A Comparative Analysis of Shortest Path Algorithms on the East Bank Gopher Way Map and Minneapolis Map

Esthi Erickson, Elizabeth Steenberg, Angel Tovsen, Gaoxiang Luo*

University of Minnesota, Twin Cities
Department of Computer Science and Engineering
(eric3722, steen270, tovse006, luo00042)@umn.edu

Abstract

Route planning is an important aspect in many fields of computer science and is also applicable to everyday activities like travel in order to optimize navigation efforts. This can be seen by the amount of attention and research the topic has received over the past couple of decades. Our study aims to build off of prior work and apply it to our Gopher Way tunnel system on the East Bank campus and the city of Minneapolis. These tunnels are often difficult to navigate for new and returning students and staff, so managing to figure out the most efficient path from one place to another is no easy feat. We hypothesized that the A* and its variants search algorithms would perform best cost and time wise for both coordinate graphs. We then applied multiple search algorithms to these coordinate graphs and found that Breadth-First Search performed best time wise for the Gopher Way map, but A* with the use of the Manhattan distance heuristic performed best for the Minneapolis map. This information provides important insight into route planning on small and large coordinate graphs and can easily be applied to other graphing situations.

1 Introduction

The issue our project is addressing refers to the Gopher Way tunnels within the East Bank Campus of the University of Minnesota - Twin Cities. These tunnels are often hard to understand for both new and returning students, and the most efficient way to use them is often unknown. In an attempt to solve this problem, we will study and experiment with different traversals through the tunnels using popular search algorithms found within the AI community to find the most optimal paths between East Bank destinations. We plan to discuss, implement, and compare the search algorithms Dijkstra[1], Greedy Best-First, Breadth-First, and different variations of A* with two different weights and four different heuristics, with the coordinates corresponding to the buildings and locations that the tunnels connect. Figure 1 shows the current Gopher Way tunnels located on East Bank, with the maroon lines representing the tunnels themselves. While this map is helpful, it is not commonly used, and has not been studied in the way we plan to study it, therefore our project will lead to new and useful information available to anyone interested.

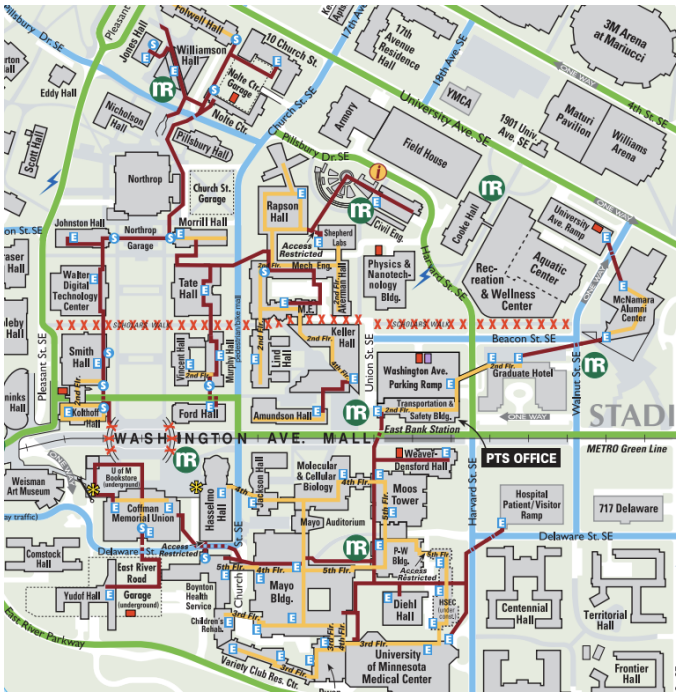

Figure 1: East Bank Campus - Gopher Way tunnels

The issue of not knowing the most optimal paths between destinations within the Gopher Way tunnels of the East Bank Campus contribute to getting lost, being late to class, and misuse or no use of the tunnels at all. These tunnels were created in order to help students travel more quickly and safely to and from locations on campus, as they provide protection from the environment and other dangerous situations, and this lack of knowledge doesn't allow them to be used most efficiently.

Many problems within the AI community have been

*These authors contributed equally to this work.

[1]This report refers to Dijkstra's Search as Uniform Cost Search in some instances

studied and discussed that relate to search algorithms and optimal path traversals. One such problem we studied this past year referred to path traversals between different areas in Romania. This problem included the application of few of the search algorithms we plan to use in our project, revealing that they are indeed capable of finding the most optimal paths, with A* consistently giving optimal results. With the success found in this problem as well as many others like it, it seemed fitting to apply and adapt these past experiments to a resource so commonly misused, but with a lot of potential, right here on our campus. From the results of the Romania problem, we hypothesize that the A* search algorithm will be the most cost and time efficient when applied to the Gopher way tunnels on East Bank.

In carefully studying and applying the coordinates of the tunnels to the most efficient search algorithms, we hope that our results aid in assisting all students and users of the tunnels to get to and from their most important locations safely and efficiently. Due to the lack of space that the tunnels cover on the East Bank Campus, our data set was relatively small. In order to account for this, we also compared the search algorithms and their path finding ability on the Minneapolis data set which was found to be much larger and therefore more informative. Running the experiments on both data sets still led to helpful results, which confirmed our trust in these algorithms and their ability to find optimal traversal paths. In this paper, our main contribution involves:

- We proposed the Gopher Way map dataset for graph search by manually collecting the data from Google map.
- We suggested the breadth first search works the best for Gopher Way tunnels route planning for various reasons.
- We figured that Manhattan A* search algorithm has the most promising performance in Minneapolis map in terms of time and memory efficiency.

2 Related Work

2.1 Overview

The shortest path algorithm is one of the well-studied topics within AI community, and it brings numerous applications to the human society, including route planning, traffic control, computer sames and even path finding in social networks [15]. According to a recent survey by Madkour et al. [9], typically the search space of shortest path algorithm can be either trees or graphs, where tree can be binary trees or multi-branch trees, and graphs can be general graph consisting of vertices, planar graph with continues edges that need not be straight, etc. In addition, the graph can be static, where the vertices and the edges remain the same, or dynamic, where the vertices and the edges can be updated or deleted over time. More specifically, the edges in a graph can be either directed or undirected, as well as the weights over the edges can either be negative or non-negative. Furthermore, the majority of shorest-path algorithms fall into two major categories [16]. The first one is single-source shortest path (SSSP), where the goal is to find the shortest path form one vertex to all other vertices, where the other is all-pairs shortest-path (APSP), which namely is to find the shortest paths between all pairs of vertices. Due to the variability of how one defines a search problem, there is no one general algorithm that is capable of solving all variants of the shortest-path problems. The choice of which algorithm is the best fit depends on the characteristic of the graph and the required application.

2.2 Shortest Path Algorithms

Algorithm 1 Breadth-First Search

BFS($problem$)
  $node \leftarrow NODE(problem.INITIAL)$
  if $problem.IS-GOAL(node.STATE)$ **then**
    return $node$
  else
    $frontier \leftarrow$ a fifo queue, with node as an element
    $reached \leftarrow \{problem.INITIAL\}$
  end if
  while $!IS-EMPTY(frontier)$ **do**
    $node \leftarrow POP(frontier)$
    for all $child$ in $EXPAND(problem, node)$ **do**
      $s \leftarrow child.STATE$
      if $problem.IS-GOAL(s)$ **then**
        return $child$
      end if
      if s is not $reached$ **then**
        add s to $reached$
        add $child$ to $frontier$
      end if
    end for
  end while
  return $failure$

Breadth-First Search Algorithm In the Breadth-First Search (BFS) algorithm, represented with pseudo code in Algorithm 1, the breadth of a node is explored before moving on to the next depth level. In other words the search moves along the edges between nodes, and unvisited sibling nodes take higher priority over child nodes. If all sibling nodes are explored, the search will return to the parent node to close the loop consisting of the current node, sibling nodes, and parent node [11]. Breadth-first search can be useful in local map-based exploration for autonomous mapping [11]. It can construct and continuously update a frontier tree using only local information and decide which node to visit next based on the BFS algorithm [11]. Unlike Dijkstra's and A*, this search does not assign costs or weights to nodes or edges, and will not get stuck or caught in a loop. In that sense, it is a simpler algorithm to conceptualize and implement but also may prove to be less efficient at finding the

goal or finding the shortest path, and tends to require a lot of memory. Breadth-First Search is one approach that has been used in finding the shortest path solution in a Cartesian area [5]. In this application, an object moves from point A (x_1, y_1) to point B (x_2, y_2) and should only bend (change directions) at the grid points and always step in parallel with the x-axis or y-axis. The object also should never cross the path that it has already explored [5]. BFS is a wide search compared to other search algorithms, since it visits all of a node's siblings and adjacent nodes before moving on to its children. This search algorithm has the advantage that it will not get stuck, and if there is more than one solution (which in the case of our research there likely will be), it will find all of the solutions including the smallest path solution [5]. The drawbacks that come with this are that the algorithm requires lots of memory since it stores all explored nodes (Cartesian coordinates in this case) in a tree and can have a long runtime. Although BFS is not the most efficient search algorithm memory-wise or cost-wise, it is important for our team to include this search in our experiment so that we have the ability to determine all possible paths from one node (campus building) to another, and so that we can compare its efficiency with Dijkstra's, Theta* and A*.

Greedy Best-First Search Algorithm The Greedy Best First algorithm is one that bases its state expansion decisions on the current heuristic values at that state. It works under the assumption that the lower heuristic values correlate with cheaper paths to the goal state. It is less used than other search algorithms, though, as it does not always provide the most optimal solution [8]. The inability to ensure an optimal solution puts this algorithm low on the list of ones to use when trying to find path traversals. This algorithm, with pseudo code represented in Algorithm 2, works by keeping track of both open and closed states. Open states refer to states that are ready for expansion, and closed states refer to states that have already been expanded in the past. Keeping track of these states allows for less repetition in the search for the goal, if it exists [7]. Greedy Best-First search will either return the first solution it finds, whether its the most optimal or not, otherwise it will return a failure. While we would not necessarily recommend using this algorithm to actually find an optimal path, we included it for further research and results.

Dijkstra's Algorithm Dijkstra's algorithm is used to find the shortest distance between two points of a graph. The graph's vertices can be used to represent locations and the edges to represent the routes to those locations. Dijkstra's algorithm is usually used for non-negative weight [1, 13] and is typically used to test functionality and performance[14]. Pseudo code for this algorithm is shown in Algorithm 3. In this paper, we plan to present Dijkstra's algorithm to solve the shortest path problem for routes based on the representation of the Gopher Way tunnels on the East Bank Campus.

Algorithm 2 Greedy Best-First Search

```
GreedyBFS()
  insert(state=initialState,h=initialHeuristic,counter=0)
  into searchQueue
  while !IS − EMPTY(searchQueue) do
    currentQueueEntry ← POP(searchQueue)
    currentState ← state from currentQueueEntry
    currentHeuristic ← heuristic from
    currentQueueEntry
    startingCounter ← counter from
    currentQueueEntry
    applicableActions ← [actionsapplicable] in
    currentState
    for all index in applicableActions ≥
    startingCounter do
      currentAction ← applicableActions[i]
      successorState ←
      currentState.apply(currentAction)
      if IS − GOAL(successorState) then
        return plan
      end if
      successorHeuristic ← heuristicValue of
      successorState
      if successorState < currentHeuristic then
        searchQueue.INSERT(currentState,
        currentHeuristic, i + 1)
        searchQueue.INSERT(successorState,
        successorHeuristic, 0)
        break
      else
        searchQueue.INSERT(successorState,
        successorHeuristic, 0)
      end if
    end for
  end while
```

This means that the path distances will always be positive. Previous studies use Dijkstra's algorithm to compare other algorithms' efficiency. We will do similar comparisons to A*, BFS algorithm, and Theta*.

A* Algorithm The A* search algorithm, which was discovered and formed in 1968 by Nils Nilsson, Bertram Raphael, and Peter Hart [6], is often used in path finding and graph traversal problems. A* allows for the use of a more directed search approach, making the shortest path (assuming one exists) be found more efficiently than previously used algorithms (Dijkstra's, for example) [12]. A* uses the concepts of heuristics, current path cost functions, and admissibility. To make use of these concepts, it incorporates a distance-plus-cost heuristic function by summing the cost of the path from the start to the current node with the admissible estimate of the cost of the path from the current

Algorithm 3 Dijkstra or Uniform Cost Search

```
Dijkstra(Graph, origin, destination)
  for all vertex in Graph do
    dist[vertex] := infinity
    previous[vertex] := undefined
    dist[origin] := 0
    Q := {Graph}
  end for
  while !IS − EMPTY(Q) do
    u := vertex in Q with smallest dist[]
    if dist[u] = infinity then
      break
    end if
    if u == destination then
      break
    end if
    remove u from Q
    for all neighborV ofU do
      alt := dist[u] + costBetween(u, v)
      if al < dist[v] then
        dist[v] := alt
        previous[v] := u
      end if
    end for
  end while
  S := {}
  u := destination
  while previous[u] == defined do
    S[0] = u
    u := previous[u]
  end while
  return S
```

Algorithm 4 A* Search

```
Main()
  g(sStart) := 0
  parent(sStart) := sStart
  open := Ø
  open.INSERT(sStart, g(sStart) + h(sStart))
  closed := Ø
  while open ≠ Ø do
    s : open.POP()
    if s = sGoal then
      return pathfound
    else
      closed := closedU{s}
      [UpdateBounds(s)]
    end if
    for all s' in succ(s) do
      if s' not in closed then
        if s' not in open then
          g(s') := infinity
          parent(s') := NULL
        end if
        UpdateVertex(s, s')
      end if
    end for
  end while
  return nopathfound
```

```
UpdateVertex(s,s')
  if g(s) + c(s, s') < g(s') then
    g(s') := g(s) + c(s, s')
    parent(s') := s
    if s' in open then
      open.REMOVE(s')
    end if
    open.INSERT(s', g(s') + h(s'))
  end if
```

node to the goal node [6]. Its cleverness was not all original, though, as the incorporation of measuring the distance from the start node to the current node was taken from Dijkstra's algorithm, and the incorporation of measuring the distance from the current node to the goal node was taken from the Best-First search algorithm [12]. Studies have been done that implement and compare A* and Dijkstra both unidirectionally and bidirectionally, and in both instances A* was found to outperform Dijkstra [12]. Due to a better understanding of the unidirectional approach, we will consider that within our project, under the assumption that it will still outperform the unidirectional Dijkstra approach as the study reported.

As time progressed, more versions of this search algorithm came to light based on the discovery of different needs in path planning. [6] investigated and discussed these different adaptations in great detail, offering further information on the success and adaptable usability of the A* search algorithm. The first adaptation to A* to be discussed is through the use of the B* search algorithm, which uses the concept of estimated intervals for distances between nodes vs. the single point-valued estimates that A* uses. The graph is then explored until the node with the best interval is found, with best referring to the one that's interval estimates the shortest path. This algorithm has been found to be used more in game playing vs. path planning, though. Another adaptation to A* has been found through the use of the D* search algorithm. This algorithm allows for estimates to change during runtime based on unexpected obstacles. The ability to change during the search, versus after and then repeating the search, allows for the most optimal path (containing obstacles) to be found quicker. The difference between this algorithm and A*, other than its consideration of obstacles within the terrain, is the fact that

it starts from the goal node and works backwards versus starting from the root node. Still, without the use of the adapted D*, [12] found that A* outperformed Dijkstra in the presence of obstacles. The two final adaptations discussed were the IDA* and SMA* algorithms. IDA* stands for Iterative Deepening A*, and SMA* stands for Simplified Memory Bounded A*. Both allow for less memory usage and similar behavior to A* otherwise. These adaptations were only possible because of the initial discovery of A*, which is a combination of Dijkstra and Best-First search. Each one incorporates the best aspects of A*, being its use of heuristics, current path-cost functions, and admissibility, while also incorporating their own techniques to make the overall search better for the specific issues they cover. In the end, it still seems that A* is the most widely used algorithm for path planning, and is the one that we have the best understanding of, so that is the one of its series that we will consider in our project. See Algorithm 4 for the pseudo code of this search algorithm.

Choosing Heuristics A big factor that plays into the efficiency of the A* search algorithm is the concept of heuristic functions. A heuristic acts as an estimated cost from the current position to the goal. A common heuristic used among path finding problems is the Manhattan distance between two points, which covers the distance measured along right angles in a lattice graph. Another common heuristic is the euclidean distance, which covers the straight line distance between two points. While both of these heuristics have proven themselves to allow for optimal paths to be found quickly with A*, researchers [2] have found that with increasingly larger maps, these heuristics are not performing as well as they could. In an attempt to improve searches, both time and space wise, they presented two new heuristics to use in path finding problems. One is the dead-end heuristic, which "avoids areas that lead to a dead-end" [2]. The other is the gateway heuristic, which recognizes that "moving through certain rooms can only lead to sub-optimal paths" [2]. In order to be able to compute these heuristics, the process makes use of state space abstraction in order to allow for better guidance in finding the estimates. These heuristics were found to perform better in terms of both time and space in comparison to the Manhattan distance heuristic, which is groundbreaking for newer applications that cover larger spaces.

The heuristics mentioned above are very efficient for large map spaces, and could be applied to the Gopher Way tunnels. Still, we decided to use more common path problem heuristics in our computations, which will include ones that consider the distance between the source and the destination. Since the Gopher Way tunnels within the East Bank campus do not represent an astronomically large space, we decided to use the following distance formulas: Euclidean, Manhattan, Chebyshev, and Hamming. Euclidean distance refers to the shortest distance between two points, Manhattan refers to the sum of all straight line distances between two points, Chebyshev refers to choosing the maximum distance between two points, and Hamming refers to the number of different positions between two points.

Algorithm 5 Weighted A* Search

Main()
  $g(sStart) := 0$
  $parent(sStart) := sStart$
  $open := \emptyset$
  $open.INSERT(sStart, g(sStart) + weight * h(sStart))$
  $closed := \emptyset$

Weighted A* The Weighted A* algorithm is identical to the usual A* algorithm, aside from the fact that it uses weighted heuristics in its computations. Use of the weights allows for the algorithm to ensure bounded and sub-optimal solution(s) [4]. Different variations of weights have been studied and implemented within the current A* algorithm, all of which relax different aspects of A* leading to faster results at the cost of possibly being sub-optimal vs. optimal. Both constant and constantly changing weights have been used, with both being found to be successful in aiding faster path finding [4]. Similarly to using different heuristics, the use of different weights allows for results to be found in different ways, sometimes being faster ways. We included a few variations of the Weighted A* algorithm in order to further our research and results. For the pseudo code of this algorithm, refer to the few lines of the Algorithm 5, where the function will change from g(sStart) + h(sStart) to g(sStart) + weight * h(sStart), and the rest of the algorithm will follow that of the normal A*, seen in Algorithm 4, starting at the while loop.

Theta* Algorithm The route planning algorithm of Theta* algorithm [3] is a variant of the search algorithm A* which means it can find near optimal solutions with run times similar to those of A*. Theta* propagates information along grid edges without constraining the path to just grid edges. Since A* does not always guarantee the most optimal solution since its headings are limited Theta*, which refers to the Basic Theta* and Angle-Propagation Theta* picks up where A* leaves off. Basic Theta* has been found to be simple and easy to implement in order to find short paths quickly. Angle-Propagation Theta* achieves worst-case complexity per vertex expansion that is constant by propagating angle ranges when it expands vertices rather than linear in the number of cells like Basic Theta*. However, Angle-Propagation Theta* is more complex, not as

Algorithm 6 Basic Theta* Search

Main()

$g(sStart) := 0$
$parent(sStart) := sStart$
$open := \emptyset$
$open.INSERT(sStart, g(sStart) + h(sStart))$
$closed := \emptyset$
while $open \neq \emptyset$ **do**
  $s : open.POP()$
  if $s = sGoal$ **then**
    return $pathfound$
  else
    $closed := closedU\{s\}$
    $[UpdateBounds(s)]$
  end if
  for all s' in $succ(s)$ **do**
    if s' not in $closed$ **then**
      if s' not in $open$ **then**
        $g(s') := infinity$
        $parent(s') := NULL$
      end if
      $UpdateVertex(s, s')$
    end if
  end for
end while
return $nopathfound$

UpdateVertex(s,s')

if $lineofsight(parent(s), s')$ **then**
  if $g(parent(s)) + c(parent(s), s') < g(s')$ **then**
    $g(s') := g(parent(s)) + c(parent(s), s')$
    $parent(s') := parent(s)$
    if s' in $open$ **then**
      $open.REMOVE(s')$
    end if
    $open.INSERT(s', g(s') + h(s'))$
  end if
else
  if $g(s) + c(s, s') < g(s')$ **then**
    $g(s') := g(s) + c(s, s')$
    $parent(s') := s$
    if s' in $open$ **then**
      $open.REMOVE(s')$
    end if
    $open.INSERT(s', g(s') + h(s'))$
  end if
end if

fast and finds slightly longer paths to the solution. Theta* has been used for path-planning and game problems. For this project, we decided to use Basic Theta*, which follows the A* algorithm exactly aside from a different use of $UpdateVertex()$. See Algorithm 6 for pseudo code of the main part of the algorithm, as well as the $UpdateVertex()$ algorithm.

2.3 Further Use

The information obtained through the in depth review of these algorithms aided in our ability to study and compare different path traversals through the Gopher Way tunnels within the East Bank Campus of the University of Minnesota - Twin Cities. The choice of comparing the Dijkstra, Greedy Best-First, A* with four different heuristics and two different weights, and Breadth-First search algorithms in terms of finding the most optimal path traversals through the tunnels have allowed for important differences and discoveries to be made. The use of these algorithms on other sets of coordinates, such as the Minneapolis coordinates, would also aid in providing optimal path traversals in a timely manner.

3 Experiment

3.1 Problem Definition

Given a set of vertices V from a csv file, a source vertex s, a destination vertex d, and a set of weighted edges E over the set V, the task is to find the shortest path between the source vertex and the destination vertex (i.e., to find the path that has the minimum weight in total). We will use a graph $G = (V, E)$ to represent the entire search space, where each edge can be either uni-directional or bi-directional. The edges have explicit weights corresponding to the distance in the real world, so they cannot be negative.

As our objective is the route planning system on the gopher way map for walking navigation, we will define the search space as an un-directed (i.e. bi-directional) graph consisting of vertices with edges of positive weight among them. The graph is static, which means the search space including the weights is fixed throughout the experiments, because realistically the gopher way tunnels don't have traffic and are not influenced by the weather.

3.2 Dataset

In order to develop this route planning system, we proposed the first quantitative gopher way map of East Bank campus of UMN made by our teammates. We utilized the Google map to manually drop pin points on the major turns and intersections of Gopher Way tunnels based on the map from the freshman guide of UMN, and we exported the geocoordinates of those points in the same fashion as the provided Minneapolis map in our UMN CSCI 4511W course materials. This Gopher Way map aims to navigate walking on campus, so it's bi-directional. As shown in Figure 2, the

Figure 2: The Gopher Way map in the graph search space.

Table 1: The statistics of Gopher Way map and Minneapolis map.

Maps	Vertices	Edges
Gopher Way	91	95
Minneapolis	946	1354

mapping may look slightly different from what's presented on the UMN freshman guide, because the geo-coordinate system that we used is World Geodetic System (WGS84), which is different from the geo-coordinate system of the underlying code of aima-python package, and we're not sure how to re-project them. It, however, wouldn't influence the behaviors of running shortest path algorithms on this map. After visualizing the gopher map, we quickly realized some algorithms would end up with the same results because there are not many circles on the map. Hence, we also built the search space of Minneapolis map (see Figure 3), that has many circles, and the statistics of both maps are shown in Table 1.

3.3 Experimental Design

On the Gopher way map, we conducted three groups of experiments. First of all, we will compare and contrast breadth first search algorithm and Dijkstra's algorithm as representatives of the uninformed search and A* algorithm and greedy best first search algorithm as representatives of the informed search in terms on evaluation metrics defined in the next section. Second, we're going to compare and contrast weighted A* algorithms with different scales of weight (e.g., 1, 1.4, 2) on the heuristic function. Third, we will implement different heuristic functions on A* algorithms to tell which one yields the best and interpretable performance, so that future developers of this navigation system will have a reference. Notably, since the search space is too small to

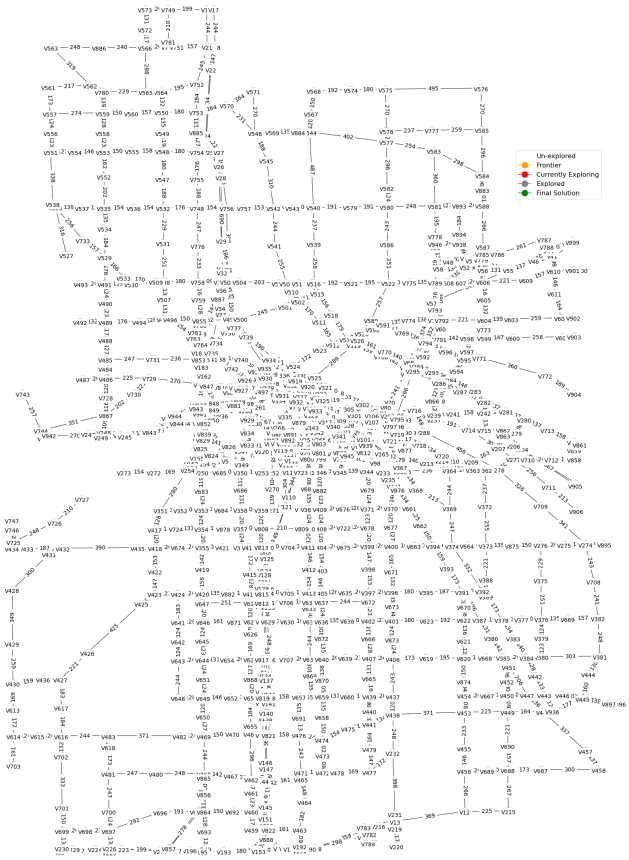

Figure 3: The Minneapolis map in the graph search space.

quantitatively compare the metrics between two vertices on the map and we'd like a general advice of which algorithm works the best on Gopher Way map, we applied the all-pairs shortest-path (APSP) fashion, which is to aggregate the results of every two vertices.

On the other hand, for the Minneapolis map we still performed experiments on these three groups of comparisons, but instead of applying the APSP method, we applied the single-source shortest path (SSSP) methodology. The reason that we did not run APSP was that it would take more than two days to aggregate all the results on every two vertices on the map based on our back-of-the-envelop calculation, and Google Colab has a limit of time use. Therefore, we used SSSP, which finds the shortest path from one vertex to all the other vertices. Since the Minneapolis map has many more vertices than the Gopher Way map, we figured that we would still obtain insightful results by using SSSP.

It's worth mentioning that we intended to implement Theta* algorithm proposed by Daniel et al. [3] as we completed a comprehensive study of it in our literature review, but we decided to exclude it in our experiments due to the

following two reasons. First, the search space that the paper is based on is a grid, which is essentially different from our graph search space. The novelty is basic Theta* algorithm is the line-of-sight check to avoid obstacles in route planning, but such a check is meaningless on graph search space unless we converted it into grid search space. Second, aima-python's underlying code supports appending the unexplored successors into the priority queue but Theta* algorithm's pseudocode suggests the priority queue shall hold both the explored successor and its parent as an element.

3.4 Experimental Setup

The code that we're using is from aima-python [10], which comes with the textbook *Artificial Intelligence - A Modern Approach* for UMN CSCI 4511W. Since it doesn't come with all the algorithms that we intend to compare, we implemented different heuristics including chebyshev distance and variants of A* algorithms such as weighted A* algorithm with different weights. The platform we're executing our code on is Google Colab with a 2-core CPU Intel(R) Xeon(R) CPU @ 2.00GHz.

3.5 Evaluation Metrics

Since our aim in this study was to figure out the most efficient path from one underground entrance to another using the Gopher Way tunnels on East Bank Campus, we thought it best to run our algorithms between every combination of two entrances, which are represented by the vertices on our map.

The metrics that we used to compare the performance of the search algorithms are the time taken to search, the total path cost, the number of successors, the number of states, and goal tests performed across each search algorithm. The number of successors, commonly known as the branching factor, is the total number of nodes that are expanded during the search. The number of states metric is fairly self-explanatory as it is the total number of states represented by the search space throughout the entire search. The "goal tests" metric is the number of times that the algorithm checks whether or not the current state is the goal state, in other words whether or not the search has reached the goal node. The number of successors, number of states and number of goal tests are all indicators of memory used during the search which is very important to consider when choosing a search algorithm.

4 Results

4.1 Heuristics

Inspired by the most commonly seen heuristic (i.e.,Euclidean distance) has a formulation of l_2 norm, we extracted the Manhattan distance and Chebyshev distance by applying l_1 norm and l_∞ norm. As you see on Table 2, Manhattan A* Search uses Manhattan as its heuristic, which is defined as

Table 2: The algorithm zoo for our experiments (A* Search and its weighted versions by default use Euclidean distance as heuristic if not specified).

Algorithm	f value	Optimality
A* Search	f = g + h	optimal
Weighted A* Search	f = g + 1.4 x h	non-optimal
Extra Weighted A* Search	f = g + 2 x h	non-optimal
Manhattan A* Search	f = g + h	optimal
Chebyshev A* Search	f = g + h	optimal
Hamming A* Search	f = g + h	optimal
Greedy Best First Search	f = h	non-optimal
Dijkstra Search	f = g	optimal
Breadth First Search	N/A	non-optimal

$$\text{Manhattan Distance}(\mathbf{x}, \mathbf{y}) = \sum_{i=0}^{m} |x_i - y_i| \qquad (1)$$

where m is the number of vertices along the path. The Chebyshev distance is defined as

$$\text{Chebyshev Distance}(\mathbf{x}, \mathbf{y}) = \max\{|x_i - y_i|\} \qquad (2)$$

where $i = 1, 2, \ldots, m$.

Although Hamming distance is not popular in graph search space, we also define it as

$$\text{Hamming Distance}(\mathbf{x}, \mathbf{y}) = \sum_{i=0}^{m} \mathbb{1}(x_i \neq y_i) \qquad (3)$$

where $\mathbb{1}$ is an indicator function outputting true or false.

4.2 Gopher Way Map

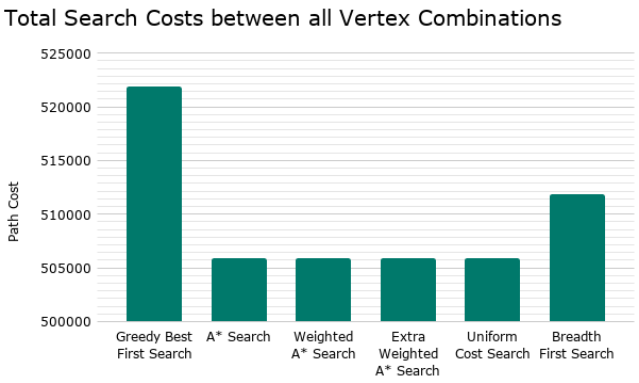

Figure 4: The search costs of different shortest-path algorithms on Gopher Way map.

While all of our search algorithms (Greedy Best First, A*, Weighted A*, Extra Weighed A*, Dijkstra's, and

Breadth First) had similar overall costs for the UMN East Bank map, we did see a slightly better cost for the three variations of A* and Dijkstra's. This is expected because Dijkstra's is an optimal search, meaning that no other search algorithm has a higher cost. Similarly, A* is also an optimal search in the case that it uses an admissible heuristic. The heuristic used in this experiment is the Euclidean distance, which is admissible, so A* in this case is optimal. The path costs are shown in Figure 4. It is important to note that the y-axis on this graph starts at 500,000 and not 0, so the cost differences may be smaller than they appear.

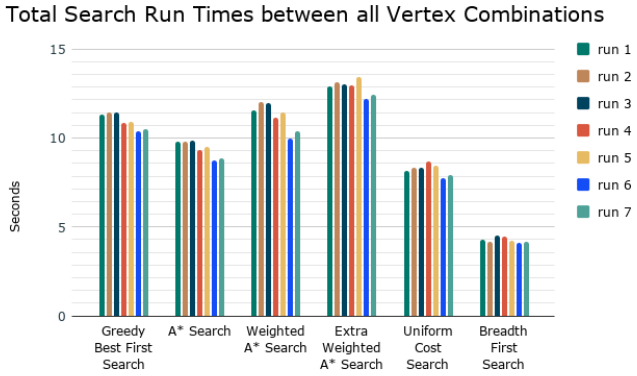

Figure 5: The search times of different shortest-path algorithms including A* variants in Gopher Way map.

The fastest algorithm in terms of time was Breadth First Search with an average total run time of 4.288 seconds. On the contrary, the algorithm that took the longest amount of time was Extra Weighted A* Search with an average total run time of 12.862 seconds, nearly 3 times that of Breadth First Search. Greedy Best First Search had an average total run time of 10.969, A*'s was 9.397409473, Weighted A*'s was 11.199, and Dijkstra's was 8.225. The individual run times for each APSP run (7 total) are shown in Figure 5.

Greedy Best First Search, Uniform Cost Search and all versions of A* Search had the same number of successors, 372,645. Breadth First Search had a slightly lower number of successors with 348,400. A* Search, Weighted A* Search, Extra Weighted A* Search and Uniform Cost Search all had the same number of states, 803,919, whereas Greedy Best First Search had 798,933 and Breadth First Search had the least with 749,867. All six of the search algorithms had the same number of goal tests, with 380,926, making goal tests an irrelevant metric. Regardless of the number of goal tests failing to indicate any algorithm using more or less memory, the number of successors and number of states both indicate that Breadth First Search used the lowest amount of memory. Figure 6 compares these three metrics which indicate memory usage across the different search algorithms.

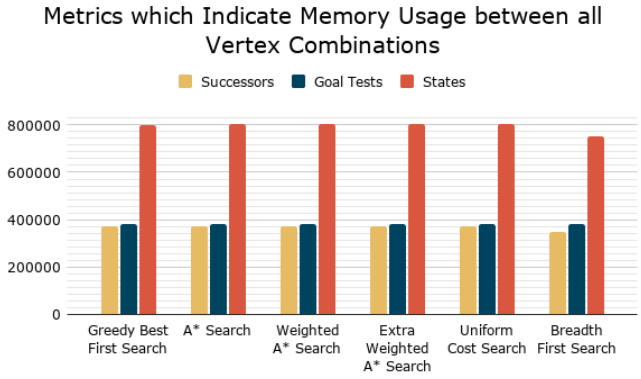

Figure 6: The memory usage-related metrics of different weighted A*s, greedy best first search, Dijkstra and breadth first search in Gopher Way map.

Table 3: The memory usage related metric of A* algorithms across **different heuristic functions** on the Gopher Way map.

Algorithm	# Successors	# Goal Tests	# States
Euclidean A*	372645	380926	803919
Manhattan A*	372645	380926	803919
Chebyshev A*	**362133**	**370414**	**782349**
Hamming A*	372645	380926	803919

4.3 Minneapolis Map

As we expected due to the nature of optimal search algorithms, all three variants of the A* Search and Dijkstra's had the same search cost on the Minneapolis map, with a cost of $1.46 * 10^{10}$. Breadth First Search had a significantly higher cost than that with $1.70 * 10^{10}$ and Greedy Best First Search had a cost in between those two with $1.65 * 10^{10}$. These costs are shown in Figure 7.

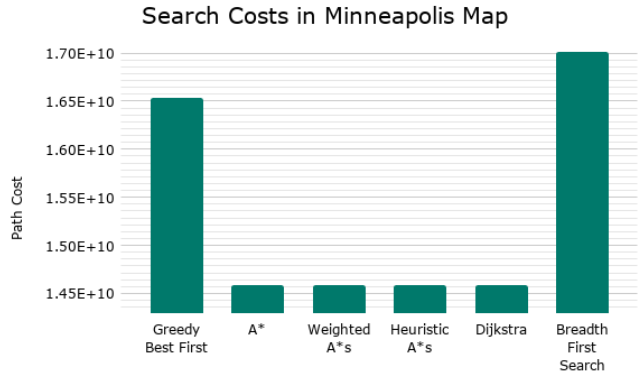

Figure 7: The search costs of different shortest-path algorithms on Minneapolis map.

Table 4: The memory usage related metric of A* algorithms across **different weights** on Euclidean heuristic on the Minneapolis map.

Algorithm	# Successors	# Goal Tests	# States
A*	446942	447888	1313767
Weighted A*	446928	447874	1313729
Extra Weighted A*	**446904**	**447850**	**1313664**

Table 5: The memory usage related metric of A* algorithms across **different heuristic functions** on the Minneapolis map.

Algorithm	# Successors	# Goal Tests	# States
Euclidean A*	446942	447888	1313767
Manhattan A*	**446931**	**447877**	**1313736**
Chebyshev A*	446982	447928	1313886
Hamming A*	446947	447893	1313782

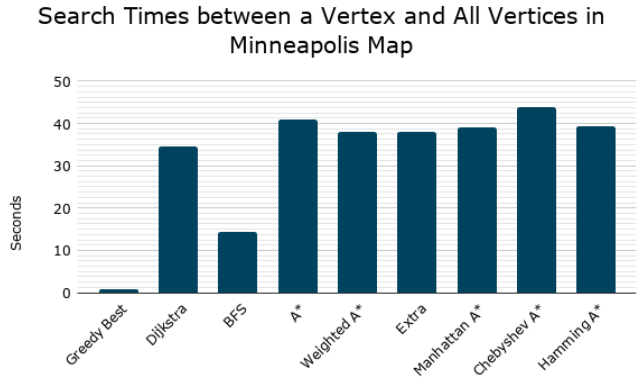

Figure 8: The search times of different shortest-path algorithms including A* variants in Minneapolis map.

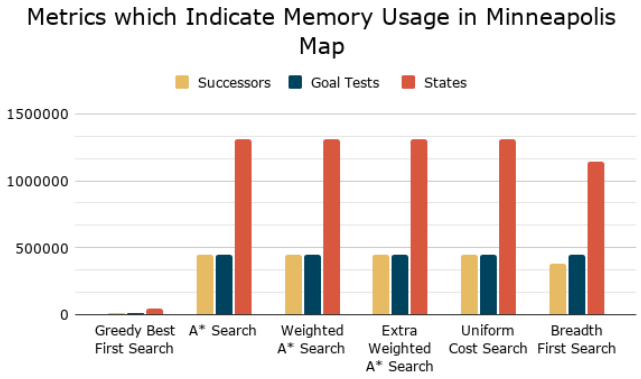

Figure 9: The memory usage-related metrics of different weighted A*s, greedy best first search, Dijkstra and breadth first search in Minneapolis map.

5 Discussion

5.1 Analysis of Gopher Way Map

When choosing a search algorithm for any application it is important to consider each algorithm's metrics and characteristics, as they can all be preferable in unique scenarios. The results described in section 4.2: Gopher Way Map agree with the AI communities consensus that Uniform Cost Search and A* Search with an admissible heuristic always find the optimal solution, i.e. the solution with the lowest possible cost. This explains why all four of these algorithms had the same path cost. If our only concern with choosing an algorithm was cost, we would suggest any of these four, but there are other metrics to consider.

As anyone in the computer science and AI communities knows, run time is extremely important when choosing an algorithm or building software. Our results favored one algorithm significantly in the run time category, with Breadth First Search having the lowest run time by a long shot. Breadth First Search does not always have a low run time in every application, so it is likely that this low run time is due to the Gopher Way map's low branching factor. The branching factor is the number of children of each node. Since the Gopher Way does not have many intersections where a person can choose between multiple paths, it has a low branching factor. The Breadth First Search algorithm adds nodes (or vertices) to a queue, one breadth level at a time. The map's low branching factor means that each breadth level has a relatively small number of nodes, so the algorithm spends a very small amount of time adding nodes to the queue. It will often add a node to the queue and then instantly pop it off and continue the search if it is the only node in the queue.

Another major consideration that should be made when choosing a search algorithm is memory usage. As described in Section 4.2: Gopher Way Map, our results indicated that Breadth First Search was most efficient in its usage of memory. This contrasts with the general principle that Breadth First Search uses more memory than most other search algorithms, but this application is unique for the reasons described in the previous paragraph. Since the search space has a low branching factor, the queue never gets very large, and therefore does not take up much memory.

Furthermore, among all the A* algorithms with different heuristic functions, we noticed that the Chebyshev A* algorithms outperform all other heuristic functions, because it has considerably lower memory usage than the others (see Table 3. The result may implies that Chebyshev distance (i.e., max function) is representative enough in a map like Gopher Way map that is mostly one directional without circles.

With all of these metrics taken into consideration, we suggest that any future developers who may create a navigation app for the University of Minnesota East Bank Gopher Way to use the Breadth First Search algorithm in their implementation. Although Breadth First Search did not achieve optimal path cost, it was relatively close and we believe that its advantages in both run time and memory usage outweigh its slightly less than optimal path cost.

5.2 Analysis of Minneapolis Map

The search space in the Minneapolis map is much larger than the Gopher map, and the customers that use a navigation software on Minneapolis map care more about going from one place to another place with the lowest cost (i.e., shortest path). Hence, Greedy Best First Search and Breadth First Search would be ruled out because the final path they found has a much higher cost than the other optimal algorithms as shown in Figure 7. Among all the optimal search algorithms, A* variants have fewer successors, goal tests and states, and this is generally true because heuristic functions help A* variants to lean toward expanding the most promising direction first.

An interesting phenomenon we discovered is that both weighted A* algorithms actually found the same optimal path that A* found in Minneapolis map, although weighted A* algorithms don't guarantee finding the optimal path. Among the weighted A* algorithms with Euclidean heuristics, although the differences are very tiny, we found the Extra Weighted A* algorithm has the fewest number of successors, goal tests and states (see Table 4, which implies that it's more memory efficient. In addition, the Extra Weighted A* algorithm ran faster on the Minneapolis map compared to A* and weighted A* algorithms (see Figure 8). Hence, we conclude that for the Minneapolis map, having more weight (e.g., twice) on the heuristic function (i.e., more aggressive on future prediction) helps with the performance in terms of search time and memory.

When it comes to the A* algorithms with different heuristics, we observed the A* with Manhattan distance as its heuristic function performed better than using any other heuristics in terms of the number of successors, goal tests, and states, which are indicators of memory usage efficiency (see Table 5). Furthermore, Manhattan distance also used slightly less time than the others (see Figure 7). The geographical interpretation of Manhattan distance makes sense on the Minneapolis map, because of the layout of the streets in Minneapolis, and the mapping consists of many blocks without diagonal connections.

To conclude this section, we recommended using the Extra Weighted A* algorithms on the navigation software of Minneapolis map, because it outperforms Manhattan A* algorithm in terms of time while having slightly lower expanded nodes.

5.3 Analysis across Two Maps

As mentioned in our overview section, there is not a universal perfect search algorithm that works the best everywhere. It heavily depends on the characteristics of the search space. We found that Chebyshev distance as heuristic works better in Gopher Way map, while Manhattan distance as heuristic works better in Minneapolis map. From the visualization of both maps (see Figure 2 3), we thought the difference maybe caused by the nature that Minneapolis map has many vertices that Gopher Way map not nearly has. Moreover, we also have a deeper appreciation of the A* algorithm after all these experiments, because the A* algorithm is just so simple and elegant but still powerful. Some modified versions of A* algorithms either from heuristics or from weights only bring little and almost negligible performance increase to the A* algorithm.

6 Conclusion

We had hypothesized that given our Gopher Way map, the A* search algorithm would be the most cost and time efficient. Based on our findings, Breadth-First-Search actually performed best in terms of overall time. The size of the coordinate set should have been something that we considered when making such hypothesis, since Breadth-First-Search in this case would make the most sense to be most cost and time efficient. Interestingly, we found that A*, Weighted A*, and Extra A* search algorithms performed best when applied to the Minneapolis coordinates. This is so be expected due to the nature of optimal search algorithms on a larger graph.

7 Future Work

Looking forward, this type of algorithm analysis has and should be applied to other coordinate system graphs like the Gopher way tunnels on East Bank and the city of Minneapolis. Future research can build upon our findings of which search algorithms are most cost and time efficient for which types of graphs. Path finding is an efficient way to save time when finding pathways along such graphs like the Gopher Way tunnel or the city of Minneapolis. While it maybe straight forward and enlightening to visualize the path each algorithm finds on the map, we didn't do it because most of our algorithms are optimal which will end up with the exact same solution for visualization.

Meanwhile, if we're able to get the geo-coordinate system of the underlying code of aima-python, as well as the means to re-project the current Gopher Way map to that specific coordinate system, we may have a nicer visualization graph and more precise results on our evaluation metrics. Also, since we propose the Gopher Way map by manually recording each coordinates from the Google map and validate them on campus by field trip, we're open to share the Gopher Way dataset to everyone who're interested in further research on the map.

8 Contribution

1. Esthi - Abstract, Related work: Dijkstra's Algorithm and Theta*, Experimental setup, Data Recording, Discussion, Conclusion
2. Gaoxiang - Overview, Problem Definition, Gopher Way Dataset, Experimental Design, Coding, visualization, Analysis of Minneapolis map
3. Elizabeth - Evaluation Metrics, Results, Experimental Design (Minneapolis section), Analysis of Gopher Way Map, wrote code for the experiment and ran the experiment to collect and record data, made graphs from data
4. Angel - Introduction, Related Work: Greedy Best First, A*, Choosing Heuristics, Weighted A*, Further Use, and all pseudo code for the algorithms.

References

[1] Ayoub Bagheri, Mohammad-R Akbarzadeh-T, and Mohamad Saraee. Finding shortest path with learning algorithms. *International Journal of Artificial Intelligence [electronic only]*, 1, 01 2008.

[2] Yngvi Björnsson and Kári Halldórsson. Improved heuristics for optimal pathfinding on game maps. In *Proceedings of the Second AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, AIIDE'06, page 9–14. AAAI Press, 2006.

[3] K. Daniel, A. Nash, S. Koenig, and A. Felner. Theta*: Any-angle path planning on grids. *Journal of Artificial Intelligence Research*, 39:533–579, Oct 2010.

[4] R. Ebendt and R. Drechsler. Weighted a search – unifying view and application. *Artificial Intelligence 173 (2009) 1310–1342*, pages 1310–1342.

[5] Robbi Rahim et al. Breadth first search approach for shortest path solution in cartesian area. *Journal of Physics: Conference Series*, pages 1–6, 2018.

[6] H. Hasanvand, R. Karimi, and M. Nosrati. Investigation of the * (star) search algorithms: Characteristics, methods, and approaches. *World Applied Programming Vol. 2, No. 4*, pages 251–256, April 2012.

[7] M. Heusner, Keller T., and Helmert M. Best-case and worst-case behavior of greedy best-first search. *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence (IJCAI-18)*, pages 1463–1470.

[8] M. Heusner, Keller T., and Helmert M. Understanding the search behaviour of greedy best-first search. *Proceedings of the Tenth International Symposium on Combinatorial Search (SoCS 2017)*, pages 47–55.

[9] Amgad Madkour, Walid G. Aref, Faizan Ur Rehman, Mohamed Abdur Rahman, and Saleh M. Basalamah. A survey of shortest-path algorithms. *CoRR*, abs/1705.02044, 2017.

[10] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall Press, USA, 3rd edition, 2009.

[11] Hyejeong Ryu and Wan Kyun Chung. Local map-based exploration using a breadth-first search algorithm for mobile robots. *International Journal of Precision Engineering and Manufacturing Vol. 16, No. 10*, pages 2073–2080, 2015.

[12] S. Sharma. Shortest path searching for road network using a* algorithm. *International Journal of Computer Science and Mobile Computing Vol. 4, No. 7*, pages 513–522, July 2015.

[13] Anupam Singh, Vivek Shahare, Nitin Arora, and Ahatsham . Path finder : An artificial intelligence based shortest path. *International Journal of Recent Technology and Engineering*, 8:5177–5181, 12 2019.

[14] Thorat Surekha and Rahane Santosh. Review of shortest path algorithm. *International Research Journal of Engineering and Technology*, 3, 2016.

[15] X. Zhao, A. Sala, H. Zheng, and B. Y. Zhao. Efficient shortest paths on massive social graphs. In *7th International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom)*, pages 77–86, 2011.

[16] Uri Zwick. Exact and approximate distances in graphs - a survey. In *In ESA*, pages 33–48. Springer, 2001.