Heuristics in the Longest Simple Path Problem

Justin J. Xie[1]

[1]Institute for Computing in Research

Abstract

Finding the Longest Simple Path (LSP) in a unit-weighted undirected graph is an NP-Hard problem. We have explored the effectiveness of various heuristics for the LSP problem, and their potential for further use. In our exploration, we implemented heuristics inspired by well-known greedy techniques. Other heuristics utilized novel graph features to improve performance. We benchmarked the performance of these heuristics on accuracy, error, and speed on small random graphs. We found that these heuristics provided a considerable speed improvement to the naïve brute-force solution. We found that sparse graphs were the most difficult for our heuristics. This result supports other research indicating that sparse graphs continue to provide challenges for heuristics and approximation algorithms to solve.

1 Introduction

The Longest Simple Path (LSP) problem is a long-standing problem of both theoretical and practical interest. A path is simple if it does not repeat vertices. Finding the longest simple path in a graph involves finding the simple path with maximum length.

Much work has been done on variations of the LSP problem. Solutions to the longest paths in directed, acyclic graphs and trees are the most commonly referenced and utilized solutions. In our research, we set out to explore the LSP problem in undirected unit-weighted connected graphs. The LSP problem is NP-hard: there is no existing solution or computer capable of finding the LSP of an arbitrary graph in polynomial time. As such, heuristic approximation might be an attractive alternative to searching for perfect solutions. In our research, we explored heuristics for this purpose.

Our main research contribution is the development and examination of high-level heuristics and principles of two different kinds: variations of the classic greedy approach, and heuristics involving constructing spanning trees from the graph in order to utilize Dijkstra's LSP algorithm for trees [3]. These heuristics classified a graph's vertices as "peripheral" and "central", using this classification along with other graph features to determine which edges to cut during tree construction.

We tried five heuristics that fall into one of the two categories mentioned above. We experimented with the implementation of these heuristics and obtained benchmark results that allow comparison to past and future explorations of the LSP problem. The data we gathered on these heuristics allowed us to gain some insight into sources of difficulty in LSP construction. We found that each heuristic had its own points of failure. The greedy algorithm failed to break ties; the graph pruning heuristic may not be cutting the most optimal edges. These issues can be further explored and possible remedies can be found for more efficient and accurate heuristics.

We found that when compared to past proposed greedy heuristics, our greedy heuristic showed similar behaviors when the number of edges in a graph was increased. It performed adequately in trees and denser graphs, where many solutions were available, but struggled with certain sparse graphs that only contained one or two possible longest simple paths. The graph pruning heuristics also showed promise for future use. On their own, they struggled to consistently cut the graph into a tree containing the longest simple path.

Our research has shown that heuristics, although far faster than brute-force algorithms, often fail to account for the bigger picture. In our exploration of the LSP problem, the greedy and graph pruning/periphery heuristics that only registered and factored in data local to their "current position" in the graph required few calculations, making them faster. Longest simple paths, however, often require a view of the graph in its entirety in order to find the exact solution. Thus, heuristics that factor in all of the graph or make an extra calculation when deciding on a "next vertex" are the ones that outperformed in the accuracy category when compared to similar heuristics.

This is also the reason for the heuristics' improved ability to find longest paths on dense graphs in comparison to sparse graphs. Dense graphs give heuristics far more options for possible longest simple paths. As a result, even when the heuristics do not consider the entirety of the graph, the path they follow is more likely to be the LSP. In sparse graphs, there are often only one or two longest simple paths, where a single wrong turn in a heuristic's path will lead to an incorrect LSP. As has been confirmed by several other papers and their heuristics, the sparse graphs continue to be the more challenging aspect of implementing heuristics into the longest simple path problem.

2 Background

The longest simple path (LSP) in a graph is the longest possible path that originates at any vertex and traverses the graph without repeating vertices. The problem is an NP-Hard problem: there is no known algorithm capable of finding the LSP on an arbitrary graph in polynomial time. As the number of vertices and the size of the graph increases, any algorithm that guarantees an accurate solution hits an "exponential wall" where even the fastest and most optimized hardware struggle to efficiently identify the longest simple path in an efficient manner.

3 Related Work

The longest simple path problem is a constantly revisited problem. While no general solution has been found, continual research into the problem has found polynomial-time solutions or even linear time solutions for many classes of graph.

One such class is the directed acyclic graph. The longest path of these graphs can be found in linear time by inverting each edge's weight and solving it as a shortest path problem using topological sort. [15]

Trees—undirected, acyclic graphs—are solvable in linear time using an algorithm by Dijkstra that we call the "Dijkstra Dangle". This algorithm finds two "extremes" of a tree using two breadth first searches: the path between them is the LSP for the tree [3].

Block graphs and weighted trees have been shown to have linear time solutions; Cacti graphs in $\mathcal{O}(n^2)$ time [17, 18]. Bipartite Permutation graphs, too, have an $\mathcal{O}(n)$ solution [19]. Interval graphs are solvable in $\mathcal{O}(n^4)$ time [7, 6], Circular-Arc graphs in $\mathcal{O}(n^4)$ time [9], Co-Comparability graphs in $\mathcal{O}(n^4)$ time [10], and Ptolemaic graphs in $\mathcal{O}(n^5)$ time [16].

The LSP problem on general graphs has been intensively studied by many. One notable approach is dynamic programming using graph partitioning [2], including parallelization [5]. Another approach uses constraint programming, with both an exact algorithm and a tabu local-search algorithm: this is more successful in niche groups of graphs [11].

Approximations and heuristics for LSP have also been a big consideration. Approximation has been applied to specific graph groups without polynomial time solutions, such as grid graphs [20], (approximated in $O(n^2)$ time), and also to more general cases. Examples of approximation algorithms include genetic algorithms [8, 12, 13], a color-coding [1] inspired approximation [14], a simulated annealing algorithm [13], and a tabu-search algorithm [13].

Two heuristic approaches similar to ours were a graph pruning heuristic and a greedy variation heuristic. A permissible grid-graph pruning heuristic was used with A* and Depth-First Branch-and-Bound complete search algorithms to find the LSP [4]. A greedy variation heuristic called the k-step Greedy Lookahead employed a method of finding partial paths by looking k vertices ahead [13].

4 Heuristics

We developed several heuristic approximations to LSP for undirected unit-weighted connected graphs. These heuristics employed past techniques, such as the greedy or pruning algorithms, but utilized different graph characteristics. In the process of applying and testing our heuristics, we confirmed properties of the LSP problem introduced in previous work. We also discovered certain graph characteristics that could be helpful for future work.

4.1 Greedy Heuristics

The first group of heuristics that were explored were ones based on the classic greedy algorithm. These algorithms depend on making the "best" decision in the current situation. In order to utilize this principle, there needed to be a graph characteristic that was identifiable and able to be recalculated on a local scale. This led us to the graph characteristic of vertex neighbor counts. A vertex neighbor count is the number of vertices adjacent to the vertex whose neighbor count is being calculated. This was a feature of the graph that could be recomputed every step.

The greedy heuristic utilizing the graph's vertex neighbor count feature, or the Greedy Neighbor heuristic, found the longest simple path in a graph by starting at each vertex. It would then traverse the graph by choosing adjacent vertices with the least available neighbors until there were no options left. Each step in the traversal required a recalculation of the available vertices along with their neighbor count. Each starting node would eventually end up with a greedy traversal of the graph, which was then used to calculate the greedy path from each vertex. Finally, the longest greedy path was output as the heuristics solution to the LSP problem in the given graph.

The greedy heuristic was based on the options for the next vertex in traversing the graph. The heuristic would prioritize visiting vertices with fewer neighbors and thus a lower number of "options" by which it could be reached. Consequentially, vertices with more neighbors were visited later, as there were more "options" to get to them. The problem with this heuristic was that it was often in a position where two or more vertices adjacent to the current vertex had the same number of available neighbors, putting the heuristic at a crossroads of tie breaking. In our

Algorithm 1 Greedy Neighbor Heuristic

```
let $G$ be the given graph
longest_path $\leftarrow$ empty path
**for** vertex in $G$ **do**
    current_path $\leftarrow$ empty path
    **while** adjacent vertices to last vertex exist **do**
        **if** $V$ with least neighbors and neighbor count $> 1$ **then**
            add $V$ to current_path
        **else if** $V$ with neighbor count $= 1$ **then**
            add $V$ to current_path
        **else**
            end traversal
        **end if**
    **end while**
    **if** length of current_path $>$ length of longest_path **then**
        longest_path $\leftarrow$ current_path
    **end if**
**end for**
return vertices_visited - 1
```

first iteration of the Greedy Neighbor heuristic, the vertex with a lower ID was chosen. While this sometimes was inconsequential, because in many cases both paths led to longest paths being found, there were also graphs where choosing the lower node ID led to a shorter path length than the desired LSP. This behavior was most noticeable in larger and sparse graphs with seven or more vertices. There needed to be a tiebreaker that could pick the correct choice instead of letting chance decide.

Algorithm 2 DFS Tiebreaker Function

```
$G \leftarrow$ tree created by DFS
$R \leftarrow$ root vertex of tree
internal path length $\leftarrow 0$
**for** vertex in tree **do**
    $D \leftarrow$ distance from vertex to $R$
    add $D$ to internal path length
**end for**

return internal path length
```

The second version of the Greedy Neighbor heuristic involved a Depth-First-Search (DFS) that acted as the tiebreaker when more than one adjacent vertex had the most neighbors. This DFS was a specialized version that traversed the remaining available vertices in the graph. It prioritized visiting vertices with lower IDs (instead of randomly deciding the next vertex). For each of the tied adjacent vertices, it created a tree graph with the root being the starting DFS vertex. For each vertex that had tied with having the most neighbors, the specialized DFS was executed to create post-DFS trees. The internal path length is the sum of the depth (distance from root) of all nodes. The internal path length of each post-DFS tree was calculated and the one with the highest was chose as the next vertex to visit in the greedy traversal.

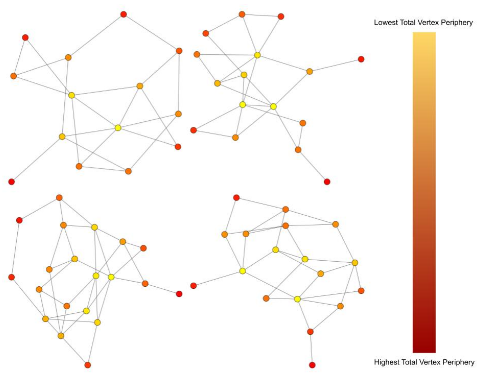

Figure 1: Example Graphs With Gradient Representation of Total Vertex Periphery

4.2 Graph Centrality, Periphery, and Pruning

The second group of heuristics was based on the centrality of the vertices. The definition of three key terms representing graph and vertex centrality are as follows:

1. **Vertex Periphery**: For any vertex v, the vertex periphery (VP) is the longest shortest path from v to any other vertex. The lower the vertex periphery of v, the more central v is.

2. **Total Vertex Periphery**: For any vertex v, the total vertex periphery (TVP) is the sum of all shortest paths from v to every other vertex. The lower the total vertex periphery of v, the more central v is. Figure 1 gives a visual illustration of TVP.

3. **Graph Periphery**: For any graph G, the graph periphery (GP) is the sum of the TVP of all vertices in G.

All graph centrality heuristics were based on the idea that given a graph G with n vertices and m edges, if you cut specific $m-(n-1)$ edges, the graph would then be pruned into a tree T, whose longest path corresponds to the longest path in

G. The longest path of T could then be found in linear time with Dijkstra's Dangle algorithm.

The first graph centrality heuristic was one that made use of the TVP values. Named the Prune Central Edges heuristic, it relied on the assumption that the edges connecting vertices with lower TVP (meaning that they were more central edges) were unnecessary in the LSP of the graph. However, that proved to be an incorrect assumption because during testing, it was found that in certain instances, the edge connecting vertices with the lowest TVP was utilized in the longest simple path.

Algorithm 3 Prune Central Edges

```
let $n$ be the number of vertices
let $m$ be the number of edges
let $G$ be the given graph

for $m - (n - 1)$ cuts do
    $M \leftarrow$ mapping of vertex TVP
    $V \leftarrow$ vertex with lowest TVP
    for neighbors of $V$ do
        $E \leftarrow$ edge from $V$ to neighbor
        temporarily remove $E$
        if $G$ is connected then
            remove $E$ from $G$
        else
            next edge
        end if
    end for
end for

execute Dijkstra's Dangle on cut graph
return longest path obtained by Dijkstra's
```

The second graph centrality heuristic also made use of the total vertex periphery values. The Stretch Lowest TVP heuristic worked by choosing a vertex v with the lowest TVP that had available edges that could be removed without disconnecting the graph. Next, the heuristic would find and cut the edge connecting to v that, when removed, would increase the TVP of v the most. This heuristic operated under the idea that the optimal tree—containing the longest path—would have the highest possible graph periphery. By removing edges from the vertices with low TVP, the heuristic would be increasing the TVP of the vertices and stretching the GP of the graph.

Through our testing, it was found that in certain graphs, the edges that would increase the TVP of a vertex were edges that were a crucial part of the graph's LSP. As a result, the principle that this exact heuristic followed was not the correct way to go about cutting the graph into a tree.

The third graph centrality heuristic made use of the graph periphery (GP). The Stretch Graph Periphery heuristic operated by looping through all removable edges (ones that did not create discontinuities in the graph) and finding the one that brought about the largest increase in the

Algorithm 4 Stretch Lowest TVP

```
let $n$ be the number of vertices
let $m$ be the number of edges

for $m - (n - 1)$ cuts do
    $V \leftarrow$ vertex with lowest TVP
    initialize highest periphery
    for edge connected to $V$ do
        temporarily remove edge
        find TVP
        if TVP $>$ highest periphery then
            highest periphery $\leftarrow$ TVP
        end if
        remove edge with highest periphery
    end for
end for

execute Dijkstra's Dangle on cut graph
return longest path obtained by Dijkstra's
```

GP when removed. Similar to the Stretch Lowest TVP heuristic, the Stretch Total Graph heuristic also worked to remove edges that lowered the graph periphery and to stretch the graph out to its widest possible tree.

Algorithm 5 Stretch Graph Periphery

```
let $n$ be the number of vertices
let $m$ be the number of edges
let $G$ be the given graph

for $m - (n - 1)$ cuts do
    initialize highest periphery
    for all edges in $G$ do
        temporarily remove edge
        $P \leftarrow$ graph periphery
        if $P >$ highest periphery then
            highest periphery $\leftarrow P$
        end if
        remove edge with highest periphery
    end for
end for

execute Dijkstra's Dangle on cut graph
return longest path obtained by Dijkstra's
```

Similar to the Stretch Lowest TVP heuristic, it was revealed that there were graphs that did not work well for the Stretch Total Graph heuristic. There were certain graphs where the edge that increased the GP the most were part of the longest simple path. Other graphs had edges that increased the GP the same. With no tiebreaker method, the edge with lower vertex IDs would be chosen to be removed. As such, there were graphs where necessary edges were removed because of ties between edges.

5 Experimental Results

As we designed our heuristics, we experimented with prototype implementations. We eventually

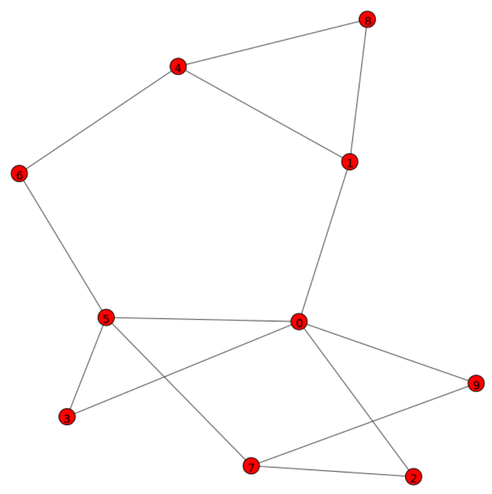

Figure 2: Example Graph, Drawn by Python-igraph

constructed a benchmark suite for a more thorough evaluation.

5.1 Experiment Setup

In order to implement our heuristics, we utilized the Python programming language (version 3.10.4) and its `igraph` package. The `igraph` package contains an extensive array of functions and variable types fit for quick and efficient manipulation as well as useful graph visualization tools. We utilized `igraph`'s ability to quickly add and remove edges and vertices, as well as its functions for plotting graphs and returning important graph characteristics such number of edges, vertices, or a vertex's neighbors. The package also included functions such as `get_simple_paths` or `shortest_paths` that made the translation of heuristics from idea to code a straightforward process.

We used `igraph`'s visualization tools to draw nicely-layed-out graphs such as the one in Figure 2. Working with graphs that were visualized this way helped to in understanding different implementations of heuristics like the greedy heuristic and new characteristics.

5.2 Experimental Approach

The first step in implementing heuristics was the proof-of-concept. By looking at characteristics of a graph, we were able to find some that might be of use in a heuristic. Characteristics such as vertex neighbor count, diameter of the graph, and vertex centrality were all examples of graph characteristics that were involved in one or more heuristics. We often used graph features for inspiration when pondering possible heuristics.

When heuristic ideas revealed themselves to be feasible and capable of attaining accurate longest simple paths, pseudocode was written to be used in the second step: Python implementation. This was where the ideas were implemented as heuristics compatible with `igraph` for testing.

5.3 Random Graph Generation

The random graphs for visualization and benchmarking were created through a randomization process called Tree-Based-Graph-Gen. This algorithm, given a n vertices and m edges, creates a random, undirected, connected graph by creating a tree with all n vertices and $n-1$ edges. The remaining $m-(n-1)$ required edges are then added randomly throughout the graph.

Algorithm 6 A random graph generation algorithm

```
let $E$ be a set of all possible edges $(i, j)$
for vertex $i$ from 2 to $n$ do
    let $j$ be a random number from $i$ to $(i-1)$
    connect $i$ to $j$
    remove $(i, j)$ from $E$
end for
for $m-(n-1)$ edges do
    if $E$ is empty then
        return fail
    end if
    pick random edge $(i-1)$ from $E$
    connect $i$ to $j$
    remove $(i, j)$ from $E$
end for
```

This "random" graph generation, however, is not truly random. Due to the way that the algorithm chooses "random" target edges during the base-tree creation phase, certain graph shapes are more likely to be built than others.

5.4 Benchmarking Methods

For the testing and benchmarking of our heuristics, we created three tests to measure crucial areas of the Longest Simple Path Problem: accuracy, error, and runtime. For consistency, the graphs used throughout all tests were the same graphs. The tests were as follows:

1. **Accuracy:** The percentage of graphs that a given heuristic gets the exact longest simple paths for.
2. **Error:** The average difference between the actual longest simple path and longest simple path found by a given heuristic.
3. **Runtime:** The wall-clock average execution time of a given heuristic over a group of graphs.

5.5 Experimental Results

The accuracy benchmark confirmed the findings of several past experiments. All heuristics were seen to have a "canyon" behavior. At any vertex count, as the edge count increases, the accuracy of the heuristics decrease to a lower bound, then increases after hitting the lower bound. In other words, the LSP of tree graphs, and dense graphs were a accurately found by the heuristics, while the sparse and non-tree graphs were the difficult

Table 1: Heuristics Accuracy on 9-Vertex Graphs

Number of Edges	Heuristics				
	greedy	dfs_greedy	prune_central	stretch_total	stretch_nodes
8	100.0	100.0	100.0	100.0	100.0
9	99.0	100.0	98.0	99.0	99.0
10	99.0	100.0	96.0	91.0	92.0
11	99.0	98.0	92.0	90.0	90.0
12	100.0	100.0	91.0	78.0	84.0
13	100.0	99.0	83.0	79.0	81.0
14	99.0	100.0	92.0	86.0	88.0
15	99.0	100.0	86.0	84.0	86.0
16	100.0	100.0	91.0	88.0	81.0
17	100.0	100.0	98.0	88.0	89.0
18	100.0	100.0	96.0	96.0	87.0
19	100.0	100.0	98.0	96.0	91.0
20	100.0	100.0	100.0	99.0	95.0
21	100.0	100.0	99.0	99.0	94.0
22	100.0	100.0	99.0	98.0	97.0
23	100.0	100.0	100.0	94.0	96.0
24	100.0	100.0	100.0	100.0	96.0
25	100.0	100.0	100.0	97.0	96.0
26	100.0	100.0	100.0	99.0	95.0
27	100.0	100.0	100.0	100.0	97.0
28	100.0	100.0	100.0	99.0	94.0
29	100.0	100.0	100.0	100.0	97.0
30	100.0	100.0	100.0	100.0	94.0
31	100.0	100.0	100.0	100.0	98.0
32	100.0	100.0	100.0	100.0	96.0
33	100.0	100.0	100.0	100.0	98.0
34	100.0	100.0	100.0	100.0	100.0
35	100.0	100.0	100.0	100.0	100.0
36	100.0	100.0	100.0	100.0	100.0

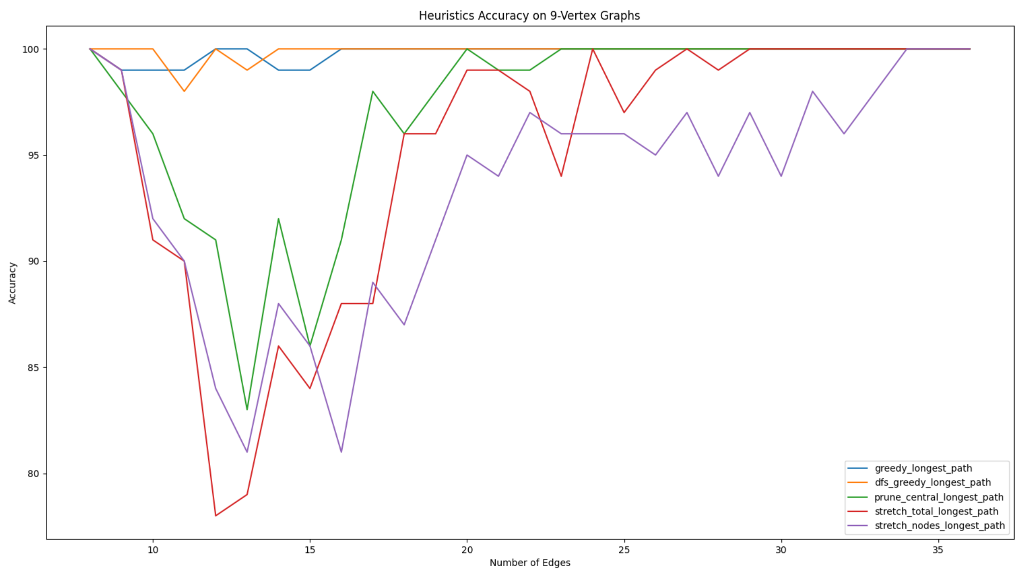

Figure 3: Heuristics Accuracy on Graphs with 9 Vertices

Table 2: Heuristics Error on 9-Vertex Graphs

Number of Edges	Heuristics				
	greedy	dfs_greedy	prune_central	stretch_total	stretch_nodes
8	0.0	0.0	0.0	0.0	0.0
9	0.01	0.0	0.02	0.01	0.01
10	0.01	0.0	0.04	0.09	0.08
11	0.01	0.02	0.08	0.10	0.10
12	0.0	0.0	0.09	0.23	0.16
13	0.0	0.01	0.18	0.21	0.19
14	0.01	0.0	0.08	0.14	0.12
15	0.01	0.0	0.14	0.16	0.14
16	0.0	0.0	0.19	0.13	0.19
17	0.0	0.0	0.02	0.12	0.11
18	0.0	0.0	0.04	0.04	0.13
19	0.0	0.0	0.02	0.05	0.09
20	0.0	0.0	0.0	0.01	0.05
21	0.0	0.0	0.01	0.01	0.06
22	0.0	0.0	0.01	0.02	0.03
23	0.0	0.0	0.0	0.06	0.04
24	0.0	0.0	0.0	0.0	0.04
25	0.0	0.0	0.0	0.03	0.04
26	0.0	0.0	0.0	0.01	0.05
27	0.0	0.0	0.0	0.0	0.03
28	0.0	0.0	0.0	0.01	0.06
29	0.0	0.0	0.0	0.0	0.03
30	0.0	0.0	0.0	0.0	0.06
31	0.0	0.0	0.0	0.0	0.02
32	0.0	0.0	0.0	0.0	0.04
33	0.0	0.0	0.0	0.0	0.02
34	0.0	0.0	0.0	0.0	0.0
35	0.0	0.0	0.0	0.0	0.0
36	0.0	0.0	0.0	0.0	0.0

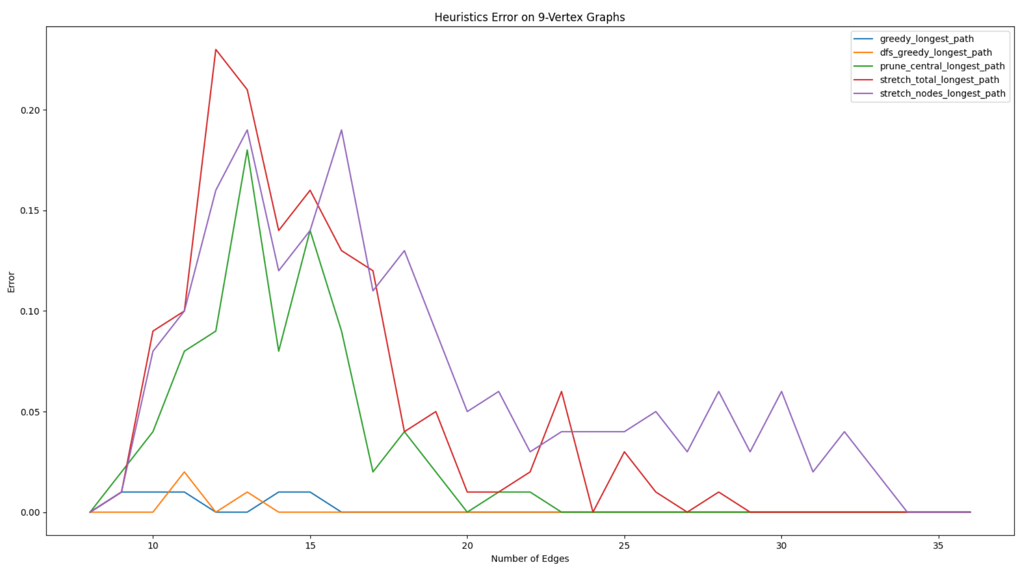

Figure 4: Heuristics Error on Graphs with 9 Vertices

Table 3: Heuristics Runtime on 9-Vertex Graphs

Number of Edges	Heuristics					
	greedy	dfs_greedy	prune_central	stretch_total	stretch_nodes	brute-force
8	0.0001339	0.0002115	1.78E-05	1.24E-05	1.28E-05	3.93E-05
9	9.44E-05	0.0003868	0.0002032	0.0011236	0.0003383	5.81E-05
10	0.0001048	0.0005775	0.0004151	0.0023543	0.0006561	9.03E-05
11	0.0001125	0.0007197	0.0005731	0.0037371	0.0009669	0.0001347
12	0.0001214	0.0008422	0.0008724	0.0053146	0.0013564	0.0002133
13	0.0001287	0.0009806	0.0009262	0.0071409	0.0016779	0.0003941
14	0.000135	0.0011354	0.0011374	0.0090412	0.0020774	0.000534
15	0.0001397	0.0013132	0.0012995	0.0109298	0.0024597	0.000801
16	0.0001446	0.0013653	0.0014624	0.0131174	0.00292	0.0011781
17	0.0001484	0.0015664	0.0016971	0.015527	0.0033403	0.0017935
18	0.0001527	0.0017702	0.0018301	0.0182468	0.0038113	0.0027262
19	0.0001562	0.0020146	0.0020375	0.0209624	0.0043504	0.0043898
20	0.0001597	0.002153	0.0021626	0.0235532	0.0048087	0.0065727
21	0.0001618	0.0023596	0.0023787	0.0255448	0.0053355	0.0092391
22	0.0001643	0.00258	0.0025342	0.0292619	0.0058146	0.0133933
23	0.0001676	0.0027988	0.0027264	0.0325536	0.0064107	0.0192997
24	0.0001705	0.0030494	0.0030376	0.0360228	0.0069574	0.025796
25	0.0001729	0.0033103	0.0029793	0.0395946	0.0075332	0.03686
26	0.0001746	0.0034828	0.0031932	0.0434956	0.0080365	0.0492349
27	0.0001771	0.0037537	0.0033473	0.0471843	0.0085708	0.0655757
28	0.0001789	0.0040668	0.0034598	0.0520643	0.0092661	0.0890664
29	0.0001811	0.0044	0.0035829	0.0569198	0.0097569	0.1137601
30	0.0001832	0.0047944	0.0037833	0.0620864	0.0103405	0.1465667
31	0.0001856	0.0052211	0.0040176	0.0665662	0.0112145	0.1921401
32	0.0001874	0.0057643	0.0041552	0.0722577	0.011959	0.2542703
33	0.0001889	0.0063387	0.0044217	0.0775441	0.0127085	0.3268791
34	0.0001915	0.0074702	0.0045202	0.0832664	0.013976	0.4175168
35	0.000193	0.0085568	0.0046966	0.08821	0.0146478	0.5437159
36	0.0001936	0.010597	0.0053172	0.0936378	0.0153948	0.7104294

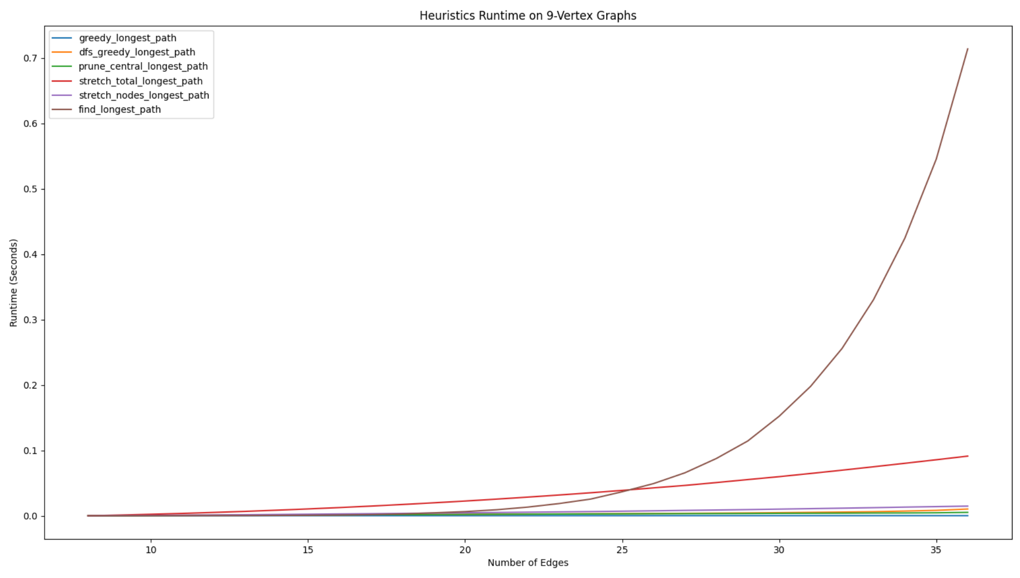

Figure 5: Heuristics Runtime on Graphs with 9 Vertices

graphs to find a correct LSP for. This confirms the claims of other studies that have found the sparse graphs to be the real challenge instances for the LSP problem.

In addition, the accuracy tests also revealed that both variants of the greedy algorithm, with and without the depth-first-search tiebreaker, were both guaranteed to get the correct LSP in dense graphs, which also confirms previous findings. For graph pruning, the accuracy shown was worse than the greedy heuristics, signalling that, as stand-alone heuristics, they are not as effective as greedy heuristics. However, it is possible that implementing them in tandem with something like A* or local search could produce desirable results.

The runtime tests we administered showed that in comparison to the naive, brute-force solution, all heuristic runtimes grew at a slower pace. The brute-force solution has a clear exponential growth in runtime as the number of edges increases, while the heuristics have runtimes that increase more gradually. The one exception to this behavior was the heuristic that stretched the graph periphery. Because of its need to loop through all edges each time, in sparse graphs, it has a slower runtime than the brute-force algorithm. Furthermore, as the number of vertices increased, the stretching heuristic's runtime also increased by a larger degree than other heuristics. Despite its relative inefficiencies, the stretching graph periphery heuristic was still more efficient than the naive solution.

6 Conclusions

The difficulties of the longest simple path problem are many and varied. We explored possible implementations of heuristics for more efficient solutions or approximations to finding the LSP. Through our experimentation, we confirmed past findings about the greedy heuristics abilities to solve the LSP problem on dense graphs, and the difficulty that sparse graphs bring to the problem. We also discussed the reasoning behind the various graph pruning heuristics. While these heuristics have so far been less effective, there is still a large space to explore.

Acknowledgements

I would like to thank Bart Massey and Cassaundra Smith for their help in the research into the longest path problem and writing of this paper. I would also like to thank Mark Galassi for providing the opportunity to be an intern at the Insitute for Computing in Research.

References

[1] Noga Alon, Raphael Yuster, and Uri Zwick. Color-coding. *Journal of the ACM (JACM)*, 42(4):844–856, 1995.

[2] Tomas Balyo, Kai Fieger, and Christian Schulz. Optimal longest paths by dynamic programming. *arXiv preprint arXiv:1702.04170*, 2017.

[3] Eindhoven Tuesday Afternoon Club, RW Bulterman, FW van der Sommen, G Zwaan, T Verhoeff, AJM van Gasteren, and WHJ Feijen. On computing a longest path in a tree. *Information Processing Letters*, 81(2):93–96, 2002.

[4] Yossi Cohen, Roni Stern, and Ariel Felner. Solving the longest simple path problem with heuristic search. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 30, pages 75–79, 2020.

[5] Kai Fieger, Tomas Balyo, Christian Schulz, and Dominik Schreiber. Finding optimal longest paths by dynamic programming in parallel. In *Twelfth Annual Symposium on Combinatorial Search*, 2019.

[6] Archontia C Giannopoulou, George B Mertzios, and Rolf Niedermeier. Polynomial fixed-parameter algorithms: A case study for longest path on interval graphs. *arXiv preprint arXiv:1506.01652*, 2015.

[7] Kyriaki Ioannidou, George B Mertzios, and Stavros D Nikolopoulos. The longest path problem is polynomial on interval graphs. In *International Symposium on Mathematical Foundations of Computer Science*, pages 403–414. Springer, 2009.

[8] Tom V Mathew. Genetic algorithm. *Report submitted at IIT Bombay*, 2012.

[9] George B Mertzios and Ivona Bezakova. Computing and counting longest paths on circular-arc graphs in polynomial time. *Electronic Notes in Discrete Mathematics*, 37:219–224, 2011.

[10] George B Mertzios and Derek G Corneil. A simple polynomial algorithm for the longest path problem on cocomparability graphs. *SIAM Journal on Discrete Mathematics*, 26(3):940–963, 2012.

[11] Quang Dung Pham and Yves Deville. Solving the longest simple path problem with constraint-based techniques. In *International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) Techniques in Constraint Programming*, pages 292–306. Springer, 2012.

[12] David Portugal, Carlos Henggeler Antunes, and Rui Rocha. A study of genetic algorithms for approximating the longest path in generic graphs. In *2010 IEEE International Conference on Systems, Man and Cybernetics*, pages 2539–2544. IEEE, 2010.

[13] John Kenneth Scholvin. Approximating the longest path problem with heuristics: A survey. Master's thesis, University of Illinois at Chicago, 1999.

[14] Maria Grazia Scutella. An approximation algorithm for computing longest paths. *European Journal of Operational Research*, 148(3):584–590, 2003.

[15] Robert Sedgewick and Kevin Daniel Wayne. *Algorithms*. W. Ross MacDonald School Resource Services Library, 2017.

[16] Yoshihiro Takahara, Sachio Teramoto, and Ryuhei Uehara. Longest path problems on ptolemaic graphs. *IEICE transactions on information and systems*, 91(2):170–177, 2008.

[17] Ryuhei Uehara and Yushi Uno. Efficient algorithms for the longest path problem. In *International symposium on algorithms and computation*, pages 871–883. Springer, 2004.

[18] Ryuhei Uehara and Yushi Uno. On computing longest paths in small graph classes. *International Journal of Foundations of Computer Science*, 18(05):911–930, 2007.

[19] Ryuhei Uehara and Gabriel Valiente. Linear structure of bipartite permutation graphs and the longest path problem. *Information Processing Letters*, 103(2):71–77, 2007.

[20] Wen-Qi Zhang and Yong-Jin Liu. Approximating the longest paths in grid graphs. *Theoretical Computer Science*, 412(39):5340–5350, 2011.