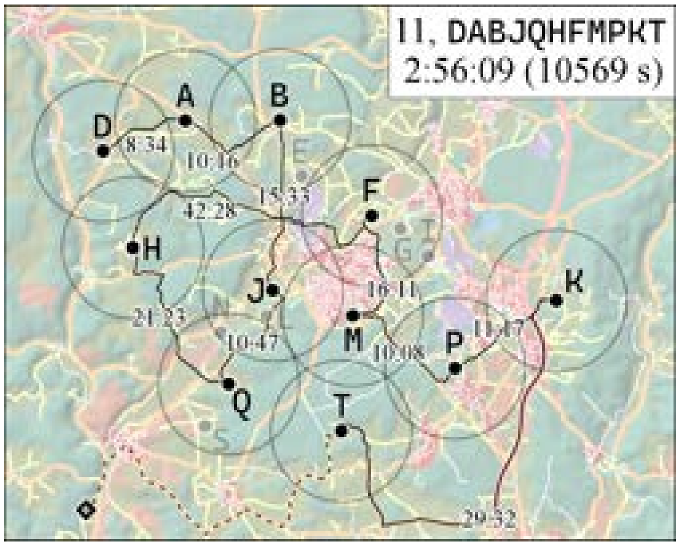

Figure 24: Result from Nearest Neighbor, D-17.

The Nearest Neighbor result for D-17 does reach 11 positions; see Figure 24 above.

The PolyCostRN results for D-17 are shown in Table 13. Remember that the definition of “top-tier” differs between D-17 and K-17 (Section 6.6), so one should not compare the top-tier columns for Tables 12 and 13 with each other. In Table 13, one can see that the number of top-tier results increases with degree and is highest for d = 9. But a peculiar detail is that the winning solution with cost 8819 s was found only with d = 4 and then only once, while the runner-up with cost 8945 s was found once with d = 4 and 45 times with $d > 4$. Despite the slightly better chances of finding top-tier solutions for D-17 than for D-20, the overall winner from D-20 (8523 seconds, Figure 16) was better than the overall winner from D-17 (8819 seconds).

We have also tried GaussCostRN on D-17. Some preliminary tests suggested that the proportion of top-tier results would peak with σ near 0.05, so we chose values for σ around 0.05 in our main tests, as shown in Table 14. The number of top-tier results for σ = 0.05 was 316, much higher than the best number for PolyCostRN, but GaussCostRN did not find any winning cost lower than 8945 seconds.

We have also revisited the lower bound for K-20 and D-20 that was computed by the overgreedy algorithm in Section 6.4. Of the ten legs found by the overgreedy algorithm (Figure 6), only the leg from `O` to `I` involves a MIS-blocker. With the three MIS-blockers removed, the overgreedy algorithm working on K-17 or D-17 would instead choose the leg from `M` to `J` with a cost of 11 min 18 s, which would raise the lower bound by 3 min 30 s or 3.7%.

7.5 Finding all Maximum Independent Sets

One may wonder how many different maximum independent sets there are in the conflict graph for W-20. When we adapted our exhaustive search algorithm for this question, we found that there were only 16 such sets. (We do not know how good the sophisticated algorithms mentioned in Section 7.3 would be for this purpose, since they are designed to find the independence number and *one* example of a maximum independent set). Again, the algorithm does not provide a short human-readable proof of this, but if you trust Figure 22, it can be used to generate the 16 sets. To begin with, each maximum independent set must contain ten nodes, and since it must contain all five of the blue nodes, it must also contain five of the ten black nodes. Of the black nodes, `g` and `h` are in conflict with each other but not with any other black node, so we must choose exactly one of them. The same is true for `r` and `q`, giving us another choice with two options, and the two choices are unrelated. Finally, we must choose three independent nodes of the six remaining black ones, the set { `d`, `i`, `n`, `k`, `f`, `c`}, which can be done in four ways:

1. { `d`, `n`, `f`}
2. { `d`, `n`, `c`}
3. { `d`, `k`, `c`}
4. { `i`, `k`, `c`}

So we have three unrelated choices with 2, 2 and 4 options each, giving $2 \times 2 \times 4 = 16$ ways to construct a maximum independent set. The number 16 is smaller than we had expected, since it is possible to design a graph with 20 nodes that has 1024 different maximum independent sets: just ensure that the nodes are grouped into 10 pairs with an edge for each pair but no other edges.

Can we exploit the low number of maximum independent sets to improve the search for solutions of W-20? One idea is to reduce W-20 to sixteen subproblems of 10 nodes each and run Random Neighbor on each of them. One such subproblem should be easier to solve than the 15-node problem W-15 of the previous section, but since there are sixteen of them, it is not obvious that this approach would improve the search. We decided to try it by running PolyCostRN 1000 times evenly distributed over the sixteen subproblems, but 1000/16 = 62.5 so we run it 62 times on each of eight subproblems and 63 times on each of the other eight. We did this using only degree 7 for the polynomial bias function. Table 15 shows the results, which were unexpected. We got 999 valid solutions of cardinality 10 which is excellent, with the only exception being an attempt that got stuck in a dead end as in Figure 18. On the other hand, we got only 43 top-tier solutions, which is significantly fewer than in the corresponding experiments with W-15 (Table 10). Since the number of top-tier solutions may be the more important metric, we conclude that the division into sixteen subproblems did not help.

Table 15: PolyCostRN used 1000 times in total on the 16 subproblems of W-20.

d	**# top-tier**	**# tens**	**Best result**
7	43	999	10787 s

In hindsight, we can say that the division into sixteen subproblems should not help in the choice between positions `g` and `h` or the choice between `r` and `q` (see Figure 22), because these choices cause no problems when running Random Neighbor on W-15: when the algorithm has chosen to use one of `g` and `h` or one of `r` and `q`, the other option will become invalid and no opportunities have been lost. It is the choice of three of the other six black positions **`d`**, **`i`**, **`n`**, **`k`**, **`f`**, **`c`** that can cause problems for W-15, because if Random Neighbor happens to choose **`i`** and **`f`** first, it cannot choose a third of the six, and the solution will get a cardinality lower than 10. But we do not know why the larger number of solutions with cardinality 10 contained fewer top-tier solutions.

Another way to exploit the division into sixteen subproblems is to further raise the lower bound found by the overgreedy algorithm. In Section 7.4 we described how the lower bound could be raised by first removing the five MIS-blockers. But when the overgreedy algorithm was run on W-15, it generated nine least-cost legs between 12 positions that cannot occur in the same solution, since a valid solution cannot use more than 10. Indeed, among the 12 are both **`c`** and **`f`** which are in conflict and both **`d`** and **`i`** which are also in conflict (even though the overgreedy algorithm is smart enough not to generate any leg between **`c`** and **`f`** or between **`d`** and **`i`**). So, one can improve the lower bound further by running the overgreedy algorithm on each of the 16 subproblems and use the minimum of the results.

For K-20 and D-20, a similar study of Figure 23 reveals that three unrelated choices must be made when constructing a maximum independent set: we have 2 options in the top left sub-graph where we must choose **`A`** but can choose either **`B`** or **`E`**, 3 options in the top right sub-graph where we can choose any of the three positions, and 5 options to choose two independent positions of the bottom left sub-graph. Thus, there are $2 \times 3 \times 5 = 30$ maximum independent sets for D-20 and for K-20. We have not tested Random Neighbor on the corresponding 30 subproblems of D-20 or K-20, because we see no reason to believe that it would be better than running Random Neighbor on D-17 or K-17. Unlike the situation for W-15 where it would be a mistake to choose **`i`** and **`f`** early, Random Neighbor on D-17 or K-17 can choose black positions in any order without losing the opportunity to eventually get the optimal number, 4 black positions, as one can see by inspecting each sub-graph in Figure 23 separately. This structural difference between the conflict graphs for W-15 and D-17/K-17 explains why all generated solutions for D-17 and K-17 achieved the maximal cardinality (Tables 12 – 14), while at most 94% did so for W-15 (Tables 10 and 11).

Finally, one could improve the lower bound for K-17 and D-17 even more by exploiting the division into 30 subproblems. We saw in Section 7.4 that the lower bound calculated for K-17 and D-17 improved the one for K-20 and D-20 by 3.7%. But the improved lower bound is based on edges between a set of eleven positions that do not form a maximum independent set, since it lacks the MIS-requisites **`H`** and **`T`** and contains some positions that are in conflict. By instead using the overgreedy algorithm on each of the 30 subproblems and taking the minimum result, one could improve the lower bound further.

8 SOLVING A TRAVELING ARTILLERYMAN PROBLEM

Altogether, our results suggest the following way to find a good solution to a TAP instance:

1. Construct the conflict graph, analyze it to find the positions that are MIS-blockers, and remove them.
2. Run the Nearest Neighbor algorithm once.
3. Also run the Random Neighbor algorithm thirty times (or more if time allows) using a bias function with a slim tail, for example ExpRankRN with $b = 0.25$, GaussCostRN with $\mu = 1$ and $\sigma = 0.05$, or PolyCostRN with d in the range from 4 to 9.
4. Pick the best of the solutions found.

This approach should give a good chance of finding a top-tier solution; that is, a good chance of finding a solution with the highest possible cardinality and with a cost at most 10% higher than the cost of the winning solution from thousands of attempts by the Random Neighbor algorithm.

9 RELATED WORK

Our paper complements previous work on shoot-and-scoot tactics. In Section 1.2 we have already described how we were inspired by the work of Temiz [3]. In this chapter we present two other related papers: Quinn and Kunkleman [16] on finding good position areas, and Shim and Atkinson [26] on using Markov chain optimization to determine when to shoot and when to scoot.

9.1 Finding Good Position Areas

Quinn and Kunkleman [16] study how terrain affects artillery systems, specifically the terrain in the Hohenfels Training Area and the surrounding Maneuver Rights Area in Germany. They suggest 40 case points, each one representing the center of a position area that should be large enough to contain many alternative firing positions. (We have used their case points as TAP positions, Section 6.1, but without considering their varying quality or sizes.)

Quinn and Kunkleman list five criteria on position areas and show how one can analyze digital geodata to find and evaluate locations, although they recommend true site reconnaissance. The five criteria differ somewhat between the friendly forces in the rural Hohenfels Training Area and the enemy forces in the more populated Maneuver Rights Area. For the friendly forces, the authors list the following criteria:

1. Level ground: no more than 5 degrees of slope.
2. Communications: some radio systems need a free line of sight (say about 5 km).
3. Size: a battery (six howitzers) needs 3×3 km; a platoon (three howitzers) needs 1.5×3 km.
4. Tree lines that provide cover and concealment.
5. No visibility from nearby villages or major roads.

The criteria No. 1, 2 and 4 are also used for the enemy's position areas. But the size required by the enemy is 3×3 km for a battery and only 1×1 km for a platoon. And criterion No. 5 is inverted: instead of avoiding villages, it is assumed that the enemy will exploit the many settlements in the Maneuver Rights Area for cover and concealment. This difference in tactics is visible in our Figure 2, where one can see the contrast between Quinn and Kunkleman's friendly case points in a rural area and their enemy case points around the town of Burglengenfeld.

Finally, they evaluate each position area as satisfying each criterion to a low, moderate or high degree – except for the size criterion, which is rated by a number: the occupiable percentage of a large enough area around the case point.

9.2 Mission Support using a Markov Chain

How does one know when to shoot and when to scoot? That is, for how long should we keep firing from our current position until it is too exposed and we need to scoot away? To answer this question, Shim and Atkinson [26] have developed a statistical model in which time is discretized into ticks or *time windows*.

A time window is considered as an action, where the unit will either fire at its target with an assumed hit probability or flee from its current position with an assumed success probability (enemy forces miss).

Shim and Atkinson assume that, every time the unit has decided to scoot, the first shot fired from the next firing position is free from risk. And there is a feedback loop when continuing to fire from the same position: the next fired round will have better chances of hitting the target. But staying in the same position increases the risk of being hit by opposing forces. The model also includes a deadline: the mission is successful only if the opposing target has been eliminated before the deadline.

The problem is simplified by assuming that the unit will either stay in a firing position and shoot until the next time window occurs or scoot immediately after its first risk-free shoot.

When firing, the probability of success is calculated as: *probability of hitting target in current time window + probability of hitting target when entering next time window*. When scooting, the success is calculated as: *Probability of hitting risk-free shoot + probability of making a successful relocation + probability of hitting target before time window runs out*. These calculations are done in reverse, where you start at the last time window and traverse backwards to estimate total success. Since each variable is dependent on what happened in previous time window, Shim and Atkinson apply a Markov chain to find a solution.

10 FUTURE WORK

In future work, one could either keep the problem statement but try to handle it better, or try to improve the problem statement.

10.1 Better Algorithms

We think better algorithms should be possible. For one thing, in our attempts to balance greed and randomness we have tried only a few types of bias functions, and apart from bias functions there are several other ways to achieve balance [14]. But it may be more important to improve the randomized greedy solutions by local search procedures, which have been very successful for the traveling salesman problem [9, 10, 11], although they may be difficult to apply to TAP. Seemingly local changes to a TAP solution can affect legs farther away, so the legs and the risk circles will need a spatial index that allows efficient detection of all legs that must be recalculated after a local change. There are many ideas in the TSP literature that one could try, but note that some of them work only for undirected graphs. Even if we wanted to study TAP for undirected terrain graphs, the risk-circle constraint means that if the sequence of used positions or a part of it is reversed, further modifications may be necessary.

10.2 Using Brute-force Search to Find an Optimal Solution

In Section 7.5 we showed how the problem instance W-20 with 20 positions could be reduced either to the equivalent problem W-15 with 15 positions or else to 16 subproblems with 10 positions each. Using W-15 instead of W-20 improved the Random Neighbor algorithm but using the 16 subproblems did not.

However, the reduction to 16 subproblems may be useful in an attempt to use brute-force search where problem size is crucial. Since we already know a good solution, we could use its cost as a threshold for the refined version of brute-force search; see Section 4.1. And since each subproblem has only 10 positions, leg costs could be precomputed reasonably fast; see Section 6.3.

The benefit of using the threshold-based refinement is hard to predict, but we think the total computation time would be a matter of days rather than weeks. The result of a successful attempt would be either certainty that the winning solution (Figure 15) is optimal or a better solution.

For K-20 and D-20, there are 30 subproblems instead of 16 and each has 11 positions instead of 10. We think that these differences would make brute-force search roughly 20 times slower.

10.3 Soft Risk Circles

The risk-circle constraint is rigid in a way that can be impractical. If two risk circles overlap slightly, the artillery unit must not pass through the overlap even if it could do so quickly. So, it is possible that our algorithms find an optimal solution for one value of the risk radius, failing to see that a much better solution would be possible with a slightly shorter radius that would separate the two circles. To avoid missing such opportunities, a user could try shorter and shorter values of the risk radius to get many different solutions, and then use her own judgment to choose one of them.

It may be better to use two radii instead. An inner radius would define a strict risk circle as before, while an outer radius would define an annulus area between the inner and the outer circle. One could allow the pathfinder to use the annulus but only reluctantly, with a degree of reluctance that can be controlled.

In an earlier paper [27], we controlled the reluctance for a softly restricted area by using a *safety factor* between 0 and 1. If the factor is 0, the area is forbidden, and if it is 1, the area is perfectly safe. In general, the true speed assumed for a vehicle shall be multiplied with the safety factor to get a *penalized* speed, which is then used to calculate the penalized travel times that the pathfinder tries to minimize. For example, if the safety factor of an area is 0.25, the pathfinder will pretend that the vehicle speed is only 1/4 of its true value there, so each minute spent there will count as 4, or one can say that each minute spent there adds 3 penalty minutes.

However, it is not clear if modifying TAP like this would be beneficial. In the original TAP there is only one problem parameter to be decided by a human expert: the risk radius. In the modified version, there would be three: an inner and an outer radius and a safety factor for the annulus area, so the space of parameter settings will be larger and harder to explore for a user.

10.4 Mixed Strategy

When we defined the traveling artilleryman problem, we made the implicit assumption that the enemy does not know where our firing positions are, nor our strategy of using them. The reviewers for GVSETS asked, what if that is false? That is, let us say that our algorithm always finds the optimal solution to a TAP instance. If the enemy knows our firing positions and can use the same algorithm, then they can predict the movements of our artillery unit.

A basic approach for this kind of situation is well-known, the Game Theory concept of a *mixed strategy* [28, chapter 6]: instead of using just the optimal solution, we should be unpredictable by first generating several good alternative solutions and then choose one randomly. (Randomized decisions were recommended also by Temiz [3, page 20].) In this section we shall not discuss how to assign probabilities to the alternative solutions since that would require a well-defined way to measure operational risks and opportunities, but we can discuss how to generate good alternative solutions.

Of course, our Random Neighbor algorithm already uses random choices, so the solution from a single attempt is unpredictable, but it may be a poor or invalid solution. And the more we repeat Random Neighbor to find a top-tier solution, the more predictable the result may be.

Previously, we have suggested running Random Neighbor at least 30 times to get a good chance of finding one top-tier solution. To find more than one, we do not know any better way than the obvious idea of running Random Neighbor many more times: running it 300 times should give a good chance of finding ten top-tier solutions.

We should also ensure that the top-tier solutions in our shortlist are not too similar, since a random choice between nearly identical alternatives is too predictable. When positions are named by single letters as we have done, a solution can be described by a string (the names of the used positions in order), so the similarity between two solutions could perhaps be defined by some variant of *edit distance* [29].

To summarize: after many attempts with Random Neighbor, one could order all solutions by quality (cardinality and cost) and discard solutions with a short edit distance to a better one; the beginning of the remaining list should then be a useful shortlist. Unfortunately, the top-tier metric of our experimental results, as tabulated in Sections 6 and 7, does not tell us how many times Random Neighbor would need to be run in this setting. To learn that, one would need to first define an edit distance threshold, and then for each batch of 1000 Random Neighbor attempts discard the top-tier solutions that are too similar to better ones, and count only the rest.

Even if we can find a mixed strategy that prevents the enemy from deducing our plans in advance, the risks will increase toward the end of the operation if the enemy knows where our firing positions are, at least if we need to use most of them. Because the enemy will know which positions we have used and can more easily predict the next one toward the end when only one or a few unused positions remain possible. In that situation, it may be better to sometimes flee to safety before using all positions allowed by the TAP rules, or to sometimes reuse an old firing position, just to be more unpredictable. The lexicographic order of solutions (Section 3.4) may then be less useful, since it always favors solutions that use as many positions as possible. To allow early escape, it may be better to instead rank solutions by *cardinality divided by cost*, representing the number of used positions per time unit.

11 CONCLUSIONS

The traveling artilleryman problem is NP-hard, but we have developed two heuristic algorithms, Nearest Neighbor and Random Neighbor (Section 5), that can produce good solutions reasonably fast for problems of modest size. We have also developed a simple "overgreedy" algorithm (Sections 4.3 and 6.4) that calculates a lower bound of the solution cost for a given cardinality (number of positions used).

We have also discovered a way to analyze conflicts between positions in a TAP instance via maximum independent sets. In this way, one can find positions that cannot possibly be used by an optimal solution, and by removing these, one can improve the performance of the heuristic algorithms (Section 7).

We have tested our algorithms and our conflict analysis on three TAP instances with 20 positions each, and measured running time and solution quality (Sections 6.5, 6.6, 7.4 and 7.5).

Running our Nearest Neighbor algorithm or Random Neighbor algorithm once on one of our TAP instances takes about 3 seconds in our implementation, but in general the time will depend on many factors like the number of positions, the size of the search area, and the terrain resolution (Section 6.2).

Measuring solution quality was problematic. For a solution of optimal cardinality (the number of positions used), we would have preferred to define its quality as the percentage by which its cost exceeds the cost of an optimal solution. But we could not measure that percentage, since we do not know the optimal solutions. We do have the simple algorithm that can calculate a lower bound for the cost of an optimal solution, so we can measure the percentage by which the cost of a solution exceeds the lower bound, but we believe that our lower bounds are not very tight. So, instead of comparing with the optimal solution or a lower bound, we compare with a "winning" solution: the best solution we have found by running the Random Neighbor algorithm many thousands of times on the problem instance.

By this metric, the Nearest Neighbor algorithm produced an excellent result for one of our TAP instances, but produced mediocre and poor results for the two others (Sections 6.5 and 7.4).

The average quality of solutions from the Random Neighbor algorithm depends on how choices are randomized, which can be done in many ways. We have explored one way that uses a bias function that assigns probability weights to alternative choices depending on their cost. We have not found a single bias function that outperforms all others on our TAP instances, but we found several that give a decent probability of finding a "top-tier" solution: that is, a solution of the highest cardinality with a cost not more than 10% higher than the winning solution from thousands of tests (Sections 6.6 and 7.4). Although the top-tier probability can be low for a single use of Random Neighbor, one should get quite a good chance by repeating Random Neighbor thirty times (Section 8).

11.1 Lessons Learned

We learned that bias functions work, but it is difficult to get their tail slim enough without also making their "neck" too steep (the beginning of the function curve). That is, if the tail is too thick, the Random Neighbor algorithm will choose costly alternatives too often, which decreases the probability of getting top-tier solutions. But if the neck becomes too steep as a side effect of slimming the tail, then the algorithm becomes too greedy when choosing among the best alternatives and produces fewer *unique* solutions, and a sizable number of the solutions will just be the same as the one found by Nearest Neighbor. So, it may be better to just cut off the tail at some threshold, for example by ignoring the worst half of the alternatives; then one would be freer to design the bias function for the remaining ones. Apart from bias functions, Resende and Silva [14] list seven other general methods to balance greed with randomness, so there is a large design space to explore.

11.2 Local Search

Local search procedures that improve tentative solutions have been very successful for the traveling salesman problem, but they seem complicated to apply to our problem; see Section 10.1.

12 REFERENCES

[1] H. Lye, "Why modern militaries still need artillery," *Global Defence Technology*, January 14, 2021. [Online]. Available: https://defence.nridigital.com/global_defence_technology_jan21/why_modern_militaries_still_need_artillery

[2] B. B. Knutson Jr, *Suppression of Enemy Air Defenses (SEAD)*. U.S. Marine Corps, MCWP 3-22.2, 2001. [Online]. Available: https://www.marines.mil/Portals/1/Publications/MCWP%203-22.2%20Suppression%20of%20Enemy%20Air%20Defenses.pdf

[3] Y. Z. Temiz, "Artillery survivability model," Master's thesis, Naval Postgraduate School, Monterey, California, June 2016. [Online]. Available: https://apps.dtic.mil/sti/tr/pdf/AD1026840.pdf

[4] R. Ruitenberg, "In Ukraine, 'shoot-and-scoot' tactics helping Caesars survive," *Defense News*, April 2, 2024. [Online]. Available: https://www.defensenews.com/global/europe/2024/04/02/in-ukraine-shoot-and-scoot-tactics-helping-caesars-survive

[5] H. Graff-Hedberg, "Archer mobile howitzer," *BAE Systems*, https://www.baesystems.com/en/product/archer.

[6] W. T. Tutte, *Graph Theory*, ser. Encyclopedia of Mathematics and its Applications. Addison-Wesley, 1984, vol. 21.

[7] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100—-107, 1968. [Online]. Available: https://sci-hub.se/10.1109/TSSC.1968.300136

[8] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. MIT Press, 2009.

[9] G. Laporte, "The traveling salesman problem: An overview of exact and approximate algorithms," *European Journal of Operational Research*, vol. 59, pp. 231–247, 1992. [Online]. Available: https://web.ist.utl.pt/~ist11038/CD_Casquilho/TSP1992EJOR_Laporte.pdf

[10] D. S. Johnson and L. A. McGeoch, "The traveling salesman problem: A case study in local optimization," in *Local Search in Combinatorial Optimization*, E. H. L. Aarts and J. K. Lenstra, Eds. London: John Wiley & Sons, 1997, pp. 215–310. [Online]. Available:

https://www.cs.ubc.ca/~hutter/previous-earg/EmpAlgReadingGroup/TSP-JohMcg97.pdf

[11] C. Rego, D. Gamboa, F. Glover, and C. Osterman, “Traveling salesman problem heuristics: Leading methods, implementations and latest advances,” *European Journal of Operational Research*, vol. 211, no. 3, pp. 427–441, 2011. [Online]. Available: https://leeds-faculty.colorado.edu/glover/fred%20pubs/429%20-%20TSP%20-%20problem%20heuristics%20-%20leading%20methods,%20implementations,%20latest%20advances.pdf

[12] D. Applegate, R. Bixby, V. Chvátal, W. Cook, and K. Helsgaun, “Optimal tour of Sweden,” https://www.math.uwaterloo.ca/tsp/sweden, 2004, accessed: 2025-01-03.

[13] G. Righini, “Efficient optimization of the Held–Karp lower bound,” *Open Journal of Mathematical Optimization*, vol. 2, article no. 9, 2021. [Online]. Available: https://ojmo.centre-mersenne.org/articles/10.5802/ojmo.11

[14] M. G. C. Resende and R. M. A. Silva, “GRASP: Greedy randomized adaptive search procedures,” in *Wiley Encyclopedia of Operations Research and Management Science*, J. J. Cochran et al., Ed. John Wiley & Sons, 2011, vol. 3, pp. 2118–2128. [Online]. Available: https://mauricio.resende.info/doc/sgrasp2009.pdf

[15] J. L. Bresina, “Heuristic-biased stochastic sampling,” in *AAAI-96: Thirteenth National Conference on Artificial Intelligence*, Portland, Oregon, 1996, pp. 271–278. [Online]. Available: https://cdn.aaai.org/AAAI/1996/AAAI96-041.pdf

[16] P. Quinn and K. Kunkleman, “Position areas for artillery (PAA) analysis in severely restricted terrain,” Center for Army Lessons Learned, Fort Leavenworth, Kansas, Tech. Rep. 24-846, January 2024. [Online]. Available: https://api.army.mil/e2/c/downloads/2024/01/25/a8b49a1b/24-846-paa-analysis-jan-22-public.pdf

[17] Carmenta, “Terrain vehicle analysis,” https://docs.carmenta.com/pages/terrain_vehicle_analysis.html, 2020.

[18] S. Berg, “Terrängtypsschema för skogsarbete,” Skogforsk (the Forestry Research Institute of Sweden), 1995. Swedish version online: https://www.skogforsk.se/kunskapsbanken/kunskapsartiklar/1995/terrangtypschema. An English translation, “Terrain classification system for forestry work,” can be ordered from Skogforsk: https://www.skogforsk.se/english.

[19] NATO Standardization Office, NATO Standard AMSP-06, “Guidance for standards applicable to the development of Next Generation NATO Reference Mobility Models (NG-NRMM),” July 2021. [Online]. Available: https://nso.nato.int/nso/nsdd/main/standards?search=AMSP-06

[20] Tekpool, “Bit count: Parallel counting – MIT HAKMEM,” https://tekpool.wordpress.com/2006/09/25/bit-count-parallel-counting-mit-hakmem, 2006.

[21] S. E. Anderson, “Counting bit sets,” in *Bit Twiddling Hacks*. Stanford, 1997–2005. [Online]. Available: https://graphics.stanford.edu/~seander/bithacks.html

[22] M. Xiao and H. Nagamochi, “Exact algorithms for maximum independent set,” *Information and Computation*, vol. 255, no. 1, pp. 126–146, 2017. [Online]. Available: https://arxiv.org/abs/1312.6260

[23] T. Akiba and Y. Iwata, “Branch-and-reduce exponential/FPT algorithms in practice: A case

study of vertex cover," *Theoretical Computer Science*, vol. 609, pp. 211–225, 2016. [Online]. Available: https://arxiv.org/abs/1411.2680

[24] H. Ayad, "Independent set and vertex cover," Lecture notes, http://www.hananayad.com/teaching/syde423/IndependentSet.pdf, 2008.

[25] J. Dahlum, D. Hespe, S. Lamm, P. Sanders, C. Schulz, D. Strash, R. F. Werneck, and R. Williger, "KaMIS – Karlsruhe Maximum Independent Sets," https://karlsruhemis.github.io.

[26] Y. Shim and M. P. Atkinson, "An analysis of artillery shoot-and-scoot tactics," *Naval Research Logistics*, vol. 65, pp. 242–274, 2018. [Online]. Available: https://faculty.nps.edu/mpatkins/docs/1_ShootScoot_Final.pdf

[27] T. Jonsson Damgaard, M. Rittri, and P. Franz, "Risk-adaptive rendezvous planning for resupply missions in the battlefield," in *Proceedings of the Ground Vehicle Systems Engineering and Technology Symposium (GVSETS)*, NDIA, Novi, MI, Aug. 13–15, 2024. [Online]. Available: https://ndia-mich.org/images/events/gvsets/2024/papers/AAIR/4%2000PM%20Risk%20Adaptive%20Rendevous%20Planning%20for%20Resupply%20Missions%20in%20the%20Battlefield.pdf

[28] S. Tadelis, *Game Theory: An Introduction*. Princeton University Press, 2013.

[29] G. Navarro, "A guided tour to approximate string matching," *ACM Computing Surveys*, vol. 33, no. 1, pp. 31 – 88, March 2001. [Online]. Available: https://users.csc.calpoly.edu/~dekhtyar/570-Fall2011/papers/navarro-approximate.pdf

[30] Desmos Studio, "Graphing Calculator," https://www.desmos.com.

[31] Copernicus Land Monitoring Service, "CORINE land cover," https://land.copernicus.eu/en/products/corine-land-cover.

[32] J. de Ferranti, "Viewfinder Panoramas," https://viewfinderpanoramas.org.

[33] HERE Technologies, https://www.here.com.

[34] GeoNames, https://www.geonames.org.

13 CONTACT INFORMATION

Thomas Jonsson Damgaard
Software Engineer
Thomas.JonssonDamgaard@carmenta.com
Carmenta Geospatial Technologies AB
Box 11354
SE-404 28 Gothenburg
SWEDEN
info@carmenta.com
`www.carmenta.com`

14 ACKNOWLEDGMENTS

Our questions on independent sets were kindly answered by Kenneth Langedal at Heidelberg University, Peter Sanders at the Karlsruhe Institute of Technology, and Darren Strash at Hamilton College.

We have also got good advice from the anonymous reviewers for GVSETS and from our co-workers Patrik Ellrén, Patrick Franz, Marcus Lundberg, Tobias Moberg, and Emilie Le Moël.

The bias function diagrams in Section 6.6 were made with Desmos [30] and Paint.

We used geodata from the following sources:

– CORINE Land Cover data [31].

– Elevations from Viewfinder Panoramas [32].

– Roads from HERE Technologies [33].

– Town names from GeoNames [34].

– Case points from Quinn and Kunkleman [16].

15 NOTATION

`a`, **`b`**, ... , **`t`**	Names of the 20 positions in TAP instance W-20.
`A`, **`B`**, ... , **`T`**	Names of the 20 positions in TAP instances K-20 and D-20.
b	Base of an exponential bias function.
c	Cost of leg (travel time in seconds).
c'	Normalized cost of leg.
d	Degree of a polynomial bias function.
p, q	Positions in a general TAP instance.
$R(p)$	The risk circle around the position p.
r	Rank of leg in a list of legs sorted by cost.
u, v, w	Nodes in a general graph.
μ, σ	Parameters of a Gaussian bias function.

16 ACRONYMS

A*	Algorithm for single-pair shortest path problem [7].
CORINE	*Coordination of Information on the Environment*, a European land-cover dataset (Section 6.1).
D-20	Eastern TAP instance, starting from **`D`** (Section 6.1).
D-17	Reduced version of D-20 with the same optimal solution (Section 7.4).
ExpRankRN	Exponential Rank Random Neighbor algorithm (Section 5.2).
GaussCostRN	Gaussian Cost Random Neighbor algorithm (Section 5.2).
K-20	Eastern TAP instance, starting from **`K`** (Section 6.1).
K-17	Reduced version of K-20 with the same optimal solution (Section 7.4).
MIS-blocker	Maximum-independent-set blocker (Section 7.4).
MIS-requisite	Maximum-independent-set requisite (Section 7.4).
NN	Nearest Neighbor algorithm (Section 5.1).
NP-hard	For NP-hard problems, optimal solutions cannot be found in polynomial time if $\mathrm{P} \neq \mathrm{NP}$ as is widely believed ([8, chapter 34]).
PolyCostRN	Polynomial Cost Random Neighbor algorithm (Section 5.2).
RN	Random Neighbor algorithm (Section 5.2).
TAP	Traveling Artilleryman Problem (Section 3).
TSP	Traveling Salesman Problem (Section 4).
W-20	Western TAP instance, starting from **`a`** (Section 6.1).
W-15	Reduced version of W-20 with the same optimal solution (Section 7.4).