P2P Shortest Path Problem and Beyond
Final Report

Michael Henderson, Valentin Koch

April 5, 2006

Abstract

This report describes the progress we have made on our project of investigating the A* algorithm vs. the Dijkstra algorithm in graphs and the shortest path problem. In this memo, we review the nature of the project and describe work we have completed. This report provides readers with practical information on the comparison of the two algorithms in different settings in the context of the shortest path problem.

Contents

1 Introduction

1.1 Project Description

The complete project description can be found in the project topic-3 document[1]. One of the motivations to choose this project was the desire to learn more about graphs and the appealing problem of finding the shortest path in complex graphs.

1.2 Literary survey

The shortest path problem has been researched extensively in the past due to its numerous real world applications. In the search for a fast algorithm it was discovered that A* together with a good heuristic could outperform a basic dynamic programming aproach. Andrew V. Goldberg and Chris Harrelson compared the use of landmarks and the triangle inequality as an A* heuristic to Dijkstra's algorithm[2]. The ALT heuristic is the main heuristic in our research.

2 Implementation

2.1 Software Design

Our program design is broken up into modules. The first module contains the classes responsible for the graph generation, storage and loading. Figure 1 shows a UML diagram of the three different graphs we are using:

- Random Graph
- Geometric Graph
- Grid Graph

The parent class of the random graph and the geometric graph is the unordered graph. The characteristic of an unordered graph is the random generation of nodes and edges. The random graph generates random edges using a probability edge factor with random weights assigned to the edges. The geometric graph generates random positions for the nodes and creates edges between the nodes according to a distance edge factor. The weight of the

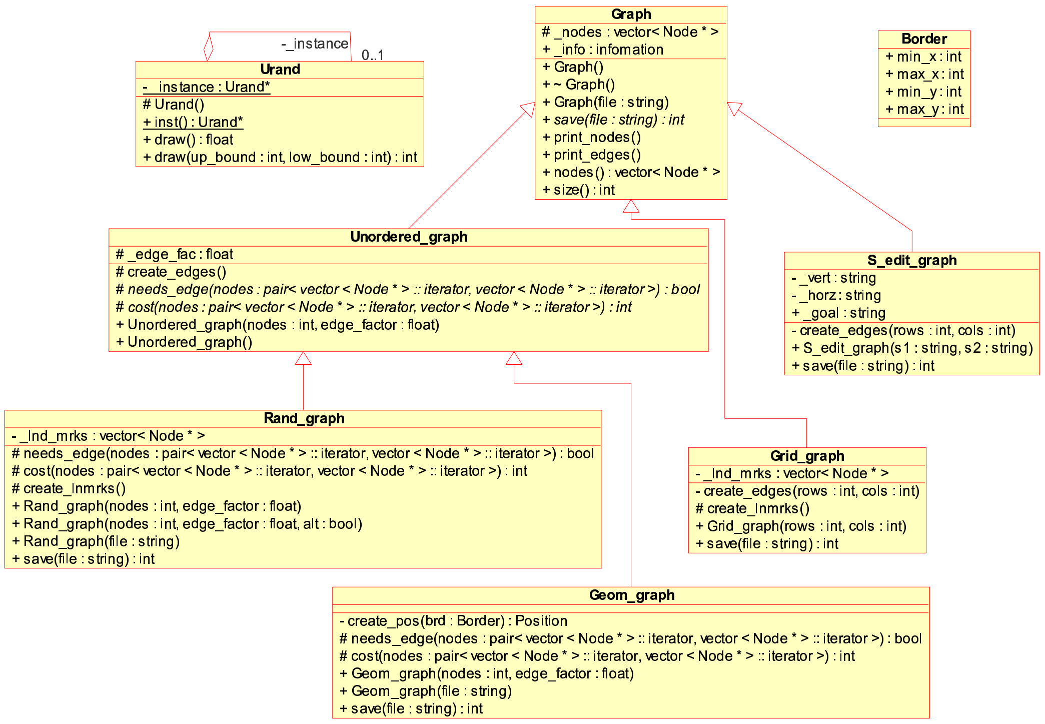
-_instance
0..1
Urand
-_instance : Urand*
# Urand()
+ inst() : Urand*
+ draw() : float
+ draw(up_bound : int, low_bound : int) : int
Graph
# _nodes : vector< Node * >
+ _info : infomation
+ Graph()
+ ~ Graph()
+ Graph(file : string)
+ save(file : string) : int
+ print_nodes()
+ print_edges()
+ nodes() : vector< Node * >
+ size() : int
Border
+ min_x : int
+ max_x : int
+ min_y : int
+ max_y : int
Unordered_graph
# _edge_fac : float
# create_edges()
# needs_edge(nodes : pair< vector < Node * > :: iterator, vector < Node * > :: iterator >) : bool
# cost(nodes : pair< vector < Node * > :: iterator, vector < Node * > :: iterator >) : int
+ Unordered_graph(nodes : int, edge_factor : float)
+ Unordered_graph()
S_edit_graph
- _vert : string
- _horz : string
+ _goal : string
- create_edges(rows : int, cols : int)
+ S_edit_graph(s1 : string, s2 : string)
+ save(file : string) : int
Rand_graph
- _lnd_mrks : vector< Node * >
# needs_edge(nodes : pair< vector < Node * > :: iterator, vector < Node * > :: iterator >) : bool
# cost(nodes : pair< vector < Node * > :: iterator, vector < Node * > :: iterator >) : int
# create_lnmrks()
+ Rand_graph(nodes : int, edge_factor : float)
+ Rand_graph(nodes : int, edge_factor : float, alt : bool)
+ Rand_graph(file : string)
+ save(file : string) : int
Grid_graph
- _lnd_mrks : vector< Node * >
- create_edges(rows : int, cols : int)
# create_lnmrks()
+ Grid_graph(rows : int, cols : int)
+ save(file : string) : int
Geom_graph
- create_pos(brd : Border) : Position
# needs_edge(nodes : pair< vector < Node * > :: iterator, vector < Node * > :: iterator >) : bool
# cost(nodes : pair< vector < Node * > :: iterator, vector < Node * > :: iterator >) : int
+ Geom_graph(nodes : int, edge_factor : float)
+ Geom_graph(file : string)
+ save(file : string) : int

Figure 1: Graph Types

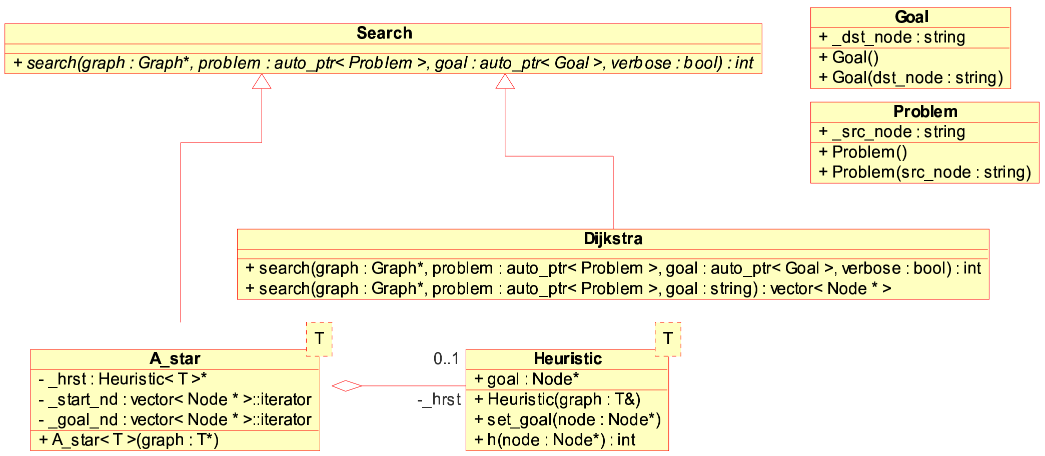
Search
+ search(graph : Graph*, problem : auto_ptr< Problem >, goal : auto_ptr< Goal >, verbose : bool) : int
Goal
+ _dst_node : string
+ Goal()
+ Goal(dst_node : string)
Problem
+ _src_node : string
+ Problem()
+ Problem(src_node : string)
Dijkstra
+ search(graph : Graph*, problem : auto_ptr< Problem >, goal : auto_ptr< Goal >, verbose : bool) : int
+ search(graph : Graph*, problem : auto_ptr< Problem >, goal : string) : vector< Node * >
T
A_star
- _hrst : Heuristic< T >*
- _start_nd : vector< Node * >::iterator
- _goal_nd : vector< Node * >::iterator
+ A_star< T >(graph : T*)
0..1
-_hrst
T
Heuristic
+ goal : Node*
+ Heuristic(graph : T&)
+ set_goal(node : Node*)
+ h(node : Node*) : int

Figure 2: Search Algorithms

edges in the geometric graph are calculated according to the real distance between the nodes on a plane. The grid graph is basically a random graph with ordered edges. Each node has at most four edges and the edges go to its upper, lower, left and right neighbour nodes. The nodes at the border sides of the graph have only three edges. All graphs use an adjaceny list to store the edges for each node.

The second module incorperated the search algorithms. Both A* and the Dijkstra were grouped in a search parent class (see Figure 2) which run on an abstract model of a graph. This provides the possibility for future extensions with additional types of graphs.

The last module of the program contains the heuristics for the A* search.

2.2 Implementation

In our implemetaion we emphasized modularity and extensibility by using an object oriented approach with C++ templates.

2.2.1 Graph Generation

The core of a graph is a STL vector containing pointers to nodes. Figure 3 shows the relation of a node and its edges.

2.2.2 Graph Storage/Retrieval

Once generated, the graphs can be saved to disk for future use. Random graphs are stored in the DIMACS[7] edge format and geometric graphs are stored as a geometric graph in DIMACS format. Grid graphs are also stored in DIMACS edge format with a nonstandard line giving the dimensions of the graph.

2.2.3 Dijkstra's Algorithm

A good explanation of the Dijkstra algorithm can be found on Samuel Rebelsky's homepage [6]. A more detailed discussion of the Dijkstra algorithm goes beyond the scope of this document and we refer to the respective literature. Our implementation of Dijkstra's used the STL algorithm min_element to extract the node with the shortest current distance to the current node. This algorithm has an O(n) running time and this should be considered when comparing CPU time with the A* search which uses a priority queue approach.

2.2.4 A* Search

An efficient implementation of A* search was found in the Boost Graph Library[3]. This implementation uses node colouring instead of lists to keep track of the status of the nodes. The priority queue in the STL is very limited. Our implementaion of the A* alorithm uses the STL heap algorithms to sort and choose the best f-value.

2.2.5 String Edit Distance

The string edit distance graph is based on a grid graph. It uses a cost of one for each vertical and horizontal edge which represent inserting or removing a

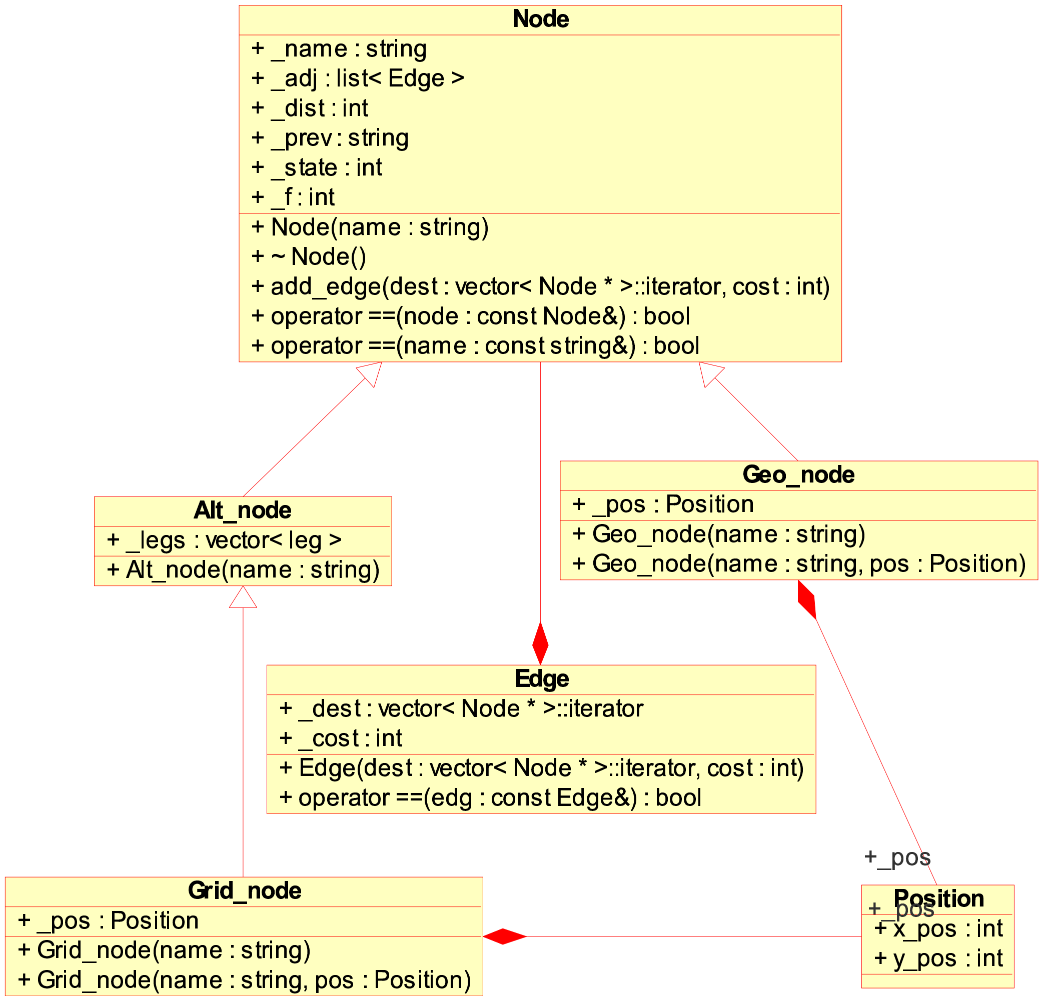
Node
+ _name : string
+ _adj : list< Edge >
+ _dist : int
+ _prev : string
+ _state : int
+ _f : int
+ Node(name : string)
+ ~ Node()
+ add_edge(dest : vector< Node * >::iterator, cost : int)
+ operator ==(node : const Node&) : bool
+ operator ==(name : const string&) : bool
Alt_node
+ _legs : vector< leg >
+ Alt_node(name : string)
Geo_node
+ _pos : Position
+ Geo_node(name : string)
+ Geo_node(name : string, pos : Position)
Edge
+ _dest : vector< Node * >::iterator
+ _cost : int
+ Edge(dest : vector< Node * >::iterator, cost : int)
+ operator ==(edg : const Edge&) : bool
+_pos
Grid_node
+ _pos : Position
+ Grid_node(name : string)
+ Grid_node(name : string, pos : Position)
Position
+_pos
+ x_pos : int
+ y_pos : int

Figure 3: Nodes and Edges

character. A diagonal edge of cost 0 is added for each character in one string that matches a character in the other.

2.2.6 Heuristics

The heuristics are implemented as templates to simplify their usage with the various kinds of graphs. The basic heuristic checks if the underlying graph type is a random graph and if not it acts as a uniform cost heuristic. In case of a random graph the heuristic uses the landmark data in the nodes and the triangle inequality to find a lower bound for the heuristic funtion. The precomputed landmark data for each node is stored in a container in each node. The precomputation of the landmark data is done during graph creation with the single-source version of Dijkstra's. The grid graphs also use this heuristic. The geometric graph uses a simple straight line distance to the goal node calculated using the positions of the current node and the goal node. The heuristic used for the string edit distance problem is the difference between the number of characters remaining in each string.

3 Experiments

3.1 Experimental Setup

All tests were run on a Intel Pentium III 1.4GHz with 1GB of Memory running Linux 2.6.5. The graphs were created in advance and saved in DIMACS format:

- 5000 node random graphs with edge probabilities ranging from .0025 to .035 in .0025 increments and also for edge probabilities of .04 and .045
- 49 100x100 geometric graphs between 100 and 2000 nodes (randomly distributed) with maximum edge length between 20 and 100 units
- Grid graphs with nodes ranging from 10000 to 50000 in increments of 10000

For the string edit distance problem we downloaded DNA sequences in FASTA format from the European Bioinformatics Institute [8]. The GNU awk parser was used to extract parts of the sequences to enable us to run the string edit

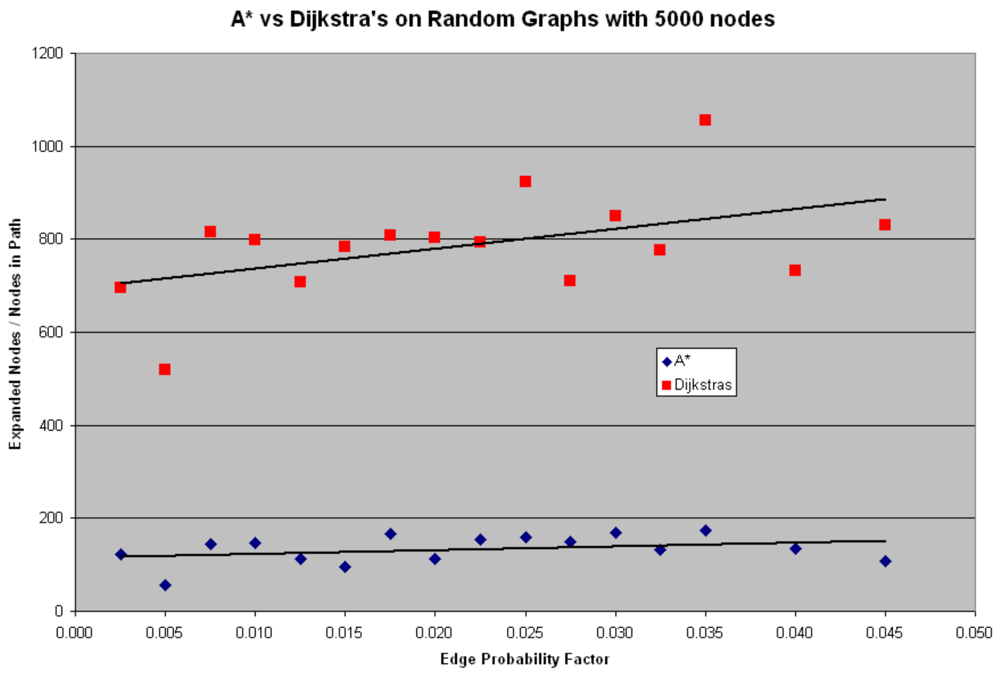

Figure 4: Random Graphs

distance comparison on the sequences. Bash scripts were created to both create and automatically run the tests many times on each graph.

3.2 Experimental Results

3.2.1 Random Graphs

For the 5000 node graphs both A* and Dijkstra's was run a total of 826 times each. Each search used randomly chosen source and destination nodes. Figure 4 shows the results of the tests. An interesting observation is that the ratio of nodes expanded over the number of nodes in the path for A* stays relatively constant while Dijkstra's gets worse as the density of the graph increases. A* has a significant adavantage to begin with and it gets better as the size of the graph increases.

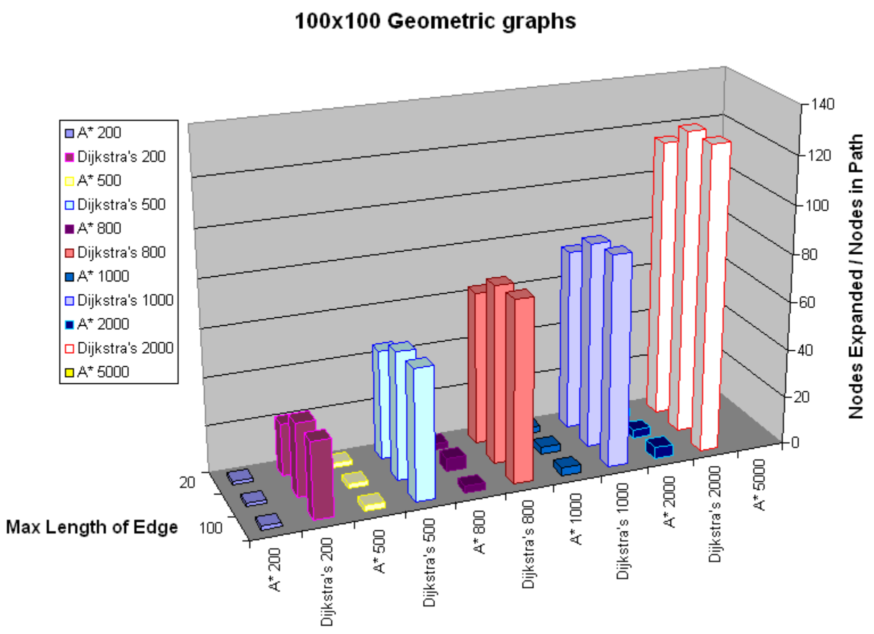

Figure 5: Geometric Graphs

3.2.2 Geometric Graphs

For 100x100 geometric graphs both A* and Dijkstra's was run 100 times each on each graph. This comes to a total of 9800 searches. In this case A* outperforms Dijkstra's significantly. As seen in figure 5 as the density of the graph increases, the advantage of A* increases.

3.2.3 Grid Graphs

On grid graphs A* and Dijkstra's performed similarly to the other graphs. We performed a total of 1800 searches each for both algorithms. Figure 6 shows the results.

3.2.4 String Edit Distance

Because of time constraints we were unable to create a good heuristic for this problem. A* was still able to outperform Dijkstra's in all cases with the

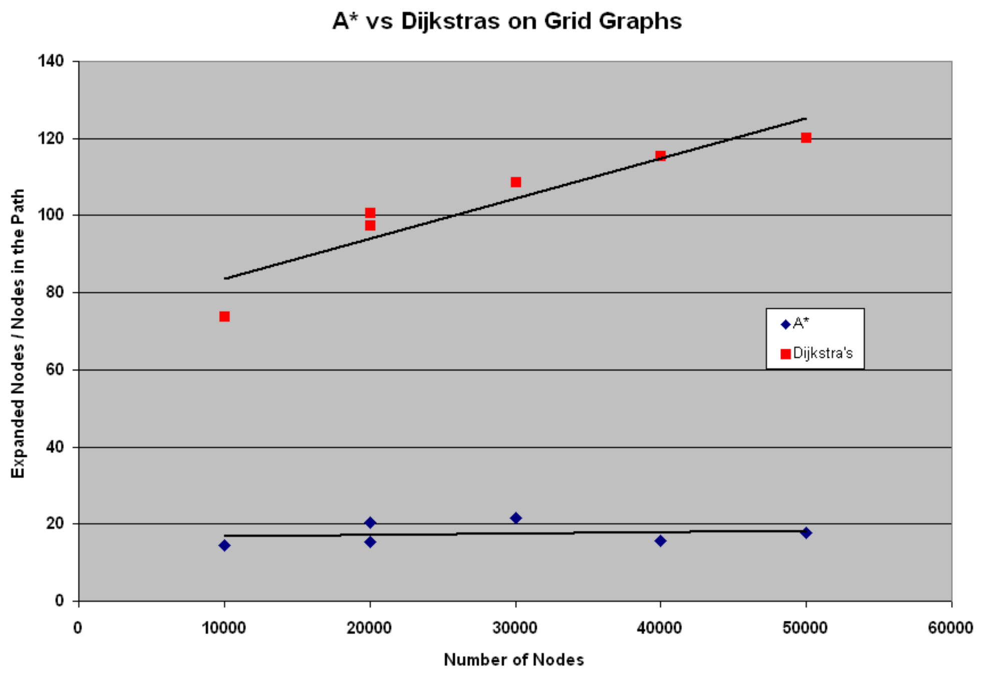

Figure 6: Grid Graphs

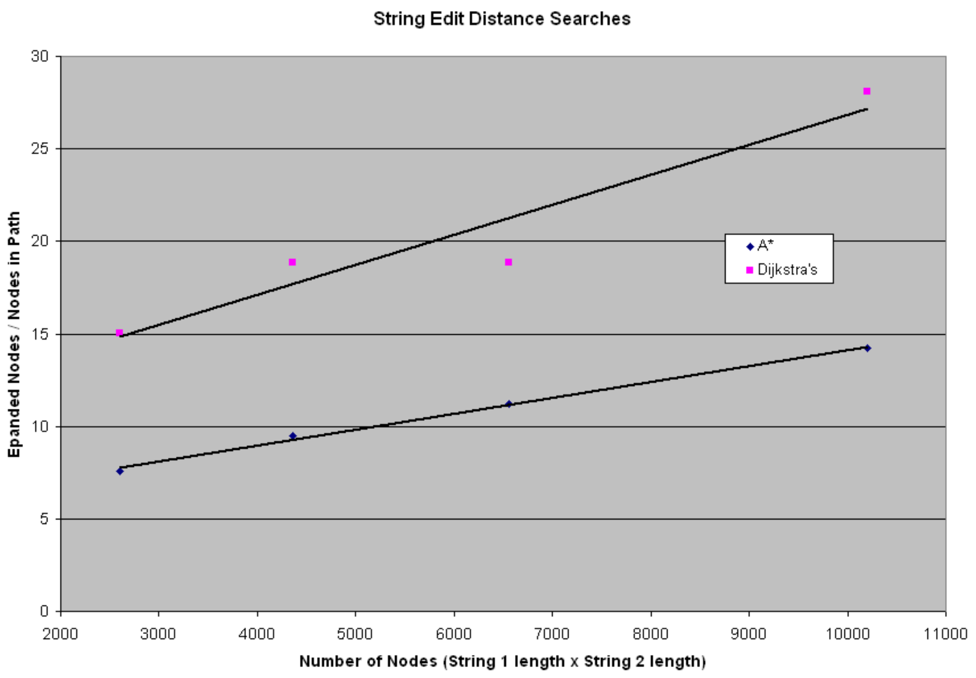

Figure 7: String Edit Distance

heuristic we used. We compared the first 50, 65, 80, and 100 characters of 2333 DNA sequences (FASTA format)[8]. Figure 7 shows the results of the experiments.

4 Conclusion

Our results show that A* outperforms Dijkstra's in every one of our experiments. However, the advantage of A* for random graphs and grid graphs using the ALT heuristic relies on preprocessing to create landmark data. For single searches on those graphs this is not an advantage, but if a large nunber of different searches were to be done on the same graph the preprocessing would become desireable. The advantage of using the A* search increases as the size of the graph increases. This may become significant for string edit searches when the size of the strings being compared becomes large.

List of Figures

References

[1] Gao, Yong, *Point-to-Point Shortest Path Problem and Beyond*, January 25, 2006.

[2] Goldberg, Andrew V. and Harrelson, Chris, *Computing the Shortest Path: A* Search Meets Graph Theory*, http://research.microsoft.com/research/sv/spa/, 2005.

[3] Boost C++ Library, *A* Heuristic Search*, http://www.boost.org/libs/graph/doc/astar_search.html.

[4] Dimacs, *Clique and Coloring Problems Graph Format*, May 8, 1993.

[5] Dimacs, *The First DIMACS International Algorithm Implementation Challenge: Problem Definitions and Specifications.*

[6] Rebelsky, Samuel A., *Dijkstra's Shortest Path Algorithm*, http://www.math.grin.edu/~rebelsky/Courses/CS152/98S/Outlines/outline.50.html, May 4, 1998.

[7] DIMACS, *Implementation Challenges with DIMACS graph format*, http://dimacs.rutgers.edu/Challenges/

[8] European Bioinformatics Institute, *IMGT/HLA Sequence Database*, http://www.ebi.ac.uk/imgt/hla/download.html