University of Denver
Digital Commons @ DU

Electronic Theses and Dissertations | Graduate Studies

1-1-2017

Explaining the Performance of Bidirectional Dijkstra and A* on Road Networks

Sneha Sawlani
University of Denver

Follow this and additional works at: https://digitalcommons.du.edu/etd

Part of the Computer Engineering Commons

Recommended Citation

Sawlani, Sneha, "Explaining the Performance of Bidirectional Dijkstra and A* on Road Networks" (2017). *Electronic Theses and Dissertations*. 1303.
https://digitalcommons.du.edu/etd/1303

This Thesis is brought to you for free and open access by the Graduate Studies at Digital Commons @ DU. It has been accepted for inclusion in Electronic Theses and Dissertations by an authorized administrator of Digital Commons @ DU. For more information, please contact jennifer.cox@du.edu,dig-commons@du.edu.

Explaining the Performance of Bidirectional Dijkstra and A* on Road Networks

Abstract

The heuristic search community traditionally uses A* as the baseline algorithm for their research methods. Research papers in the road networks community, however, often build upon Dijkstra's algorithm and use Bidirectional Dijkstra's algorithm as their baseline. This thesis investigates the performance of A* and Bidirectional Dijkstra in road networks to see how they compare and to see if there is a principled explanation for the different approaches. Our analysis reveals why Bidirectional Dijkstra can perform well in this domain, but also shows a simple mistake that can be made when building test problems that hurts the performance of A*.

Document Type

Thesis

Degree Name

M.S.

Department

Computer Science

First Advisor

Nathan Sturtevant, Ph.D.

Second Advisor

Chris Gauthier-Dickey

Third Advisor

Mei Yin

Keywords

A*, Bidirectional dijkstra, Dijkstra's algorithm, Road networks, Search algorithms

Subject Categories

Computer Engineering

Publication Statement

Copyright is held by the author. User is responsible for all copyright compliance.

This thesis is available at Digital Commons @ DU: https://digitalcommons.du.edu/etd/1303

Explaining the Performance of Bidirectional Dijkstra and A* on Road Networks

A Thesis

Presented to

the Faculty of the Daniel Felix Ritchie School of Engineering

and Computer Science

University of Denver

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

By

Sneha Sawlani

June 2017

Advisor: Nathan Sturtevant

© Copyright by Sneha Sawlani 2017.

All Rights Reserved

Author: Sneha Sawlani
Title: Explaining the Performance of Bidirectional Dijkstra and A* on Road Networks
Advisor: Nathan Sturtevant
Degree Date: June 2017

Abstract

The heuristic search community traditionally uses A* as the baseline algorithm for their research methods. Research papers in the road networks community, however, often build upon Dijkstra's algorithm and use Bidirectional Dijkstra's algorithm as their baseline. This thesis investigates the performance of A* and Bidirectional Dijkstra in road networks to see how they compare and to see if there is a principled explanation for the different approaches. Our analysis reveals why Bidirectional Dijkstra can perform well in this domain, but also shows a simple mistake that can be made when building test problems that hurts the performance of A*.

Acknowledgements

I would like to express my sincere gratitude to my thesis advisor, Dr. Nathan Sturtevant. Dr. Sturtevant has always been available for help whenever I ran into a trouble spot or had a question about my research or writing. I would like to thank him for helping me choose a thesis topic, guiding me on the right path throughout the research, and for patiently correcting my writing.

I would like to thank my committee members, Dr. Mei Yin and Dr. Chris GauthierDickey for their time and for their interest in my work.

I would like to thank the Department of Computer Science at the University of Denver for providing me all the resources I needed to complete my thesis.

I would like to thank my family for their love, support, and constant encouragement.

Contents

Acknowledgements . . . iii
List of Tables . . . vi
List of Figures . . . vii

1 Introduction **1**

2 Background **4**
2.1 Graphs and Paths . . . 4
2.2 Best First Search . . . 5
2.2.1 Dijkstra's Algorithm . . . 6
2.2.2 A* . . . 8
2.3 Bidirectional Search . . . 8
2.3.1 Bidirectional Dijkstra . . . 10
2.3.2 MM . . . 12

3 Experimental Setup **15**
3.1 Algorithms used . . . 15
3.2 Data structures . . . 18
3.3 Metrics . . . 20
3.4 Dataset and Problem Selection . . . 21
3.4.1 Dataset . . . 21
3.4.2 Problem selection . . . 22

4 Experiments **25**
4.1 Hypothesis . . . 25
4.2 Results . . . 26

4.3 Analysis 27
4.3.1 Regions 27
4.3.2 Example Problem 30
4.3.3 Analysis 35
4.3.4 Explanation 42

5 Conclusions **46**

Bibliography **48**

List of Tables

3.1 Dataset 22

4.1 The definition of the regions used for our analysis. 28
4.2 Problem Set 1: Average node expansions on graphs with edge costs representing distance. 29
4.3 Problem Set 1: Average node expansions on graphs with edge costs representing travel time. 30
4.4 Region Comparison: Bidirectional Dijkstra and A* 30
4.5 Problem Set 1: Average node expansions in regions 38
4.6 Problem Set 2: Average node expansions on graphs with edge costs representing distance. 45
4.7 Problem Set 2 : Average node expansions on graphs with edge costs representing travel time. 45

List of Figures

3.1 Grid overlay on CO map 24

4.1 A* vs Bidirectional Dijkstra 27
4.2 A visual picture of 8 of the 9 regions from Table 4.1 28
4.3 NY city map 33
4.4 Example problem: Node Expansions 34
4.5 Performance Predictor 39
4.6 Problem Set 1: Region Distribution 40
4.7 Node expansion percentage 41
4.8 Crossover points for $\Delta(NF + FF)$ and $\Delta(NR + FN + RN)$ 42
4.9 Problem Set 2: Region Distribution 45

Chapter 1

Introduction

Shortest path problems are one of the most-studied combinatorial optimization problems in the literature [10, 11, 12, 3, 15, 14]. Given a starting point and a destination, the task of point-to-point shortest path problems is to find the optimal path from the starting point to the destination. "Optimal" refers to shortest time, shortest distance, or least total path cost. Shortest path problems have a wide-range of applications in areas such as communications [14], transportation [10], game development [9] and AI [15]. Dijkstra's algorithm [2] is the "classic" solution for this problem. But querying shortest paths in large problems, such as road-networks, would result in query-times that are unacceptably slow in real world applications. For example, if we take a country sized graph like the USA with tens of millions of vertices, Dijkstra's algorithm on a server takes time in the order of seconds to answer a point-to-point query. Since servers typically handle thousands of queries per second, they need to answer in millisec-

onds, not in seconds. To this end, there has been a considerable interest in the speed-up techniques to shortest path algorithms. Two of those techniques are:

1. Goal-directed Search
2. Bidirectional Search

Goal-directed search takes into account the knowledge of goal and speeds up the search by using heuristics that guide the search toward the goal instead of exploring regions in all directions. The A* path finding algorithm is an example of this technique. This algorithm is an extension of Dijkstra's algorithm and was introduced in 1968 at Stanford Research Institute [5].

Bidirectional Search, as the name implies, searches in two directions at the same time: one forward from the initial state and the other backward from the goal. The search stops when searches from both directions meet and the optimal solution is proven. In many cases, it makes the search faster. For example, in a search problem modeled by a tree with branching factor b and solution depth d, a bidirectional search will expand $2b^{d/2}$ states instead of b^d required by unidirectional search. Bidirectional Dijkstra algorithm is an example of this technique.

The heuristic search community traditionally uses Goal-directed search techniques to speed up Dijkstra's algorithm and get A*. It then uses A* as the baseline algorithm for their research methods. Research papers in the road networks community, however, often use Bidirectional Search techniques to speed up Dijkstra's algorithm and use Bidirectional Dijkstra's algorithm as a baseline algorithm [4, 13]. The purpose of this thesis is to investigate the performance of A* and Bidirectional Dijkstra in road networks to see how they compare and to see if there is a principled explanation for the different approaches. We compared and analyzed the performance of A* and Bidirectional Dijkstra on road networks of the USA. Our analysis reveals why Bidirectional Dijkstra can perform well in this domain, but also shows a simple mistake that can be made when building test problems that can hurt the performance of A*.

Chapter 2

Background

In this chapter, we introduce basic terminology used throughout the thesis and describe existing work in this field.

2.1 Graphs and Paths

Road networks can be represented as a directed graph where nodes correspond to junctions and edges correspond to roads that connect two junctions. Edge cost is the distance or time it takes to travel between two junctions.

The input to a search problem is a directed graph $G = (V, E)$ with a node set V of size n and an edge set $E \subseteq V \times V$ of size m. A weight function $w : E \mapsto R_0^+$ assigns a nonnegative weight $w(u, v)$ to each edge (u, v).

A path P in G from a node u_1 to a node u_k is a sequence of nodes $\{u_1, u_2, \ldots, u_k\}$ such that $(u_i, u_{i+1}) \in E$. The length $w(P)$ of a path P is the sum of the weights of the edges that connect two consecutive nodes in P. $P^* = \{s, \ldots, t\}$ is a shortest path if there is no path P' from s to t such that $w(P') < w(P^*)$. The

distance $d(s,t)$ from s to t in G is the length of a shortest path from s to t or ∞ if there is no path from s to t. C^* is used to denote the length of shortest path from start to goal.

$h(n)$ is a heuristic function that takes a node n and returns a non-negative real number that is an estimate of the path cost from node n to a goal node. A heuristic function is *admissible* if it never overestimates the cost of reaching the goal. A heuristic function is *consistent*, if for every node n and each successor s of n, the estimated cost of reaching the goal from n is no greater than the step cost of getting to s plus the estimated cost of reaching the goal from s. That is:

$$h(n) \leq w(n,s) + h(s) \tag{2.1.1}$$

A search problem is defined by (G, w, s, t, h) and the goal is to return a path from s to t with cost $w(P^*)$.

2.2 Best First Search

Best First Search is a search technique which explores the graph by expanding the node with the least cost first. A priority function is used to assign cost to each candidate node. The algorithm maintains two lists, one containing a list of candidate nodes yet to explore (*Open*), and the other containing a list of visited nodes (*Closed*). Since all unvisited neighbors of every visited

node are included in the *Open* list, the algorithm is not restricted to only exploring neighbors of the most recently visited node as it would be in a depth first search. In other words, the algorithm always chooses the best of all unvisited nodes, rather than being restricted to only a small subset, such as immediate neighbors. The advantage of this strategy is that if the algorithm reaches a dead-end node, it will continue to try other nodes. A priority function $pr(n)$ is used to find the "best" (usually the least-cost) node to expand next in the search. The priority function used for each algorithm is described later. Algorithm 1 contains pseudocode for Best First Search.

2.2.1 Dijkstra's Algorithm

Dijkstra's algorithm solves the single-source shortest-path problem on a weighted, directed graph $G = (V, E)$ for the case in which all edge weights are nonnegative. It was conceived by computer scientist Edsger W. Dijkstra in 1956 and published three years later [2] .

Dijkstra's algorithm uses the best-first search approach to solve the single source shortest path problem. It uses the following priority function:

Algorithm 1 Best First Search(G, w, s, t, h)

```
s_pr ← pr(s)
Open ← {s}
for all n ∈ V − {s} do
    n_pr ← ∞
end for
while Open ≠ ∅ do
    n ← delete-min(Open)
    if n = t then
        return Path(s, n)
    end if
    Closed ← Closed ∪ {n}
    for all succ ∈ n_successors do
        if succ ∈ Closed then
            continue
        else
            priority ← pr(succ)
            if succ ∈ Open then
                if succ_pr > priority then
                    succ_pr = priority
                end if
            else
                succ_pr = priority
                Open ← Open ∪ {succ}
            end if
        end if
    end for
end while
return failure
```

$$pr(x) = g(x) \tag{2.2.1}$$

Here, $g(x)$ is the path cost from start node to x.

2.2.2 A*

A* is an extension of Dijkstra's algorithm [5]. When goal-directed search techniques are applied to Dijkstra's algorithm, we get A*. A* also uses the best-first search approach. It uses the following priority function:

$$pr(x) = f(x) \tag{2.2.2}$$

$$f(x) = g(x) + h(x) \tag{2.2.3}$$

Here, $h(x)$ is the path cost estimate from x to the goal node.

h is a heuristic function used to approximate distances from the current location to the goal state. For A* to guarantee correctness, i.e., always find the optimal path, the heuristic function must be admissible. For A* to guarantee that it never expands a node more than once, the heuristic function must be consistent.

2.3 Bidirectional Search

A bidirectional search does simultaneous forward and backward searches. The backward search is done from the goal to the start, using the reverse of the standard operators. In a problem space without invertible operators, the reverse operators for

a state s are those which generate s when applied to some other state. Both search directions have a frontier of the nodes generated so far. Any time the frontiers intersect — meaning the same node has been generated in both directions — a path has been found from the start to the goal. The first path is not necessarily the cheapest, so search continues until a solution is proven optimal.

Bidirectional search is faster than unidirectional search in many cases. For example, in a search problem modeled by a tree where both search directions have a branching factor b, and the distance from start to goal is d, each of the two searches have complexity $O(b^{d/2})$ (assuming the search meets in the middle) and the sum of these two search times is asymptotically much less than $O(b^d)$ that would result from unidirectional search.

A bidirectional search algorithm maintains an open and a closed list for each direction of the search. $Open_F$ refers to the open list for forward search and $Open_B$ refers to the open list for backward search. $Closed_F$ and $Closed_B$ are defined analogously. $prmin_F$ and $prmin_B$ are used for the minimum priority on $Open_F$ and $Open_B$. Different bidirectional search algorithms can use different priority functions for nodes on the open list. They can also

use different strategies for alternating between forward and backward search. Also, different termination conditions can be used as long as the solution is proven optimal with the given termination condition.

2.3.1 Bidirectional Dijkstra

Bidirectional Dijkstra runs Dijkstra's algorithm in both directions. The priority of node n on $Open_F$ is defined to be:

$$pr_F(n) = g_F(n) \tag{2.3.1}$$

$pr_B(n)$ is defined analogously. There are many strategies of alternating between forward and backward search. Different strategies can yield different performance. The strategies we used in the experiments are discussed in next chapter. In each iteration, Bidirectional Dijkstra expands a node either from forward or backward frontier. p is the cost of the least cost path found so far. ϵ is the least cost edge in the state space. Initially ∞, p is updated whenever a better solution is found. Bidirectional Dijkstra stops when

$$p \leq prmin_F + prmin_B + \epsilon \tag{2.3.2}$$

Algorithm 2 contains the pseudocode for Bidirectional Dijkstra.

Algorithm 2 Bidirectional Dijkstra(G, w, s, t, h)

```
s_pr ← g(s)
t_pr ← g(t)
Open_F ← {s}
Open_B ← {t}
for all n ∈ V − {s, t} do
    n_pr ← ∞
end for
p ← ∞
while Open_F ≠ ∅ AND Open_B ≠ ∅ do
    prmin_F = get-min(Open_F)
    prmin_B = get-min(Open_B)
    if prmin_F + prmin_B + ε ≥ p then
        return path for p
    end if
    if Forward frontier is expanded then
        n = delete-min(Open_F)
        Closed_F = Closed_F ∪ n
        for all succ ∈ n_successors do
            if succ ∈ Closed_F then
                continue
            else
                priority ← pr(succ)
                if succ ∈ Open_F then
                    if succ_pr > priority then
                        succ_pr = priority
                    end if
                else
                    succ_pr = priority
                    Open_F ← Open_F ∪ {succ}
                end if
            end if
            if succ ∈ Open_B AND g_F(succ) + g_B(succ) < p then
                p ← g_F(succ) + g_B(succ)
            end if
        end for
    else
        //Expand backward frontier analogously
    end if
end while
return failure
```

2.3.2 MM

MM is the first bidirectional heuristic search algorithm whose forward and backward searches are guaranteed to"meet in the middle" i.e., never expand a node beyond the solution midpoint [6].

MM runs an A*-like search in both directions, except that MM orders the Open list in a novel way. The priority of node n on $Open_F$, $pr_F(n)$ is defined to be:

$$pr_F(n) = max(f_F(n), 2g_F(n)) \tag{2.3.3}$$

$pr_B(n)$ is defined analogously. $c = min(prmin_F, prmin_B)$. On each iteration, MM expands a node with priority c. p is the cost of the cheapest solution found so far. Initially ∞, p is updated whenever a better solution is found. ϵ is the cost of least cost edge in the state space. MM stops when [6]

$$p \leq max(c, fmin_F, fmin_B, gmin_F + gminB + \epsilon) \tag{2.3.4}$$

MM has the following properties:

- **P1.** MM's forward(backward) search never expands a state with $g_F > C^*/2$ ($g_B > C^*/2$), i.e., its forward and backward searches meet in the middle.
- **P2.** MM never expands a node whose f-value exceeds C*.

- **P3.** MM returns a path of cost C*.
- **P4.** If there exists a path from start to goal and MM's heuristics are consistent, MM never expands a state twice.

Algorithm 3 contains pseudocode for MM.

Algorithm 3 MM(G, w, s, t, h)

```
g_F(start) ← 0
g_B(goal) ← 0
Open_F ← {start}
Open_B ← {goal}
p ← ∞
while Open_F ≠ ∅ AND Open_B ≠ ∅ do
    c ← min(prmin_F, prmin_B)
    if p ≤ max(c, fmin_F, fmin_B, gmin_F + gmin_B + ϵ) then
        return path for p
    end if
    if c = prmin_F then
        //Expand in forward direction
        choose n ∈ Open_F for which pr_F(n) = prmin_F and g_F(n) is minimum
        move n from Open_F to Closed_F
        for all successors succ of n do
            if succ ∈ Open_F ∪ Closed_F AND g_F(succ) ≤ g_F(n) + cost(n, succ)
then
                continue
            end if
            if succ ∈ Open_F ∪ Closed_F then
                remove succ from Open_F ∪ Closed_F
            end if
            g_F(succ) ← g_F(n) + cost(n, succ)
            add succ to Open_F
            if succ ∈ Open_B then
                p ← min(p, g_F(succ) + g_B(succ))
            end if
        end for
    else
        //Expand in backward direction analogously
    end if
end while
```

MM Region Analysis

Holte et al. [6] provides a framework that can be used to compare algorithms by doing region analysis. It divides the state-space into 9 disjoint regions. These regions are denoted by two letter acronyms. The first letter indicates the distance from the start(N=near, F=far, R=remote) and the second letter indicates the distance from the goal(N=near, F=far, R=remote). A state s is defined to be "near to the start" if $d(start, s) \leq C^*/2$, "far from the start" if $C^*/2 < d(start, s) \leq C^*$, and "remote" if $d(start, s) > C^*$. Classification with respect to distance from the goal is made analogously.

Chapter 3

Experimental Setup

3.1 Algorithms used

We used the following two algorithms in our experiments.

- A*: We used the A* algorithm for unidirectional heuristic search. Since in our problem sets, we have two different sets of edge costs (distance-based and time-based), we used different heuristics for each. Searches with distance-based edge costs used the Euclidean distance heuristic and searches with time-based edge costs used Euclidean distances weighted by the maximum speed possible on any edge in the state space. These heuristics are described below:
  - Distance-based edge costs: Searches with distance-based edge costs used the Euclidean distance as the heuristic. Euclidean distance is the straight-line distance between two points in Euclidean space.

In cartesian coordinates, if $p = (p_1, p_2, ...p_n)$ and $q = (q_1, q_2, ..., q_n)$ are two points in Euclidean n-space, then the euclidean distance e from p to q is given by the Pythagorean formula:

$$e(p, q) = \sqrt{(q_1 - p_1)^2 + (q_2 - p_2)^2 + ...(q_n - p_n)^2}$$

Euclidean distance is admissible and consistent.

- Time-based edge costs: For searches with time-based edge costs, dividing the distance by speed gives the time estimate to travel to the goal. To ensure admissibility, we divide Euclidean distances by the maximum speed possible over all edges in the state space to get the heuristic.

- Bidirectional Dijkstra

  We studied three variants of Bidirectional Dijkstra. The difference comes from the way we alternate between forward and backward search.

  - Variant 1: This variant expands the side with the lowest g-cost as suggested by [7] - also called MM0 [6]. This ensures that the searches meet in the middle. Nodes that are less than or equal to C*/2 distance away from the start are expanded by forward search and nodes that are less than or equal to C*/2 distance away from the

goal are expanded by backward search. We refer this variant as BD1 for short.

– Variant 2: This variant strictly alternates between forward and backward search as suggested by [1]. This ensures that the number of nodes expanded by forward search is equal to the number of nodes expanded by backward search. In this variant, the two searches need not meet at the solution midpoint. This variant performs better than variant 1 because it balances the amount the work done in each direction. However, if the branching factor is not equal in both directions, giving priority to the direction with lower branching factor should give better performance than expanding equal number of nodes in each direction. This is because lower branching factor implies lesser density of nodes. Thus giving preference to the direction with lower branching factor makes the search explore sparse regions quickly and make the two searches meet in the middle of densest regions. This reduces node expansions in dense regions. To that end, we studied variant 3.

- Variant 3: This variant uses the cardinality criterion from [8], giving preference to the frontier with the smaller open list. If the branching factor is same in both directions, this variant works similar to variant 2. But in other cases, it gives preference to the frontier with smaller open list (lower branching factor). In this variant also, the two searches need not meet at the solution midpoint.

Among the three variants, variant 3 performs the best and we use this variant for our analysis. We refer this variant as BD3 for short.

3.2 Data structures

Open and Closed List: Deciding the data structures depends mainly on the operations that need to be performed on the data and the number of times each operation will be performed. In our algorithms, we need to perform the following operations on Open and Closed Lists.

1. Get and Remove the best node (minimum f cost) from open list. A binary heap can be used to store nodes in the open list. Getting and removing the node with minimum f-cost in a binary heap takes O(log n) time.

2. Check if a node is present in open or closed list: Using a set or hash map as open and closed list, we can perform this operation in O(1) time.
3. Update the cost of a node in the open list if a better path is found. Using a binary heap with *update-key* operations, we can perform this operation in O(log n) time.

To best combine all the three requirements, we used a specialized data structure. A hash table is created that maps the hash of a node to the node-id. An indexed array is created that maps the node id to the node. The binary heap stores the node id for all the nodes in the open or closed list . It does comparisons based on a comparison key (which is f cost of a node in most cases).

With the above data structure, the asymptotic running time of the primary operations on it are:

1. *Get and Remove best node(minimum f cost) from open list* takes $O(logn)$ time, where n is the number of nodes in open and closed list combined.
2. *Check if a node is present in open or closed list* takes $O(1)$ time. The hash table is checked for membership of hash value of the node in the table.

3. *Update the cost of a node in the open list if a better path is found* takes $O(logn)$ time. First the location of the node needs to be found in the heap. This operation can be done in $O(1)$ time. The indexed array gives reference to the node object and the node object stores the location of the node in the heap. The node at this location can be moved to the correct position in the heap using *heapify-up* or *heapify-down* operation in $O(logn)$ time.

3.3 Metrics

Different metrics can be used to compare the performance of different algorithms. Three of them are: number of nodes expanded, number of nodes generated, and time.

1. Nodes Expanded: This metric compares the number of nodes expanded by each algorithm. A node is called 'expanded' when it is removed from the open list and its neighbors are added to the open list. The lesser the number of nodes expanded, the better the algorithm. This metric is generic. It is not influenced by implementation details, and is machine and language independent. However, not being influenced by implementation details ignores some important factors. For example, A* has one very large open list and Bidirectional

Dijkstra has two small open lists. This does affect the performance of operations on Open and Closed list. But with this metric, that performance difference is not considered. This is a standard metric in the research community and so we have used this metric in our experiments.

2. Nodes Generated: This metric compares the number of nodes generated by each algorithm. A node is called 'generated' when it is added to the open list. This metric is generic too. We have not used this metric because it is correlated with nodes expanded.

3. Time: This refers to measuring and comparing the time taken by each algorithm to solve the same problems. Since time taken to solve a problem depends on factors like implementation details (the data structures used etc.), it is not machine and language independent. So we have not used this metric to compare and analyze the performance of different algorithms.

3.4 Dataset and Problem Selection

3.4.1 Dataset

We used four maps of USA road networks from DIMACS 9th challenge core instances. These maps are listed in Table 3.1. Each

map comes in two versions: physical distance and travel time arc lengths. Physical distance arc lengths correspond to the distance between two nodes. Travel time arc lengths correspond to the time it takes to travel between two nodes. These maps range from 264k to 1.5M nodes and 733k to 3.8M edges.

Map	**# of nodes**	**# of edges**	**# of problems**
FLA state	1,070,376	2,712,798	4,851
CO state	435,666	1,057,066	9,900
San Francisco Bay area	321,270	800,172	9,200
NY city	264,346	733,846	9,200

Table 3.1: Dataset

3.4.2 Problem selection

As mentioned in chapter 2, a search problem is defined by (G, w, s, t, h) and the goal is to return a path with cost $w(P^*)$. For each map, the graph G, weight function w, and heuristic function h stay constant for all test problems. The start node s and goal node t needs to be selected for each test problem. We aim to select start and goal nodes from all regions of the map and to cover variety of path costs between them (small, medium, and long). To that end, we have two problem sets:

Problem Set 1: To create this data set we begin by overlaying a 10x10 grid of equal-sized rectangles on the map, dividing it into 100 slots, as shown in Figure 3.1. We then estimate the longest path in the graph (*maxDistance*) via Dijkstra search from points near the edge of the map. Given this, we use the pseudo-code in Algorithm 4 to build the problem set. We select a random node at various distances from random points in each slot by performing a Dijkstra search until we reach a node at that distance. This produces at most 10,000 test pairs, but due to limits on path lengths, it creates fewer problems on some maps.

Problem Set 2: After extensive experimentation with Problem Set 1, we found that the approach was biased, which we will describe shortly. As a result, we created a second data set. This set is identical to Problem Set 1 except that for each (s, t) pair in the data set, we also include (t, s).

Algorithm 4 Problem Selection

1: problems $\leftarrow \{\}$
2: **for** $i \in \{1, 2, ..., 100\}$ **do**
3:     **for** $j \in \{1, 2, ..., 100\}$ **do**
4:         $s \leftarrow$random node in slot$_i$
5:         $t \leftarrow$nearest node at dist. $\geq \frac{j}{100} * maxDistance$ from s
6:         Add (s, t) to problems
7:     **end for**
8: **end for**

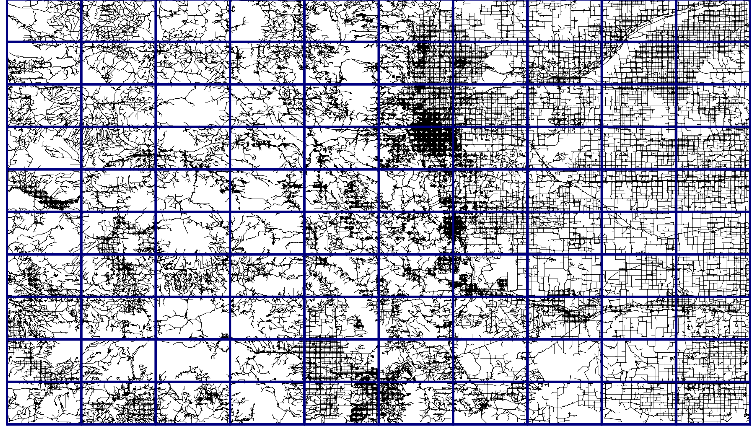

Figure 3.1: Grid overlay on CO map

Chapter 4

Experiments

4.1 Hypothesis

As seen in [4, 13], research papers in the road networks community often build upon Dijkstra's algorithm and use Bidirectional Dijkstra as their baseline algorithm in contrast to the heuristic search community which uses A* as their baseline algorithm. We run experiments on standard road networks from the USA to gain a deeper understanding of the performance of Bidirectional Dijkstra and A* on road networks and see if there is a principled explanation for different approaches. We look for the characteristics of road networks that can make one algorithm perform better than the other. Road networks are usually not equally dense throughout the map. There are regions of higher and lower density corresponding to cities and countryside. Our hypothesis is that density difference in different regions of a road map gives some advantage to Bidirectional Search over Unidirectional

Search. We also suspect that problem selection approach on road networks influences the performance of unidirectional search in comparison to bidirectional search. We ran experiments to test these hypotheses. They are explained in the next section.

4.2 Results

We ran A* and Bidirectional Dijkstra on the four maps found in Table 3.1. For each of the listed maps, we have distance and travel time graphs. Distance graphs have edge costs that correspond to the distance between two nodes. Travel time graphs have edge costs that correspond to the travel time between two nodes. We run experiments on both kinds of graphs for each map. We plot the work distribution for a single map in Figure 4.1. BD3 has the same performance on both problem sets; for now we look at the performance of A* on problem set 1. From this figure we can see that A* has better performance on the shorter problems and BD3 has better performance on the longer problems. This raises the question of why A* does worse (or BD3 does better) on these longer problems. To better understand and explain the difference in the behavior of the algorithms, we used the region analysis introduced by Holte et al [6]. The region analysis is explained in next section.

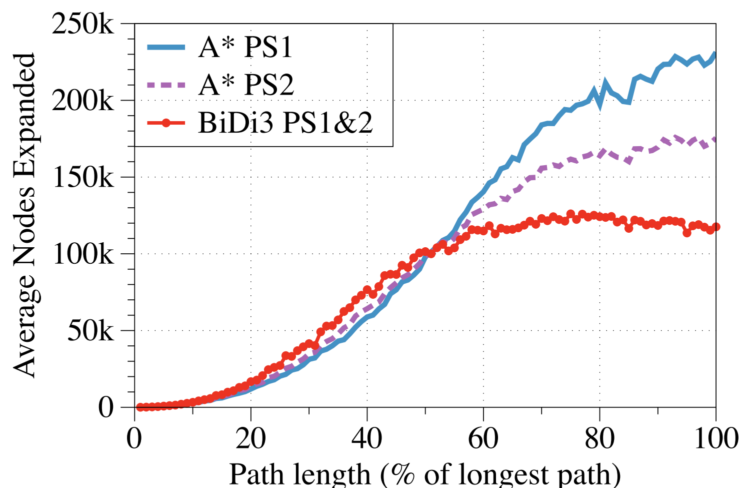

Figure 4.1: A* vs Bidirectional Dijkstra

4.3 Analysis

4.3.1 Regions

The region analysis introduced by Holte et al. in [6] divides the state space into 9 regions based on how far a state is from start and goal. If the distance between start and goal is C^*, the 9 regions are found in Table 4.1. These regions are illustrated in Figure 4.2. The regions are denoted by two letter acronyms. The first letter indicates the distance from start(N=near, F=far, R=remote) and the second letter indicates the distance from goal(N=near, F=far, R=remote). A state s is said "near to start" if $d(start, s) \leq C^*/2$, "far from start" if $C^*/2 < d(start, s) \leq C^*$, and "remote" if $d(start, s) > C^*$.

Region	Distance from Start	Distance from Goal
NN	$d \leq C^*/2$	$d \leq C^*/2$
NF	$d \leq C^*/2$	$C^*/2 < d \leq C^*$
NR	$d \leq C^*/2$	$C^* < d$
FN	$C^*/2 < d \leq C^*$	$d \leq C^*/2$
FF	$C^*/2 < d \leq C^*$	$C^*/2 < d \leq C^*$
FR	$C^*/2 < d \leq C^*$	$C^* < d$
RN	$C^* < d$	$d \leq C^*/2$
RF	$C^* < d$	$C^*/2 < d \leq C^*$
RR	$C^* < d$	$C^* < d$

Table 4.1: The definition of the regions used for our analysis.

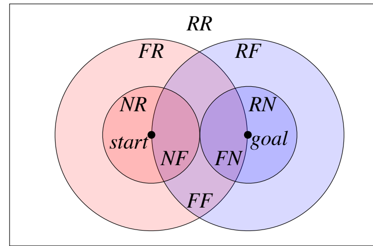

Figure 4.2: .
The NN region is not shown.

Classification with respect to distance from goal is made analogously.

We call the regions that are near to the start (NN, NF, and NR) as N* and regions that are near to the goal (NN, FN, and RN) as *N.

As long as an admissible heuristic is used, we expect A* to only expand nodes in NN, NF, NR, FN, FF, and FR because A* never expands a node with f-*cost* $> C^*$, where C^* is the optimal path cost.

Bidirectional Dijkstra (variant 1) only expand nodes in N* and *N. This is because forward and backward search never expand a node greater than C*/2 distance away from start and goal respectively. This causes them to meet exactly in the middle. And thus no expansions are done in FF, FR, RF, and RR regions.

Bidirectional Dijkstra (variant 2 and variant 3) may do more work than variant 1 in FF, FR, and RF regions because the two searches need not meet exactly in the middle but they tend to perform better in NF or FN than variant 1 and overall they perform better than variant 1.

Our experiments studied the performance of A* and BD3 in each of these regions. We found that there were not significant node expansions in NN, FR, RF, and RR by both algorithms. So, the regions of interest are: NF, NR, FN, FF, and RN.

Table 4.2 and Table 4.3 show the average nodes expanded by both algorithms in each of these regions:

State	A^*_{Total}	$BD3_{Total}$	A^*_{NF}	$BD3_{NF}$	A^*_{NR}	$BD3_{NR}$	A^*_{FN}	$BD3_{FN}$	A^*_{FF}	$BD3_{FF}$	A^*_{RN}	$BD3_{RN}$
CO dist	**128,846**	141,760	59,217	**46,141**	**14,230**	16,386	**37,487**	44,617	17,879	**14,916**	**0**	15,965
NY dist	**93,004**	94,667	44,731	**35,622**	**12,081**	12,212	**21,445**	28,581	14,199	**8,710**	**0**	8,923
FLA dist	**508,912**	531,785	234,781	**191,607**	**72,540**	81,537	161,307	**60,739**	40,264	**30,851**	**0**	50,561
BAY dist	**78,334**	97,654	33,265	**28,408**	**11,356**	13,873	**23,281**	29,453	10,333	**8,305**	**0**	15,180

Table 4.2: Problem Set 1: Average node expansions on graphs with edge costs representing distance.

State	A^*_{Total}	$BD3_{Total}$	A^*_{NF}	$BD3_{NF}$	A^*_{NR}	$BD3_{NR}$	A^*_{FN}	$BD3_{FN}$	A^*_{FF}	$BD3_{FF}$	A^*_{RN}	$BD3_{RN}$
CO time	174,335	**128,117**	78,856	**49,194**	21,212	**12,358**	**36,960**	37,053	36,195	**16,740**	**0**	9,959
NY time	106,611	**77,939**	47,073	**30,989**	16,699	**9,168**	**16,969**	19,925	23,525	**11,425**	**0**	5,855
FLA time	**465,797**	498,913	221,600	**179,177**	**54,768**	73,797	**139,786**	159,719	49,445	**37,913**	**0**	43,961
BAY time	106,894	**91,420**	46,906	**31,744**	15,314	**10,827**	**22,922**	26,878	21,039	**11,012**	**0**	9,327

Table 4.3: Problem Set 1: Average node expansions on graphs with edge costs representing travel time.

Considering the averages in the table above, Bidirectional Dijkstra performs better than A* in NF and FF regions and A* performs better than Bidirectional Dijkstra in NR, FN, and RN region.

Region	Stronger Algorithm
NF	Bidirectional Dijkstra (8 of 8 maps)
NR	A* (5 of 8 maps)
FN	A* (7 of 8 maps)
FF	Bidirectional Dijkstra (8 of 8 maps)
RN	A^* (8 of 8 maps)

Table 4.4: Region Comparison: Bidirectional Dijkstra and A*

4.3.2 Example Problem

Figure 4.3 shows a problem instance on NY city map. Points S and G represent start and goal respectively. Figure 4.4(a) shows the nodes expanded by A* for finding the path from start to goal. It expands about 96% of the total nodes and the majority of the nodes in N* and FF. This trend is found in all experimen-

tal results. In the majority of the experimental results, it gets significant pruning in the regions near the goal (*N). But for this problem instance, the Euclidian distance heuristic is very weak, so it does not get significant pruning in *N. Figure 4.4(b) shows the nodes expanded by A* in the reverse search. It expands about 60% of the total nodes and the majority of the nodes in N* (*N of forward search). Since the heuristic is more accurate in the reverse direction, it gets about 30% pruning in *N (N* of forward search).

The reasons that A* performs better in the reverse direction are: 1) The heuristic is stronger in the reverse direction, and 2) The size of N* is one-third of the size of *N in the reverse direction. We see that A* can give different performance for the same problem instance based on the direction of search. But, A* does not have the ability to switch directions. It must choose a direction and then only search in that one direction. Thus, it cannot use the knowledge of N* and *N region sizes that would indicate which search direction is better.

Bidirectional search, however, will discover this during search and can focus its search in the smaller region. Figure 4.4(c) shows the nodes expanded by BD1. It expands about 60% of the to-

tal nodes. It does not expand any node in the FF region while A* expands majority of the nodes in that region. The nodes pruned between (a) and (c) are primarily in FF. (FF region is near where the two searches meet). Since N* is much bigger than *N for this problem instance, BD1 does much more work in the forward direction than in the backward direction increasing the total number of node expansions. We can get significant performance improvement by expanding more nodes in the smaller frontier to maximize savings in the larger frontier. BD3 does that.

Figure 4.4(d) shows the nodes expanded by BD3. It expands about 50% of the total nodes. This variant gives preference to the frontier with a smaller open list and thus does some more work in the backward direction and saves a lot of work in the forward direction. It trades-off extra work of about 20% in FF region to get savings of about 40% in N* regions. Another reason why BD3 does well is that the problem we are studying has unbalanced regions. If the region sizes were all equal, the balancing wouldn't matter.

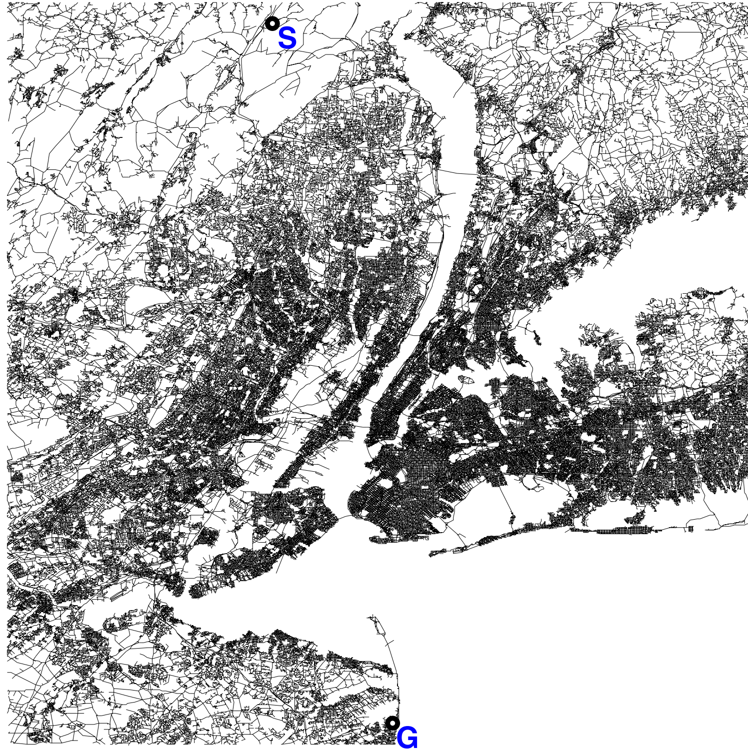

Figure 4.3: NY city map

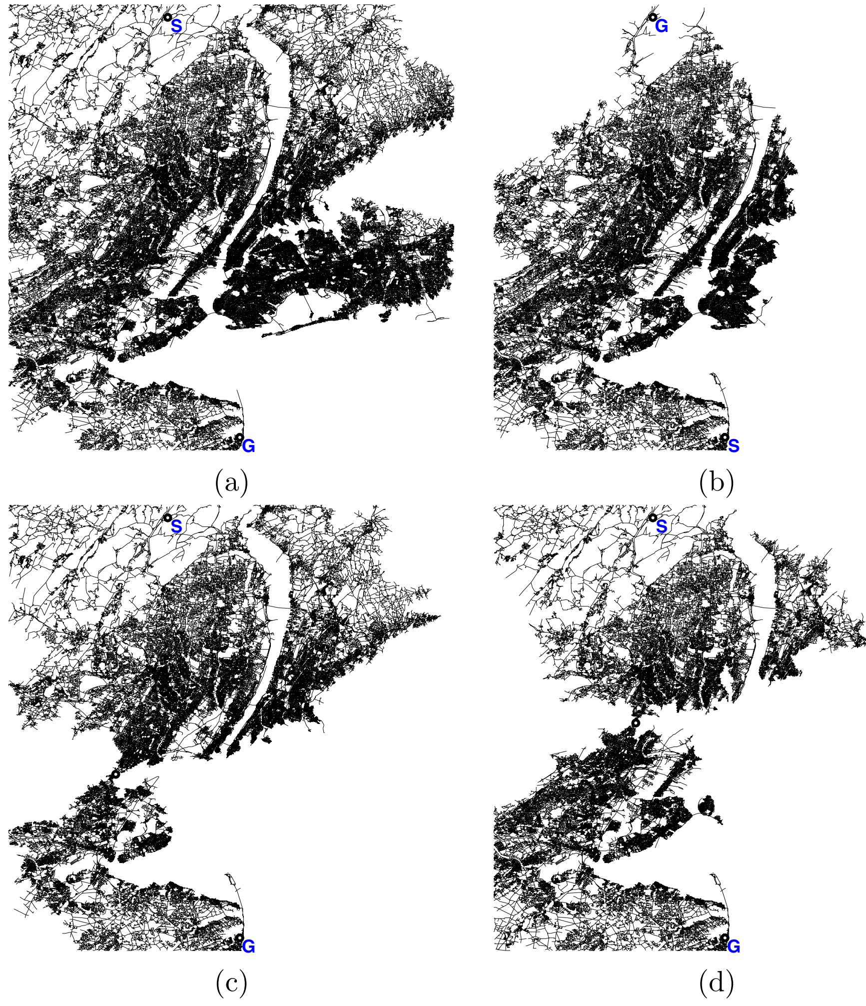

Figure 4.4: Example problem: Node Expansions

4.3.3 Analysis

Now we analyze the aggregate results over all problems.

1. Region Performance

   1. NF : We expect A* to expand many nodes in NF. These nodes have $g_F(n) \leq C^*/2$. So A* would prune them only if the condition $h_F(n) > C^*/2$ is true for a node n. With Euclidian distance as a heuristic, we indeed found that A* expands the majority of nodes in NF. One might expect BD3 to expand more nodes in NF than A* because A* prunes NF with a heuristic while BD3 does not use a heuristic. However, BD3 does some pruning based on other factors:

      * We use the termination condition $gmin_F + gmin_B + \epsilon$ in Bidirectional Dijkstra. So the forward search does not need to expand all nodes in NF before it meets the backward search. The use of ϵ will prevent expansion of some nodes in NF.
      * If the density of nodes in FN and RN is less than the density of nodes in NF and NR or the edge costs of nodes in FN and RN are greater than the edge costs

of nodes in NF and NR on average, then the backward search will move faster than the forward search and thus the two searches will not meet exactly in the middle. Instead, they will meet in the region NF and thus the forward search does not expand all nodes in NF before it meets the backward search.

We found that BD3 performs better than A* in NF on average on all maps.

2. NR: We expect A* to expand fewer nodes in NR than in NF. Nodes in NR region have $g_F(n) \leq C^*/2$. So A* would prune them if the condition $h_F(n) > C^*/2$ is true for a node n. However, the real distance from a node in these regions to the goal node lies between C^* and $3C^*/2$. Even if the heuristic underestimates the cost by 2 to 3 times, it will still prune nodes in NR. BD3 also prunes nodes in this region if forward search intersects with the backward search before expanding all nodes in NR. The trade-off between how accurate the heuristic is in A* and how quickly the two frontiers meet in BD3 determines which algorithm performs better in NR.

In 62% of our maps, on average A* performs better than BD3 in this region.

3. FN : We expect A* to expand far fewer nodes than BD3 in FN. These nodes have $g_F(n) > C^*/2$ and being relatively close to the goal, we expect the heuristic values for those nodes to be very accurate. BD3 also prunes some nodes in FN by the analogous reasoning as for the region NF. However, because of a relatively more accurate heuristic in FN, it cannot quite offset the pruning done by A* in this region.

4. FF: BD3 trades a small amount of work in FF to save work in NF and FN (most of the savings are in NF). The trade-off between accuracy of a heuristic and how many nodes BD3 expands in FF determines which algorithm performs better in this region. In all of our maps, on average BD3 performs better than A* in this region.

5. RN: We expect A* to perform better than BD3 in this region because A* does not expand any node in this region while Bidirectional Dijkstra expands some nodes in this region.

2. Performance Difference

   Here we combine strong and weak regions for each algorithm to see if there is a correlation between the size of these regions and the performance comparison of both algorithms.

   – NF+FF: BD3 is stronger than A* in NF+FF. This is shown in Table 4.5

   – NR+FN+RN: A* is stronger than BD3 in NR+FN+RN. This is shown in Table 4.5

State	A^*_{NF+FF}	$BD3_{NF+FF}$	$A^*_{NR+FN+RN}$	$BD3_{NR+FN+RN}$
CO Dist	77,096	**61,058**	**51,717**	76,970
NY Dist	58,930	**44,332**	**33,527**	49,717
FLA Dist	275,046	**222,458**	**233,848**	305,122
BAY Dist	43,598	**36,713**	**34,638**	58,454
CO Time	115,052	**65,935**	**58,173**	59,371
NY Time	70,598	**42,414**	**33,668**	34,949
FLA Time	271,045	**217,091**	**194,555**	277,478
BAY Time	67,946	**42,757**	**38,236**	47,033

Table 4.5: Problem Set 1: Average node expansions in regions

   – $\Delta(\Delta(NF+FF), \Delta(NR+FN+RN))$: $\Delta(NF+FF)$ is the difference in nodes expanded by A* and Bidirectional Dijkstra in $(NF+FF)$ region and $\Delta(NR+FN+RN)$ is the difference in nodes expanded by A* and Bidi-

rectional Dijkstra in $(NR+FN+RN)$ region. The difference between $\Delta(NF+FF)$ and $\Delta(NR+FN+RN)$ is directly correlated with the difference in total performance of A* and Bidirectional Dijkstra. We plotted a scatter plot to confirm this observation. The greater the value of $\Delta(NF+FF)$ and the smaller the value of $\Delta(NR+FN+RN)$, the better Bidirectional Dijkstra performs against A^*. Figure 4.5 shows this observation on a single map. X-axis in the chart represents the difference in $\Delta(NF+FF)$ and $\Delta(NR+FN+RN)$. Y-axis represents the difference in total nodes expanded by A* and Bidirectional Dijkstra.

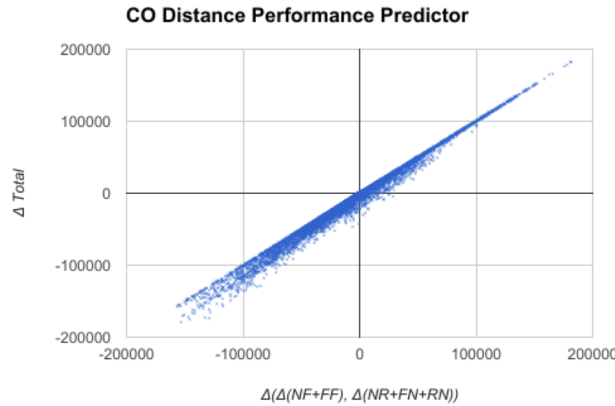

Figure 4.5: Performance Predictor

3. Map-Based Explanation

– There is a crossover point between the size of NF+FF and NR+FN+RN regions. In Figure 4.6 we plot the size of each of these regions as the problem length increases. We find that NR+FN+RN starts shrinking after a certain point, while NF+FF keeps growing. That is, on harder problems the region sizes become more unbalanced.

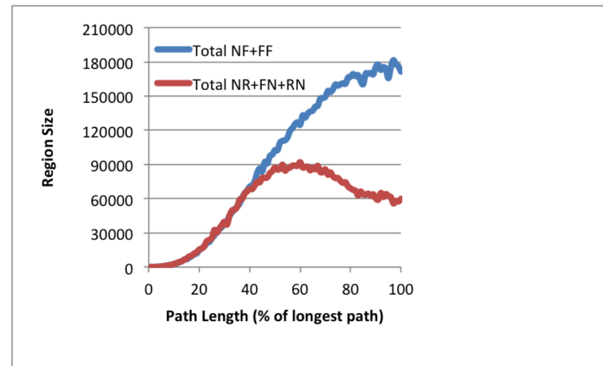

Figure 4.6: Problem Set 1: Region Distribution

– The density of nodes around the goal node decreases for large distances. This is because, for majority of the test problems for large distances, starting point is selected uniformly across the map but because of longer path lengths goal tends to be pushed toward the edges of the map and the density of nodes near the edges of a map is less than the density of nodes in the center of the map.

– A* does less pruning in sparse regions than it does in dense regions. We know that density of nodes around goal decreases for large distances. We define node expansion percentage as the fraction of nodes expanded in a region. We calculated node expansion percentage by both algorithms in NR+FN+RN region. Figure 4.7 shows this on a single map. We see that A*'s node expansion percentage increases with path cost (and low density) and it eventually crosses over Bidirectional Dijkstra's node expansion percentage (which stays relatively constant throughout the density difference).

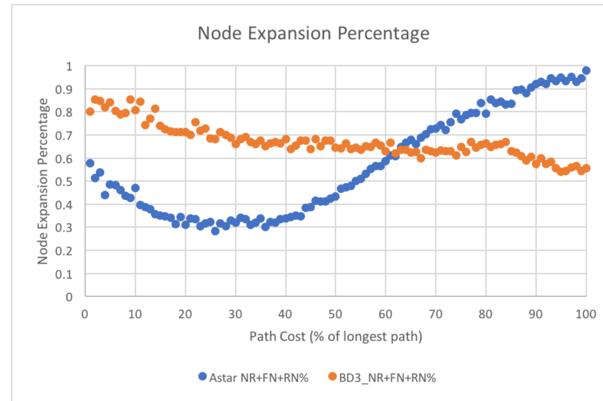

Figure 4.7: Node expansion percentage

– The crossover point for total nodes expanded is close to the crossover point for $\Delta(NF+FF)$ and $\Delta(NR+FN+RN)$. This is shown in Figure 4.8

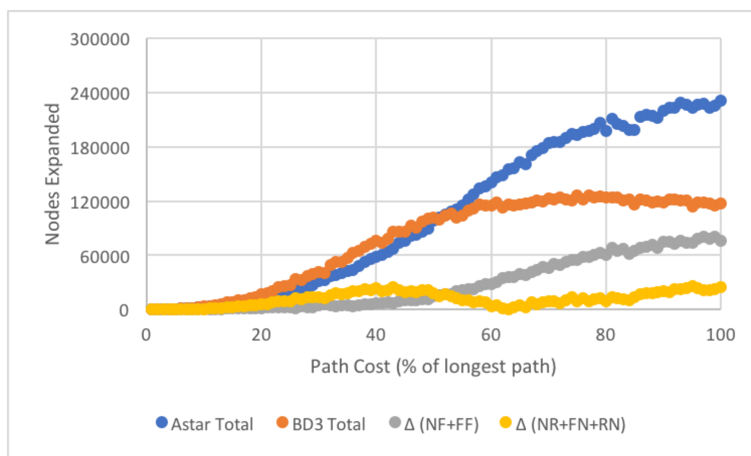

Figure 4.8: Crossover points for $\Delta(NF + FF)$ and $\Delta(NR + FN + RN)$

4.3.4 Explanation

The analysis indicate that $\Delta(NF + FF)$ and $\Delta(NR + FN + RN)$ can predict which algorithm will perform better. The analysis also show that there is a relation between the density of nodes or size of the region NR+FN+RN and the strength of A*. We see that Bidirectional Dijkstra performs better than A* in the NF+FF region. A* performs better than Bidirectional Dijkstra in the NR+FN+RN region. The overall performance depends on the size of these two regions and the pruning each algorithm does in these regions.

- Size of two regions: We see that there is a crossover point between the size of regions NF+FF and NR+FN+RN. After a certain point, the size of NR+FN+RN becomes less

than NF+FF. Since the region where A* is strong becomes smaller than the region where A* is weak, it does not get significant pruning to offset pruning done in regions NF+FF by Bidirectional Dijkstra.

- Pruning in each region: We see that the pruning done in NR+FN+RN by A* keeps decreasing as the density of the region decreases and there is crossover point after which A* does less pruning in NR+FN+RN than Bidirectional Dijkstra. In other words, as the regions around goal get sparser, A* performs worse. A* does less pruning for sparse graph than for dense graphs.

Both of the above factors favor Bidirectional Dijkstra over A* for scenarios where NF+FF becomes much bigger than NR+FN+RN and the density of nodes around goal becomes low. In such scenarios, the bidirectional nature of the search allows it to adapt. BD3 gives preference to the frontier with lower branching factor and thus fewer nodes are explored in denser regions of the state space. A* has to keep exploring the regions in the forward direction and thus does not have this ability to adapt.

Part of this density difference is from problem selection. We expect region sizes to be balanced on average, but they are not.

To offset the bias introduced by density difference, we ran experiments with Problem Set 2. As shown in Tables 4.2 and 4.7 and in Figure 4.1, the total BD3 numbers are unchanged, however now the work done in each region is balanced. This is because BD3 now gets the savings described in the last section half the time in the forward direction and half the time in the backwards direction. But, A* does significantly better because NF+FF is more balanced in comparison to NR+FN+RN. This is illustrated in Figure 4.9. This confirms our explanation that density difference is making the difference here. However, this change still does not make A* better than Bidirectional Dijkstra in all graphs. Consider the following example to understand the reason: We run a test problem from start and goal where density around goal is much less than density around start. Bidirectional Dijkstra expands 50% of the total nodes and A* expands 75% of the total nodes. Now if we run the same test problem from goal to start, Bidirectional Dijkstra expands 50% of the total nodes and A* expands 40% of the total nodes. Since we are looking at the average results, Bidirectional Dijkstra expanded 50% nodes overall and A* expanded 57% of nodes.

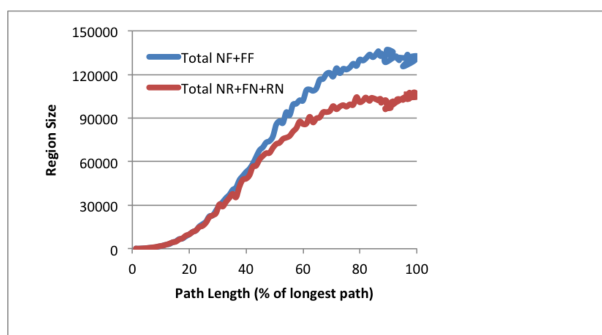

Figure 4.9: Problem Set 2: Region Distribution

This does make A* stronger than before. However, it was not able to completely offset the bias.

State	A^*_{Total}	$BD3_{Total}$	A^*_{NF}	$BD3_{NF}$	A^*_{NR}	$BD3_{NR}$	A^*_{FN}	$BD3_{FN}$	A^*_{FF}	$BD3_{FF}$	A^*_{RN}	$BD3_{RN}$
CO dist	**126,464**	141,138	56,996	**45,220**	**14,341**	16,072	**38,355**	45,220	17,374	**14,838**	**0**	16,072
NY dist	**85,747**	94,721	37,978	**32,113**	**9,571**	10,562	**25,050**	32,113	12,718	**8,741**	**0**	10,561
FLA dist	**479,827**	531,785	212,715	**182,315**	**56,877**	66,049	170,099	**182,315**	40,116	**30,851**	**0**	66,049
BAY dist	**78,623**	97,654	33,814	**28,931**	**12,301**	14,500	**22,102**	28,930	10,310	**8,305**	**0**	14,500

Table 4.6: Problem Set 2: Average node expansions on graphs with edge costs representing distance.

State	A^*_{Total}	$BD3_{Total}$	A^*_{NF}	$BD3_{NF}$	A^*_{NR}	$BD3_{NR}$	A^*_{FN}	$BD3_{FN}$	A^*_{FF}	$BD3_{FF}$	A^*_{RN}	$BD3_{RN}$
CO time	162,438	**128,117**	64,192	**43,124**	18,100	**11,159**	45,369	**43,123**	33,832	**16,740**	**0**	11,159
NY time	92,269	**77,939**	35,335	**25,457**	11,624	**7,512**	**23,103**	25,457	20,724	**11,425**	**0**	7,512
FLA time	**434,880**	498,913	198,247	**169,448**	**40,978**	58,879	**148,374**	169,448	47,165	**37,913**	**0**	58,879
BAY time	98,253	**91,420**	39,919	**29,311**	12,950	**10,077**	**25,615**	29,311	19,149	**11,012**	**0**	10,077

Table 4.7: Problem Set 2 : Average node expansions on graphs with edge costs representing travel time.

Chapter 5

Conclusions

This thesis compares A* and Bidirectional Dijkstra search on Road Networks. We saw that one algorithm does not always perform better than the other. The characteristics of a map and a test problem play a role in determining which algorithm should be preferred. We used a region analysis framework to understand the behavior of both algorithms and found that Bidirectional Dijkstra performs better than A* in NF and FF regions and A* performs better than Bidirectional Dijkstra in NR, FN, and RN regions. We found that Bidirectional Dijkstra performs better than A* when the $(NF+FF)$ region is bigger than the $(NR+FN+RN)$ region. In such scenarios, we show that Bidirectional Dijkstra has better performance because the bidirectional nature of the searches allow it adapt – expanding fewer nodes in the denser regions of the state space. Additionally, we show that a simple problem selection scheme can be biased against A* at longer path lengths. These results suggest more broadly where bidirectional

search is likely to be effective – in state spaces where there is variable density in the different regions. In such problems Bidirectional Dijkstra is able to adapt and expand fewer states where A* cannot.

Bibliography

[1] Kazi Shamsul Arefin and Aloke Kumar Saha. A new approach of iterative deepening bi-directional heuristic front-to-front algorithm (IDBHFFA). *International Journal of Electrical and Computer Sciences (IJECS-IJENS)*, 10(2), 2010.

[2] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[3] Hazem EI-Affy, David Jacobs, and Larry Davis. Multi-scale video cropping. *MM*, pages 97–106, 2007.

[4] Andrew V. Goldberg and Chris Harrelson. Computing the shortest path: A* search meets graph theory. In *Proc. SODA '05 Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 156–165, 2005.

[5] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Trans. Systems Science and Cybernetics*, 4(2):100–107, 1968.

[6] Robert C. Holte, Ariel Felner, Guni Sharon, and Nathan R. Sturtevant. Bidirectional search that is guaranteed to meet in the middle. In *AAAI Conference on Artificial Intelligence*, 2016.

[7] T. A. J. Nicholson. Finding the shortest route between two points in a network. *The Computer Journal*, 9(3):275–280, 1966.

[8] Ira Pohl. Bi-directional and heuristic search in path problems. Technical Report 104, Stanford Linear Accelerator Center, 1969.

[9] Steve Rabin and Nathan R Sturtevant. Pathfinding architecture optimization. *Game AI Pro: Collected Wisdom of Game AI Professionals*, pages 241–252, 2013.

[10] Dimitris Sacharidis and Panagiotis Bouros. Routing directions: Keeping it fast and simple. In *Proceedings of the 21st ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 164–173, 2013.

[11] Vitaly Surazhsky, Tatiana Surazhsky, Danil Kirsanov, Steven J. Gortier, and Hugues Hoppe. Fast exact and ap-

proximate geodesics on meshes. *SIGGRAPH*, pages 553 – 560, 2005.

[12] Frank W. Takes and Walter A. Kosters. Adaptive landmark selection strategies for fast shortest path computation in large real-world graphs. *WI-IAT*, pages 27 – 34, 2014.

[13] Dorothea Wagner, Thomas Willhalm, and Christos Zaroliagis. Geometric containers for efficient shortest-path computation. *Journal of Experimental Algorithmics*, 2005.

[14] Yue Wang, Jie Gao, and Joseph S.B. Mitchell. Bounday recognition in sensor networks by topological methods. In *Proceedings of the 12th annual international conference on Mobile computing and networking*, 2006.

[15] Songhua Xing and Cyrus Shahabi. Scalable shortest paths browsing on land surface. *GIS*, pages 89 – 98, 2010.