Heuristics for the Generalized Traveling Salesman Problem

Molly Grove
Division of Science and Mathematics
University of Minnesota, Morris
Morris, Minnesota, USA 56267
grove266@morris.umn.edu

ABSTRACT

The generalized traveling salesman problem (GTSP) is a variation of the traveling salesman problem, a classic NP-hard optimization problem. The goal of the GTSP is to find a minimum-cost cycle that visits one node from every subset, or category, of vertices. This paper presents several algorithms for finding approximate solutions to the GTSP.

Keywords

generalized traveling salesman problem, hybrid algorithms, parameterized algorithms, consultant-guided search, variable neighborhood search, combinatorial optimization

1. INTRODUCTION

Many problems with practical applications involve completing a required task with as little cost as possible, whether cost is defined as time, money, distance, or some other measure. While some problems, such as finding the shortest route from one location to another, are relatively easy to solve, others do not have known easy solutions. One example of this is the traveling salesman problem. The traveling salesman problem is a well-known problem where a salesman must travel to all of a specific selection of cities in one trip and wishes to find the shortest route available without repeating any cities. This problem has many applications, including some such as task scheduling where no actual traveling is involved.

The *generalized* traveling salesman problem is like the traveling salesman problem except that instead of going to every city, the cities are divided into groups and the traveler must go to one city in each group. The 'cities' in each group may be relatively close together, as would be the case for a traveler going to one city in each of several states, or they may not be, as would be the case for someone running errands who needs to go to one of each of several types of stores. This means that the generalized traveling salesman problem has potential for even more variation than the traveling salesman problem.

UMM CSci Senior Seminar Conference, December 2015 Morris, MN.

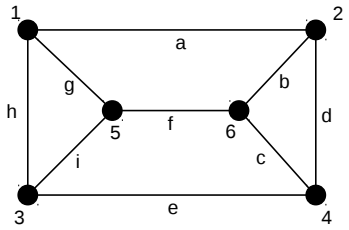

Figure 1: A graph with vertices and edges labeled

Like the traveling salesman problem, the generalized traveling salesman problem has many applications, and cannot be solved optimally without essentially trying all possibilities. For this reason, approximate algorithms, which give a solution that is good, but not always optimal, are used.

This paper presents some approximate algorithms, or *heuristics*, for the generalized traveling salesman problem. Section 2 provides some background necessary to understand the algorithms. Section 3 introduces a parameterized algorithm used for road networks. Section 4 describes a hybrid algorithm that combines several search techniques. Section 5 introduces some search algorithms that can be combined with variable-neighborhood search, as well as some instances that illustrate why combining them is beneficial.

2. BACKGROUND

To understand the problem and heuristics, we must first understand some graph theory, some computation theory, and the precise definition of the generalized traveling salesman problem.

2.1 Graphs

This section defines some graph theory terms necessary to understand the generalized traveling salesman problem.

A *graph* $G = (V, E)$ is an ordered pair of two sets: a set V of *vertices* and a set E of *edges*. Each edge is defined by a pair of vertices. For example, an edge (a, b) connects vertices a and b. The edges may have weight or cost values, $c_{i,j}$, where (i, j) is an edge in E. Vertices are said to be *adjacent* if they are connected by an edge. Vertices are sometimes referred to as *nodes*, and I will be using these terms interchangeably. Figure 1 is an example of a graph with vertices and edges indicated by numbers and letters, respectively.

In non-technical terms, a graph is a set of points (vertices or nodes) with lines (edges) connecting them. Cost values

can be thought of as the cost of traversing the edges. For some applications, this corresponds to the length of the edge.

Given a graph $G = (V, E)$, a *path* is a sequence of unique nodes and edges $(v_1, e_1, v_2, e_2, ..., e_{n-1}, v_n)$ where each node is adjacent to the previous one and the edges between the nodes connect the two nodes. For example, in Figure 1, $(4, e, 3, i, 5, g, 1)$ is a path. Notice that there are no repeated nodes. A *cycle* is a path except that $v_1 = v_n$; that is, a path that starts and ends at the same place. One example of a cycle in Figure 1 is $(2, a, 1, g, 5, f, 6, b, 2)$. Notice that the sequence has no repeated nodes except the first one, which is repeated at the end of the sequence.

2.2 The class NP-hard

In this section, I define some terms used to classify problems based on computational complexity.

A *decision problem* is a problem that, when solved, results in a 'yes' or 'no' answer [9]. A decision problem is said to be in the class P if, in the worst case, it can be solved by an algorithm in polynomial time [4]. A problem can be solved in *polynomial time* if an algorithm with an input of size n can solve the problem in no more than n^k steps, where k is a constant that does not depend on n.

Exact algorithms are generally used for problems in the class P, because most computers can use polynomial time algorithms to solve them quickly enough for most practical purposes, even when n is fairly large. However, there are many problems for which no polynomial-time solution has been found.

A decision problem is in the class *NP* if a polynomial-time algorithm can *verify* it, meaning that the algorithm can check in polynomial time if a given candidate for a solution is indeed a solution. Note that this definition does not exclude problems in P. In fact, P is a subset of NP.

There are some problems that can be proven to be at least as hard as every problem in NP. These are referred to as *NP-hard*. A problem is NP-hard if every problem in NP can be *polynomially reduced* to it [4]. This means that if there is a polynomial-time algorithm that can solve an NP-hard problem, the algorithm can be changed into an algorithm to solve any problem in NP in polynomial time.

Because the traveling salesman problem is not a decision problem, it is not in NP. However, it is NP-hard, along with the generalized traveling salesman problem and many other optimization problems. Problems that are not in P, including NP-hard problems, are much more computationally intensive when the input is large, since the number of steps increases more quickly (often exponentially) as the input size increases. For this reason, approximate algorithms are often used for NP-hard optimization problems. These algorithms don't always lead to the optimal solution. However, some of them can generate solutions that are good enough for practical purposes in a reasonable amount of time.

2.3 Generalized Traveling Salesman Problem

The generalized traveling salesman problem is, as the name suggests, a generalization of a different problem, the traveling salesman problem.

The traveling salesman problem (TSP) is defined as follows: Given a graph $G = (V, E)$, find the minimum-cost cycle that contains all nodes in G. The TSP is known to be NP-hard [4].

In the generalized traveling salesman problem (GTSP), instead of going to all nodes, the nodes are divided into disjoint subsets and the cycle has to go to one node in each subset. Mathematically, it can be defined as follows: Given a graph $G = (V, E)$ and disjoint subsets $V_1, V_2, ..., V_n$ of the set V, find the minimum cost cycle containing one node from each subset V_i.

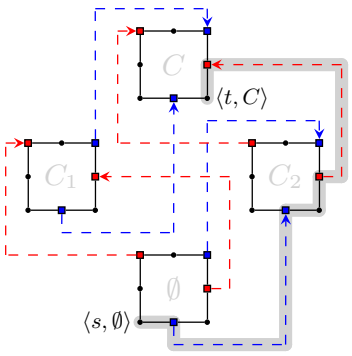

Figure 2: The product graph G_C with the shortest path highlighted (taken from [8])

Rice and Tsotras [8] show that the TSP can be reduced to the GTSP, so the GTSP is also NP-hard.

2.4 The Generalized Traveling Salesman Path Problem

The generalized traveling salesman *path* problem (GTSPP) is similar to the generalized traveling salesman problem, except that the starting and ending node are defined, and they are not necessarily the same node. Formally, it can be defined as follows: Given a graph $G = (V, E)$, disjoint subsets $V_1, V_2, ..., V_n$ of the set V, and nodes s and t in V, find the minimum cost path from s to t containing one node from each subset V_i.

Rice and Tsotras [8] show that the GTSP can be reduced to the GTSPP. Any GTSPP algorithm can also be used for the GTSP, by setting $s = t$ and taking the minimum cost over all possible choices of s. Since the GTSP is NP-hard, this means that the GTSPP is also NP-hard.

3. PARAMETERIZED ALGORITHMS AND THE GTSPP

One method of finding an approximate solution to the GTSPP or the GTSP is to use parameterized algorithms. Parameterized complexity theory defines complexity not just by the size of the input, but also by other parameters. This helps distinguish between algorithms that are inefficient in theory and practice and algorithms that are inefficient in theory but actually not that inefficient for many practical purposes.

This section summarizes the algorithm presented by Rice and Tsotras [8]. The algorithm that follows is most useful in cases where the nodes in each node subset are scattered. This algorithm is for the GTSPP, from Section 2.4. However, as mentioned before, the same algorithm could be used for the GTSP. This algorithm also allows repeated nodes.

3.1 The Product Graph

The algorithm starts by constructing a *product graph*. For a set C of node subsets, the product graph is defined as

$G_C = (V \times \mathcal{P}(C), E_1 \cup E_2)$, where $\mathcal{P}(C)$ is the *power set*, or set of all subsets, of C. E_1 is the set of edges in G copied for each subset of C, and E_2 is a set of edges from each node in each proper subset c of C to the same node in every other subset c' such that c is a proper subset of c' but there are no proper subsets c'' of c' such that c is a proper subset of c''. This is shown in Figure 2. The original graph G is duplicated for every subset of C. The E_1 edges are within each duplicate and shown as solid lines. The E_2 edges go between the duplicates of G, and are shown as dashed lines.

This construction turns the problem into a shortest-path type problem from s in the first state, the empty set, to t in the last state, C. Reaching a node from a node subset results in advancing to a state that reflects having already visited that subset.

3.2 Contraction Hierarchies

The algorithm uses a technique known as contraction hierarchies (CH) on the product graph. This process assigns a unique rank, $\phi(u)$, to each node u. The method of ranking nodes used in this algorithm is from Geisberger et al. [1]. It assigns higher values to nodes in more shortest paths.

After the nodes are ranked, for every unique shortest path from v to w containing u, if $\phi(u)$ is less than $\phi(v)$ and $\phi(w)$, a short-cut edge is added with a cost equal to the sum of the costs of (u, v) and (v, w). The "upward" graph, $G^{\uparrow}$, is defined as $(V, E^{\uparrow})$, where $E^{\uparrow}$ contains all edges (u, v) such that $\phi(u) < \phi(v)$, and the "downward" graph, $G^{\downarrow}$, is defined as $(V, E^{\downarrow})$, where $E^{\downarrow}$ contains all edges (u, v) such that $\phi(u) > \phi(v)$.

3.3 Establishing the Upper Bound

The next step in the algorithm is to calculate an upper bound, μ, on the possible solution. This is accomplished by constructing a path on the CH product graph starting with s and going along the shortest (or minimum cost) edge such that the destination node is in a node subset where no node has been visited, until all subsets have been visited. After all subsets have been visited, the total cost of the path plus the distance from the last node to t is an upper bound μ on the possible cost (meaning that the optimal solution cannot have a higher cost). This is a variation of an approximate algorithm for the TSP known as the "nearest neighbor" algorithm.

3.4 Δ-Corridors

The next step in the algorithm is to establish Δ-corridors. Δ-corridors are based on the idea that it's not necessary to consider nodes that are far away from the space between s and t, assuming that there are nodes from all node subsets near the direct path between s and t.

This step establishes V_Δ, defined as $V_\Delta = \{v \in V | d(s, v) + d(v, t) \leq \Delta\}$. To find this, Dijkstra's algorithm, a shortest-path algorithm, is used on the CH graph once from s on $G^{\uparrow}$ and once from t on $G^{\downarrow}$. The details of this step can be found in [8]. The algorithm uses $\Delta = \mu/(1 + \epsilon)$, where $\epsilon \geq 0$. If $\epsilon = 0$, this algorithm is an exact algorithm.

3.5 A* Search

In A^* search, a value is assigned to each node with the formula $f(v) = d(v) + h(v)$, where $d(v)$ is the cost of the minimum cost known path from the starting node s to v and $h(v)$ is the estimated cost for minimum cost path from v to the end node t.

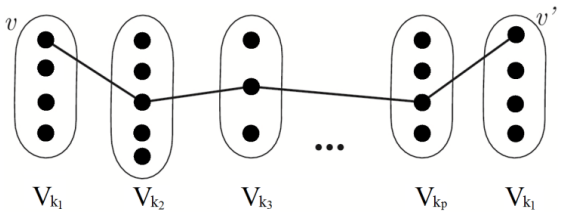

Figure 3: The local-global approach (taken from [5])

The algorithm begins by inserting all nodes adjacent to s into a set F, the "fringe" set, and setting $d(s) = 0$. For all other nodes, $d(v)$ is defined as ∞.

Next, a node u with minimum f value (as defined above) is removed from F. For every node v adjacent to u, if $d(v) > d(u) + c(e)$, where e is the edge between u and v, then $d(v)$ is set to be $d(u) + c(e)$ and v is inserted into F. This is repeated until t is reached and chosen to be removed from F, when $d(t)$ is returned.

The algorithm uses A^* search on the CH product graph within $V_{\mu/(1+\epsilon)}$, the Δ-corridor established in Section 3.4. Rice and Tsotras [8] prove that their A^* search heuristics will return the correct shortest path.

After A^* search is completed, $min\{\mu, w(P^*)\}$ is returned, where μ is the upper bound calculated in Section 3.3 and P^* is the path calculated in A^* search. Rice and Tsotras [8] prove that this algorithm runs in $O^*(2^k)$ time, where k is the number of node subsets and the O^* notation means that polynomial factors are omitted.

3.6 Experimental Results

Rice and Tsotras [8] conducted experiments using the road network of North America.

In the first experiment, the algorithm was implemented with the number of subsets V_i fixed at 5, varying ϵ and the numbers of nodes in each subset. They found that query time decreased from 1 second to 0.002 seconds as the numbers of nodes increased from 10 to 1,000,000. The query time also decreased as ϵ increased. The cost of the solutions found are worse than optimal by less than 25%, and are significantly lower in most cases.

In the second experiment, the number of nodes in each subset was fixed at 10,000 and the number of nodes and ϵ were varied. The results showed errors of less than 5%.

4. LOCAL-GLOBAL SEARCH AND CONSULTANT GUIDED SEARCH

Often, combining several algorithms to make a hybrid algorithm can generate an algorithm that is more effective than the original algorithms alone. This can be useful for solving NP-hard optimization problems like the GTSP. In this section, I present an algorithm from Pop and Iordache [5]. This algorithm is a hybrid of two algorithms: local-global search and consultant-guided search (CGS). Unlike the algorithm in Section 3, this algorithm assumes that the nodes in each node subset are close together.

4.1 Local-Global Search

Local-global search is a technique that distinguishes between edges connecting nodes within the same subset and

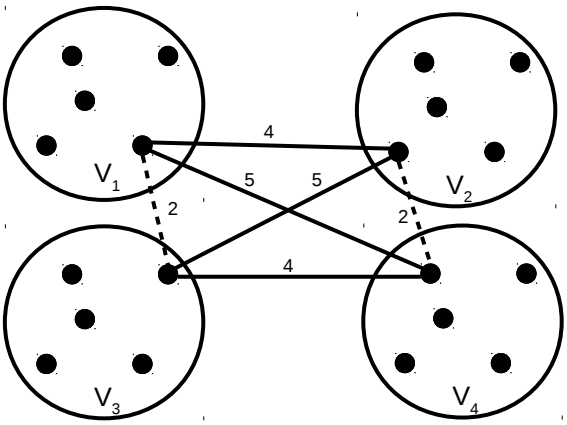

Figure 4: Local-global search given the sequence (V_1, V_2, V_3, V_4), with costs as labeled

edges connecting nodes of different subsets.

A new graph, G', is constructed with a supernode that represents each subset V_i. Supernodes are denoted with the name of the subset V_i.

Given a sequence of supernodes $V_{k_1}, V_{k_2}, ..., V_{k_n}$, the local-global search algorithm finds the minimum-cost cycle that visits one node from each subset in the order of the sequence. This is accomplished by duplicating V_{k_1} and putting it at the end of the sequence, then considering each path from each node in V_{k_1} to the corresponding node in $V_{k_{n+1}}$, visiting the subsets in the order of the sequence, as shown in Figure 3.

This algorithm is a polynomial time algorithm. This may seem like it contradicts the GTSP being an NP-hard problem. However, the algorithm does not actually find the overall optimal solution for the GTSP; rather, it only finds the optimal solution for a particular sequence of node subsets. One example of this is shown in Figure 4. This is indeed the shortest cycle given the sequence (V_1, V_2, V_3, V_4), but it's not the best solution overall. An optimal solution would use the same nodes with the dotted edges and the sequence (V_1, V_2, V_4, V_3), but local-global search doesn't rearrange the sequence order. Using the algorithm to find the optimal solution over all possible sequences is an exponential time algorithm, as expected.

4.2 Consultant-Guided Search

Consultant-guided search is also used. The following information is from Iordache [3].

Consultant-guided search (CGS) is a swarm intelligence algorithm, meaning that it is decentralized. It is based on the way people take advice from consultants.

The algorithm has simulated individuals that take on two roles, consultants and clients. Each iteration, each virtual client chooses a virtual consultant based on the consultants' *reputation*. A consultant's reputation increases when the consultant's clients succeed. A *success* is defined as a solution that is better than any other solution that has been found by the algorithm. Reputation generally decreases over time unless prevented by successes. The exception to this rule is that for some consultants who have had a very high reputation at some point, the reputation is prevented from going below a certain level.

Another factor that influences a client's choice is the consultant's *personal preference*. This factor is different for each problem where CGS is used.

In addition to reputation, each consultant also has a *strategy*, which they use to help the client to solve the problem. Using the strategy, the consultant offers a suggestion, and the client may or may not take the suggestion offered. This introduces some randomness into the construction process.

The consultants also have a *sabbatical* mode, where they do not give advice to clients and change their strategy. Sabbatical mode is activated if the consultant's reputation goes below a certain level, and is deactivated after a certain amount of time.

4.3 The Hybrid Algorithm

The algorithm Pop and Iordache propose is a hybrid of the local-global and CGS techniques.

The consultant uses virtual distances between subsets as described in Section 4.1, with each supernode at the center of mass of the subset, to generate a cycle that visits all supernodes. This cycle is improved with methods that are beyond the scope of this paper, and can be found at [5]. The consultant then uses this cycle to advise the clients. In each step of generating the cycle, the consultant randomly chooses the next supernode from the supernodes not yet visited within a candidate list of the n closest supernodes. The probability of choosing supernode j from supernode i is given by the formula

$$p_{ij}^k = \frac{1/d_{ij}}{\sum_{l \in \mathcal{N}_i^k} (1/d_{il})}$$

where $\mathcal{N}_i^k$ is the set of possible supernodes and d_{ij} is the distance between the two supernodes i and j.

When a client chooses a consultant, the consultant recommends the next subset for the client to visit. The consultant accomplishes this by looking at the client's current supernode and finding it in the consultant's constructed cycle. If the client has not visited either the supernode before or after the client's current supernode, the consultant will recommend the one that has not yet been visited, or randomly pick one of the two if neither one has been visited. If the client has already visited both supernodes, the consultant does not make a recommendation.

The client may or may not take this recommendation. If there are still supernodes within the candidate list that have not been visited by the client, the client's choice is dictated by the formula

$$j = \begin{cases} v & v \neq \text{null} \wedge q \leq q_0 \\ \text{random}\left(\mathcal{N}_i^k\right) & \text{otherwise} \end{cases}$$

where i is the current node subset, v is the recommended node subset, q_0 is a parameter, and q is a random variable between 0 and 1 (inclusive).

If all supernodes in the candidate list have been visited, the client chooses a supernode not in the candidate list.

After a sequence of supernodes is chosen, the local-global search technique in Section 4.1 is used to find the best overall cycle.

4.4 A variant using confidence

Pop and Iordache also present a variant where each consultant's confidence in each edge of a cycle factors in the client's decision to take the suggestion. In this variant, each edge in a consultant's strategy cycle has a strength. When

a consultant uses an edge in the cycle for the first time, it is given a strength of 0. Each time the consultant re-uses the same edge, the edge's strength is incremented. A client is more likely to choose the recommended supernode if the edge between the supernodes has a higher strength.

4.5 Experimental Results

The algorithm and the variant with confidence were tested on GTSP instances adapted from TSP instances from the TSPLIB library [7]. The results from the algorithm variant with confidence were statistically similar to the best known heuristic at the time the paper was written, and in some cases, the variant with confidence was significantly better.

5. VARIABLE NEIGHBORHOOD SEARCH

This section presents another algorithm that combines several techniques, this time using variable neighborhood search. Although this algorithm was originally presented by Hu and Raidl [2], most of what is presented below is from Pourhassan and Neumann [6]. Pourhassan and Neumann present some instances that illustrate the strengths and weaknesses of each individual technique and the combined algorithm.

5.1 Cluster-Based Local Search

The cluster-based approach starts with a permutation of the node subsets (or *clusters*), then finds the optimal set of nodes within the clusters for this permutation. First, a permutation of clusters $\pi = (V_{k_1}, V_{k_2}, ..., V_{k_m})$ is chosen. The *2-opt neighborhood* of π is defined as

$$N(\pi) = \{\pi' | 1 \leq i < j \leq m,$$
$$\pi' = (V_{k_1}, ..., V_{k_{i-1}}, V_{k_j}, V_{k_{j-1}}, ..., V_{k_i}, V_{k_{j+1}}, ..., V_{k_m})\}$$

where m is the number of clusters. In other words, part of π is reversed (specifically, the part between V_{k_i} and V_{k_j}). Note that this does not include all possible permutations of the node subsets.

Cluster-based local search (CBLS) searches through every permutation from the 2-opt neighborhood of π and finds optimal nodes for the permutation. For each permutation, if the cost of the cycle is less than the cost of the current lowest cost cycle, that cycle is stored as the new lowest cost cycle and π is set to the new permutation. This finds the minimum cost cycle from the permutations that are tried.

5.2 Node-Based Local Search

The node-based approach starts with a set P of one node from each cluster, then finds the minimum-cost cycle within that set. Note that the second step is basically the standard TSP, as described in Section 2.3. Because the TSP is NP-hard, this step cannot be solved optimally in polynomial time, although this doesn't matter much if the number of clusters is small. Alternatively, an approximate algorithm for the TSP may be used here. Pourhassan and Neumann [6] present two algorithms used in this step–one that finds an optimal solution and one that approximates a solution.

The neighborhood of the set P of nodes can be described formally as

$$N'(P) = \{P' | P' = \{p_1, ..., p_{i-1}, p'_i, p_{i+1}, ..., p_m\},$$
$$p'_i \in V_i \backslash \{p_i\}, 1 \leq i \leq m\}$$

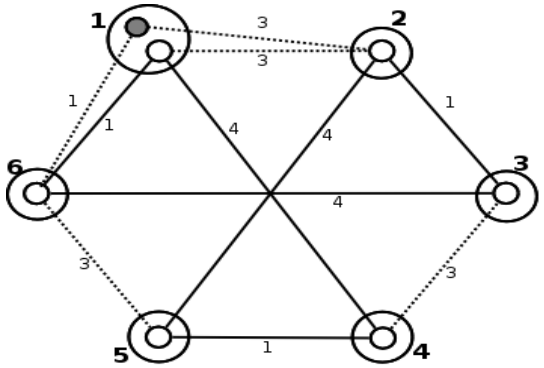

Figure 5: The first instance (taken from [6])

In other words, the neighborhood of P is given by all possible sets P' where one node p_i is removed from P and a different node in the same subset as p_i is added.

The algorithm first finds the solution for P, then searches the neighborhood of P'. If the solution found for P' is better than the current best solution, the new solution is set as the current best and P' is set as P. The algorithm terminates when there are no better solutions found for the neighborhood of P.

5.3 Variable Neighborhood Search

The Variable Neighborhood Search algorithm combines cluster-based local search and node-based local search into one algorithm. The combined algorithm starts with a possible solution (P, π), where P is a set of nodes and π is a permutation of clusters, then uses two neighborhood types based on the two algorithms, which can be defined as

- $N_1(P) = \{(P', \pi') | \pi' \in N(\pi), P' = \text{optimal set of nodes with respect to } \pi'\}$,
- $N_2(\pi) = \{(p', \pi') | P' \in N'(P), \pi' = \text{ order of clusters obtained by 2-opt from } \pi \text{ on } G[P']\}$

First, N_1 is used along with the CBLS algorithm to find a possible solution. When this is found, N_2 is used with NBLS, using P and π as found by N_1.

5.4 Example Instances

Approximate algorithms usually have some cases that are easier for them than others. This section presents some of these cases.

The first instance is one that is difficult for the cluster-based local search technique, but easy for the node-based local search technique. The graph for this instance is shown in Figure 5. The optimal solution would visit the nodes using the edges on the sides of the graph, and not use the edges with cost 4. However, Pourhassan and Neumann [6] prove that if the starting permutation is $\pi = (1, 4, 5, 2, 3, 6)$, then the cluster-based local search algorithm will never find one of these solutions, but node-based local search does.

The second instance is difficult for node-based local search, but easy for cluster-based local search. This instance is shown in Figure 6. The instance includes one cluster with a black node that is closer to the other clusters and a white node that is farther away. The optimal solution uses all black

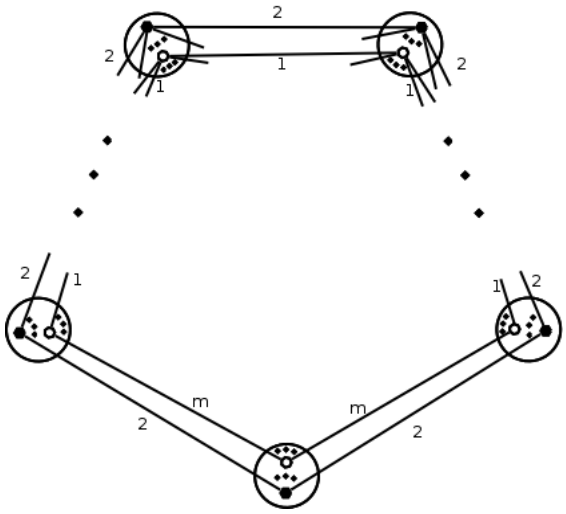

Figure 6: The second instance (taken from [6])

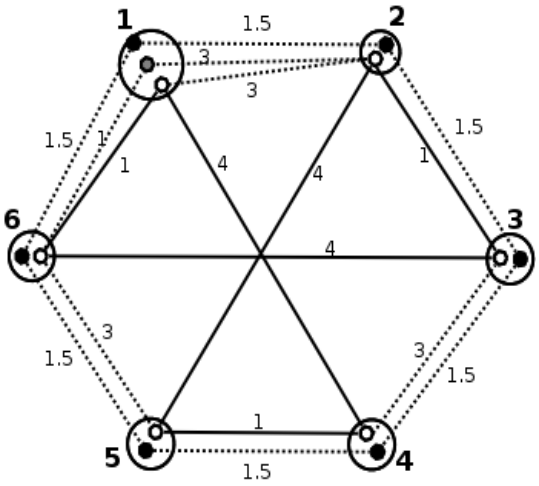

Figure 7: The third instance (taken from [6])

nodes. Pourhassan and Neumann [6] prove that cluster-based local search is able to find an optimal solution, whereas node-based local search gets stuck in local optima.

The third instance is shown in Figure 7. This instance combines the first two, with the basic structure of the first one, but with varying lengths of edges between nodes of the same clusters. This makes the instance hard for both cluster-based and node-based local search. Variable-neighborhood search, however, doesn't get stuck and can find a good approximation. This illustrates the advantages of variable-neighborhood search.

6. CONCLUSIONS

I have summarized three heuristics for the GTSP. Although the GTSP is NP-hard and exact algorithms are often time-consuming, approximate algorithms are effective for many practical purposes. Additionally, as Rice and Tsotras [8] show, there are specific cases where in practice, exact algorithms are feasible.

There are many different applications of the GTSP, and sometimes, the application changes which algorithm would work best. Rice and Tsotras [8] apply their algorithm for road trips, which assumes that elements of each node subset are spread out. Pop and Iordache [5] mention that they assume the nodes in each subset are near each other. As Pourhassan and Neumann [6] point out, factors like the distance between node subsets and nodes in subsets make a significant difference in how well an algorithm will work. A good heuristic should produce good approximations in a variety of cases, but when choosing a heuristic, it's important to consider the application.

7. ACKNOWLEDGEMENTS

I would like to thank Elena Machkasova, Nic McPhee, and Max Magnuson for their feedback on this paper. I would also like to thank Michael Rice and Vassilis Tsotras for answering questions about their work and providing a copy of one of their earlier papers.

8. REFERENCES

[1] R. Geisberger, P. Sanders, D. Schultes, and D. Delling. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *Experimental Algorithms*, pages 319–333. Springer, 2008.

[2] B. Hu and G. R. Raidl. Effective neighborhood structures for the generalized traveling salesman problem. In *Proceedings of the 8th European Conference on Evolutionary Computation in Combinatorial Optimization*, EvoCOP'08, pages 36–47, Berlin, Heidelberg, 2008. Springer-Verlag.

[3] S. Iordache. Consultant-guided search: A new metaheuristic for combinatorial optimization problems. In *Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation*, GECCO '10, pages 225–232, New York, NY, USA, 2010. ACM.

[4] C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Prentice-Hall, INC., Englewood Cliffs, NJ, USA, 1982.

[5] P. C. Pop and S. Iordache. A hybrid heuristic approach for solving the generalized traveling salesman problem. In *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation*, GECCO '11, pages 481–488, New York, NY, USA, 2011. ACM.

[6] M. Pourhassan and F. Neumann. On the impact of local search operators and variable neighbourhood search for the generalized travelling salesperson problem. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, GECCO '15, pages 465–472, New York, NY, USA, 2015. ACM.

[7] G. Reinelt. TSPLIB - a traveling salesman problem library. *ORSA Journal on Computing*, 3(4):376–384, 1991.

[8] M. N. Rice and V. J. Tsotras. Parameterized algorithms for generalized traveling salesman problems in road networks. In *Proceedings of the 21st ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, SIGSPATIAL'13, pages 114–123, New York, NY, USA, 2013. ACM.

[9] M. Sipser. *Introduction to the Theory of Computation*. Cengage Learning, Boston, MA, USA, 2013.