Quantization in Suboptimal Heuristic Search

BY

Brian Cawley

DISSERTATION

Submitted to the University of New Hampshire
in Partial Fulfillment of
the Requirements for the Degree of

Bachelor of Science

in

Computer Science

May, 2019

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Quantization in Suboptimal Heuristic Search

by

Brian Cawley

University of New Hampshire, May, 2019

Loosely inspired by radix sort and scaling algorithms, we propose three simple new algorithms for suboptimal heuristic search. Whereas previously-proposed methods exhibit behavior lying between A* and greedy search (best-first on cost-to-go), our methods exhibit behavior lying between A* and speedy search (best-first on search distance-to-go) or between greedy search and speedy search. We also present a bounded suboptimal variant. We study these methods theoretically and experimentally, finding that they exhibit promising performance in certain situations, making them useful tools in the growing arsenal of suboptimal heuristic search.

CHAPTER 1

Introduction to Search

1.1 Search

Searching is a common technique used to solve problems in artificial intelligence. Applications of searching include planning, scheduling, routing, and sequence alignment.

The sliding tile puzzle, shown in Figure 1-1, is one domain in which searching can be applied. The sliding tile puzzle is grid of numbered tiles with one empty space into which the tiles can slide. The goal of the puzzle is to rearrange the tiles such that the numbers are numerically ordered. To solve a problem by searching, a search tree is built to represent the chains of possible moves that can be made. The beginning of the search tree for this example of the sliding tile puzzle can be found in Figure 1-2. Stemming from the initial node, there are four possible moves that can be made, resulting in four new nodes. The search tree can then expand from these nodes until a goal node is found.

The general maintenance for searching algorithms includes an open list and a closed list. The open list contains all nodes that have not yet been expanded, meaning that the children of these nodes have not yet been looked at. In terms of the search tree, the open

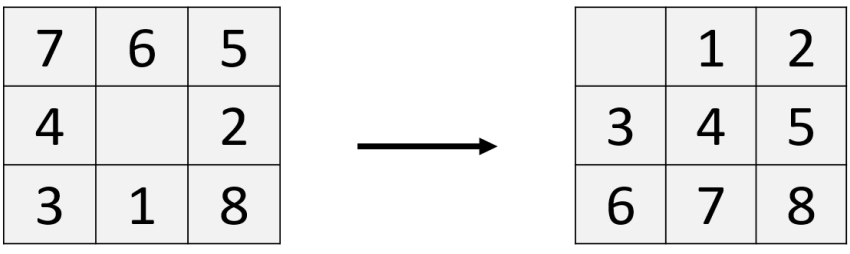

Figure 1-1: Sliding tile puzzle with some initial state (left) and goal state (right).

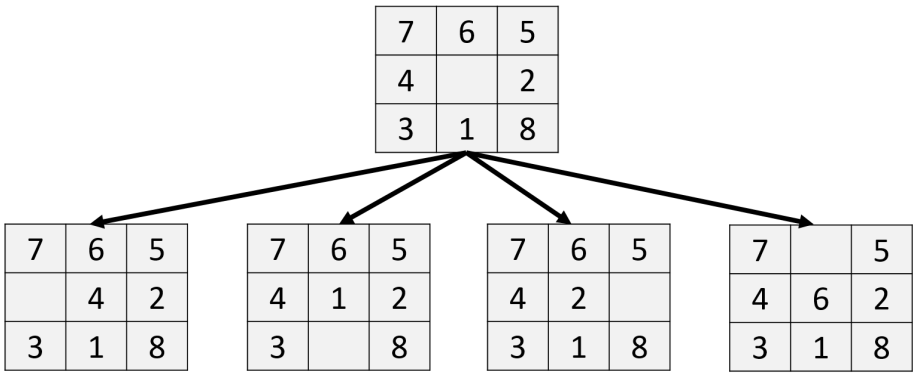

Figure 1-2: Beginning of the search tree for the sliding tile puzzle.

list is made up of all nodes on the frontier (the leaves of the search tree). One way in which search algorithms differ is in the criteria that is used to determine which of all the nodes in the open list should be chosen to be expand next. The closed list is reserved to contain all nodes that have already been generated. The purpose of maintaining the closed list is to avoid duplicates states so that unnecessary work is not performed during the search.

Searching algorithms can be divided into two families: uninformed searching algorithms and heuristic searching algorithms. Uninformed searching algorithms include depth-first search, breadth-first search, and Dijkstra's algorithm (Dijkstra, 1959). Taking Figure 1-2 as an example, one could continue to expand the search tree until the the entire search space is exhausted, such as in breadth-first search. However, this is far too costly for large problems. Thus, people often use more intelligent strategies for expanding the search tree to find a goal. Heuristic searching algorithms such as A* search (Hart, Nilsson, & Raphael, 1968) and Greedy search use domain-dependent knowledge in order to better evaluate which nodes may lead to a goal fastest.

1.2 Contributions

The contributions of this thesis begin with an analysis of a scaling version of Dijkstra's algorithm introduced by Gabow (Gabow, 1983). This scaling algorithm has a theoretical

smaller upper bound of time complexity in comparison to Dijkstra's algorithm. We give a further analysis of Gabow's algorithm and also provide a few additional enhancements.

The rest of the thesis introduces three new suboptimal searching algorithms, one of which has a bounded suboptimality guarantee. The first algorithm introduced is Quantized Edge Cost Greedy (QEG). QEG is designed to yield performance between the two extremes of Speedy and Greedy. It works by limiting the number of distinct edge costs allowed in the search, preserving some separation between cheap and expensive edge costs while grouping similarly promising nodes. Next, we introduce f-Bucket-Speedy (FBS). FBS groups nodes that have similar f values and chooses between them using d. FBS essentially uses a Speedy search on a subset of the nodes on the open list that contains the most promising nodes according to f in a manner reminiscent of A^*_ϵ(Pearl & Kim, 1982). Finally, we introduce a bounded suboptimal variant of FBS, named Bounded f-Bucket Speedy (BFBS). We evaluate these algorithms through a set of experiments using common benchmark domains. Overall, this work provides us with a better understanding of how f and d can be used in tandem to speed up a search.

Part I

Uninformed Search

CHAPTER 2

Uninformed Searching by Scaling

2.1 Introduction

Dijkstra's algorithm (pseudocode in Figure 2-1) is a best-first search algorithm that finds shortest paths in graphs from a single source node. The algorithm maintains an 'open list' (typically backed by a binary heap) of unexpanded nodes ordered by the cost expended to reach each respective node. In other words, the nodes are sorted by "cost so far" $g(n)$. The node with the smallest $g(n)$ is selected for expansion until all nodes have been expanded. The process of expanding a node n consists of removing n from the open list and updating $g(m)$ for each neighbor m if a better path to m has been found through n. The search is terminated once a goal node has been selected for expansion. Overall, Dijkstra's algorithm achieves a time bound of $O(E \log V)$.

In this chapter, we analyze a scaling version of Dijkstra's algorithm (Dijkstra, 1959) introduced by Gabow (Gabow, 1983). In general, scaling algorithms solve problems by first considering only the most significant bit of each input value, finding a solution, and iteratively adding bits of precision to get a more refined solution on each iteration. Radix sorting, in which a small compact range of values allows for an $O(n)$ sort instead of $O(n \log n)$, is an example. In this chapter, we will be exploring the use of scaling algorithms in shortest path problems in graphs. [1]

One bottleneck for the running time of Dijkstra's algorithm is the use of a binary heap to maintain the open list of unexplored nodes. Namely, the decrease key operation of a

[1]Many thanks to Rob Holte (University of Alberta) and Wheeler Ruml (University of New Hampshire) for suggesting this direction.

```
Dijkstra(V, E)
1. g(V_s) ← 0
2. for v ∈ V
3.     g(v) ← ∞
4. heap-insert(V_s)
5. while not heap-empty()
6.     n ← argmin_{n∈open} g(n)
7.     if n is goal
8.         return
9.     for (n, child) ∈ E
10.        if g(child) > g(n) + w(n, child)
11.            g(child) ← g(n) + w(n, child)
12.            if heap-contains(child)
13.                heap-decrease-key(child)
14.            else
15.                heap-insert(child)
```

Figure 2-1: Pseudocode for Dijkstra's Algorithm

binary heap is $O(\log V)$. Gabow's algorithm works by scaling down edge weights in such a way that every vertex fits in a bucket list priority queue of size $|E|$, resulting in a decrease key operation of constant time. The scaling version achieves an overall time bound of $O(E \log M)$ where M is the weight of the highest weighted edge.

2.2 Gabow's Algorithm

Gabow's algorithm is used to find a shortest path from a start node to a goal node in a graph containing only nonnegative edge weights (a limitation shared with Dijkstra's algorithm). Like Dijkstra's algorithm, Gabow's algorithm can also be used to find shortest paths from some start node to every other node. For the sake of this analysis, we are most interested in the single-source, single-goal variation as it is more pertinent to later research in this dissertation.

In order to understand Gabow's algorithm, a 'near-optimum' solution to the shortest path problem is first defined. A near-optimum solution to the shortest path problem is said to adhere to the following three conditions. (i) $g(s) = 0$ where s is the start node. (ii) For every edge (i,j) in the graph, $g(i) + w_{ij} \geq g(j)$. (iii) Every node i in the graph has a shortest path from the start node that costs between $g(i)$ and $g(i) + |E|$.

Pseudocode for Gabow's algorithm can be found in Figure 2-2. The algorithm works by scaling down the edge weights in the graph by 2 until the most expensive edge weight M is no larger than $|E|/|V|$ (lines 1-2) (note that in the pseudocode, the parameter s represents the integer that edge weights are scaled by in a given layer). This creates at most $\log M$ scales of the graph G. Starting from the most scaled version of G, the algorithm then propagates up through the layers of scaled graphs, performing a conversion procedure (a procedure that converts a near-optimum solution to an optimal one) on each layer (line 3).

Initially, $g(i)$ is set to 0 for all $i \in V$. Note that this results in a near-optimum solution to the shortest path problem for graph G. To see this, first it is obvious that condition (i) is met. Condition (ii) is met because the weights of all edges are required to be non-negative else the shortest path problem could not be solved using Dijkstra's. Finally condition (iii)

Gabow(V, E, s)

1. if $M > |E|/|V|$
2. Gabow$(V, E, s \cdot 2)$
3. Conversion-Procedure(V, E, s)

Conversion-Procedure(V, E, s)

4. for $(i, j) \in E$
5. $\overline{w}_{ij} \leftarrow g(i) + (w_{ij}/s) - g(j)$
6. Dijkstra(V,E) ▷ Using modified edge weights $\overline{w}$
7. for $i \in |V|$
8. $g(i) \leftarrow g(i) * 2$

Figure 2-2: Pseudocode for Gabow's Algorithm

is met because in order for there to be any shortest path in G, $|E| > 0$.

The conversion procedure converts a near-optimum solution to an optimum solution for a given graph G. The procedure begins by calculating modified edge weights $\overline{w}$ from the original edge weights (non-scaled edge weights from G) w for every edge. For an edge from i to j, $\overline{w}_{ij} = g(i) + w_{ij} - g(j)$ (lines 4-5). Then, use Dijkstra's algorithm with a bucket list priority queue to find shortest paths from the start node to every other node (line 6). This results in g values representing the optimal solution for graph G. Finally, to achieve a near-optimum solution for the less-scaled graph above G, multiply $g(i)$ by 2 for each node i lines (7-8).

Dijkstra's algorithm in the conversion procedure uses a bucket list priority queue of size $|E|$ to represent the open list. The bucket list is an array-backed data structure where the bucket at index i is a doubly linked list containing all nodes n that currently have $g(n) = i$. Initially, the start node is placed in bucket 0 and every other node is placed in bucket $|E|-1$. When updating the cost so far of a node, simply remove the node from its current linked list and place the node at the front of the linked list at the new index. Since the decrease key operation is constant time, Dijkstra's algorithm runs in $O(E)$ using the bucket list priority queue.

2.2.1 Scaling Factor

In graphs where $|E| > |V|$, the scaling factor can be increased to improve performance. Instead of scaling edge weights by a factor of 2, use a refined version that scales edges weights by $2 + |E|/|V|$. Then in the conversion procedure, for every node i multiply $g(i)$ by this new scaling factor. This results in $\log_{2+|E|/|V|} M$ total scales, slightly reducing the running time of the algorithm.

2.3 Analysis of Gabow's Algorithm

For a graph G, let $\overline{G}$ be the once-scaled version of G. The only difference between G and $\overline{G}$ is in the edge weights. G uses edge weights w, where w is the original weight of the graph.

$\overline{G}$ uses modified edge weights $\overline{w}$. Let $\overline{w}_{ij}$ be the modified weight of the edge between i and j where $\overline{w}_{ij} = g(i) + w_{ij} - g(j)$. Define P_{st} to be the list of vertices on the shortest path from s to t and let $c_w(P_{st})$ be the cost of P_{st} using edge weights w.

2.3.1 Correctness

First, we prove that Gabow's algorithm is correct by showing that the modified edge weights do not change shortest paths.

Theorem 1 *All paths from s to t where t is a goal node in $\overline{G}$ are shortened by $g(t)$, which does not change shortest paths to goals.*

Proof: Consider the shortest path $P_{st} = \langle v_1, ..., v_k, ..., v_n \rangle$ in graph G where $v_1 = s$ and $v_n = t$ that uses the original edge weights w. Then $c_w(P_{st}) = w_{v_1,v_2} + ... + w_{v_{k-1},v_k} + w_{v_k,v_{k+1}} + ... + w_{v_{n-1},v_n}$. Now compute $c_{\overline{w}}(P_{st})$, the cost of the same path using the modified edge weights:

$$
\begin{aligned}
c_{\overline{w}}(P_{st}) &= \overline{w}_{v_1,v_2} + ... + \overline{w}_{v_{k-1},v_k} + \overline{w}_{v_k,v_{k+1}} + ... + \overline{w}_{v_{n-1},v_n} \\
&= (g(v_1) + w_{v_1,v_2} - g(v_2)) + ... + \\
&\quad (g(v_{k-1}) + w_{v_{k-1},v_k} - g(v_k)) + (g(v_k) + w_{v_k,v_{k+1}} - g(v_{k+1})) + ... + \\
&\quad (g(v_{n-1}) + w_{v_{n-1},v_n} - g(v_n)) \\
&= g(v_1) + w_{v_1,v_2} + ... + w_{v_{k-1},v_k} + w_{v_k,v_{k+1}} + ... + w_{v_{n-1},v_n} - g(v_n) \\
&= w_{v_1,v_2} + ... + w_{v_k-1,v_k} + w_{v_k,v_k+1} + ... + w_{v_n-1,n} - g(v_n)
\end{aligned}
$$

□

Since the shortest paths to the goals do not change when using modified edge weights, the set of g values representing the optimal solution for a graph $\overline{G}$ becomes near-optimum for G when these g values are multiplied by the scaling factor.

2.3.2 Early Termination

In the next two sections, we introduce optimizations for Gabow's algorithm to improve the running time. We begin with the termination policy. In finding a shortest path using Dijkstra's algorithm, one can terminate the search once a goal node is selected for expansion. In Gabow's algorithm, the search can also be terminated early; however, the condition under which it is stopped is different.

On the final iteration of the conversion procedure, the search can terminate once a goal node is found, similar to Dijkstra's algorithm. For every other call to the conversion procedure, do the following. First, we compute an upper bound u. This is done by using infinity if this is the first call to Dijkstra's, otherwise the shortest path found in the scaled graph $\overline{G}$ and set u to be the cost of this path using the edge weights in G. Then, when finding a shortest path in G, we can stop when the index of the current bucket in the bucket list is greater than or equal to u. Now we prove that this modification preserves correctness.

Theorem 2 *The cost of the shortest path in G is less than the upper bound u that is computed using the shortest path found in $\overline{G}$*

Proof: Let $\overline{P}_{ij}$ be the shortest path from i to j in graph $\overline{G}$. Now, let $u = c_w(\overline{P}_{ij})$. Note that $\overline{P}_{ij}$ is feasible in graph G since the two graphs are the same except for different costs. $\overline{P}_{ij}$ may be the shortest path in G or it could be longer. So if P_{ij} is the actual shortest path in G, then $c_w(P_{ij}) \leq u$. □

This ensures that any node that may be on the shortest path in G is considered before Dijkstra's algorithm is terminated early because of the upper bound condition. More specifically: following Theorem 2.3.2, for any node n that may be on the shortest path in graph G, its $g(n)$ will be updated before the upper bound u is reached in Dijkstra's algorithm on graph $\overline{G}$.

2.3.3 Smart Expansion

The upper bound u detailed in the previous section can also be leveraged during the process of expanding a node. Recall that in Dijkstra's algorithm, g values are initialized to $E-1$ and are then lowered. When updating the the distance of a node's successors, only update the old distance if the new distance is less than the upper bound. If the successor happens to be on the shortest path, it will be updated at a later time as the successor of a different parent node. In order to prove that these expansion are unnecessary, we first prove the following:

Theorem 3 *The cost of a path P is no less than the cost to reach any node on P.*

Proof: Let P_{ij} be the shortest path from i to j. Recall that Dijkstra's algorithm requires a graph that contains only nonnegative edge weights, so g values are nondecreasing through each node of a path. Since there exists a path P_{ik} for each node k in P_{ij}, it follows that $P_{ik} \leq P_{ij}$. □

Theorem 4 *A node n in G that has $g(n)$ less than u can not be on the shortest path in G.*

Proof: Let $\overline{P}_{ij}$ be the shortest path from i to j in graph $\overline{G}$. Now let u $= c_w(\overline{P}_{ij})$ and P_{ij} be the shortest path in G. For the sake of contradiction let node k be on P_{ij} where $g(k) > u$. Recall from Theorem 2 that $u \geq c_w(P_{ij})$. It follows that $g(k) \geq c_w(P_{ij})$. This contradicts Theorem 3 thus k cannot be in P_{ij}. □

2.4 Experimental Results

Gabow's algorithm was tested against Dijkstra's algorithm in graphs that represent road networks in the United States. These graphs were used as a benchmark in the 9^{th} Implementation DIMACS Challenge (in semi-admissible heuristics, 2005). The vertices represent various locations across the United States and the edges that connect them represent the distance between the two cities. The number of vertices range from 2.64×10^5 to 2.39×10^7

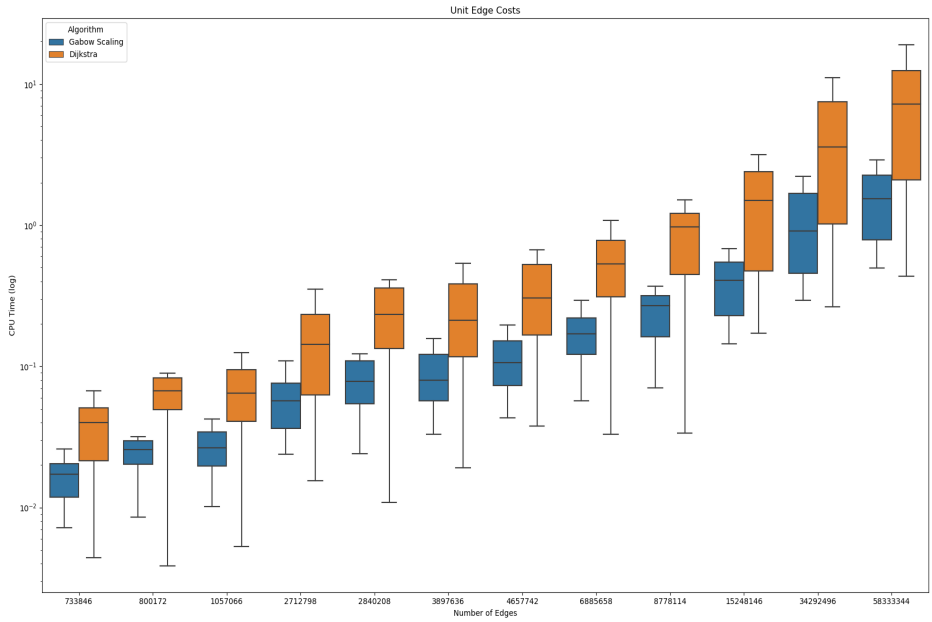

Figure 2-3: CPU time of Dijkstra vs. Gabow for unit cost edge weights.

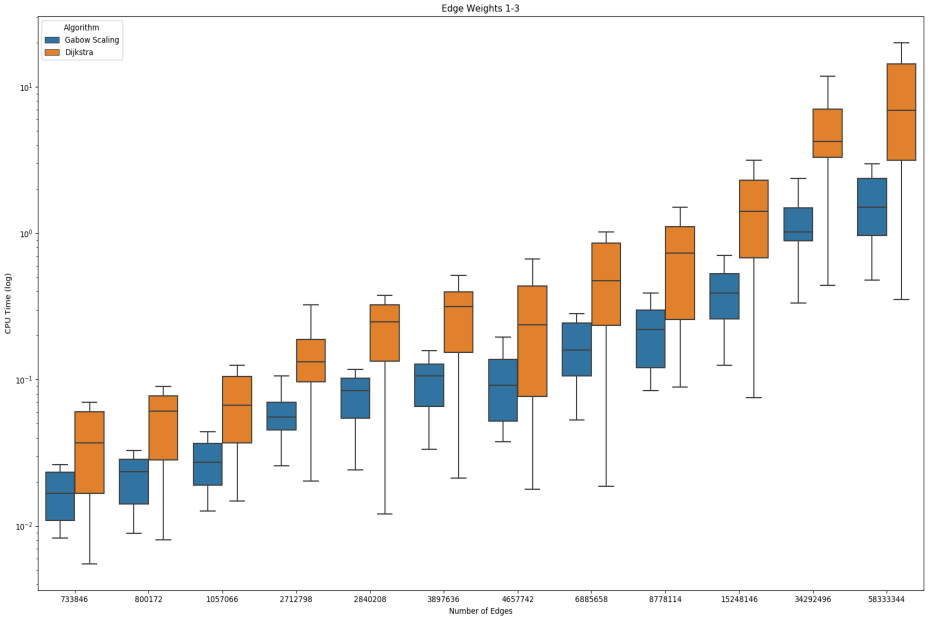

Figure 2-4: CPU time of Dijkstra vs. Gabow for edge weights 1-3.

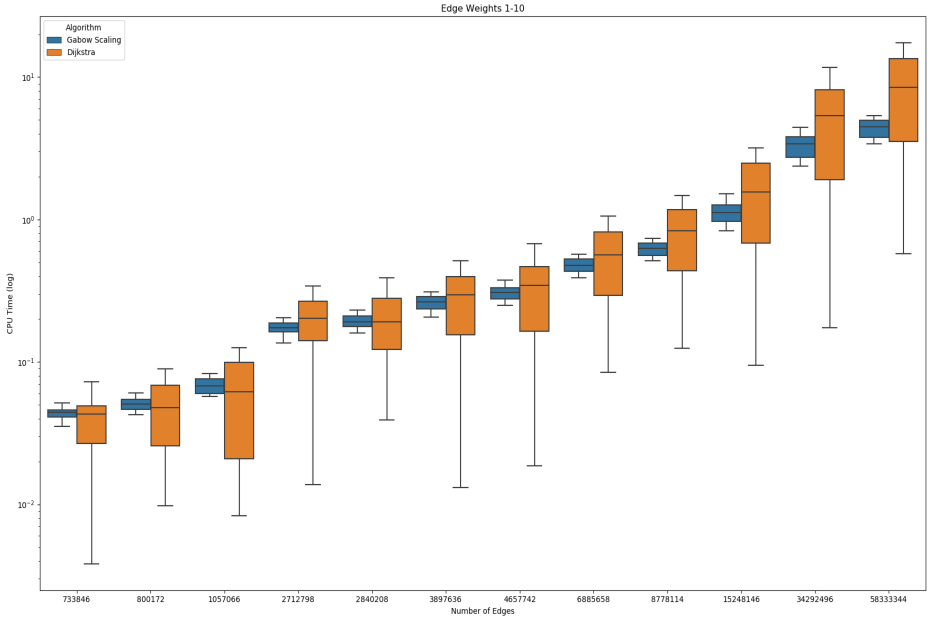

Figure 2-5: CPU time of Dijkstra vs. Gabow for edge weights 1-10.

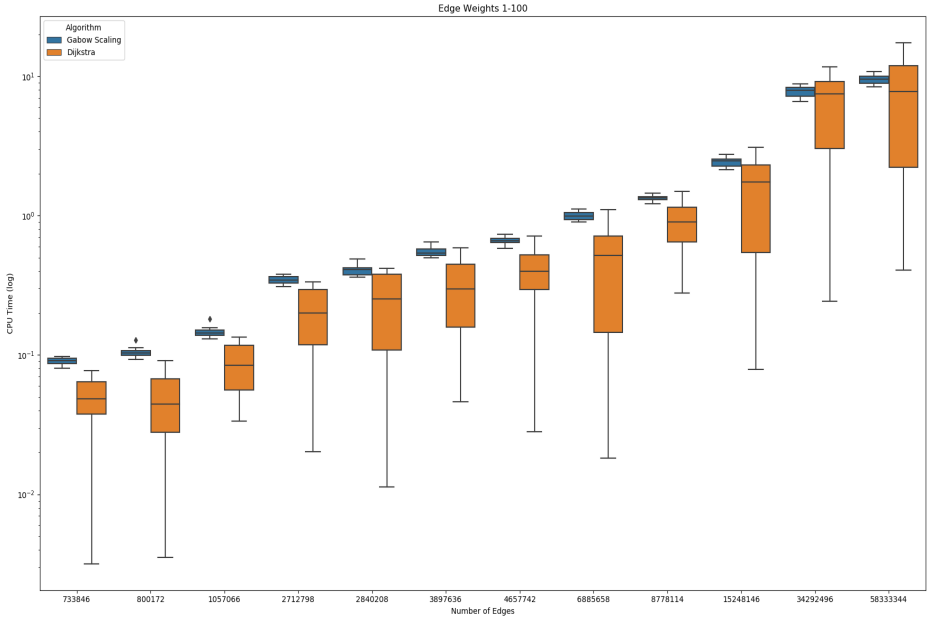

Figure 2-6: CPU time of Dijkstra vs. Gabow for edge weights 1-100.

and the number of edges range from 7.33×10^5 to 5.83×10^7 respectively. 100 random start and goal pairs were chosen.

Figures 2-3 - 2-6 compare the performance of Gabow's algorithm against Dijkstra's algorithm. Using unit cost edge weights, Gabow's algorithm significantly outperforms Dijkstra's algorithm. This is expected as it this is essentially comparing using a binary heap to maintain the open list (Dijkstra) to using the bucket list priority queue (Gabow). As the range of edge weights grows, Dijkstra quickly begins to outperform Gabow's algorithm, this is especially evident in Figure 2-6.

It is noteworthy that Gabow's algorithm does a seemingly consistent amount of work no matter the start, goal pair. On the other hand, the running time of Dijkstra's algorithm varies quite a bit depending on the distance between the start and goal pair. The consistency of Gabow's algorithm can likely be attributed to the fact that the algorithm always uses Dijkstra's algorithm on $log(M)$ times (one for each scale of G), regardless of the distance between the start and goal vertices. Without the optimizations mentioned earlier, the time it takes for Gabow's algorithm to return would be completely independent from the distance between the start and goal vertices.

2.5 Discussion

The idea to directly transform the graphs of heuristic-based searching algorithms was considered. This could be done by constructing edge weights between each of the nodes and using Gabow's algorithm. For reasons described below, we decided to only take inspiration from the idea of scaling rather than apply it directly.

First we considered scaling a greedy best first search that maintains an open list sorted by the estimated cost to reach the goal $h(i)$ for a node i. However, it is not obvious how to compute edge weights such that every edge weight is nonnegative. For example, taking the difference in heuristic values between parent and child i.e. $w_{ij} = h(i) - h(j)$ is not guaranteed to be nonnegative because some children will not be toward goals.

Unlike greedy-best first search, the transformation would be feasible in the context of A*

search. In A*, the open list is ordered by $f(i)$ where $f(i) = g(i) + h(i)$. Using a consistent heuristic, for every node i that has a child j, $f(i) \leq f(j)$. Thus, nonnegative edge lengths can be computed using the difference in f values between parent and child.

$$\begin{aligned} w_{ij} &= f(j) - f(i) \\ &= g(j) + h(j) - g(i) - h(i) \\ &= g(i) + c(i,j) + h(j) - g(i) - h(i) \\ &= c(i,j) + h(j) - h(i) \end{aligned}$$

A separate issue however is that the performance of the algorithm would seemingly take a crucial hit for shortest paths that have very small costs. Consider the the iteration where edge weights are scaled the most. In this context, all edges would be viewed as having similar costs. Edges that may lead to a very poor solution are given an equal chance of being on the path that the algorithm chooses to follow. Thus, this scaling version of A* search may expand more nodes than plain A*, which seems like an unpromising direction in which to move forward.

2.6 Conclusion

In this chapter, we compared Gabow's algorithm to Dijkstra's algorithm. While the theoretical running of Gabow's algorithm is superior to Dijkstra's algorithm, we found that Dijkstra's algorithm often outperforms Gabow's algorithm in practice. The exception is when there is a very small range of edge weights in which case Gabow's algorithm dominates. Most importantly, this technique of scaling is what motivates much of the remaining work of this thesis.

Part II

Heuristic Search

CHAPTER 3

Quantized Edge Cost Greedy

3.1 Introduction

Optimal solutions to search problems can be found using A* search (Hart et al., 1968), however this is often impractical due to the abundance of time and memory needed to obtain an optimal solution. Suboptimal algorithms trade potentially increased solution cost for greatly reduced computation time. These algorithms typically give greater consideration (or complete consideration in the case of Greedy) to h, the heuristic estimate on remaining cost needed to reach a goal. Researchers have recently looked into how d, the estimated search distance-to-go, can play a role in speeding up a search.

Greedy search is a best-first search on h, some heuristic estimate of the remaining cost to reach a goal. In Greedy search, the open list is sorted by h and the node with the smallest $h(n)$ is selected for expansion. Following low h can lead to local minima, thus Greedy search is not guaranteed to find optimal solutions.

Speedy search (Ruml & Do, 2007) is a best first search using d, where d is the number of estimated actions to reach the goal. In other words, Speedy search favors nodes that appear to be close to the goal. The strength of Speedy is in finding solutions very quickly by disregarding edge costs in the search. This approach typically uses less CPU time than a Greedy search but often leads to worse solutions in comparison. As we explain in more detail below, it has been conjectured that because it treats all actions equally, d has shallower local minima than h and hence can guide a search more quickly to a goal.

Additional inspiration for the algorithms introduced in this paper comes from scaling algorithms (Gabow, 1985). In general, scaling involves reducing the precision of the values that represent some input in order to produce the output more efficiently. Radix sorting, in

which a small compact range of values allows for an $O(n)$ sort instead of $O(n \log n)$, is an example. As we discuss more below, the application of this scaling idea in heuristic search allows us to use a more efficient data structure to represent the open list.

In Speedy search, all actions cost 1 while in Greedy search, edge costs can vary greatly depending on the domain. Both algorithms work at opposite ends of the spectrum regarding how precisely the edge costs are represented in the search. Naturally, this calls for an investigation into what lies in the space between Speedy and Greedy. The algorithm introduced below helps fill in some of the gaps between Greedy and Speedy.

3.2 Quantized Edge Cost Greedy

Speedy search does not differentiate between cheap and expensive edge costs. When dealing with domains that have a large operator cost ratio, this can be problematic in terms of solution quality (Wilt & Ruml, 2011). This leads us to the following broad idea: differentiating between cheap and expensive actions is important while generalizing the cost of actions offers some benefit. To embrace this idea, QEG limits the number of edge costs that can be represented in the search. The intuition is this would serve to separate cheap from costly actions as well as keep some of the generality that has made Speedy search so successful.

QEG works as follows. Given a user-supplied value x, the search allows for 2^x distinct edge costs. It is easiest to begin with an example to show this – we will consider QEG with $x = 1$. With 1 bit, edge costs are restricted to be either 1 or 2. In order to apply this to a domain, consider the heavy tiles 15-puzzle. In this domain, the cost of moving a tile is equal to the number on the face of the tile. Applying QEG to heavy tiles, the cost of moving tiles 1-7 becomes 1 while the cost of moving tiles 8-15 becomes 2. Furthermore, QEG with $x = 2$ is restricted to 4 edge costs: 1, 2, 3, 4. Again, using heavy tiles as an example domain, these costs are respective to the intervals [1,3], [4,7], [8,11], and [12,15]. In the general case for QEG, 2^x equal intervals are created that correspond to costs from 1 to 2^x using the lower and upper bound on all possible edge costs in the domain (of course, one could consider unequal intervals, but we leave this idea for future work). The heuristic

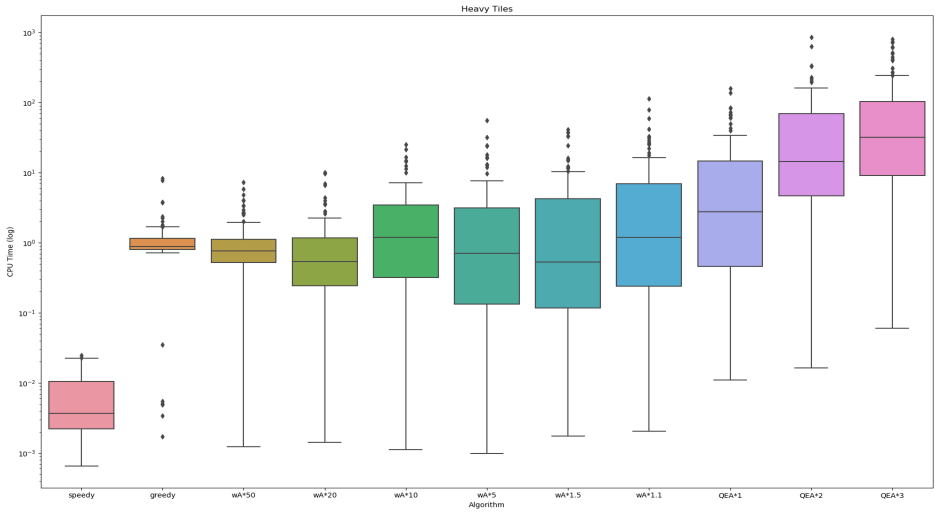

Figure 3-1: CPU time of QEA* on heavy tiles

in QEG is computed using these modified edge costs. Then, the search proceeds exactly like Greedy search (the open list is ordered by h).

3.3 Quantized Edge Cost A*

The modified edge costs in the previous section can also be used in A* search. In A* search, the open list is sorted by f where $f(n) = g(n)+h(n)$. To use the modified edge costs, simply compute $g(n)$ and $h(n)$ using the modified edge costs.

Figures 3-1 and 3-2 show the computation time and solution cost respectively on the heavy tiles domain. The number next to QEA* represents the number of bits that were used to represent edge costs in the domain. As we can see from the two figures, there is a slight increase in solution quality as the number of bits increases. This increase in solution quality also comes with an increase in computation time. Because there were very minor differences in QEA* with different values for x, further evaluation of this algorithm was not pursued on other domains.

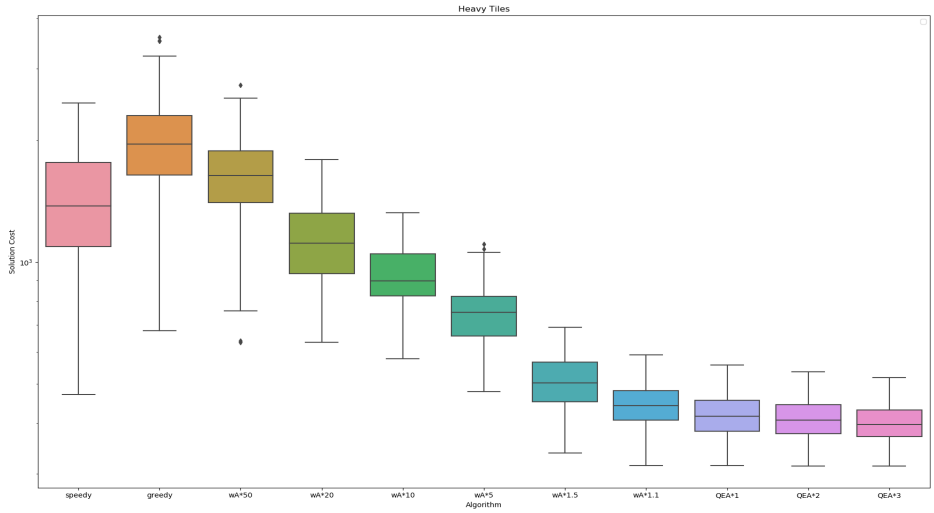

Figure 3-2: Solution cost of QEA* on heavy tiles

Variant	Cost Function
Heavy	$face$
Inverse	$\frac{1}{face}$
Square Root	$\sqrt{face}$
Reverse Inverse	$\frac{1}{16-face}$

Table 3-1: Various cost functions for sliding tile puzzles.

3.4 Experimental Results

We evaluate the performance of Greedy, Speedy, and QEG on the sliding tile puzzle. All experiments were performed using an upper limit of 8GB of RAM. All three algorithms that were evaluated use use a binary heap to maintain the open list and also decide tie-breaking in favor of low d. For QEG, we ran experiments with $x = 1, 2, 3$.

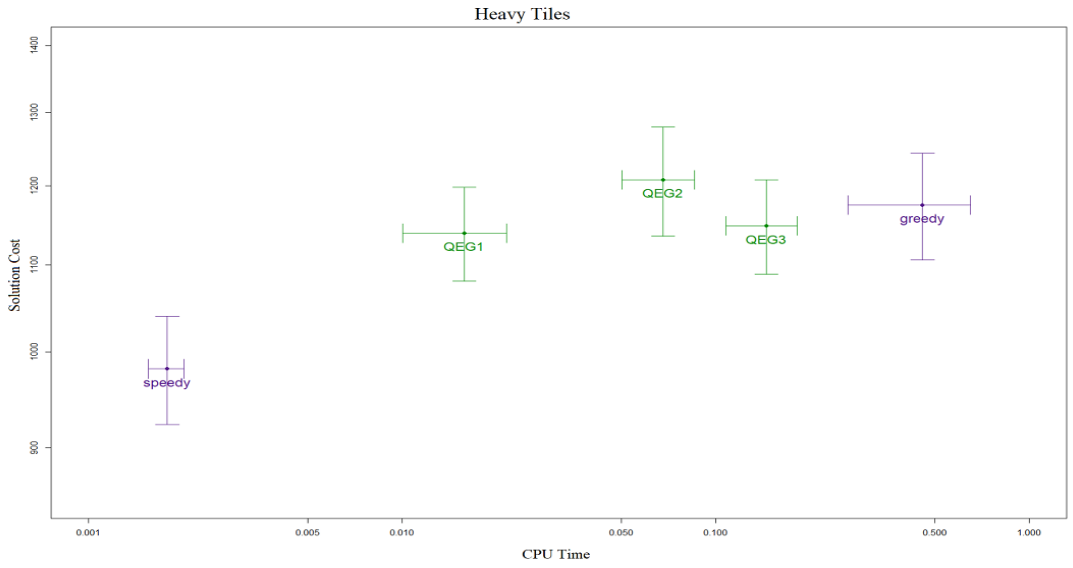

Figure 3-3: Solution cost vs CPU time for suboptimal searches on heavy tiles.

3.4.1 Sliding Tiles

For the sliding tile puzzle, we test the performance of the algorithm on the 100 instances of the 15-puzzle published by Korf (Korf, 1985). Furthermore, we looked at several variations of the sliding tile domain (Table 3-1). Regardless of the specific tiles domain, the Manhattan distance heuristic was used and weighted appropriately for the cost function. Additionally, linear conflicts (Korf & Taylor, 1996) were computed to provide a more accurate heuristic. For all variations of the tiles domain, the goal state contains the blank in the top-left of the puzzle grid with the rest of the tiles placed row-wise in numerical order.

The results of the tiles experiments can be found in Figures 3-3-3-6. The QEG number represents x. The error bars represent 95% confidence intervals on the mean. The QEG searches fill in the space between Speedy and Greedy appropriately. In general, as the number of bits used to represent edge costs increase, the computation time also increases. Unfortunately for the sliding tile puzzle, Speedy dominates Greedy thus there is no benefit to using QEG on the tiles variants.

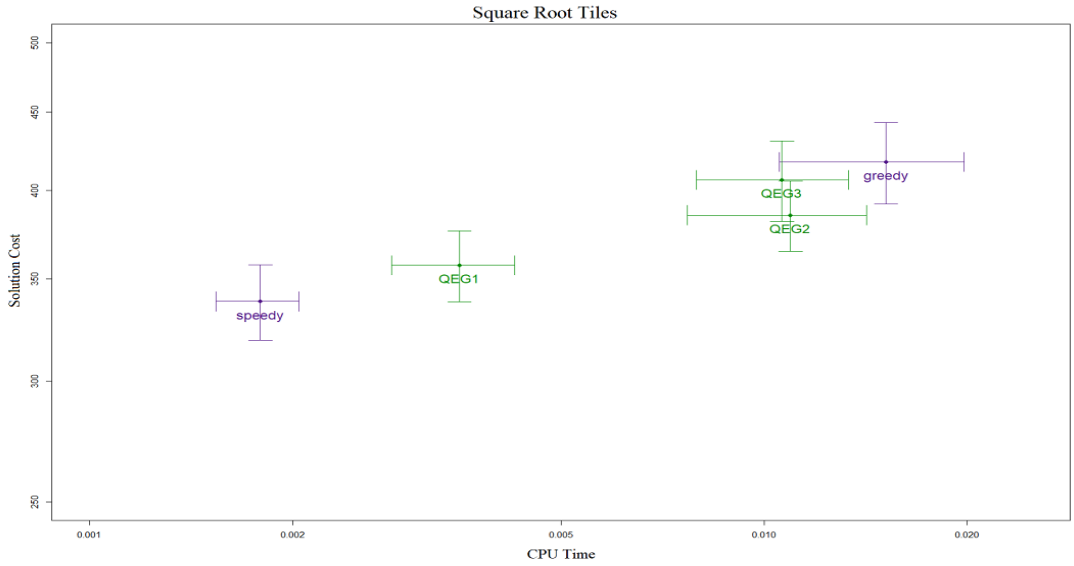

Figure 3-4: Solution cost vs CPU time for suboptimal searches on square root tiles.

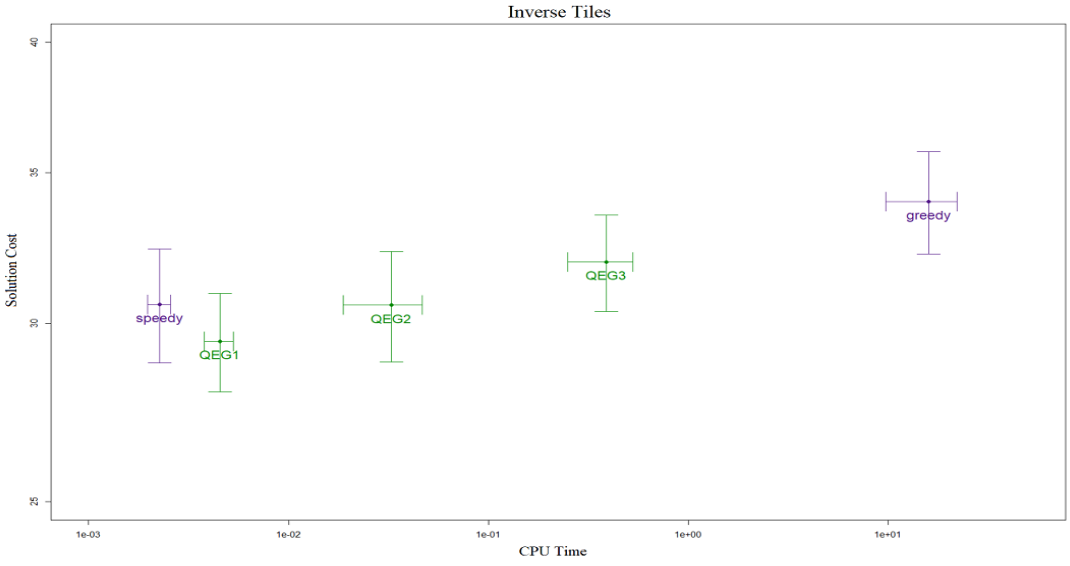

Figure 3-5: Solution cost vs CPU time for suboptimal searches on inverse tiles.

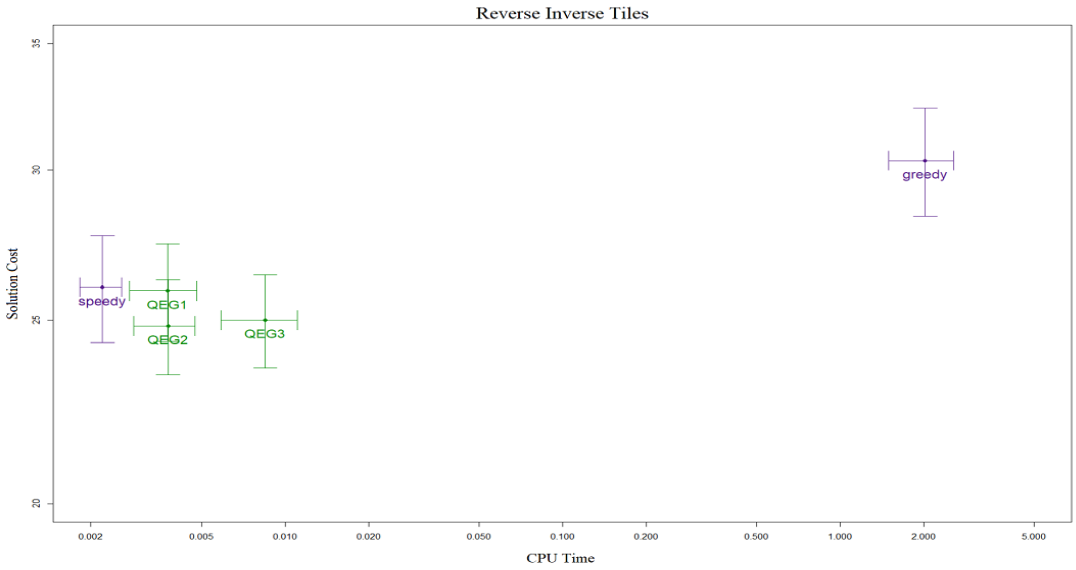

Figure 3-6: Solution cost vs CPU time for suboptimal searches on reverse inverse tiles.

3.5 Discussion

Our empirical investigation only considered non-unit cost tiles variants in which h and d differ. As mentioned earlier, one possible future direction for QEG could be to use unequal intervals within the lower and upper bound of all possible edge costs in the domain. Smartly allocating these intervals according to a specific domain may lead to a boost in performance.

The limitation of bits to represent edge costs in A* was also briefly covered with QEA*. While there seemed little promise on the heavy tiles domain, another future direction could be to evaluate the performance of QEA* on other domains. Furthermore, improvements could be made to QEA* using the more intelligent, unequal distribution of intervals mentioned above.

3.6 Conclusion

In this chapter, we introduced QEG, an algorithm that provides performance between Greedy and Speedy by limiting the number of bits that can be used to represent edge costs. We looked at QEG's performance on a few variants of the sliding tile puzzle. Be-

cause Speedy actually dominates Greedy on the tile variants, QEG is not of any use in this domain. However, QEG still fills the void between the extremes of the two best-first searches.

CHAPTER 4

f-Bucket Speedy

4.1 Introduction

Classical approaches for finding suboptimal solutions tend to lie somewhere between A* search and a Greedy best-first search such as weighted A* (Pohl, 1970). Weighted A* (Pohl, 1970) is arguably the most well-known bounded suboptimal searching algorithm, in which the open list is sorted by $f(n) = g(n) + w \cdot h(n)$ and w is some user-supplied weight. The popularity of weighted A* can be attributed to three reasons. First, it is very implementation friendly considering its close resemblance to A*. It typically uses a binary heap for the open list which tolerates any arbitrary w. Second, the algorithm is flexible in that the user can specify which weight w to use in order to meet their needs in terms of computation time and solution quality. Finally, weighted A* also provides a bound of suboptimality – an assurance on the quality of the solution that it produces. More specifically, the solution found using Weighted A* is no worse than a factor of w when compared to the optimal solution.

In this chapter we introduce f-Bucket Speedy (FBS), a suboptimal searching algorithm that behaves between A* and Speedy search. FBS groups nodes that have similar f values and chooses between them using d. FBS essentially uses a Speedy search on a subset of the nodes on the open list that contains the most promising nodes according to f in a manner reminiscent of A^*_ϵ(Pearl & Kim, 1982).

It has been well-known that, for domains in which g and h values fall in a compact range of integers, a 'bucket list' data structure can be used for A*'s open list to achieve constant time insertion and removal, avoiding the logarithmic overhead of a binary heap (Dial, 1969; Edelkamp, Schroedl, & Koenig, 2010). A bucket list is implemented using an array that

stores, at index i, a linked list of all nodes with $f(n) = i$. Recall that this bucket list was also leveraged to represent the open list in Gabow's algorithm in an earlier chapter. This technique is crucial for efficiency of FBS.

4.2 f-Bucket Speedy

In this section we propose a simple, unbounded suboptimal searching algorithm that restricts the number of f values that can be represented on the open list. FBS maintains an open list that is comprised of a user-supplied number of 'buckets'. Each bucket in the open list contains the nodes in a certain range of f values relative to the rest of the f values currently on the open list. Each bucket in turn is sorted by d the estimated search distance-to-go. Pseudocode for FBS can be found in Figure 4-1. The search works by looping through the following process until a goal node is found: extract the node with the smallest d from the bucket holding the smallest f values (line 3), insert each child of this node into the bucket priority queue (line 7), rebucket when necessary (line 9). As we explain in detail below, the bucketing of all nodes into a small finite number of buckets allows for the use of a more efficient data structure than the traditional binary heap.

When a node is inserted into the open list, it is placed into one of k buckets based on how it's f value compares to the f values of other nodes on the open list. The search begins by expanding the initial start node. The minimum and maximum f values of this node's children are assigned to f_{min} and f_{max} respectively. Then $k-1$ equally-sized intervals are made in between the f_{min} and f_{max}. The kth bucket is reserved for nodes whose f values are greater than the current f_{max}. On insertion, the correct bucket is chosen according to the node's f value. As an example, consider using 3 buckets with minimum and maximum f values of 15 and 25 respectively. With $k = 3$, nodes with an f value in the interval $[15, 20)$ are placed in the first bucket, nodes with an f value in the interval $[20, 25)$ are placed in the second bucket, and finally all nodes in the interval $[25, \infty)$ are placed in the third bucket.

In keeping the number of f values low, the search roughly places good nodes in the first bucket and unpromising nodes in the last bucket. Essentially, FBS levels the playing

```
FBS()
1. bpq-insert(start)
2. while !bpq-empty()
3.     n ← bpq-extract-min()
4.     if n is goal
5.         return
6.     for child ∈ successors(n)
7.         bpq-insert(child)
8.         if bpq[k − 1].size ≥ bpq.size / 2
9.             bpq-rebucket(f_min, f_max, k)
```

Figure 4-1: Pseudocode for FBS

field for nodes that have similar f values and turns to d to choose the most promising node to expand next. Note that since the algorithm tie-breaks on d, when k is equal to 1, the algorithm becomes Speedy search and as k approaches infinity, the algorithm behaves like A*.

4.2.1 2-Level Bucket Priority Queue

In searching algorithms, the open list is commonly represented by a binary heap. By quantizing the f values and tie-breaking on d, we are able to use a more efficient data structure for the open list: a 2-level bucket priority queue.

The 2-level bucket priority queue consists of two arrays. The top-level array is of length k and represents the f buckets. The first $k-1$ buckets in the top-level array then point to a respective second array that represents a series of buckets that is sorted by d. The k^{th} bucket does not need to order nodes by d because nodes will never be pulled from here (see below). Each entry in the second-level array is an unsorted doubly linked list of nodes that all have $d(n)$ equal to the index of the entry.

When a node is inserted into the open list, it is first indexed into the correct top-layer f bucket. Then in the second level of the bucket priority queue, the node is inserted into the front of the list at the index corresponding to the node's d value. Nodes are inserted and extracted from the front of its linked list i.e. in a last in first out manner. Since FBS limits the number of f values and d is guaranteed to be an integer, we can use the 2-level bucket priority queue to back the open list regardless of the domain. As a result, inserting a node into both levels of the bucket list is constant time, so the overall insertion is also a constant time operation. Additionally, we keep track of a node's index in the second level bucket list allowing direct access for constant time removal.

4.2.2 Rebucketing

As the search progresses, the last bucket in the top level priority queue will naturally begin to fill up since f values tend to grow larger along any path when using an admissible heuristic. Eventually, there may come a time where the last bucket contains every node in the open list, resulting in a Speedy search. To counter this, we ensure that the kth bucket does not become too populated. Once more than 50% of all nodes in the open list are in the kth bucket, a rebucket procedure is called.

This procedure begins by updating f_{min} and f_{max}. The new f_{min} is assigned to be the lower bound on the bucket that contains the next node that would be extracted. If we wanted to find the actual f_{min}, a second sorted data structure would need to be maintained to keep track of actual f values on the open list. This would hinder the performance of the constant time insertion and removal, so we opt for the less precise but more efficient selection of f_{min}. The node with the largest f value is kept track of when inserting nodes into the open list. This node will never be extracted from the open list because nodes are never extracted from the last bucket as a consequence of the 50% rebucket policy. Once f_{min} and f_{max} are updated, the intervals are recalculated in accordance with these two values. Finally, every node in the open list is removed and then re-inserted into its correct bucket according to the new intervals. With constant time insertion and removal, the overall time

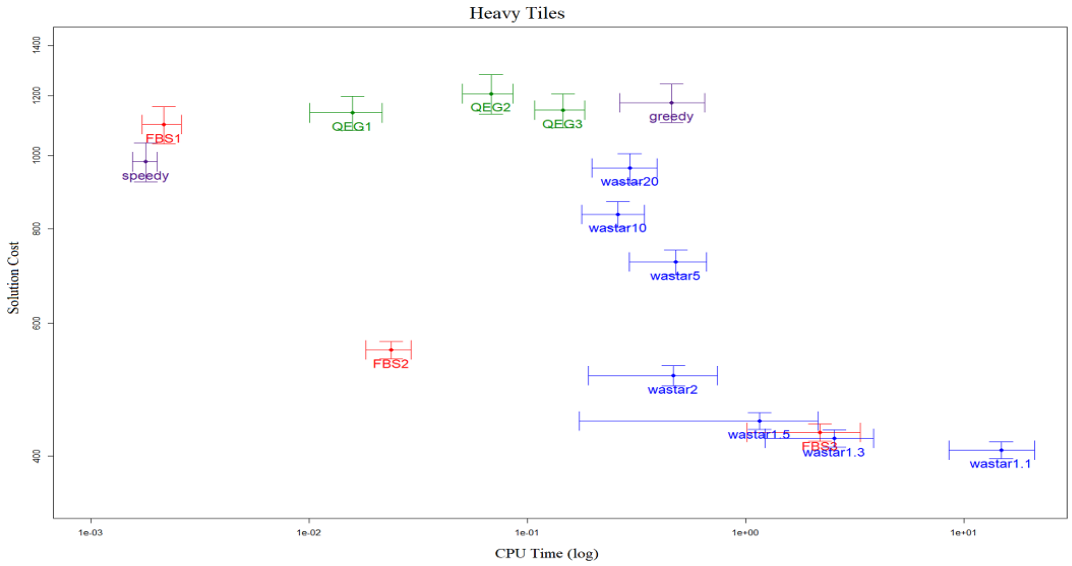

Figure 4-2: Solution cost vs CPU time for suboptimal searches on heavy tiles.

complexity of this rebucket procedure is $O(n)$. Note that FBS is possibly incomplete in an infinite state space due to rebucketing, but any search on d is already incomplete anyways.

4.3 Experimental Results

We evaluate the performance of weighted A* and FBS on the following benchmark domains: sliding tile puzzle, grid pathfinding, vacuum, and pancakes. All experiments were performed using an upper limit of 8GB of RAM. Weighted A* uses a binary heap to maintain the open list and decides tie-breaking in favor of low d. FBS uses the 2-level bucket priority queue to maintain the open list. Additionally, we experimented with using 1, 2, and 3 buckets for FBS. In the unit cost variant of these benchmark domains, $h = d$. Because we are interested in exploring distance estimates in suboptimal search, we primarily consider non-unit cost variants.

4.3.1 Sliding Tiles

The experiments for the sliding tile puzzles were performed exactly as described in the previous chapter. The results of these experiments can be found in Figures 4-2-4-5. The

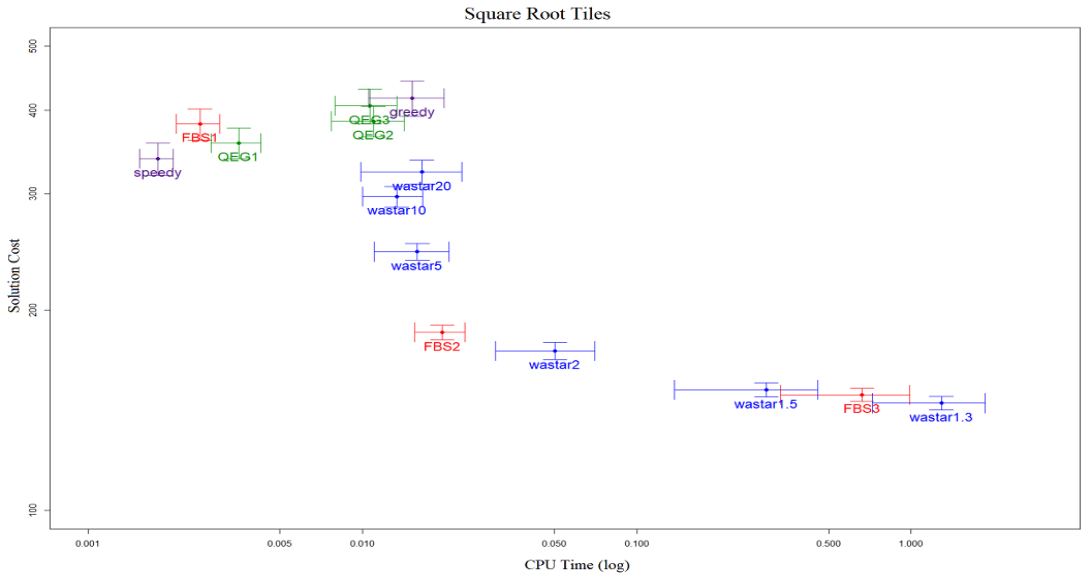

Figure 4-3: Solution cost vs CPU time for suboptimal searches on square root tiles.

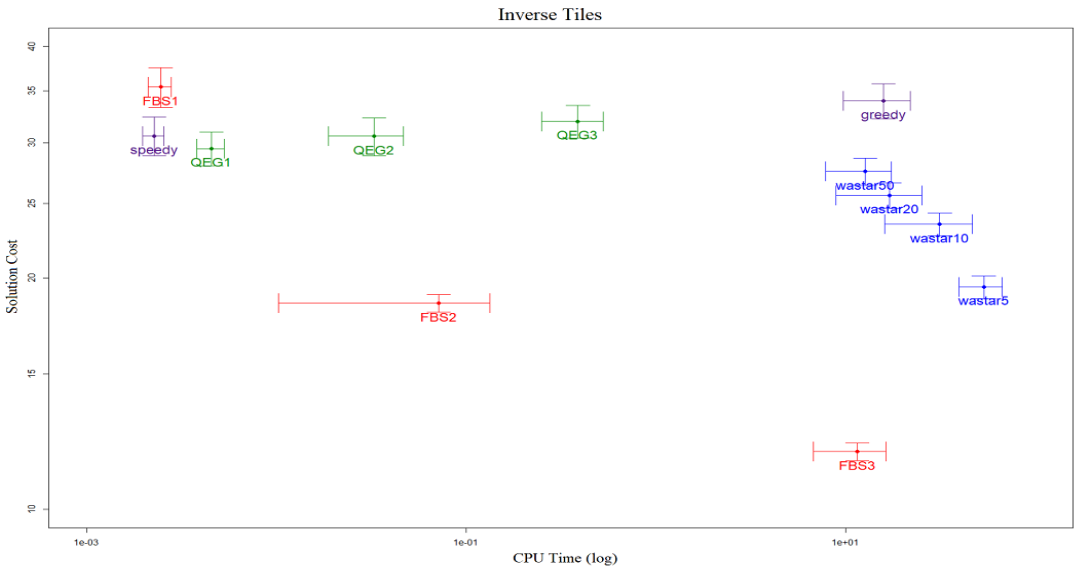

Figure 4-4: Solution cost vs CPU time for suboptimal searches on inverse tiles.

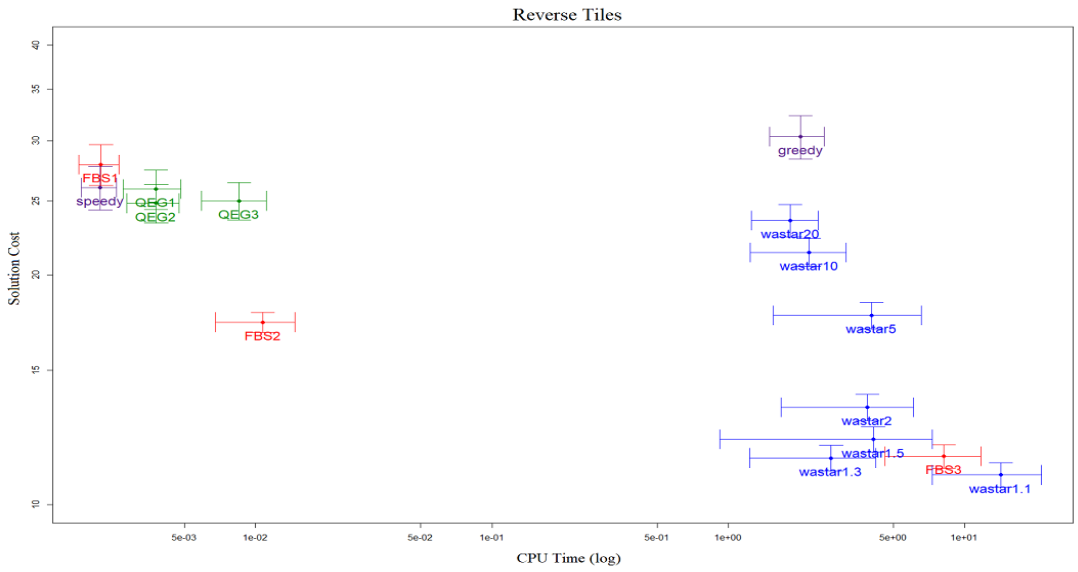

Figure 4-5: Solution cost vs CPU time for suboptimal searches on reverse inverse tiles.

	%
FBS1	100
FBS2	36
FBS3	28
wA*2	100

Table 4-1: Performance of FBS vs. weighted A* on heavy vacuum.

QEG number represents x, the FBS number represents the number of buckets used, and the weighted A* number is the weight used in the search. The error bars represent 95% confidence intervals on the mean. The QEG searches fill in the space between Speedy and Greedy appropriately. In general, as the number of bits used to represent edge costs increase, the computation time also increases. FBS performs well specifically on tiles variants where weighted A* struggles, namely the inverse, reverse inverse, and square root variants. Additionally, FBS has a node expansion rate 1.84 times faster on average compared to weighted A* on the sliding tile puzzle.

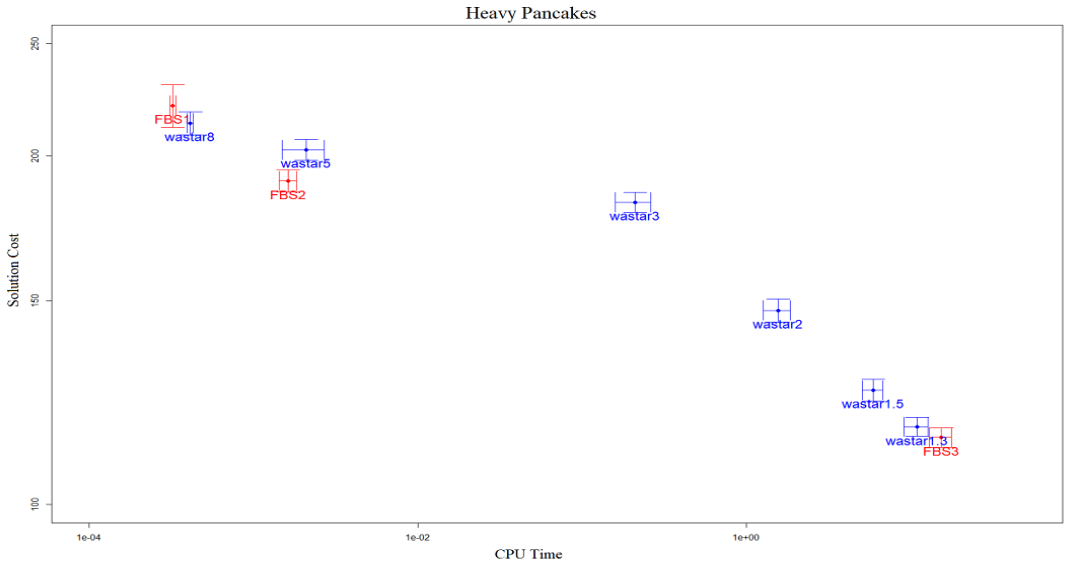

Figure 4-6: Solution cost vs CPU time for suboptimal searches on heavy pancakes.

4.3.2 Pancakes

In the pancake puzzle domain, there is a stack of pancakes that each have different sizes. The goal of the puzzle is to end with a stack where each pancake is resting on a larger pancake than itself. This goal is achieved through flipping smaller portions of the stack until all pancakes are in their goal position. We consider two different variants of the cost function in our experiments.

The first variant is heavy pancakes. In heavy pancakes, the cost to flip a stack of pancakes is the max of the two elements on either side of the stack. The gap heuristic (Helmert, 2010) works by adding 1 to the heuristic for every two adjacent pancakes that are not within one of each other. For heavy pancakes, we use an improvement to the gap heuristic. For every gap in the stack, the minimum of two pancakes that make up the gap is added to the heuristic (Gilon, Felner, & Stern, 2016). We ran the algorithms on 100 randomly generated 16-pancake puzzles with this variant. Figure 4-6 shows the results of these experiments. Note that Speedy, QEG, and Greedy were left out of the plot because these algorithms performed nearly identically to FBS with 1 bucket. FBS exhibits similar behavior to its performance on the sliding tile puzzle. As the number of buckets increases

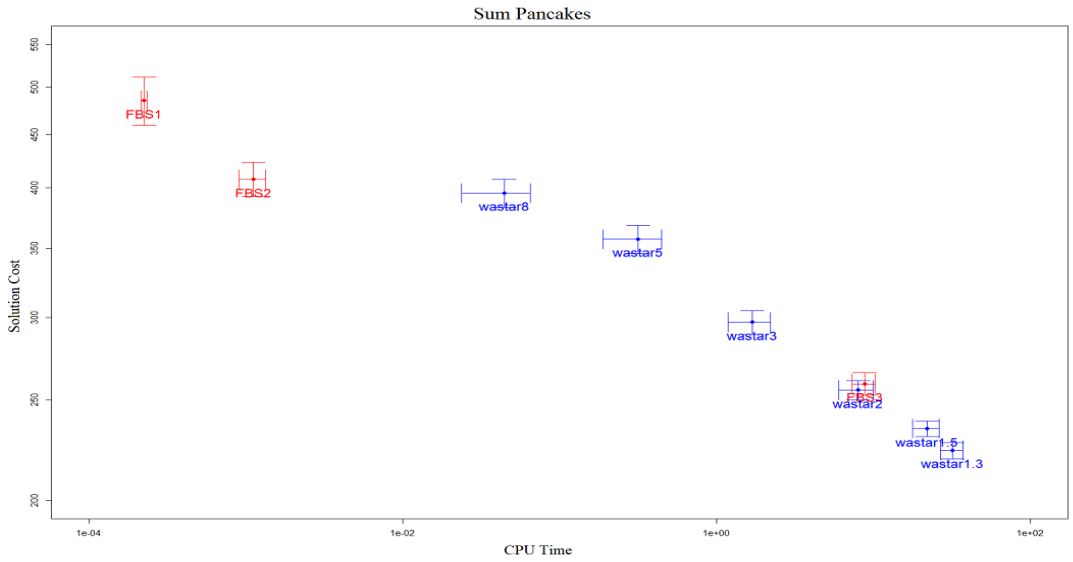

Figure 4-7: Solution cost vs CPU time for suboptimal searches on sum pancakes.

from 1 to 3, FBS finds more favorable solutions but needs more computation time. On this domain, it has little advantage over weighted A*. On average, FBS expands 1.36 times more nodes per second than weighted A* on heavy pancakes.

The second variant of the pancake puzzle that we consider is sum pancakes. In this domain, the cost of flipping a stack of pancakes is the sum of all the indices of the pancakes being flipped. We also use a modified gap heuristic for this variant: first the cost that it would take to fix the the gap closest to the bottom of the stack (the one that is most expensive to correct) is added to the heuristic and then 1 is added to the heuristic for every other gap in the stack. In this variant, our heuristic is very close to the unit gap heuristic and relatively uninformative thus we consider the smaller 14-pancake puzzle. The performance of the algorithms on 100 randomly generated instances can be found in Figure 4-7. Like heavy pancakes, many of the algorithms on the Speedy-Greedy spectrum are not included because of their similarity to FBS with 1 bucket. FBS is able to scale to substantially faster times than weighted A*. In terms of node generation, FBS is faster than weighted A* by a factor of 1.76. Using 3 buckets, FBS performs similarly to weighted A* with a weight of 2 on sum pancakes and whereas 3-bucket FBS is closer to weighted A* with a weight of 1.3

on heavy pancakes.

4.3.3 Vacuum World

In this domain, a robot is in a grid that is also populated with obstructed cells and dirt spots (Russell & Norvig, 2016). The robot is tasked with cleaning up all of the dirt spots. The robot has five available actions: move in the four cardinal directions and vacuum if the robot is in a cell that is dirty. We consider the heavy vacuum variant where 1 is added to the cost of moving the robot for each dirt that has been collected. The heuristic used for this domain is the sum of the displacements of the four dirty cells that are farthest from the robot in the cardinal directions, multiplied by the number of remaining dirty cells. For our experiments we considered a 200x200 grid with 35% obstacles and 6 randomly placed dirt spots. All grid instances were ensured to be solvable.

Table 4-1 shows the percentage of grid instances the algorithms were able to solve within 8GB of RAM. Out of 100 instances, only about a third were solved by 2-bucket FBS. Using three buckets further decreases the number of solvable instances. On the other hand, weighted A* is able to solve all 100 instances with a weight as small as 2. Following Thayer, Ruml, and Kreis (Thayer et al., 2009)'s analysis of A^*_ϵ, we attribute the poor performance of FBS to the way the nodes are bucketed. Nodes close to the goal have low d and will typically have large f. Hence, these nodes are placed in an f bucket that the search is unlikely to choose from for expansion. The work done before these nodes are chosen for expansion can be time consuming and unhelpful for the search.

4.3.4 Summary

FBS shows very promising performance on many of the domains that we tested. For example in inverse tiles, FBS completely dominates weighted A*. However, the bucketing approach performs poorly on heavy vacuums and life grid pathfinding.

4.4 Discussion

Our empirical investigation primarily considered non-unit cost domains in which h and d differ. It would be interesting to see how FBS fares on additional unit-cost domains.

We compared FBS to weighted A*. A^*_ϵ and EES are two other bounded-suboptimal searches that exploit distance-to-go information. However, implementing them efficiently requires a red-black tree whose left prefix is synchronized with a heap. The complexity of implementing this is a significant obstacle to widespread adoption. In contrast, FBS requires only arrays, thus the algorithm is simple to implement.

One possible future direction for FBS could be to consider unequal intervals for the f buckets spanning between f_{min} and f_{max}, similar to the proposal from the discussion of QEG. Currently the intervals are uniforms however choosing an intelligent distribution of the intervals in accordance with the domain could lead to better performance.

4.5 Conclusion

We have presented a simple new method for suboptimal heuristic search that exhibit novel behavior. FBS provides performance between A* and Speedy and also has a very fast node expansion rate. To our knowledge, this is the first search algorithms that use both cost-to-go and distance-to-go information but do not require a red-black tree for efficient implementation. The empirical performance suggests that it can be a useful tool in the growing arsenal of suboptimal heuristic search algorithms.

CHAPTER 5

Bounded f-Bucket Speedy

5.1 Introduction

The final contribution of this work is a simple new bounded suboptimal searching algorithm. Because optimal solutions are often impractical due to insufficient resources, it is common to turn to bounded suboptimal searching algorithms. A bounded suboptimal searching algorithm guarantees that the solution returned is within some pre-specified bound of the optimal solution.

One relevant bounded suboptimal searching algorithm is A^*_ϵ (Pearl & Kim, 1982). A^*_ϵ maintains two data structures of nodes on the frontier: the open list and the focal list. The open list orders nodes by their f values. The focal list is a prefix of the open list that contains all nodes whose $f(n) < w \cdot f_{min}$. The nodes on the focal list are then sorted by d. The node at the front of the focal list is selected for expansion until a goal node is expanded. Decreasing the bound requires the algorithm to expand nodes with low f, causing A^*_ϵ's behavior roughly to interpolate between A* and Speedy. One downfall of $A^*_\epsilon$is it's difficulty to implement efficiently because a red-black tree is required to maintain the open list.

In this chapter we introduce a bounded variant of FBS called Bounded f-Bucket Speedy (BFBS). BFBS uses a similar bucketing strategy to FBS but differs in the criteria of placing nodes in the first level representing the f buckets. We evaluate BFBS and compare it to weighted A*. Overall, this work provides us with a better understanding of how f and d can be used in tandem to speed up a search.

5.2 A Bounded Suboptimal Variant

We now present a bounded suboptimal variant of FBS that is called Bounded f-Bucket Speedy (BFBS). BFBS shares some similarities with A^*_ϵ but is much simpler to implement. Recall that A^*_ϵ only considers expanding nodes within a certain factor of f_{min}. These promising nodes are held in the focal list which is ordered by a second heuristic function. The search then only expands nodes that reside in this subset of the entire open list.

Similarly, we propose separating the nodes into two buckets based on their f values. Let the first bucket contain every node in the open list whose $f(n) < w \cdot f_{min}$ where w is some user-supplied weight. Let the second bucket contain every other node on the open list that does not meet this condition. The search proceeds by simply choosing the node with the smallest d in the first bucket to expand next. If the first bucket ever empties completely, we first reassign f_{min} to be the node with the smallest f value in the second bucket. We know that this node will not be extracted because it resides in the bucket that we are not pulling nodes from thus the node with smallest f value in the second bucket can be kept track of during insertion. Second, we rebucket each node according to the new f_{min}. Since we are only expanding nodes that reside in the first bucket, we are guaranteed that the solution will be no more a factor w worse than the optimal solution.

Theorem 5 *If opt is the optimal solution cost, the solution returned by BFBS costs less than* $w \cdot opt$.

Proof: Searching with an admissible heuristic guarantees that $f_{min} \leq opt$. Since nodes are only expanded from the first bucket where $f(n) < w \cdot f_{min}$ it follows that $f(goal) < w \cdot opt$. □

One major advantage of this algorithm over A^*_ϵ is its simplicity. Essentially, BFBS performs a Speedy search on all nodes within a factor of f_{min} while only recalculating f_{min} when absolutely necessary. An efficient implementation of A^*_ϵ would normally require a red-black tree to maintain the open list. At the very least, the use of a heap would be needed to make A^*_ϵ viable. BFBS uses the 2-level bucket priority queue that not only provides

constant time insertion and removal but is also very simple and intuitive.

5.3 Experimental Results

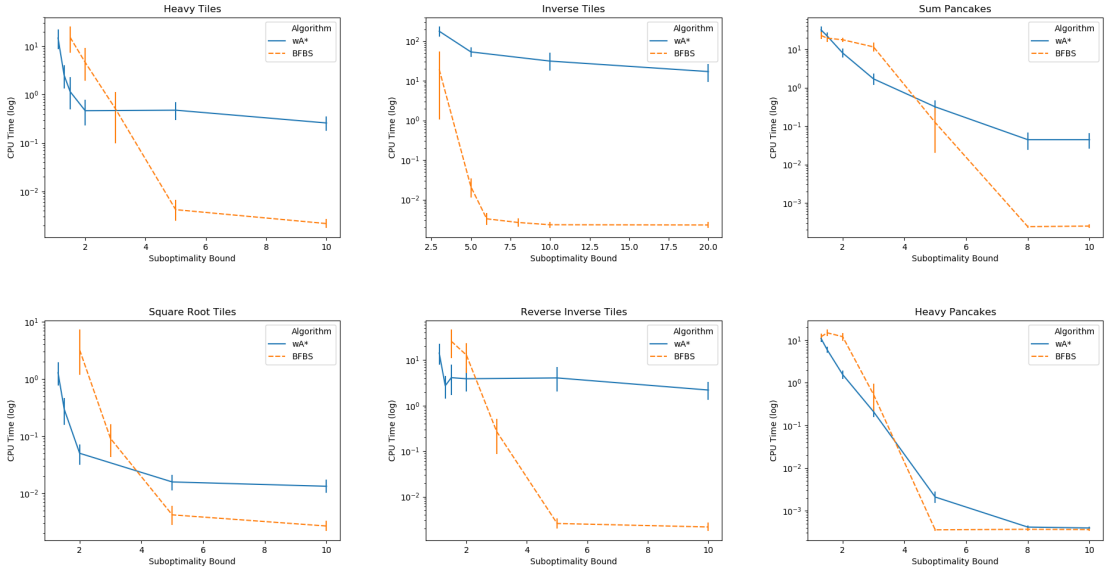

Figure 5-1: Performance of BFBS versus weighted A* on various domains.

We tested the performance of BFBS against weighted A* using the same experimental setup discussed above. The results of these experiments can be seen in Figure 5-1. BFBS is very competitive in these domains in comparison to weighted A*. The promising performance of FBS seems to be preserved in this bounded variant. For example in the inverse tiles domain, BFBS dominates weighted A* regardless of the suboptimality bound. In general, BFBS performs much better than weighted A* when using a large suboptimality bound. This can be seen consistently across all domains. But, as the suboptimality bound decreases, the performance of the two algorithms tends to converge until weighted A* overtakes BFBS.

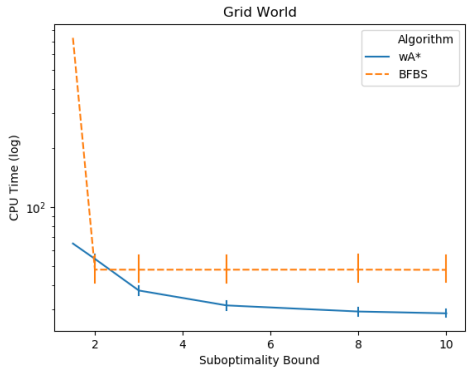

Figure 5-2: BFBS performs poorly on unit grids.

	b	%
BFBS	6	100
	2	51
	1.5	4
wA*	6	100
	2	100
	1.5	86

Table 5-1: Performance of BFBS vs. weighted A* on life grids.

5.3.1 Grid Pathfinding

Grid pathfinding is a simplified version of the vacuum world domain. A robot exists in a grid in some start position and is able to move in the cardinal directions. The objective is for the robot to end up in some goal position in the grid. The algorithms were tested on 100 randomly selected 2000 by 1200 grids that had 35% obstacles. The starting position of the robot is the bottom left corner of the grid and the goal position is the bottom right corner. All grid instances were ensured to be solvable. We use the unit cost function and also consider the "life" variant where the cost to move from a cell is equal to the y-coordinate of the cell. We use the life variant to separate the shortest path from the cheapest path i.e. $h \neq d$.

We speculate that similar to FBS, BFBS shares a shortcoming with A^*_ϵ when using very small bounds (Thayer et al., 2009). Nodes close to the goal have low d but will likely be placed in the second f bucket because of their high f values. These nodes have no chance of being selected until the first bucket empties and the rebucketing process is performed. This emptying of the first bucket is time consuming and likely does not contribute to the search. Figure 5-2 illustrates how drastically the CPU time increases as a result of a small change in the bound. Additionally, Table 5-1 shows the inability to solve some of the life grid instances as the bound decreases. Thus, on some domains, BFBS is not viable when using a small bound for the search.

5.4 Discussion

We compared BFBS to weighted A*. Beam search (Wilt, Thayer, & Ruml, 2011) is a very effective suboptimal search that sometimes outperforms weighted A* (Wilt, Thayer, & Ruml, 2010). However, beam search cannot use distance-to-go information and it is not clear how it could be modified to provide bounded suboptimality.

One necessary direction for the future of BFBS is to evaluate its performance on additional unit-cost domains. Along these same lines, further work should be done to characterize exactly which domains that BFBS can be expected to perform well on. As Figure 5-2 showed, there is a drastic change in performance due to a slight change in the suboptimality bound that is not quite understood.

A^*_ϵ and BFBS are very similar algorithms. Both essentially use a focal list, which is a subset of nodes on the open list that are within a certain bound of f_{min}. The difference is in how the focal lists are maintained. BFBS uses a draining approach and does not refill the first bucket until it is completely empty. A^*_ϵ continually removes and adds nodes to the focal list as the search progresses. Further experimentation needs to be done to compare these two methods of maintaining the focal list.

5.5 Conclusion

In this chapter we introduced BFBS, a simple bounded variant of FBS. We showed that BFBS is very effective in comparison to weighted A*, especially when using large bounds of suboptimality. We also showed the poor performance of BFBS on two domains when using suboptimality bounds close to 1. Further experimentation is needed to understand this phenomenon.

CHAPTER 6

Conclusion

6.1 Overall Lessons

Our work began with an analysis of Gabow's algorithm, a scaling version of Dijkstra's algorithm. While theoretically better, we found Gabow's algorithm only to be useful in practice on problems that have very small ranges of edge costs. However, the idea of scaling and using the efficient bucket list priority queue fueled our work in heuristic search.

We then introduced three new simple methods for suboptimal heuristic search that exhibit novel behavior. Where weighted A* provides performance between A* and Greedy, QEG provides performance between Greedy and Speedy. QEG is of no interest in domains in which $h = d$. FBS provides performance between A* and Speedy. It has a very fast node expansion rate, but can perform poorly on some domains that have not yet been characterized. Its bounded suboptimal variant, BFBS, works well for large suboptimality bounds. To our knowledge, these are the first search algorithms that use both cost-to-go and distance-to-go information but do not require a red-black tree for efficient implementation. Their empirical performance suggests that they can be useful tools in the growing arsenal of suboptimal heuristic search algorithms. We hope this works spurs further investigation into suboptimal search, the use of distance-to-go information, and how f and d can be used together.

6.2 Future Work

6.2.1 Distribution of Intervals

QEG and FBS were both introduced using uniform intervals for representing edge costs and separating f buckets respectively. As mentioned in the discussions for these two algorithms, one possible future direction could be to consider a better distribution for these intervals. Unequal intervals could take advantage of domain-dependent knowledge. For example, for a domain that has a very common edge cost, one could choose to Consulting Sturtevant, Felner, and Helmert (Sturtevant et al., 2017)'s work is one approach to this problem that may be a good place to begin.

6.2.2 Understanding Domains

We showed that FBS and BFBS have poor performance on some domains like grid world and heavy vacuum. Thayer et al. (Thayer et al., 2009)'s work with A^*_ϵ gives a reasonable explanation for this phenomenon. However, this does not fully explain why the drop in performance with FBS and BFBS is not as drastic as in grid world and vacuum world.

One speculation is that the poor performance is a result of having a poor heuristic h. In this case, f_{min} is not a valid representation of the optimal solution. To further characterize, when using an admissible heuristic, f_{min} would be an extreme underestimate of the optimal solution. FBS and BFBS both select the node for expansion from the left-most bucket containing nodes. Since nodes close to the goal have large f values, these nodes will likely be placed in low priority f buckets. As a result, unnecessary work will be done when expanding the less promising nodes in the left-most bucket. Furthermore, rebucketing is an expensive procedure ($O(n)$), and will likely need to be done at least once before these promising nodes are finally expanded. One contradiction to this speculation is the performance of FBS and BFBS on sum pancakes. Both algorithms did not show a drastic change in performance on this domain, despite using a variation of the gap heuristic that hardly differed from the unit-cost gap heuristic.

Another alternative speculation for the poor performance of FBS and BFBS pertains to the relation between edge costs and final solution cost. In grid world and vacuum world, solution costs are generally high in comparison to edge costs. Under the same assumption that towards the end of the search, nodes close to the goal are in low priority (right-most) f buckets due to a high f value, the sooner that rebucketing can occur, the sooner the search can progress. If there are many cheap actions to take, the algorithm will take a lot longer before using the rebucket procedure. It would be interesting to evaluate FBS and BFBS on grid world with 8-way movement versus movement only in the cardinal directions. In 8-way movement, one less move is needed to reach a diagonal cell, which is one less node to process before rebucketing occurs.

Bibliography

(2005). 9th DIMACS implementation challenge.. Available at http://www.diag.uniroma1.it/challenge9/download.shtml.

Dial, R. B. (1969). Algorithm 360: Shortest-path forest with topological ordering [h]. *Communications of the ACM*, *12*(11), 632–633.

Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische Mathematik*, *1*(1), 269271.

Edelkamp, S., Schroedl, S., & Koenig, S. (2010). *Heuristic Search: Theory and Applications*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Gabow, H. N. (1983). Scaling algorithms for network problems. *24th Annual Symposium on Foundations of Computer Science (sfcs 1983)*.

Gabow, H. N. (1985). Scaling algorithms for network problems. *Journal of Computer and System Sciences*, *31*(2), 148–168.

Gilon, D., Felner, A., & Stern, R. (2016). Dynamic potential searcha new bounded suboptimal search. In *Ninth Annual Symposium on Combinatorial Search*.

Hart, P. E., Nilsson, N. J., & Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, *4*(2), 100–107.

Helmert, M. (2010). Landmark heuristics for the pancake problem. In *Third Annual Symposium on Combinatorial Search*.

Korf, R. E. (1985). Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, *27*(1), 97–109.

Korf, R. E., & Taylor, L. A. (1996). Finding optimal solutions to the twenty-four puzzle. In *Proceedings of the AAAI National Conference on Artificial Intelligence (AAAI-96)*, pp. 1202–1207.

Pearl, J., & Kim, J. H. (1982). Studies in semi-admissible heuristics. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *4*(4), 392–399.

Pohl, I. (1970). Heuristic search viewed as path finding in a graph. *Artificial Intelligence*, *1*(3-4), 193–204.

Ruml, W., & Do, M. B. (2007). Best-first utility-guided search.. In *IJCAI*, pp. 2378–2384.

Russell, S. J., & Norvig, P. (2016). *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited,.

Sturtevant, N., Felner, A., & Helmert, M. (2017). Value compression of pattern databases. In *Proceedings of the AAAI National Conference on Artificial Intelligence (AAAI-17)*, pp. 912–918.

Thayer, J. T., Ruml, W., & Kreis, J. (2009). Using distance estimates in heuristic search. In *Nineteenth International Conference on Automated Planning and Scheduling*.

Wilt, C., Thayer, J., & Ruml, W. (2011). Selecting a greedy search algorithm. Tech. rep., University of New Hampshire.

Wilt, C. M., & Ruml, W. (2011). Cost-based heuristic search is sensitive to the ratio of operator costs. In *Proceedings of the Symposium on Combinatorial Search (SoCS-11)*.

Wilt, C. M., Thayer, J. T., & Ruml, W. (2010). A comparison of greedy search algorithms. In *third annual symposium on combinatorial search*.