Revisiting Heuristics for the Longest Simple Path Problem

Justin J. Xie *

Westview High School
Portland, OR 97229
justinjxie@gmail.com

Abstract

Finding the Longest Simple Path (LSP) in a unit-weighted undirected graph is an NP-Hard problem. Therefore, effective heuristics for this problem are highly desirable. In this paper, we revisit this topic and explore the effectiveness of several heuristics for the LSP problem and their potential for further use. In our exploration, we implemented heuristics that are inspired by well-known greedy techniques as well as heuristics that utilize novel graph features to improve performance. We benchmarked the accuracy, error, and speed of these heuristics on small random graphs. We found that these heuristics provide a considerable speed improvement over the naïve brute-force solution. We also found that sparse graphs are the most difficult for our heuristics. This result reconfirms prior research indicating that sparse graphs continue to post challenges for LSP heuristics and approximation algorithms to solve.

1 Introduction

The Longest Simple Path (LSP) problem is a long-standing problem of both theoretical and practical interests. 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 [6]. Much work has been done on variations of the LSP problem. Solutions for directed, acyclic graphs and trees are the most commonly referenced and utilized [17, 3]. However, the LSP problem in undirected unit-weighted connected graphs is NP-hard. As such, effective heuristics are highly sought-after.

In this research, we revisit the LSP problem for undirected unit-weighted connected graphs and explore several heuristics for this problem. We have implemented and evaluated heuristics of two major categories: variations of the classic greedy approach and graph pruning heuristics involving constructing spanning trees from the graph in order to utilize Dijkstra's LSP algorithm for trees [3]. The latter graph pruning heuristics classify a graph's vertices as "peripheral" or "central" and use the classification with other graph features to determine which edges to cut during tree construction.

We have experimented with the implementations of five heuristics that fall into these two categories and obtained benchmark results that allow comparisons to past and future explorations of the LSP problem. We found that when compared to previously proposed greedy heuristics, our greedy heuristics showed similar behaviors when the number of edges in a graph increased. These heuristics performed adequately in trees and denser graphs but struggled on sparser graphs due to challenges in tie-breaking effectively. When testing the graph pruning heuristics, we found that they may not cut the most optimal edges. However, these heuristics were effective at reducing complex graphs into graphs where algorithms such as A* and local search can be executed. These problems can be further explored and possible remedies can be found for more effective heuristics.

Through this study, we gained some insight into sources of difficulty in the LSP construction. Our research has shown that these 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 heuristics that only register and factor in data local to their "current position" in the graph require few calculations, making them faster. The LSP problem, however, often requires a view of the graph in its entirety in order to find a correct 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 outperform in the accuracy category when compared to other heuristics.

*This research was done during the author's internship at the Institute for Computing in Research during summer 2022.

This may also be 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 one of the LSP. In sparse graphs, there are often only one or few longest simple paths, where a single wrong turn by a heuristic will lead to an incorrect solution. As has been noted by previous research [9], the sparse graphs continue to be the more challenging aspect of implementing heuristics for the LSP problem.

2 Related Work

The LSP problem is frequently revisited. 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 graphs. One such class is the directed acyclic graph. The longest path of such a graph can be found in linear time by inverting each edge's weight and solving it as a shortest path problem using a topological sort [17]. Trees (undirected, acyclic graphs) are solvable in linear time using an algorithm proposed by Edsger Dijkstra. This algorithm finds two "extremes" of a tree using two breadth first searches and the path between these two "extremes" would be the LSP for the tree [3].

More specialized graphs have also been studied. Block graphs and weighted trees have been shown to have linear time solutions, Cacti graphs in $\mathcal{O}(n^2)$ time [19, 20], Bipartite Permutation graphs in $\mathcal{O}(n)$ time [21], interval graphs in $\mathcal{O}(n^4)$ time [8, 7], Circular-Arc graphs in $\mathcal{O}(n^4)$ time [11], Co-Comparability graphs in $\mathcal{O}(n^4)$ time [12], and Ptolemaic graphs in $\mathcal{O}(n^5)$ time [18].

The LSP problem on general graphs has been intensively studied by many as well. One notable approach is dynamic programming using graph partitioning [2] and later also using parallelization [5]. Another approach uses constraint programming, with both an exact algorithm and a tabu local search algorithm which are more successful in niche groups of graphs [13].

Approximations and heuristics for LSP have also been a major consideration. Approximation has been applied to specific graph groups without polynomial time solutions, such as grid graphs [22] (approximated in $O(n^2)$ time), and also to more general cases. Examples of approximation algorithms include genetic algorithms [10, 14, 15], a color-coding [1] inspired approximation [16], a simulated annealing algorithm [15], and a tabu-search algorithm [15].

Two heuristic approaches close to ours were a graph pruning heuristic and a variation of greedy heuristic. The permissible grid-graph pruning heuristic has been used with A* and Depth-First Branch-and-Bound complete search algorithms to find the LSP [4]. The variation of greedy heuristic, namely the k-step Greedy Lookahead, employs a method of finding partial paths by looking ahead k vertices [15].

3 Heuristics

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

3.1 Greedy Heuristics

The first group of heuristics that we explored were based on greedy algorithms. These algorithms depend on making the "best" decision given a "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 *unvisited neighbor count* of a vertex, which is the number of a vertex's neighbors that have yet to be visited by the algorithm.

As shown in Algorithm 1, the Greedy Neighbor Heuristic algorithm finds the LSP in a graph by starting at each vertex and traverses the graph by choosing adjacent vertices with the least unvisited neighbors until there are no options left. Each step in the traversal requires a recalculation of the unvisited vertices along with their unvisited neighbor counts. Each starting vertex will eventually end up with a greedy traversal of the graph, which is then is used to calculate the greedy path from each vertex. Finally, the longest greedy path is output as the heuristic's solution to the LSP problem for the given graph.

The greedy heuristic is based on the options for the next vertex in traversing the graph. It prioritizes visiting vertices with fewer neighbors and thus a lower number of "options" by which that vertex can be reached. Consequentially, vertices with more neighbors are visited later because there are more "options" to get to them. The problem with this al-

Algorithm 1 Greedy Neighbor Heuristic

```
let G be the given graph
longest_path ← empty path

for each v in G do
    add v to current_path
    mark v as visited
    last_vertex ← v
    while last_vertex has unvisited neighbors do
        S ← set of unvisited neighbors of last_vertex
        select w from S where w has non-zero, least
unvisited neighbors (apply tie breaker if needed)
        if w is none then
            select w from S where w has zero unvis-
ited neighbors
        end if
        add w to current_path
        mark w as visited
        last_vertex = w
    end while
    if current_path longer than longest_path then
        longest_path ← current_path
    end if
    mark all vertices as unvisited
end for
return longest_path
```

gorithm is that it is often in a position where two or more vertices adjacent to the current vertex have the same number of unvisited neighbors, putting the heuristic at a crossroad of tie-breaking. In our first iteration of the Greedy Neighbor heuristic, the heuristic defaults to choosing the vertex with a lower ID (randomly assigned number). While this is inconsequential in many cases, there are graphs where choosing the lower node ID leads to a shorter length path than the desired LSP. This behavior is most noticeable in larger sparse graphs with seven or more vertices. There needs to be a tiebreaker more effective than choosing the lower ID.

As shown in Algorithm 2, we introduced a tiebreaker function. Given a vertex in the graph, the algorithm first traverses the unvisited vertices in the graph using a specialized Depth-First-Search (DFS) that generates a tree rooted at the given vertex. The specialized DFS traverses unvisited vertices in the graph prioritizing vertices with lower IDs. Next, the algorithm returns the internal path length of the tree which is the sum of the depths (distances from root) of all vertices. We then apply this tiebreaker function to Algorithm 1 when two or more vertices have the same lowest number of unvisited neighbors. For each of these vertices, the tiebreaker function is executed

Algorithm 2 DFS Tiebreaker Function

```
T ← tree created by the specialized DFS
R ← root of T

internal_path_length ← 0
for each v in T do
    D ← distance from v to R
    add D to internal_path_length
end for

return internal_path_length
```

and the vertex with the highest internal path length is selected. If this function fails to break the tie, we return to breaking the tie with the lowest vertex ID.

3.2 Graph Centrality, Periphery, and Pruning

The second group of heuristics are based on the centrality of vertices. Below are the definitions of three key terms representing vertex and graph centrality.

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 in a graph.

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 where m is greater than $n-1$, if a specific $m-(n-1)$ number of edges are cut, the graph would 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 algorithm.

As shown in Algorithm 3, the first graph centrality heuristic is the Prune Central Edges heuristic, which utilizes the TVP values. It relied on the assumption that the edges connecting vertices with lower TVPs (meaning that they were more central edges) were unnecessary in the LSP of the graph. However, this proved to be an unreliable assumption because during testing, it was found that in certain instances,

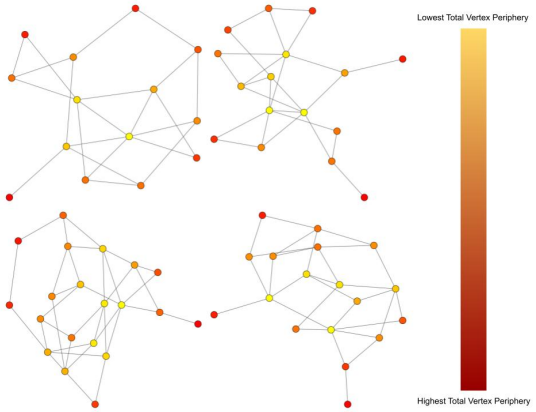

Figure 1: Example Graphs With Gradient Representation of Total Vertex Periphery

the edge connecting vertices with the lowest TVPs is indeed utilized in the longest simple path.

Algorithm 3 Prune Central Edges

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

for i = 0 to m − n do
    M ← mapping of vertices and their TVPs
    v ← vertex with lowest TVP in M
    S ← list of neighbors of v in ascending TVP
    for each neighbor in S do
        E ← edge from v to neighbor
        temporarily remove E
        if G is connected then
            remove E from G
            break
        else
            add E back to G
        end if
    end for
end for

execute Dijkstra's Algorithm on G
return longest path found by Dijkstra's Algorithm
```

As shown in Algorithm 4, the second graph centrality heuristic also utilizes the TVP values. The Stretch Lowest TVP heuristic works by choosing a vertex v with the lowest TVP that has available edges that could be removed without disconnecting the graph. Next, the heuristic finds and cuts the edge connecting to v that, when removed, maximizes the TVP of v. This heuristic operates under the assumption that the optimal tree—containing the longest path—would have the highest possible graph periphery. By removing edges from the vertices with low TVPs, the heuristic increases the TVPs of the vertices and stretches the GP of the graph.

Algorithm 4 Stretch Lowest TVP

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

for i = 0 to m − n do
    v ← vertex with lowest TVP
    highest_TVP ← 0
    for each edge E connected to v do
        temporarily remove E from G
        if G is connected then
            calculate TVP for v
            if TVP > highest_TVP then
                highest_TVP ← TVP
            end if
        end if
        add E back to G
    end for
    remove the edge corresponding to highest_TVP
end for

execute Dijkstra's Algorithm on G
return longest path obtained by Dijkstra's
```

However, through our testing, we found that in certain graphs, some edges that were cut in order to maximize the TVP of a vertex were also a crucial part of the graph's LSP. As a result, the principle that the Stretch Lowest TVP heuristic follows is not always a reliable approach to pruning a graph into a tree.

As shown in Algorithm 5, the third graph centrality heuristic utilizes the Graph Periphery (GP). The Stretch GP heuristic operates by looping through all removable edges (ones that do not create discontinuities in the graph) and cutting ones that maximize the value of the GP. Similar to the Stretch Lowest TVP heuristic, the Stretch GP heuristic also works to remove edges that lower the GP and stretches the graph out to its widest possible tree.

Similar to the Stretch Lowest TVP heuristic, it was revealed that there are graphs that the Stretch GP heuristic is not well suited for. Some of these graphs involve edges that maximize the GP of graph but are also crucial the longest simple path. Other graphs have edges that increase the GP the same. For tie breaking, edges with lower vertex IDs are chosen to be removed. However this tie breaking method sometimes remove edges in the actual longest path.

Algorithm 5 Stretch Graph Periphery

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

for i = 0 to m − n do
    highest_GP ← 0
    for each edge E in G do
        temporarily remove E from G
        if G is connected then
            GP ← graph_periphery
            if GP >highest_GP then
                highest_GP ← GP
            end if
        end if
        add E back to G
    end for
    remove edge with highest_GP
end for

execute Dijkstra's Algorithm on G
return longest path found by Dijkstra's Algorithm
```

Although the three graph centrality properties (VP, TVP, and GP) by themselves cannot serve as the single reliable metric for pruning graphs into tree, they do provide options for handling different types of graphs. With enough computing resources, various alternatives may be pursued in parallel and the one with the best results could be adopted.

4 Experimental Results

We have experimented with the implementations of our heuristics in Python and constructed a benchmark for thorough evaluation of these heuristics and future ones. This benchmark of graphs was created by random graph generation algorithm in Algorithm 6. All experiments were conducted on a Dell Inspiron 14 7000 laptop with an Intel Core i7-4500U processor and 8GB of memory running the Ubuntu 20.04 Operating System.

4.1 Implementation of Heuristics

To implement our heuristics, we used Python 3.10.4 and its `igraph` package. The `igraph` package contains an extensive array of functions and variable types fit for efficient manipulation of graphs. 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 as the number of edges, vertices, or neighbors to a vertex. The package also includes functions such as `get_simple_paths` or `shortest_paths` that make the translation of our heuristics from idea to code a straightforward process. In addition, we used the visualization tools of `igraph` to produce nicely-laid-out graphs such as the one in Figure 2, which helped in our understanding of graph characteristics and the strengths and weaknesses of different heuristics. The implementations of our heuristics can be found at https://github.com/JJ-Xie/Heuristics-Longest-Path-Project.git.

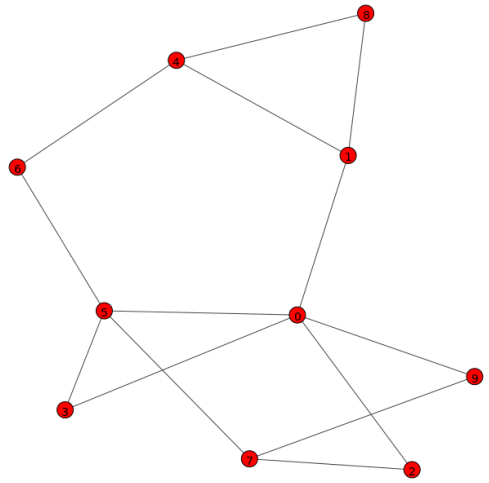

Figure 2: Sample Graph Plotted by Python-igraph

4.2 Random Graph Generation

The random graphs for benchmarking and visualization were created through a randomization process called Tree-Base Graph Generation as shown in Algorithm 6. Given n for the number of vertices and m for the expected number of edges, this algorithm creates a random, undirected, and connected graph by first creating a tree with all n vertices and $n-1$ edges. The remaining $m-(n-1)$ edges are then added randomly throughout the graph. This "random" graph generation, however, has a limitation. 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.

4.3 Benchmarking Criteria

To evaluate and benchmark our heuristics, we used three criteria to measure crucial aspects of the LSP problem: accuracy, error, and runtime. For consistency, the same set of graphs were used throughout

Algorithm 6 Random Graph Generation

let n be the number of given vertices
let m be the number of expected edges, $m \geq n - 1$
let S be the set of all possible edges for n vertices
let G be the graph to be generated

for vertex i from 1 to $n - 1$ **do**
    let j be a random number from 0 to $(i - 1)$
    add edge (i, j) to G
    remove (i, j) from S
end for

if $m = n - 1$ **then**
    return G
end if

for $k = 0$ to $m - n$ **do**
    if S is empty **then**
        return error
    end if
    pick random edge $E = (i, j)$ from S
    add edge (i, j) to G
    remove (i, j) from S
end for

return G

all evaluation process. The criteria are as follows:

1. **Accuracy:** The percentage of graphs that a given heuristic finds the actual LSP.

2. **Error:** The average difference between the actual LSP and the path found by a given heuristic across all trials.

3. **Runtime:** The average wall-clock execution time of a given heuristic over a group of graphs.

4.4 Results

Table 1 and Figure 3 show the accuracy benchmarking results of our heuristics on graphs with 9 vertices with the number of edges ranging from 8 to 36. All heuristics showed a canyon-like behavior. In other words, at any vertex count, as the edge count increases, the accuracy of the heuristics decreases to a lower bound then increases. In other words, the longest simple paths of trees and dense graphs are consistently more accurately found by the heuristics, while the sparse and non-tree graphs are the difficult graphs to find a correct LSP for. This reconfirms the observations by prior studies that sparse graphs are the real challenging instances for the LSP problem.

In addition, the accuracy benchmarking also revealed that both variants of the greedy algorithm, with and without the depth-first-search tiebreaker, were getting the correct LSPs in dense graphs, reconfirming previous findings. For the graph pruning heuristics, the accuracy results were weaker than the greedy heuristics, signalling that, as stand-alone heuristics, graph pruning is not as effective as the greedy method. However, it is possible that implementing them together with other heuristics A* or local search could produce more desirable results.

Table 2 and Figure 4 show the error benchmarking of our heuristics on the same set of graphs. It essentially bears the same trend as the accuracy benchmarking: these heuristics do well on dense graphs, but their performances degrade significantly on spare graphs, which remains a challenge.

Table 3 and Figure 5 show the runtimes of our heuristics. In comparison to the naïve brute-force solution, all heuristic runtime trendlines grow significantly slower. 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 linearly. Because of its need to loop through all edges each time, the Stretch GP heuristic tended to have higher runtimes than the other heuristics. Despite its relative inefficiencies, the Stretch GP heuristic was still more efficient than the naïve brute-force solution.

5 Conclusions

There are many difficulties in finding the LSP in any given graph. We explored and implemented several heuristics with the goal of more efficient solutions to finding the LSP. Through our experimentation, we reconfirmed past findings about the greedy heuristic's abilities to solve the LSP problem on dense graphs, as well as the challenges 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 in sparse graphs, there is still room for further exploration.

Acknowledgements

I would like to thank Bart Massey and Cassaundra Smith of Portland State University for their guidance and 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 Institute for Computing in Research during Summer 2022.

Table 1: Accuracy of Heuristics on 9-Vertex Graphs

Number of Edges	Heuristics				
	greedy	dfs_greedy	prune_central	stretch_GP	stretch_TVP
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	92.0	91.0
11	99.0	98.0	92.0	90.0	90.0
12	100.0	100.0	91.0	84.0	78.0
13	100.0	99.0	83.0	81.0	79.0
14	99.0	100.0	92.0	88.0	86.0
15	99.0	100.0	86.0	86.0	84.0
16	100.0	100.0	91.0	81.0	88.0
17	100.0	100.0	98.0	89.0	88.0
18	100.0	100.0	96.0	87.0	96.0
19	100.0	100.0	98.0	91.0	61.0
20	100.0	100.0	100.0	95.0	99.0
21	100.0	100.0	99.0	94.0	99.0
22	100.0	100.0	99.0	97.0	98.0
23	100.0	100.0	100.0	96.0	94.0
24	100.0	100.0	100.0	96.0	100.0
25	100.0	100.0	100.0	96.0	97.0
26	100.0	100.0	100.0	95.0	99.0
27	100.0	100.0	100.0	97.0	100.0
28	100.0	100.0	100.0	94.0	99.0
29	100.0	100.0	100.0	97.0	100.0
30	100.0	100.0	100.0	94.0	100.0
31	100.0	100.0	100.0	98.0	100.0
32	100.0	100.0	100.0	96.0	100.0
33	100.0	100.0	100.0	98.0	100.0
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

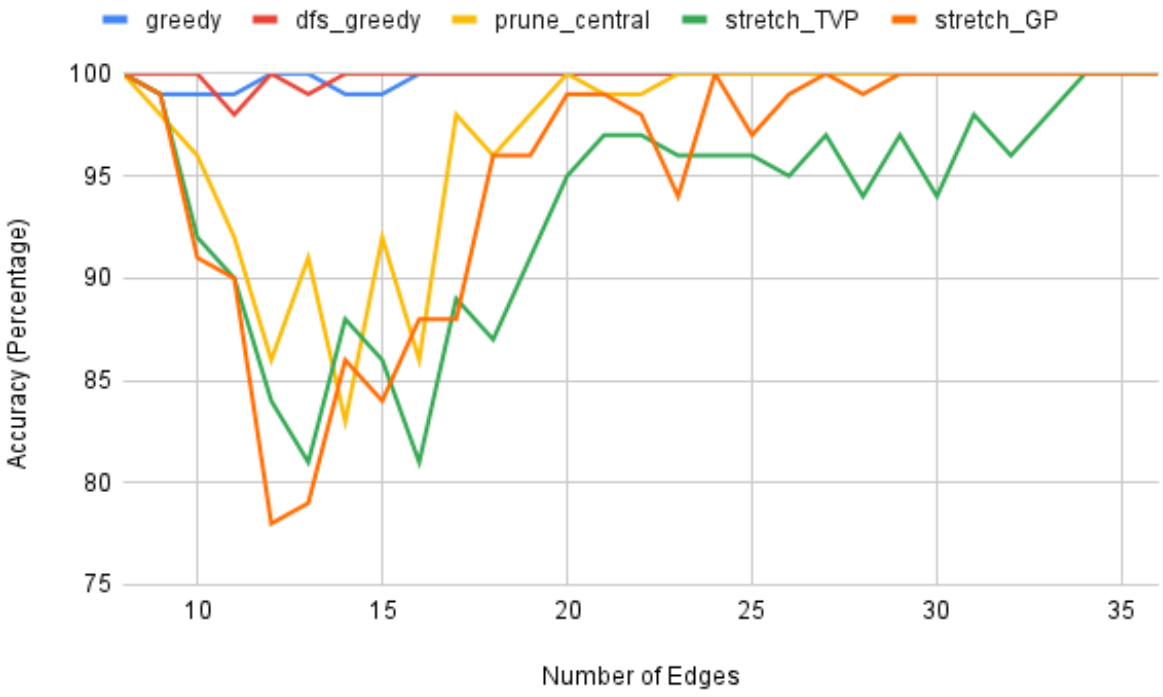

Figure 3: Accuracy of Heuristics on Graphs with 9 Vertices

Table 2: Error of Heuristics on 9-Vertex Graphs

Number of Edges	Heuristics				
	greedy	dfs_greedy	prune_central	stretch_GP	stretch_TVP
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	0.08	0.09
11	0.01	0.02	0.08	0.1	0.1
12	0.0	0.0	0.09	0.16	0.23
13	0.0	0.01	0.18	0.19	0.21
14	0.01	0.0	0.08	0.12	0.14
15	0.01	0.0	0.14	0.14	0.16
16	0.0	0.0	0.19	0.19	0.13
17	0.0	0.0	0.02	0.11	0.12
18	0.0	0.0	0.04	0.13	0.04
19	0.0	0.0	0.02	0.09	0.05
20	0.0	0.0	0.0	0.05	0.01
21	0.0	0.0	0.01	0.06	0.01
22	0.0	0.0	0.01	0.03	0.02
23	0.0	0.0	0.0	0.04	0.06
24	0.0	0.0	0.0	0.04	0.0
25	0.0	0.0	0.0	0.04	0.03
26	0.0	0.0	0.0	0.05	0.01
27	0.0	0.0	0.0	0.03	0.0
28	0.0	0.0	0.0	0.06	0.01
29	0.0	0.0	0.0	0.03	0.0
30	0.0	0.0	0.0	0.06	0.0
31	0.0	0.0	0.0	0.02	0.0
32	0.0	0.0	0.0	0.04	0.0
33	0.0	0.0	0.0	0.02	0.0
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

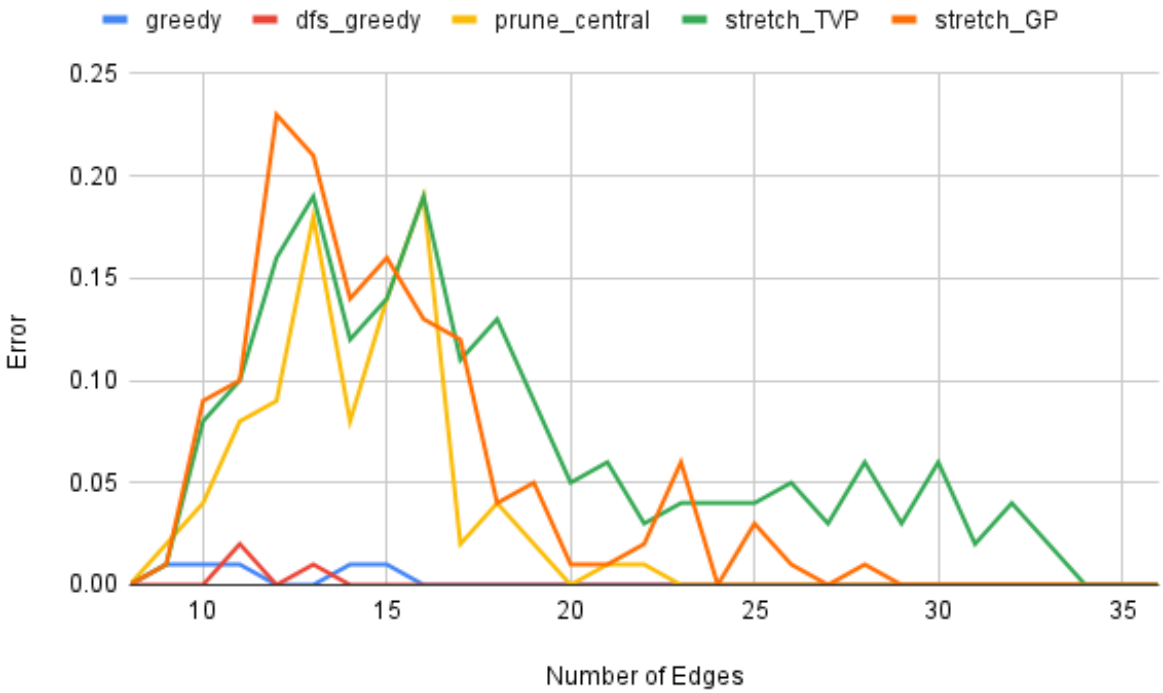

Figure 4: Error of Heuristics on Graphs with 9 Vertices

Table 3: Runtime of Heuristics on 9-Vertex Graphs

Number of Edges	Heuristics					
	greedy	dfs_greedy	prune_central	stretch_GP	stretch_TVP	brute-force
8	0.0001339	0.0002115	0.0000178	0.0000128	0.0000124	0.0000393
9	0.0000944	0.0003868	0.0002032	0.0003383	0.0011236	0.0000581
10	0.0001048	0.0005775	0.0004151	0.0006561	0.0023543	0.0000903
11	0.0001125	0.0007197	0.005731	0.0009669	0.0037371	0.0001347
12	0.0001214	0.0008422	0.0008724	0.001356	0.0053146	0.0002133
13	0.0001287	0.0009806	0.0009262	0.0016779	0.0071409	0.0003941
14	0.000135	0.0011354	0.0011374	0.0020774	0.0090412	0.000534
15	0.0001397	0.0013132	0.0012995	0.0024597	0.0109298	0.000801
16	0.0001446	0.0013653	0.0014624	0.00292	0.0131174	0.0011781
17	0.0001484	0.0015664	0.0016971	0.0033403	0.015527	0.0017935
18	0.0001527	0.0017702	0.0018301	0.0038113	0.0182468	0.0027262
19	0.0001562	0.0020146	0.0029375	0.0043504	0.0209624	0.0043898
20	0.0001597	0.002153	0.0021626	0.0048087	0.0235532	0.0065727
21	0.0001618	0.0023596	0.0023787	0.0053355	0.0255448	0.0092391
22	0.0001643	0.00258	0.0025342	0.0058146	0.0292619	0.0133933
23	0.0001676	0.0027988	0.0027264	0.0064107	0.0325536	0.0192997
24	0.0001705	0.0030494	0.0030376	0.0069574	0.0360228	0.025796
25	0.0001729	0.0033103	0.0029793	0.0075332	0.0395946	0.03686
26	0.0001746	0.0034828	0.0031932	0.0080365	0.0434956	0.0492349
27	0.0001771	0.0037537	0.0033473	0.0085708	0.0471843	0.0655757
28	0.0001789	0.0040668	0.0034598	0.0092661	0.0520643	0.0890664
29	0.0001811	0.0044	0.0035829	0.0097569	0.0569198	0.1137601
30	0.0001832	0.0047944	0.0037833	0.0103405	0.0620864	0.1465667
31	0.0001856	0.0052211	0.0040176	0.0112145	0.0665662	0.1921401
32	0.0001874	0.0057643	0.0041552	0.011959	0.0722577	0.2542703
33	0.0001889	0.0063387	0.0044217	0.0127085	0.0775441	0.3268791
34	0.0001915	0.0074702	0.0045202	0.013976	0.0832664	0.4175168
35	0.000193	0.0085568	0.0046966	0.0146478	0.08821	0.5437159
36	0.0001936	0.010597	0.0053172	0.0153948	0.0936378	0.7104294

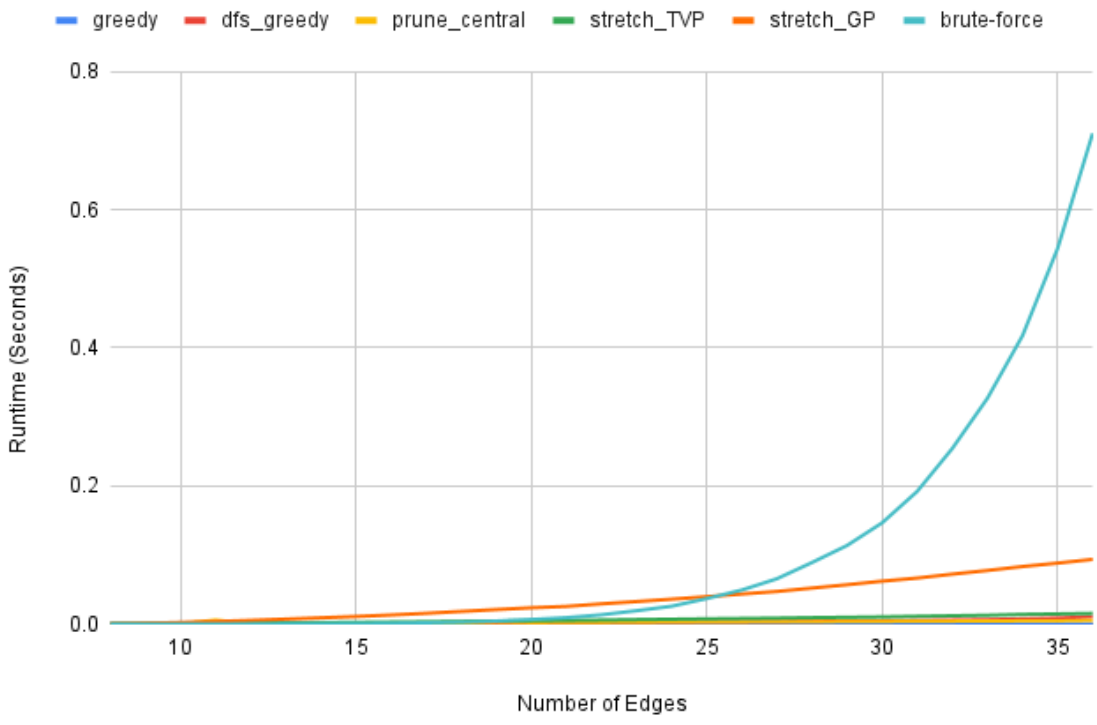

Figure 5: Runtime of Heuristics on Graphs with 9 Vertices

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] Michael R. Garey and David S. Johnson. *Computers and intractability. A guide to the theory of NP-completeness*. Freeman, 1979.

[7] 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.

[8] 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.

[9] David Karger, Rajeev Motwani, and G. D. S. Ramkumar. On approximating the longest path in a graph. *Algorithmica*, 18(1):82–98, 1997.

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

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

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

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

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

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

[20] 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.

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

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