UNIVERSITY OF CALIFORNIA
Los Angeles

Computing Optimal Solutions to the Orienteering Problem

A thesis submitted in partial satisfaction
of the requirements for the degree
Master of Science in Computer Science

by

Miles Stephenson

2019

ABSTRACT OF THE THESIS

Computing Optimal Solutions to the Orienteering Problem

by

Miles Stephenson
Master of Science in Computer Science
University of California, Los Angeles, 2019
Professor Richard E. Korf, Chair

We have found two admissible heuristics that we use within a branch and bound framework to compute optimal solutions to the Orienteering Problem on both complete and incomplete graphs. Our approach exponentially outperforms naive methods in terms of both node expansions and runtime. A detailed description of our heuristics and overall algorithm are provided. Finally, we analyze the performance of our approach on a variety problem of instances.

The thesis of Miles Stephenson is approved.

Rafail Ostrovsky

Guy Van den Broeck

Richard E. Korf, Committee Chair

University of California, Los Angeles

2019

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to Professor Richard Korf. He is responsible for introducing me to many of the topics discussed in this thesis. Additionally, the many hours of conversation we spent discussing this problem were invaluable in helping me refine my approach.

CHAPTER 1

Introduction

1.1 Motivation

Consider the following problem: A salesperson leaves on a business trip of some predetermined length to sell products to residents of a region. The amount of sales revenue in a city is known to be proportional to its population, but travelling between cities in the region via car takes time. The salesperson's trip starts and ends at specific locations, but there are no restrictions on cities visited intermediately. After selling products to residents in the first city, the salesperson would like to know how best to plan the remainder of his or her trip. How should the salesperson plan his or her path through the region to maximize the amount of sales revenue, given the amount of time available?

The above is a real world example of the type of problem that can be optimally solved by the methods described in this thesis. Furthermore, the orienteering problem has recently been used as a model in several practical applications, such as the tourist trip design problem as well as the mobile-crowdsourcing problem. [GLV16]

1.2 Problem Description

The orienteering problem (OP) is an NP-hard routing problem on a connected (not necessarily complete), undirected edge-weighted graph G = (V, E). The weight of an edge represents the cost of traversing that edge, that is moving from one node to another via that edge. Additionally, each vertex has associated with it a positive integer reward which is collected upon the initial visit to the node. Nodes can be revisited, but no additional reward is gained

from subsequent visits. Starting and ending at specific initial and terminal nodes, and given an edge budget, the objective is to determine a subset of nodes to visit, and in which order (i.e., a path), such that the sum of the rewards collected from nodes is maximized and the edge budget is not exceeded. Unlike the TSP, a solution to the OP need not be a cycle, unless the source and destination nodes are the same.

An optimal solution to the OP is implicitly defined as a path through the graph such that the sum of the rewards collected is greater than or equal to the sum of rewards possible via any other valid path through the graph.

1.3 Complexity of the OP

The OP can be shown to be NP-hard through a reduction from Hamiltonian cycle. To see this, consider the following example. Suppose we have a connected graph with unit edge weights and unit node rewards. Let the source and destination node be the same and let the initial edge budget be equal to the number of nodes in the graph. Let the number of nodes in this graph be V, where $V > 2$.

Suppose we have an algorithm A that computes optimal solutions to the orienteering problem. There are two cases. If the value of the solution returned by A is equal to V, then the resulting path must be a Hamiltonian cycle in the graph. If the solution value is less than V, then we know that a Hamiltonian path does not exist in this graph. This completes the reduction, and shows the OP is NP-hard.

CHAPTER 2

Problem Space

We considered the problem space described below during the development of our solution method. Other problem space representations are possible, such as the set of edges included in the current partial path through the graph, but we found this representation to be the most intuitive.

2.1 State Representation

A state of the OP is uniquely defined by the following:

- The current location on the graph (a node)
- The set of nodes visited thus far
- The cumulative sum of rewards collected so far via this partial path
- The amount of edge budget remaining

2.2 Operators

There is only a single operator, which is moving from the current node to any of the adjacent nodes, at the cost of the edge connecting it to the current node. This action is only possible when the cost of the edge is less than or equal to the remaining edge budget. After each such move, the edge budget is decremented by the cost of the edge traversed.

2.3 Initial State

The initial state is defined as a state in which no edge budget has been spent, that is, the entire initial edge budget remains and the sum of rewards collected equals only the reward associated with the initial node. Furthermore, in the initial state the current node is the initial node and the path contains only the initial node.

2.4 Goal State

A terminal state is implicitly defined as a state such that insufficient edge budget remains to visit any other unvisited node from the current state. The current location on the graph (the final node on the path) in the goal state must be the specified destination node. A terminal state is considered optimal if the sum of rewards collected is greater than or equal to the sum possible via any other valid path.

CHAPTER 3

Comparison to Existing Methods

3.1 Problem Variations

There are many variants of the orienteering problem. The version of the OP we considered is as follows:

- The path must begin and end at specific nodes (possibly the same)
- The path need not be a cycle
- Nodes can be visited multiple times, but no additional reward is earned from subsequent visits
- The graph is not necessarily complete
- The graph is not necessarily embedded in a plane

Our variant of the OP is the most general when compared to what appears in the literature. Namely, we do not require the solution's path to be a cycle, nor do we prohibit nodes from being revisited. Because of these generalizations, optimal zero-one integer programming methods published in [FGT98] are not applicable to our variant of the OP, as far as we can tell. Of the optimal methods we found after an extensive survey of the literature, the OP variant and methods published in [BH17] are the most similar and directly applicable to our problem.

3.2 Orienteering with Category Constraints

Of the many orienteering problem variants published in the literature, our problem setup and solution methods are most similar to Orienteering with Category Constraints, described in [BH17]. The focus of that work is similar to ours, namely using heuristic search methods to optimally solve the OP. The main difference in their problem variant is the addition of category constraints, where each node in the graph belongs to a category and there is a limit imposed on the number of nodes in each category that a solution's path can visit. Their approach is currently considered to be the state-of-the-art for this variant of the OP.

Our variant of the OP can be solved via their methods if we consider only a single category and set the constraint for that category equal to the number of nodes in the graph. Admittedly, this represents one of the worst case scenarios for their method, however the purpose of this comparison is to determine the effectiveness of both their heuristic and a best-first-search. We found our approach significantly outperformed their algorithm in every test on instances of our OP variant. This is due partially to the overhead of maintaining the double-ended priority queue used in their best-first-search, however the vast majority of the relative efficiency of our approach is due to the superiority of our heuristics. To be specific, when we compared our approach to theirs, we used our AUE heuristic. While both heuristics are admissible, the AUE heuristic will always produce an upper bound that is less than or equal to the value produced by the heuristic described in their paper. A detailed explanation of the AUE heuristic, as well as a definition of admissibility in this context are provided subsequently. After describing our approach, we will show how, with slight modification, our heuristics could be employed to solve the OP with category constraints with considerably greater efficiency than what is currently considered the state-of-the-art.

3.3 Approximation Methods

After a survey of the literature, it seems that the vast majority of researchers concentrate on approximation methods. This differs from the objective of this thesis, which was to compute

optimal solutions to the OP. As such, these methods offered little valuable insight.

CHAPTER 4

Our Approach

To find optimal solutions, we need an admissible search algorithm. Exponential-space algorithms like A* are impractical on large instances of the OP. This is because in practice we will quickly exhaust the available memory. Since it is trivial to find *a solution*, this suggests depth-first branch and bound. Next, we need an admissible heuristic function. Since the OP is a maximization problem, admissibility means that the heuristic will never underestimate the additional rewards that could be collected from a given state. This property allows our branch and bound algorithm to safely prune branches of the search tree that could not possibly result in optimal solutions. A brief description of depth-first branch and bound applied to the OP is provided in the section below.

4.1 Depth First Branch and Bound

Depth-first branch and bound performs a single depth-first search of the problem-space tree. Assume the search is done left-to-right without loss of generality. As soon as the left-most branch terminates, we have a candidate solution. Let the value of this solution be α, which is the value of the best complete solution found so far. Initially, α is set to 0. As the search continues, the sum of rewards collected along the path plus the (admissible) heuristic evaluation of the current state is computed and compared to α. This sum represents an upper bound on the solution we could achieve by continuing down the search tree from the current state. If this upper bound is less than α, the branch below that node (in the search tree) is pruned, because it can only lead to solutions whose value is less than or equal to the best solution found so far. If a complete solution is found whose sum of rewards value

is greater than α, α is updated to this greater value, and the corresponding best path is replaced by the path leading to this state. When the search terminates by exhausting the entire search tree, the best solution found is returned as the optimal solution.

With an admissible heuristic, depth-first branch and bound is guaranteed to return an optimal solution. Since this is a depth-first search, the space complexity is linear in the maximum search depth. [Kor13]

4.2 Preprocessing the Graph

Both of our heuristics require the length of the shortest path from the current node to every other node in the graph. Since the heuristic function is called at every node generation, it should run as efficiently as possible. Calculating the lengths of the shortest paths from the current node to all other nodes is $O(n^2)$ in terms of time complexity. As such, doing so at every node generation would be extremely inefficient. This cost can be amortized by computing the length of the shortest path from every node to every other node once, as a preprocessing step before the search begins, and caching the values. This is relatively cheap, requiring only $O(n^3)$ time just once. This results in an NxN matrix (where N is the number of nodes), in which shortest path lengths between any two nodes can be looked up in constant time. In practice, the time required to generate this matrix is trivial relative to time spent searching for the optimal solution.

With this information, nodes whose shortest path distance from the current state is greater than the remaining edge budget can be removed from consideration, since they are inaccessible from the current node via any legal path. Our heuristic functions will use this information to reduce the number of nodes they consider every time they are called.

CHAPTER 5

Heuristic Evaluation Functions

To prune suboptimal branches of the search tree within a branch-and-bound framework, we need an admissible heuristic function. Since we are interested in maximal solutions, our heuristic must never underestimate the additional rewards that could be achieved from a given state. In this section, we discuss two such heuristics, one of which generally expands fewer nodes, but is more computationally expensive. Later, we analyze both heuristics on a variety of graphs.

5.1 Definitions of Accessibility

In our descriptions of the heuristic evaluation functions, we refer to the accessibility of nodes and edges. Below we formally define what we mean in both contexts.

- We say that a node x is *accessible* if in the current state there is sufficient edge budget to travel to x from the current node, as well as from x to the destination node via shortest paths. For efficiency, we determine this using the pairwise shortest path matrix we described previously. Formally, x is accessible if it satisfies the following:

$$s.p.\ dist(curr, x) + s.p.\ dist(x, dest) \leq budget$$

- We say that an edge is *accessible* if the nodes at both of its endpoints are accessible, as defined above.

5.2 Affordable Untraversed Edges Heuristic

The first heuristic we discuss works by establishing an upper bound on the *number* of previously unvisited nodes that could be visited from the current state. Then, to establish an upper bound on the additional rewards that could be achieved, it is assumed that all of these nodes have the greatest possible rewards. This heuristic requires nodes to be sorted in descending order by reward and for edges to be sorted in ascending order by weight. We refer to this heuristic as the *Affordable Untraversed Edges* heuristic, or AUE for short.

The key observation made in the construction of this heuristic is that to collect an additional reward, we must visit a previously unvisited node. A lower bound on the cost we would incur to visit a new node is the weight of the minimum weight accessible untraversed edge. We use the matrix of pairwise shortest path values we precomputed to consider only nodes and edges in the accessible region of the graph. The heuristic establishes an upper bound first by determining the greatest number of untraversed edges (edges with at least one endpoint at an unvisited node) in the accessible region of the graph that could possibly be traversed given the amount of edge budget remaining. We establish this number by "buying" the cheapest edge in the accessible region of the graph until insufficient budget remains to buy the next edge. Let this number of edges be m. Then, the greatest m rewards in accessible unvisited nodes are summed. This sum is returned as the heuristic evaluation of the current state.

With this implementation of our heuristic, its time complexity is dominated by the sorting of the edges and nodes. In the next section, we show an optimization which allows us to improve this to linear time complexity.

5.3 AUE Heuristic Implementation Optimization

Instead of sorting both the nodes and edges every time the AUE heuristic is called, we sort only twice as a preprocessing step. We sort the edges by weight in ascending order, and the nodes by reward in descending order.

With the nodes and edges stored this way, there is no need to sort each time the heuristic is called. We simply scan across the pre-sorted array of edges, check accessibility (i.e., both endpoint nodes are accessible), incrementing the total number of edges we can afford and decrementing the budget until we can't afford the next edge. Then, we do another linear scan across the pre-sorted array of nodes, check if the node is accessible and previously unvisited, and if so, increment the additional rewards possible by the value of the reward associated with the node. Determining the accessibility of a node can be done in constant time using the pairwise shortest path matrix we precomputed. This results in an improvement in the time complexity of the heuristic. After factoring out the sorting calls, it now runs in $O(V + E)$, where V and E are the number of nodes and edges in the graph, respectively.

5.4 Admissibility of the Affordable Untraversed Edges Heuristic

Theorem 1. *The affordable untraversed edges heuristic never underestimates the sum of additional rewards that could be achieved from the current state via any valid path to the terminal node.*

Proof. The first step in the proof is to determine the greatest number of previously untraversed edges in the reachable graph (the graph comprised entirely of accessible nodes and edges) that could be traversed given the remaining edge budget. By sorting the untraversed edges from least to greatest cost, and "buying" edges until insufficient edge budget remains to "buy" another edge, the greatest total number of edges is "bought". [See proof in Lemma below]

Next, observe that from any node, in order to visit another node an edge must be traversed. Specifically, from the current node, to visit an unvisited node, at the very least, a single untraversed edge must be traversed. We have established an upper bound on the number of untraversed edges that could be traversed given the remaining edge budget. Let this number of edges be m. Having established this upper bound on the number of untraversed edges, an upper bound on the additional rewards achievable given the current state

Data: current state

Result: upper bound on additional rewards

affordable edges $\leftarrow 0$;

budget $\leftarrow$ *current state's budget*;

for *each edge in pre-sorted edge list* **do**
    if *edge has not been traversed* ***and***
      edge_cost $\leq$ *budget* **then**
        affordable edges ++;
        budget $-=$ *edge cost*;
    end
end

additional_rewards $\leftarrow 0$;

for *each node in pre-sorted node list* **do**
    if *affordable edges* $== 0$ **then**
        break;
    end
    if *node is accessible and unvisited* **then**
        additional rewards $+=$ reward from node;
        affordable edges $-= 1$;
    end
end

return *additional rewards*;

Algorithm 1: Affordable Untraversed Edges Heuristic

is simply the sum of the greatest m rewards. This value is clearly an upper bound, since the assumption is that the greatest possible number of unvisited nodes is visited, and that associated with this subset of nodes is the subset of rewards with the greatest sum. □

Lemma 2. *"Buying" edges from least to greatest cost results in the maximum number of edges "bought"*

Proof. Let B be the edge budget and E be a list of accessible, untraversed edges, sorted from least to greatest cost. Let m be the number of edges that can be bought before B is less than the cost of the next edge (B is decremented by the cost of the edge each time an edge is bought). To show that m is the maximum number of edges that can be bought, we must show that buying any other edge at any point, instead of the next one in sorted order, cannot possibly result in a greater number of total edges bought. Observe that since edges were bought in increasing order of cost, every remaining edge that was not purchased will have cost greater than or equal to every edge we bought. Since we stopped buying edges when the cheapest edge could not be afforded, this means buying any other edge would require us to "return" at least one of the edges we bought.

There are two possible cases. In the first case, suppose we return one edge, and are then able to afford another one we had not originally bought. In this case, we end up with the same number of total edges, m. In the second case, suppose we return an edge, and still do not have enough budget to afford another one. So, we return edges until we have enough to buy another one. However, in this case since we returned at least two edges to add only a single new edge, we will end up with at most $m - 1$ edges. Notice that it is not possible for us to return a single edge and then be able to buy more than one new one with the resulting additional budget. This is the case because we bought edges in ascending order by cost, as such, the cost of every edge we bought is less than or equal to the cost of every edge we did not buy. Therefore, greedily buying the cheapest edge until $B < min[edge \in E]$ results in the greatest total number of edges bought. □

5.5 Knapsack Problem Analogy

After studying the *AUE* heuristic, we realized that in some scenarios, we were able to derive a tighter upper bound on the additional rewards that could be achieved by drawing an analogy to the knapsack problem.

The knapsack problem is a combinatorial optimization problem. Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is maximized. It derives its name from the problem faced by someone who is constrained by a fixed-size knapsack and must fill it with the most valuable items. [KT05]

There two most popular variants of the knapsack problem are the *0/1 Knapsack Problem* and the *Fractional Knapsack Problem*. The distinction is that in the 0/1 knapsack problem the knapsack may only include complete items. In the fractional knapsack problem, fractions of items can be included. The 0/1 knapsack problem is NP-Complete. The fractional knapsack can be solved in $O(nlogn)$ time. [KT05]

The second heuristic we discuss works primarily by modeling the computation of the heuristic for the current state of the OP as a fractional knapsack problem. We consider the set of the items available for placement into our knapsack to be the set of accessible unvisited nodes. For each of these items, their associated value is the reward associated with that node. Finally, the cost, or weight, associated with each item is the minimum weight accessible incident edge. In this regard, this is similar to how the *AUE* heuristic establishes an upper bound. The cost of the minimum weight accessible edge incident to a node represents a lower bound the cost to visit that node from *any* other accessible node in the graph.

5.6 Fractional Knapsack Heuristic

The *fractional knapsack* heuristic works in two steps. We need to know the set of unvisited nodes which are accessible from the current node. This depends on the set of nodes previously

visited and on the edge budget. Since we know that the solution to the OP **must** visit the destination node, we know we will incur a cost of at least the minimum weight accessible edge into that node. Let this edge be m. Because of this, we can immediately decrement the remaining edge budget by the weight of m before proceeding to the next step of the algorithm. An important detail in this step is to check that after decrementing the edge budget, the node at the other end of m, the penultimate node in this potential solution's path, is still accessible. If this condition is not satisfied, we repeat this step, letting m be the next lowest weight edge with an endpoint in the destination node. If the destination node has not yet been visited, we add its reward to the additional rewards value, the value returned by the heuristic.

With this decremented edge budget, we can now begin the second step of the heuristic. This step is similar to the fractional knapsack problem, We sort the accessible unvisited nodes (computed in step 1) by the ratio of their rewards divided by the weight of each node's minimum weight accessible incident edge. These ratios represent that cheapest reward to cost ratios for all remaining rewards. To establish an upper bound on the additional rewards we could achieve, we "buy" the nodes' rewards at these rates until insufficient budget remains, or no nodes remain for us to buy. At each iteration, we increment the additional rewards value by the node's reward and decrement our edge budget by the associated cost (the min. weight accessible incident edge) It is important to note that we may not be able to afford the entire last node. In this case, we only increment our additional rewards value by the fraction that we can afford. This final step guarantees admissibility of the heuristic.

Determining the minimum weight accessible edge incident to each node depends on the current state of the OP, specifically the amount of edge budget remaining. Because of this, each time the heuristic is evaluated, it must first determine the minimum weight edge incident to each unvisited node and then sort the nodes according to the resulting ratios. In the worst case, each node will have an edge to every other node. Finding the minimum edge would require a linear scan of all $n-1$ edges, as such the *Fractional Knapsack* heuristic runs in $O(n^2)$, where n is the number of nodes in the graph.

Pseudocode for the heuristic is shown below:

Data: current state

Result: upper bound on additional rewards

additional rewards $\leftarrow 0$;

node set $\leftarrow$ *All unvisited nodes accessible given the current state's budget*;

$B \leftarrow$ *current state's budget*;

if *current node* $\neq$ *destination node* **then**

  if *destination node has not been visited* **then**

    additional rewards $+=$ *reward from destination node* ;

    remove destination node from node set;

  end

  $m \leftarrow$ *minimum weight accessible edge incident to destination node*;

  $B' = B-$ *weight of* m;

  penultimate node $\leftarrow$ *node at* m*'s other endpoint*;

  while *penultimate node is inaccessible given* B' **do**

    $m \leftarrow$ *next lowest weight accessible edge incident to destination node*;

    penultimate node $\leftarrow$ *node at* m*'s other endpoint*;

    $B' = B-$ *weight of* m;

  end

end

sort node set by ratio of reward to each nodes minimum weight accessible incident edge;

while $B' > 0$ *and node set is not empty* **do**

  min edge $\leftarrow$ *current node's minimum weight accessible incident edge*;

  reward ratio $\leftarrow$ *min*$(1, \frac{B'}{\text{min edge}})$;

  additional rewards $+=$ *current node's reward * reward ratio*;

  $B'-=$ *min edge*;

  remove current node from node set;

end

return *additional rewards*;

Algorithm 2: Fractional Knapsack Heuristic

5.7 Admissibility of the Fractional Knapsack Heuristic

Theorem 3. *The fractional knapsack heuristic never underestimates the sum of additional rewards that could be achieved from the current state via any valid path to the terminal node.*

Proof. We will prove admissibility of the *Fractional Knapsack* heuristic in two steps. First, we formulate the computation of the heuristic for a given state of the OP as a fractional knapsack problem and note that the greedy solution to that problem is optimal. Then, we show that the optimal solution to the fractional knapsack problem we are solving must be greater than or equal to the sum of additional rewards that could be achieved from the current state of the OP.

As we described in 5.5, the *Fractional Knapsack* heuristic works by modeling the heuristic computation of the current state of the OP as a fractional knapsack problem. The objective of the fractional knapsack problem is to select items, each of which has a weight and a value, for inclusion in a collection (the knapsack) so that the total weight of the knapsack is less than or equal to a given limit and the total value is maximized. To make this precise the fractional knapsack problem we are solving is formulated as follows:

- The set of items available for inclusion in the knapsack is the set of accessible unvisited nodes.
- The weight of each item is the weight of that node's minimum weight accessible incident edge.
- The value of each item is that node's reward.
- The knapsack weight limit is the remaining budget in the current state of the OP, minus the weight of the minimum weight accessible edge incident to the destination node (described in 5.6).

The fractional knapsack problem is solved optimally by sorting the items in descending order by the ratio of their value to their weight. Items are then added to the knapsack until its weight limit is met, or until all of the items are included. If the entire last item cannot be

included in the knapsack, only the fraction that will "fit" is added and the objective function value is incremented only by the same fraction of the item's value. Only the last node is ever fractionally included in the optimal knapsack solution. [Gol14].

The intuition for this is as follows. Imagine there is some commodity we are interested in buying. The commodity is available in fixed size lots at varying rates. We want to acquire as much of this commodity (reward) as possible given the amount of money (edge budget) we have. To accomplish this, we buy all of the commodity we can at the cheapest rates available until we run out of money, or all of the commodity has been purchased.

Let the optimal solution to fractional knapsack problem we are solving be opt_{KS}.

To show admissibility of the heuristic, we must show that the value it returns never underestimates the sum of the additional rewards that could be achieved from the current state. Let opt_{OP} be the optimal solution to the OP we are solving. Recall that the knapsack heuristic optimistically assumes that every accessible unvisited node can be visited at cost w_i, where w_i is the weight of the minimum weight accessible edge incident to node n_i. Furthermore, observe that w_i represents a lower bound on the cost of including node n_i in an OP solution. Because of this fact, the cost of opt_{OP} must be greater than or equal to the sum of the associated w_i's for each node n_i in opt_{OP}. In other words, the cost of the optimal OP solution must be greater than or equal to the cost of including the same set of nodes in the fractional knapsack problem we were solving previously. Consequently, there will be as much, or more, edge budget remaining after including the same set of nodes (the nodes in opt_{OP}) in the fractional knapsack. With this extra budget, potentially more nodes could be purchased and added to the fractional knapsack, which would result in an increase in its value. Even if insufficient budget remains to buy any entire new accessible unvisited node, the heuristic would add the fraction it could, which would still increase the value of the knapsack. Succinctly, any extra budget could only increase the value of the knapsack. If there is no extra budget, the value will not increase. Either way, it is never less than the value of opt_{OP}.

It is important to note that the set of nodes in opt_{KS} and opt_{OP} will not be the same, in

general.

Since opt_{KS} is optimal, by definition it includes the subset of nodes which results in the maximum total rewards. To make this point stronger, opt_{KS} may include a fraction of a node's reward, which further increases its value. This is not legal in the OP. As such, the value of opt_{KS} will always be greater than or equal to the value of opt_{OP}.

Because the fractional knapsack heuristic optimistically assumes nodes can be purchased at minimal cost it will never underestimate the additional sum of rewards that could be achieved from the current state. □

CHAPTER 6

Overall Algorithm

6.1 Computing Optimal Solutions

With an admissible heuristic in hand, we can now combine it with depth-first branch-and-bound to compute solutions which are guaranteed to be optimal. Since our algorithm is depth-first, it is implemented recursively. The pseudocode for the algorithm is provided in 3 and 4.

6.2 Node Ordering

By ordering the nodes based on a desirability metric, it may be possible to proceed down higher reward yielding branches of the search tree earlier. This would allow more pruning to happen earlier in the search by increasing the value of α (described in 4.1) sooner . In our approach, we expand nodes greedily by reward. To be precise, from the current node in our search tree we always expand the child nodes in descending order of reward. We can expand nodes in this order without any sorting overhead per recursive call because the nodes are pre-sorted as described in 4.2.

Of course, other node ordering schemes are possible, such as the other obvious greedy approach where we instead expand nodes in ascending order of edge weight. Most other ordering schemes would require sorting the child nodes at every recursive call, however, we found the overhead of doing so outweighed the benefit in nearly every case. In practice, we found node ordering does not have much of an effect on the total number of node expansions, and thus runtime, required to optimally solve problems.

Data: current state, source, destination, opt path, best yet

if *budget remaining* $<$ *min distance to dest node* **then**
  return

end

additional rewards $\leftarrow 0$;

if *current node is unvisited* **then**

  additional rewards $+=$ current node's reward;

end

mark current node as visited;

if *accumulated rewards* $+$ *additional rewards* $>$ *best yet* **then**

  opt path $\leftarrow$ *current path*;

  best yet $\leftarrow$ *accumulated rewards* $+$ *additional rewards*;

end

H value $\leftarrow$ *Heuristic(current state)*;

solution upper bound $\leftarrow$ *total rewards* $+$ *H value*;

if *solution upper bound* $<=$ *best yet* **then**
  return

end

for *each adjacent node* **do**

  Solve DFBNB(updated state);

end

Algorithm 3: Solve DFBNB

Data: source, destination, edge budget

opt path $\leftarrow$ [];

best yet $\leftarrow$ *sum of rewards along min cost path from initial to terminal node*;

Solve DFBNB(source, destination, opt path, opt reward);

return *opt path, best yet*

Algorithm 4: Solve

We tested this by solving identical problem instances twice. The first time we would solve using our standard approach (with $\alpha = 0$ initially) to find the optimal solution value. The second time, we would set the initial value of α to be the optimal solution value for the problem. Starting the search with the optimal solution value did not result in a significant reduction in number of node expansions. This indicates that the majority of the time is spent verifying the optimal solution, rather than finding it.

CHAPTER 7

Experimental Results

The characteristics of the graphs greatly affect the time required to optimally solve instances of the orienteering problem and there are many parameters to vary when generating graphs. The parameters we considered are listed below.

- Number of Nodes
- Branching Factor
- Mean Reward
- Reward Distribution
- Mean Edge Weight
- Edge Weight Distribution

Every graph we tested our methods on was randomly generated and is not embedded in a plane. As such, distances between nodes in our graphs do not satisfy the triangle inequality.

7.1 Comparison to Brute Force

The first set of experiments we ran was to test the performance of our heuristic search approach against a simple brute-force search. This comparison provides a baseline in terms of runtime and number of node expansions required to solve problem instances. Additionally, the brute-force search serves as a sanity check; if the solution returned by the brute-force

Edge Budget	BForce runtime (s)	*AUE* runtime (s)	BForce Node Expansions	*AUE* Node Expansions
325	0.11	0.06	7,548,921	69,586
350	0.48	0.14	31,291,460	148,171
375	1.83	0.35	127,720,954	336,983
400	7.33	0.94	524,290,665	829,418
425	29.80	2.35	2,128,027,157	2,129,324
450	121.50	5.16	8,683,104,698	4,464,869
475	492.47	12.51	35,136,414,735	10,574,550

Table 7.1: Comparison to brute-force

search matches the solution returned by our heuristic algorithms, it provides evidence that there are no bugs in our implementation.

To avoid instance specific artifacts, we generated ten random graphs using the parameters listed below. We then report the average number of node expansions and runtime required to optimally solve the instances. The tables below report this information for a range of edge budget values.

- Number of Nodes: 50
- Branching Factor: 5
- Reward Distribution

  Uniform, Min 800, Max 1200
- Edge Weight Distribution

  Uniform, Min 20, Max 40

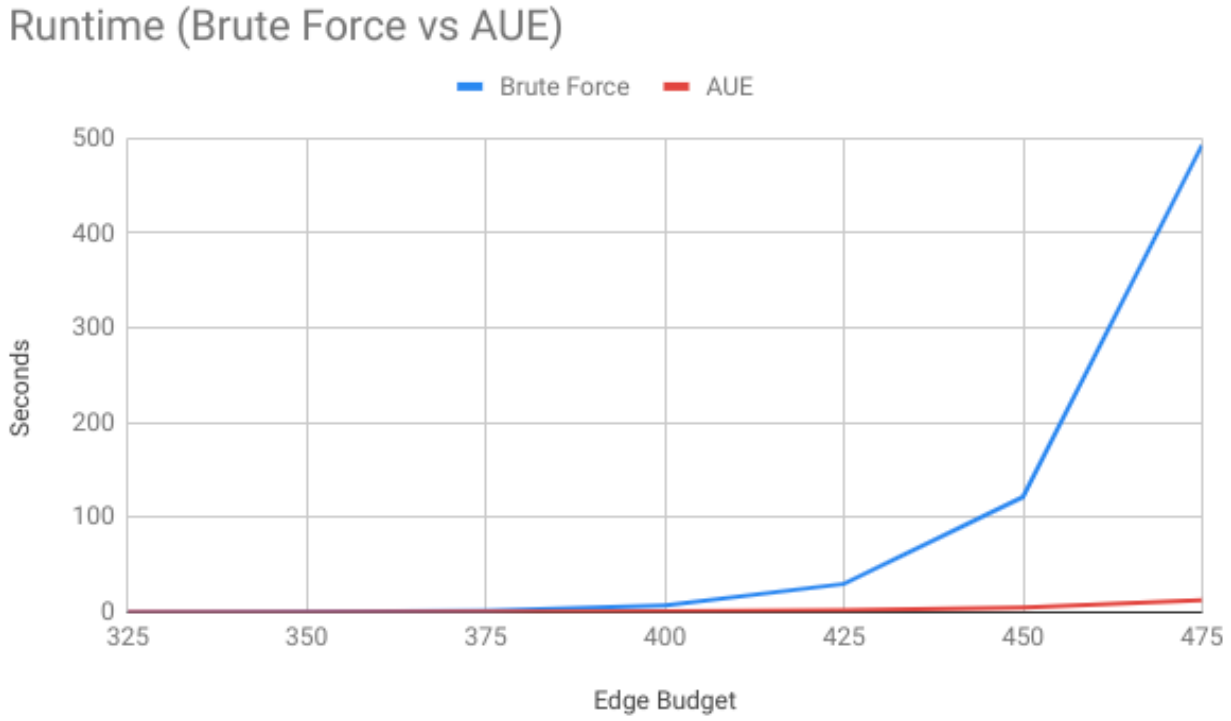

Figure 7.1: Brute force versus AUE runtime

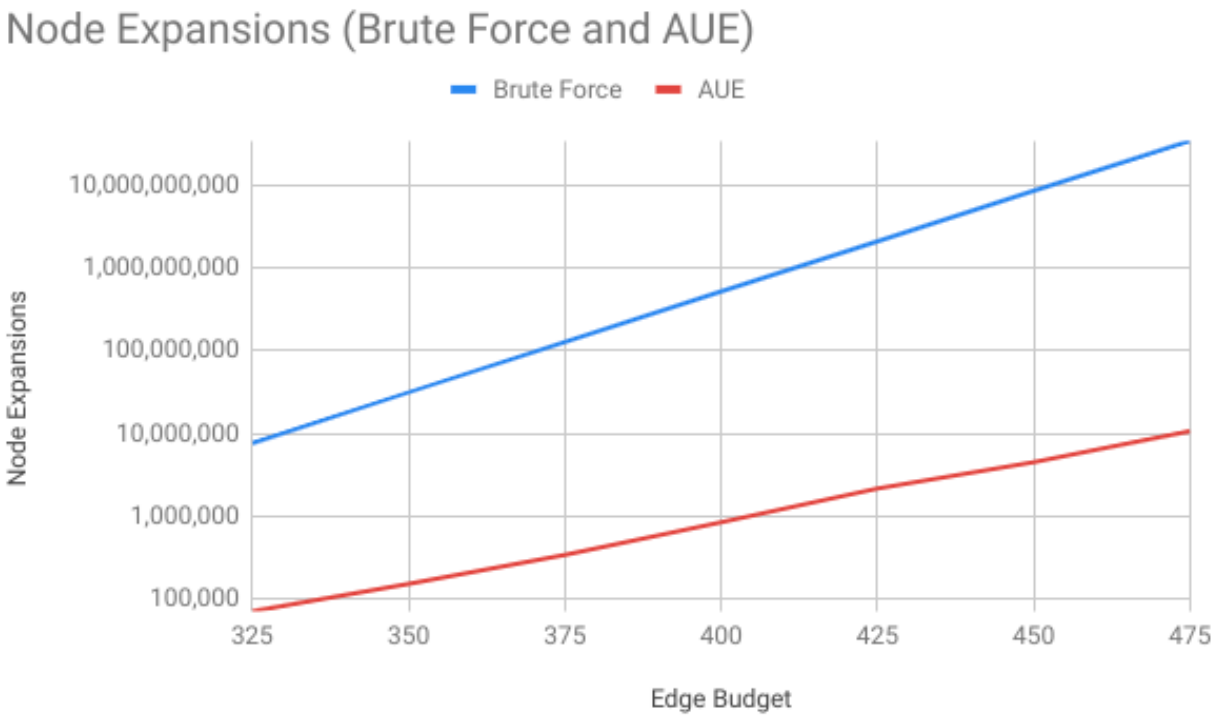

Figure 7.2: Brute force versus AUE node expansions

The plots above show the runtime and number of node expansions for both algorithms. Notice the vertical axis in plot showing node expansions is logarithmically scaled. The amount of pruning our heuristic accomplishes is evident in the exponentially smaller number of node expansions.

We limited the edge budget in our tests using the brute force algorithm because even a small increase in budget results in a huge increase in runtime required to solve the problem. To increase confidence in the correctness of our solutions, we ran the brute-force algorithm on smaller instances of every graph we test in subsequent sections. We did not publish these results, but the results are quite similar; the heuristic algorithms always run exponentially faster.

7.2 Application of AUE Heuristic to Orienteering with Category Constraints

7.2.1 Bolzoni and Helmer's heuristic

The heuristic published in [BH17] works by considering the set of accessible (via shortest path) unvisited nodes. After reducing the set of nodes in consideration, the next step in their algorithm is to sort the nodes in descending order, by reward, in each category and to sum up the greatest $(max_k - incl_k)$ rewards, where max_k and $incl_k$ represent the node limit imposed on the k^{th} category, and the number of nodes from the k^{th} category already included in the solution's path, respectively. Finally, the sum of the sums of rewards from each category is returned as the heuristic evaluation of the current state. Their heuristic can be used to solve our OP variant if we consider only a single category, and place no constraint on that category. In this scenario, it will simply return the sum of all rewards in unvisited accessible nodes.

While this heuristic is clearly admissible, the upper bound it produces is not very tight. This is because it does not take into account the maximum *number* of new nodes that could be visited from the current state. Instead, it naively assumes that all accessible nodes could be visited, and uses only the category constraints to bound this number.

To make this point clear, consider the following example. Imagine a region of the graph shaped like a wheel, where the current node is at the center of the wheel, and the spokes of the wheel are edges of weight r to nodes along the circumference. Let there be arbitrarily many nodes along the circumference. Suppose also that each of these nodes along the circumference has an edge of weight r leading to the destination node. Now, suppose the remaining budget in the current state is $2r$. By the heuristic described in [BH17], *every* node along the circumference of this wheel would be considered accessible, when in reality only a single node could be visited.

This would not be the case if instead we used our AUE heuristic. In that case, we'd see the cheapest accessible untraversed edge has weight r (in this scenario, all edges have weight

r) and would determine the greatest number of previously untraversed edges, and hence new nodes, we could include in our solution's path must be 2. The final step would then be to sum the greatest two rewards and return that as the heuristic evaluation.

7.2.2 AUE Heuristic Adaptation

With this motivating, albeit contrived, example in mind, we will now explain our proposed improvements to the heuristic used for the OP with category constraints. Begin by running the first step of the AUE heuristic on the current state to determine an upper bound on the number of new nodes that could be included in our solution's path. Let this number be m_{AUE}. Next, sort the entire set of nodes by reward in descending order. This step could be factored out of the heuristic, and only done once as an initialization step as we explained in 4.2. Finally, conduct a linear scan across this sorted list of nodes; if the current node is accessible, unvisited and the category limit for that node's category has not been reached, add its reward to a running total. Continue this process until m_{AUE} node's rewards have been added, or all the category limits have been reached. This total is then returned as the heuristic evaluation of the current state. The addition of the category constraints does not change the admissibility of the AUE heuristic.

7.2.3 AUE Heuristic Superiority

The superiority of our proposed heuristic for orienteering with category constraints comes from the lower upper bound we establish on the number of new nodes that could be included in a solution's path. With the heuristic proposed in [BH17], the upper bound on the number of new nodes is established primarily using the category constraints.

Notice that the AUE heuristic begins with the set of accessible nodes and looks for untraversed edges that could potentially lead to new nodes. Clearly an upper bound on the number of new nodes that could be included in a solution's path from any state is the number of accessible unvisited nodes. This gives us the following:

$$m_{AUE} \leq |\{accessible\ unvisited\ nodes\}|$$

Finally, both heuristics work by summing the greatest rewards from each category until some stopping condition is met. The approach in [BH17] stops after all categories are at capacity, or when there are no accessible unvisited node's rewards left to add. Our proposed approach using the AUE heuristic stops after either m_{AUE} rewards have been added, all categories are at capacity, or when there are no accessible unvisited node's rewards left to add. Since we know m_{AUE} is less than or equal to the number of remaining accessible unvisited nodes our approach could never sum more node rewards than theirs. In other words, our approach will always stop summing rewards before, or at the same time as theirs. Since the rewards are summed in descending order, the upper bound produced by our approach will never exceed theirs, and in practice will often be less. Furthermore, since the AUE heuristic's value is always less than or equal to theirs and because the heuristic value is used to determine when pruning can occur, the number of nodes expanded by a search using the AUE heuristic will always be less than or equal to the number using their heuristic.

7.2.4 Experimental Results

We ran a handful of tests on instances of our problem using our implementation of the approach described in [BH17], as well as using our approach with the AUE heuristic. The results of our tests are summarized in 7.3. The results show a search AUE using the heuristic significantly outperforms a search paired with the heuristic from [BH17]. Solving larger problem instances using their heuristic quickly becomes infeasible.

Table 7.2 describes the parameters of the graphs we used in our testing. Table 7.3 shows the average (across ten randomly generated instances) amount of time and number of node expansions required to optimally solve problems with the amount of edge budget indicated in the budget column. The graph ID column relates the two tables, indicating the set of parameters used to generate each graph.

Graph ID	Nodes	Branching Factor	Reward Distribution	Edge Weight Distribution
1	50	3	Uniform [800,1200]	Uniform [20, 40]
2	50	20	Uniform [800,1200]	Uniform [20, 40]

Table 7.2: B.H. comparison test graph characteristics

Graph ID	Budget	$Time_{AUE}$ (s)	$Nodes_{AUE}$	$Time_{B.H.}$ (s)	$Nodes_{B.H.}$	$Time_{B.H}/Time_{AUE}$	$Nodes_{B.H}/Nodes_{AUE}$
1	500	0.07	112,960	11.2	56,906,712	160.0	503.8
1	600	0.36	568,603	675.8	33,655,030,045	1,877.2	59,188.9
2	220	0.90	599,763	7.17	29,645,864	7.9	49.4
2	240	3.03	1,836,829	50.5	207,315,452	16.6	112.9

Table 7.3: Comparison to B.H. approach

CHAPTER 8

Comparison of Our Heuristics

In the following section, we will analyze the relative performance of the *AUE* and *Fractional Knapsack* heuristics when used within our depth-first branch and bound method. We conduct our tests by varying one parameter at a time in an attempt to ascertain the sensitivity of the heuristics to each parameter. Finally, we study specific instances which represent near best and worst case scenarios for both algorithms.

8.1 Test Simplifications

We held several values constant in our testing to reduce the dimension of the test space. Specifically, we fix the number of nodes, mean edge weight, and mean node reward value in all of our graphs. Below we explain and justify each of these simplifications. Additionally, branching factor is one of the parameters we vary in our testing, however, in any given graph every node will have the same number (the branching factor) of incident edges. Our algorithm does not rely on any of these simplifications.

All graphs on which we tested our methods have 50 nodes. The number of nodes in the graph becomes somewhat arbitrary after a point, as the majority in the nodes become inaccessible via shortest path at deeper search depths (where the vast majority of the time is spent) and are not considered by our heuristics. More concisely, our approach focuses only on the region of the graph that is accessible given the current edge budget. Furthermore, the greatest number of nodes included in any optimal path produced during our testing was less than 40. Computing optimal paths with more nodes requires more runtime than we allowed for in any of our tests.

The distribution of edge weights is a parameter we vary in our testing, however, we hold the mean edge weight constant at a value of 30. We make this simplification so that similar edge budget values yield similar length optimal paths. Varying this value alone affects primarily the number of nodes included in the optimal path.

By similar reasoning, we vary the distribution of node rewards, but hold the mean node reward constant at a value of 1,000. Varying this value would change only the magnitude of the optimal solution values, but has no effect on the difficulty of solving the problem.

8.2 Testing Methodology

For each set of parameters, we generated 10 random graphs. Since we are interested in the relative performance of our heuristic algorithms in large problems, we do not hold the initial edge budget constant, but instead select a value for the budget such that the longer running of the two searches takes approximately one hour to solve a single problem instance. We do this because, for example, with the same edge budget the optimal solution can be computed in a fraction of a second on a graph with branching factor 3, but can take nearly a day on a graph with branching factor 30. We select this edge budget value by solving a single instance with increasing edge budget values until the runtime required is approximately one hour. To be clear, we do not report the results of these initial program executions; they are done only to select an edge budget value. With this edge budget value selected, we then compute the optimal solution on each graph using depth-first branch and bound with both heuristics and log the the runtimes and number of node expansions.

The results we show for a given set of parameters represent the average of these values across all 10 of the random graphs.

8.3 Branching Factor

The first parameter we vary in our testing is the branching factor of our graph.

The following tests were conducted on graphs with branching factors as shown in 8.1.

B Factor	$Time_{AUE}$ (s)	$Nodes_{AUE}$	$Time_{KS}$ (s)	$Nodes_{KS}$	$Time_{KS}/Time_{AUE}$	$Nodes_{KS}/Nodes_{AUE}$
3	552.5	1,252,009,059.0	6,636.2	962,933,306.8	12.012	0.769
4	1,473.0	2,468,619,796.0	6,616.7	860,279,091.5	4.492	0.348
5	1,144.6	1,437,370,964.0	2,658.9	348,294,965.0	2.323	0.242
10	1,223.2	932,374,228.9	2,045.9	251,532,126.6	1.672	0.270
20	3,464.0	1,543,101,256.0	5,249.1	587,404,057.9	1.515	0.381
30	4,255.4	1,461,410,488.0	9,419.3	929,423,325.0	2.214	0.636
40	764.2	215,384,375.7	1,158.6	128,551,368.0	1.516	0.597
49	510.7	144,685,110.4	1,640.2	169,999,737.1	3.212	1.175

Table 8.1: Relative heuristic performance varying branching factor

The other parameters were held constant. They are listed below.

- Edge Distribution:

  Uniform, Min 20, Max 40

- Node Reward Distribution:

  Uniform, Min 800, Max 1200

The dip in the ratios shows that the Knapsack heuristic's relative performance improves as the branching factor of the graphs increases, but only up to a point. When the branching factor is high, there will be more low weight accessible edges in the accessible region of the graph. Since the *AUE* heuristic buys edges in increasing order of weight, it will buy the cheaper edges first and end with a higher total number of edges bought. This higher total number will result in more node rewards being included in the sum eventually returned as the heuristic evaluation.

Since the *Knapsack* heuristic associates edges with nodes, it cannot simply look for the cheapest subset of nodes, but instead the cheapest untraversed edge incident to each node. This stipulation generally forces the heuristic to use higher weight edges to add new node rewards, which results in a lower total value returned by the heuristic.

The reason the *Knapsack* heuristic does not maintain its relative advantage in complete and nearly complete graphs is likely because when there are many edges incident to each

Reward Range	$Time_{AUE}$ (s)	$Nodes_{AUE}$	$Time_{KS}$ (s)	$Nodes_{KS}$	$Time_{KS}/Time_{AUE}$	$Nodes_{KS}/Nodes_{AUE}$
[1, 10000]	606.0	596,087,633.7	2,271.4	232,285,033.5	3.748	0.390
[1, 2000]	751.5	673,891,570.5	1,306.8	142,376,307.2	1.739	0.211
[200, 1800]	327.9	276,099,775.2	820.7	61,231,630.2	2.503	0.222
[400, 1600]	312.7	276,846,625.3	740.2	54,732,415.9	2.367	0.198
[600, 1400]	360.6	300,429,008.9	1,080.4	80,618,878.4	2.996	0.268
[800, 1200]	1,144.6	1,437,370,964.0	2,658.9	348,294,965.0	2.323	0.242
[1000, 1000]	5.9	6,193,726.5	857.7	91,971,554.4	145.510	14.849

Table 8.2: Relative heuristic performance varying reward distribution

node, the set of minimum weight incident edges may not differ much from the set of a lowest weight edges in the graph (how the AUE heuristic works). Furthermore, the overhead of finding the minimum weight accessible incident edge for each node becomes more expensive when there are mode edges. With V nodes in the graph, finding the set of minimum edges will require $O(V * b)$ time where b is the branching factor of the graph, whereas the AUE heuristic runs in linear time. Even in cases where the search using the *Knapsack* heuristic expands half as many nodes as the AUE heuristic, the superior time complexity of the AUE heuristic still results in a faster computation of the optimal solution.

8.4 Node Reward Distribution

The next parameter we varied was the distribution of node rewards. Again, the other parameters were held constant. They are listed below.

- Edge Distribution:

  Uniform, Min 20, Max 40

- Branching Factor: 5

The node rewards were randomly selected from a uniform distribution. 8.2 lists the minimum and maximum values for each distribution.

The results show that relative performance of our heuristics is not very sensitive to changes to the reward distribution, with one obvious exception. When we fixed the node

rewards to all be the same value, the *AUE* heuristic vastly outperformed the *Knapsack* heuristic, running on average 156 times faster. This result is not surprising; the *Knapsack* heuristic works by associating node rewards with minimum incident edge weights, whereas the *AUE* heuristic does not associate node rewards with edge weights. In this case, since all of the node rewards are identical, there is no information to be gained by associating edges with nodes. Furthermore, this situation is very favorable for the *AUE* heuristic. Recall that it returns the sum of the m greatest rewards, where m is the most nodes that could be added to the solution's path. The *AUE* heuristic will overestimate badly when few of these high reward nodes can be added to the solution's path, however, since the rewards in this situation are identical, this source of potential overestimation is eliminated.

8.5 Edge Weight Distribution

The next parameter we varied in our testing was the distribution of edge weights. As with the previous tests, we held the other parameters constant. They are listed below.

- Node Reward Distribution:

  Uniform, Min 800, Max 1200

- Branching Factor: 5

The edge weights were randomly selected from a uniform distribution. 8.3 lists the minimum and maximum values for each distribution.

We've found that the relative performance of our two heuristics is sensitive to the distribution of edge weights. The search paired with the *AUE* heuristic was faster in all of our tests, but as the edge weight distribution became very narrow its advantage increased dramatically. Drawing weights from the distribution from 28 to 32 the search with the *AUE* heuristic was over one hundred times faster. In the extreme, when all edges were assigned the same weight, it ran well over one thousand times faster.

The reason for this behavior is likely the same as why the *AUE* heuristic was vastly

Edge Range	$Time_{AUE}$ (s)	$Nodes_{AUE}$	$Time_{KS}$ (s)	$Nodes_{KS}$	$Time_{KS}/Time_{AUE}$	$Nodes_{KS}/Nodes_{AUE}$
[1, 1,000]	153.7	822,129,853.0	730.4	137,148,747.5	4.752	0.167
[1, 60]	9.8	29,040,007.0	139.4	20,496,780.3	14.276	0.706
[5, 55]	499.5	674,559,401.8	2,243.3	289,462,479.0	4.491	0.429
[10, 50]	205.7	237,503,824.7	1,222.6	142,671,263.5	5.944	0.601
[15, 45]	18.3	19,599,805.7	68.4	8,740,700.5	3.733	0.446
[20, 40]	1,144.6	1,437,370,964.0	2,658.9	348,294,965.0	2.323	0.242
[25, 35]	50.5	41,803,333.7	361.3	27,593,810.8	7.156	0.660
[28, 32]	6.2	5,865,794.1	430.2	32,739,926.9	69.125	5.581
[30, 30]	3.9	5,685,917.3	3,014.3	444,018,632.7	763.927	78.091

Table 8.3: Relative heuristic performance varying edge weight distribution

superior when all nodes were assigned the same reward. When all edges have the same weight, no information is gained by associating node rewards with edge weights. This is situation is also quite favorable for the *AUE* heuristic. Recall that it determines an upper bound on the number of additional nodes that could potentially be added to the solution's path by buying the cheapest (accessible) edges until insufficient budget remains to buy the next edge. It will overestimate especially badly when many of these cheap accessible edges do not actually result in the addition of new rewards to the solution's path. In this case, since all of the edge weights are identical, this potential source of overestimation is eliminated.

8.6 Edge Budget

The final parameters we varied in our testing was the edge budget. The goal of our test was to ascertain the sensitivity of the relative performance of our heuristics to the edge budget value. In general, a greater edge budget value will result in a deeper, and longer running search. We conducted this test by optimally solving problem instances (using both heuristics) with increasing budget values on ten randomly generated graphs using the same set of parameters. As before, the values reported for each budget value represent the mean runtime ratio across all ten graphs.

The set of parameters we used are listed below.

Budget	$Time_{AUE}$ (s)	$Nodes_{AUE}$	$Time_{KS}$ (s)	$Nodes_{KS}$	$Time_{KS}/Time_{AUE}$	$Nodes_{KS}/Nodes_{AUE}$
325	4.7	2,372,594.1	18.1	1,394,437.1	3.817	0.588
350	12.5	6,853,050.3	43.7	3,675,615.4	3.508	0.536
375	30.6	18,605,381.6	86.6	8,941,427.1	2.825	0.481
400	78.1	51,004,397.3	184.5	20,944,496.7	2.361	0.411
425	202.1	136,151,389.1	426.4	49,628,013.2	2.110	0.365
450	529.6	392,782,275.1	982.5	121,648,520.0	1.855	0.310
475	1,223.2	932,374,228.9	2,045.9	251,532,126.6	1.672	0.270

Table 8.4: Relative heuristic performance varying edge budget

- Edge Weight Distribution:

  Uniform, Min 20, Max 40

- Node Reward Distribution:

  Uniform, Min 800, Max 1200

- Branching Factor: 10

We see in table 8.4 that the relative performance of the search paired with the *Knapsack* heuristic improves with increased edge budget, however it begins to level off at approximately a factor of 1.6 in relative runtime. We've observed cases where the search with the *Knapsack* heuristic is up to 25% faster than the AUE heuristic, however, all of these searches were run for at least an entire day. It is possible the *Knapsack* heuristic may outperform the AUE in very long running problems, but we did not run enough sufficiently large problem instances to know whether this behavior was instance specific, or in fact a general trend.

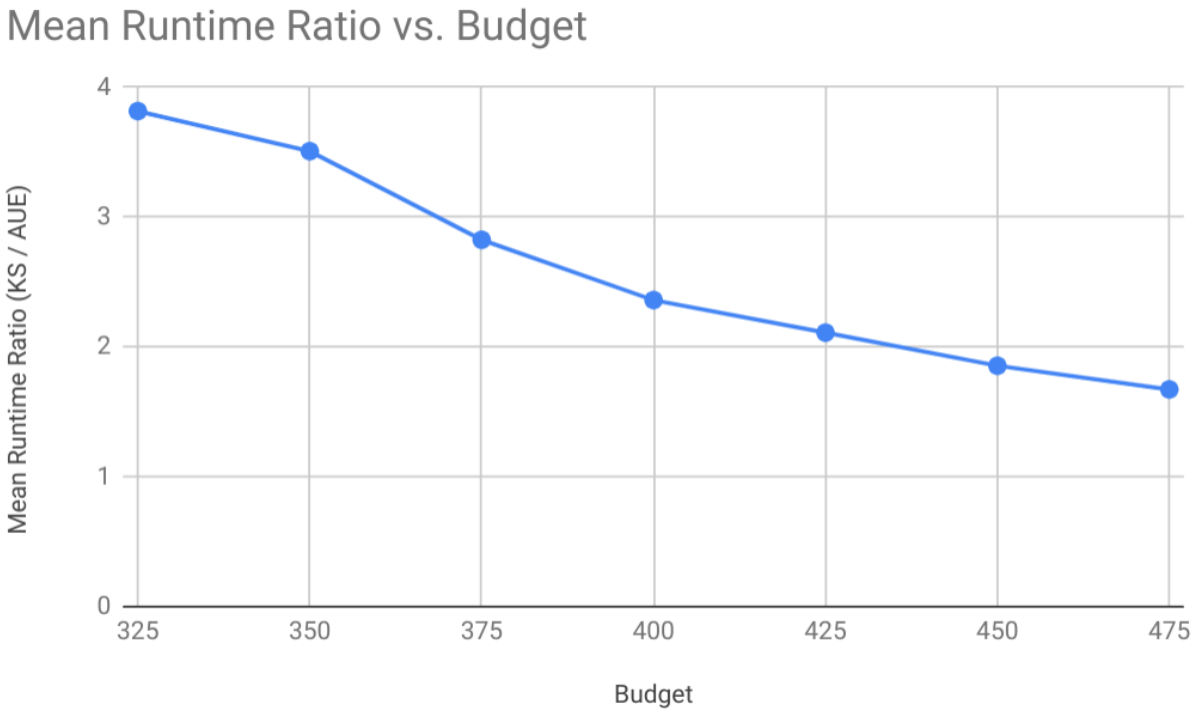

Figure 8.1: Ratio of runtimes of heuristics varying edge budget

CHAPTER 9

Special Cases

9.1 AUE Heuristic Best Case Scenario

In the analysis of the results from the tests on varying the distribution of node rewards and edge weights, we saw the *AUE* had its best relative performance when we had constant node rewards and edge weights. Combining these two cases on a complete graph represents the best case scenario for the *AUE* heuristic. Notice that since every node has the same reward, and can be accessed from every other node at the same cost, the *AUE* heuristic is actually a perfect heuristic on this graph. That is, the value it returns is always exactly the additional rewards that can, and will be, achieved from the current state. In this case, the search with the *AUE* heuristic is many orders of magnitude faster than with the *Knapsack* heuristic.

9.2 AUE Heuristic Worst Case Scenario

One of the theoretical worst case scenarios for the *AUE* heuristic is nearly the opposite of its best case scenario. Specifically, a graph with a wide distribution of both node rewards and edge weights and a relatively small branching factor could often cause the *AUE* heuristic to overestimate severely. This will occur when the heuristic sees many very low weight edges and assumes it can add several highest reward nodes to the path, when in reality traversing high weight edges is necessary to add reach new nodes. By associating edge weights with node rewards, the *Knapsack* heuristic should not overestimate as severely.

We generated ten random instances with the parameters described above, however in our tests we still saw the search with the *AUE* heuristic solve problems several times faster.

The heuristic behavior described above is likely very instance specific, thus it is not likely for such a graph to be generated specifically. Furthermore, we saw previously that the relative performance of the *Knapsack* is often best in very long (multiple day) searches. To summarize, we believe this represents one of the theoretical worst cases, but we do not have sufficient data to support this claim due to the amount of time required to generate the data.

CHAPTER 10

Summary of Results

In the set of experiments we ran, we did not find a set of parameters where the *Knapsack* heuristic outperformed the AUE heuristic on average in terms of runtime. However, in nearly every situation the search using the *Knapsack* heuristic expanded fewer nodes.

We believe the reason the *Knapsack* heuristic expands more nodes than the AUE heuristic in certain cases is due to the fact that the final value it adds to the upper bound it returns can represent only part of a node's reward. Consider the case where insufficient edge budget remains to afford a single untraversed accessible edge in the graph. Let this edge be e. In this case, the AUE will return 0. However, the *Knapsack* heuristic will return $\frac{B}{e}$ times the reward in one of the nodes this edge e is incident to, where B is the remaining edge budget. Detecting this case involves running the majority of the logic of the AUE heuristic before running the *Knapsack* heuristic. In practice, we found it was still faster just to run the search using the AUE heuristic instead of running both and taking the minimum value returned, even though more nodes are expanded.

We saw that the *Knapsack* heuristic's relative performance improved in larger searches, and even observed a handful of cases where it outperformed the AUE heuristic in terms of runtime. Unfortunately, all of the cases where it outperformed the AUE heuristic required more than a day of runtime. Because of this, we do not have enough data to report average case performance on such large searches.

CHAPTER 11

Future Work

11.1 Heuristic Improvement

The greatest source potential improvement in our approach would be through improving our heuristics. A search with the *Knapsack* heuristic generally expands fewer nodes than one with the AUE heuristic. Optimizing it is an obvious avenue for improvement, however its current implementation is the most efficient one we have discovered thus far.

The two heuristics described in this thesis represent the best approaches we have realized to date. Of course, it is possible there are other admissible heuristics that would produce lower upper bounds. In practice, the effectiveness of a heuristic depends on how expensive it is to compute versus the tightness of the upper bound it produces. An admissible heuristic with a favorable tradeoff between the two could offer a potentially huge speedup.

11.2 Parallelization

As with many problems that can be solved using branch and bound, computing optimal solutions to the OP is highly parallelizable. By using an efficient parallel search algorithm, such as distributed tree search, which is designed for effective load balancing on irregular trees, there should be a nearly linear speedup by using multiple processors. [FK88] We did not implement parallelization, but the effectiveness of our heuristics should not be affected by it. While parallelization will only offer a linear speedup, as opposed to an exponential improvement possible via a superior heuristic, it is still a good option to consider for especially large problems.

11.3 Approximation

11.3.1 Weighted Search

Our approach could easily be modified to be an approximation algorithm by allowing pruning to happen sooner. The idea is similar to a weighted A* search. This would sacrifice the optimality guarantee, but the solution quality could be bounded depending on the weighting factor. [Kor13]

11.3.2 Anytime Property of Branch and Bound

Our approach could easily be adapted to an approximation algorithm by taking advantage of the anytime property of the branch and bound algorithm. The anytime property means that we can stop our branch and bound search at any time and we will have *a solution*, albeit not necessarily an optimal one. This would require virtually no modification of our code. We would simply run the algorithm for as much time as is available and then report best solution found so far when the program is stopped. [Kor13]

CHAPTER 12

Conclusion

The focus of this thesis was to discover a more efficient way to optimally solve this generalized variant of the orienteering problem. With the methods described, using depth-first branch and bound along with either of our admissible heuristics it is now possible optimally solve instances of the OP in exponentially less time than possible with a simple brute-force search, or methods using a less sophisticated heuristic evaluation function. With this approach it is possible to compute optimal solutions to instances of the OP on graphs with up to fifty nodes in hours or days, as opposed to months or years.

REFERENCES

[BH17] Paolo Bolzoni and Sven Helmer. "Hybrid Best-First Greedy Search for Orienteering with Category Constraints." In *International Symposium on Spatial and Temporal Databases*, pp. 24–42. Springer, 2017.

[CMS14] Vicente Campos, Rafael Martí, Jesús Sánchez-Oro, and Abraham Duarte. "GRASP with path relinking for the orienteering problem." *Journal of the Operational Research Society*, **65**(12):1800–1813, 2014.

[EGV14] Lanah Evers, Kristiaan Glorie, Suzanne Van Der Ster, Ana Isabel Barros, and Herman Monsuur. "A two-stage approach to the orienteering problem with stochastic weights." *Computers & Operations Research*, **43**:248–260, 2014.

[FGT98] Matteo Fischetti, Juan Jose Salazar Gonzalez, and Paolo Toth. "Solving the orienteering problem through branch-and-cut." *INFORMS Journal on Computing*, **10**(2):133–148, 1998.

[FK88] Chris Ferguson and Richard E Korf. "Distributed Tree Search and Its Application to Alpha-Beta Pruning." In *AAAI*, volume 88, p. 128, 1988.

[GAL84] Bruce Golden, Arjang Assad, Larry Levy, and Filip Gheysens. "The fleet size and mix vehicle routing problem." *Computers & Operations Research*, **11**(1):49–66, 1984.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.

[GLV87] Bruce L Golden, Larry Levy, and Rakesh Vohra. "The orienteering problem." *Naval Research Logistics (NRL)*, **34**(3):307–318, 1987.

[GLV16] Aldy Gunawan, Hoong Chuin Lau, and Pieter Vansteenwegen. "Orienteering problem: A survey of recent variants, solution approaches and applications." *European Journal of Operational Research*, **255**(2):315–332, 2016.

[Gol14] Mordecai Golin. "Greedy Algorithms: The Fractional Knapsack, Lecture Notes Design and Analysis of Algorithms.", November 2014.

[IID08] Taylan Ilhan, Seyed MR Iravani, and Mark S Daskin. "The orienteering problem with stochastic profits." *Iie Transactions*, **40**(4):406–421, 2008.

[KML18] Gorka Kobeaga, María Merino, and Jose A Lozano. "An efficient evolutionary algorithm for the orienteering problem." *Computers & Operations Research*, **90**:42–59, 2018.

[Kor13] Richard Korf. "Heuristic Search." Course notes for CS261A (Problem Solving and Search), UCLA Computer Science Department, 3 2013.

[KT05] Jon Kleinberg and Eva Tardos. *Algorithm Design*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005.

[MSL16] Yi Mei, Flora D Salim, and Xiaodong Li. "Efficient meta-heuristics for the multi-objective time-dependent orienteering problem." *European Journal of Operational Research*, **254**(2):443–457, 2016.

[PSL17] Pamela J Palomo-Martínez, M Angélica Salazar-Aguilar, Gilbert Laporte, and André Langevin. "A hybrid variable neighborhood search for the orienteering problem with mandatory visits and exclusionary constraints." *Computers & Operations Research*, **78**:408–419, 2017.

[RB91] Ram Ramesh and Kathleen M Brown. "An efficient four-phase heuristic for the generalized orienteering problem." *Computers & Operations Research*, **18**(2):151–165, 1991.

[SG10] John Silberholz and Bruce Golden. "The effective application of a new approach to the generalized orienteering problem." *Journal of Heuristics*, **16**(3):393–415, 2010.

[Tsi84] Theodore Tsiligirides. "Heuristic methods applied to orienteering." *Journal of the Operational Research Society*, **35**(9):797–809, 1984.