Interactive Comparison of Path Finding Algorithms

Jasmine Chan (jjc297), Jonya Chen (jc957), Ava Tan (ajt222)

December 18, 2014

1 INTRODUCTION

For our CS 4701 Foundations of Artificial Intelligence Practicum project, we want to present certain topics that we have learned throughout the course with the creation of a fun and interactive game interface. Our core project focus is on search algorithms, as they are a central topic within artifical intelligence. Often, we are presented with a goal-based agent, such as a problem-solving agent, in which the agent aims at satisfying a goal as well as maximizing their performance measure - through the use of search algorithms, we can solve these problems. In the case of our project, we focus on taking in a problem and through the use of search, returning a solution in the form of an action sequence. Then, the actions can be carried out during the execution phase, which will lead the agent to the desired goal.

Within the artificial intelligence community, there is a large interest in the need for fast and accurate graph traversal search algorithms – specifically, those that are able to find a path from a start to goal node all with minimum cost. Some example real world applications include game analysis, such as chess or sudoku, optimization of computer chip layouts, streamlined logistics for package delivery systems, and much more. We represent our game as a similar graph, with the optimal solution being the lowest cost path to the goal node.

We also want to present the user with a visual representation of the agent reaching its desired goal through the use of various search algorithms. So we created a game which demonstrates these search strategies by allowing a user to help our character, Pacman, reach his end goal of food. Our grid map GUI is completely interactive, as it allows the user to set a specific start node, end node, and obstacles in order to create a unique maze for Pacman to navigate his way through. The user is then able to select from a variety of implemented search algorithms to execute and watch the algorithm in action. Eventually, an optimal path to the end node (if one exists) will be returned. Finally, Pacman will follow the path produced by the algorithm from the start to end node and accomplish his goal of eating his food.

As for the actual search methods, we decided to implement a series of algorithms that have been discussed in class. Each algorithm implemented is described in more detail later on in this documentation. The ones we chose to include in our project are as follows:

- Breadth-First Search
- Depth-First Search
- Depth-Limited Search
- Dijkstra's Algorithm
- Uniform Cost Search
- Bidirectional Search
- Iterative Deepening Depth-First Search
- A* Search
- Greedy Best-First Search

We focus on implementing several uninformed search algorithms, as well as informed search algorithms. We are then able to analyze the differences between these algorithms. For example, uninformed search algorithms are given no information about the problem other than its definition and as a result, should rarely solve problems efficiently. On the other hand, informed search methods tend to perform quite well given some guidance on where to look for solutions. As a result, we should be able to view noticeable differences between each of these executed algorithms. Not only may this be observed visually through the use of our GUI, but differences can also be analyzed through various quantitative evaluation measures, such as the length of the path or the time it took for the operation to complete. Ideally by the end of our project, we should be able to determine under which conditions is it best to use certain search algorithms over others.

2 Description of Algorithms Implemented

2.1 Breadth-First Search

Breadth-First Search, commonly referred to as just BFS, is a type of uninformed search algorithm that works with just two main strategies: 1) visiting and inspecting a node in a graph, and 2) visiting the neighboring nodes of said node. Another way to describe how breadth-first search operates is that, starting from the root node, it always expands the shallowest unexpanded nodes first.

Before delving into the actual implementation details, we will first discuss the characteristics and properties of BFS. Breadth-first search was first implemented in the late 1950's by E.F. Moore, who is most commonly recognized for the development of the Moore finite state machine. He originally wrote the algorithm for BFS to find the shortest path out of a maze. Thus, we see that the algorithm is *complete* for finite graphs, meaning that if the maximum branching factor of the search tree is finite, then the algorithm is guaranteed to terminate.

The *time complexity* of BFS is $O(b^{d+1})$, and this is because, starting from the root node, the algorithm searches every possible node at every depth until the depth of the goal/solution is reached. b here represents the maximum branching factor of the tree, and d represents the depth of the goal/solution. The biggest concern by far, with this algorithm, is *space complexity*. Like time complexity, it is $O(b^{d+1})$, but this is more of a problem since every node searched is stored in memory for the purpose of reconstructing a solution path.

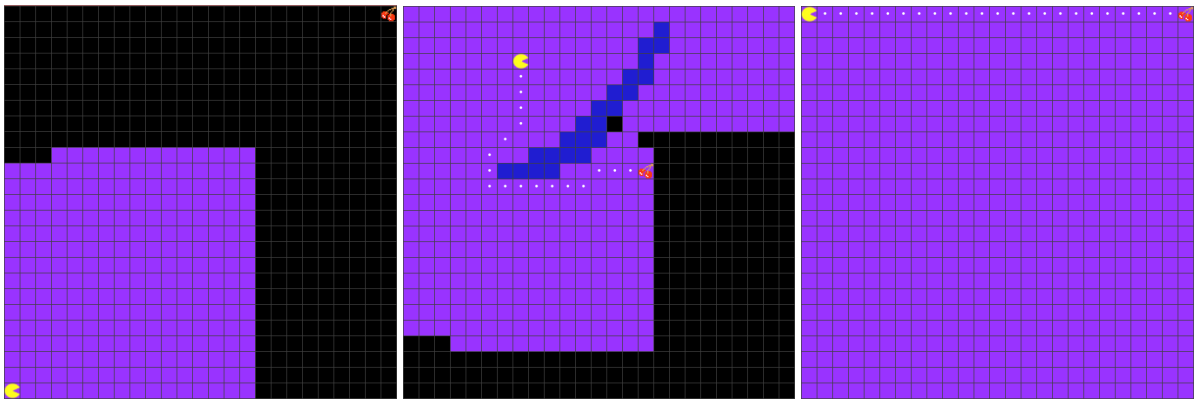
Lastly, we note that BFS will always find an optimal path, i.e, one that is the shortest – given that the step costs are all identical. There exist other search algorithms (ones that we have implemented and will discuss) that are better suited for searching a graph with weighted edges.

The pseudocode for BFS is given below.

```
1  public double BFS(G, src, goal)
2    new queue Q, new set V
3    add src to V, add src to Q
4      while (Q is not empty) loop {
5      t <= Q.remove()
6      if t is goal {
7        return t
8        }
9      for all edges e in G adjacent to t loop {
10     u <= G.adjacentVertex(t,e)
11       if u is not in V {
12         add u to V, add u to Q
13         }
14       }
15     }
16   return none
17 end BFS
```

Note that a lot of implementation details have been abstracted away by the pseudocode above.

Refer to Figure 2.1 below for screenshots of the DFS algorithm running under varying scenarios.



(a) BFS in action, no wall. **(b)** Path found by BFS, with walls. **(c)** A worst-case scenario for BFS.

Figure 2.1: Screenshots of running the breath-first search algorithm in varying scenarios.

In Figure 2.1c, we note that Pacman winds up searching the entire graph in order to find the cherry that is just across from him. This illustrates a kind of "worst-case scenario" for which BFS might not be as efficient as some other search algorithm.

2.2 Depth-First Search

Depth-First Search, commonly referred to as just DFS, is a type of uninformed search algorithm that works on the principle of always expanding the deepest unexpanded node. In this regard, DFS is different from BFS, which prefers to explore the shallowest unexpanded nodes first. The main implementation detail that makes DFS different from BFS is the manner in which the fringe nodes are stored – we see that a *queue* is used for BFS. DFS is different in terms of

implementation in that instead of a queue, it uses a *stack* to store nodes, such that the nodes can be extracted in a last in, first out manner.

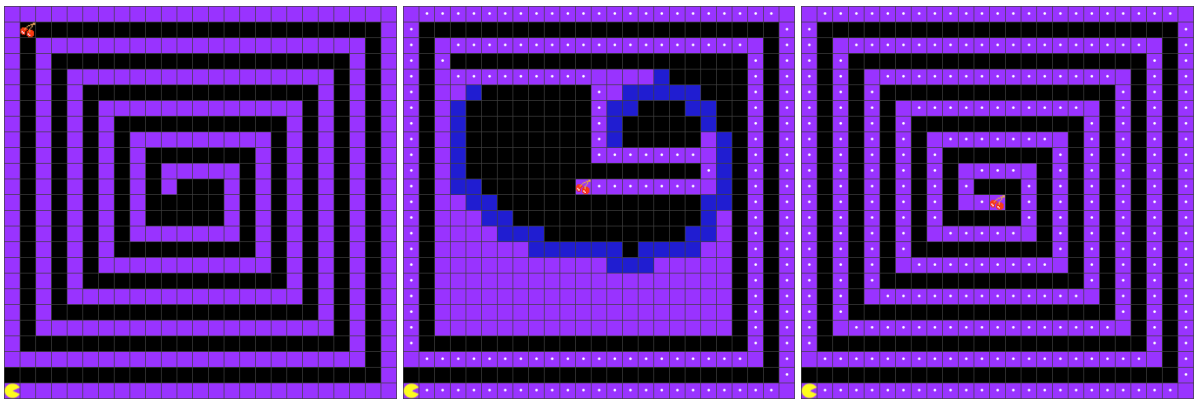
Now, we consider the characteristics of DFS. Firstly, we node that for infinite-depth spaces, the algorithm is *not* complete, as it can continue forever down the a path. DFS is, however, complete for finite graphs. For graphs that contain cycles or loops, DFS can become "trapped," and thus ought to be modified to prevent repeating states. The *time complexity* of DFS is $O(b^m)$, where m is the maximum depth of the state space. This is considered bad if m is much larger than d. If multiple solutions exist and are densely packed, DFS may be faster than BFS! We also must note that DFS's greatest attribute is space complexity, which is in fact linear. It is $O(bm)$.

Lastly, we note that DFS will not always find an optimal solution.

The pseudocode for DFS is given below.

```
1  public double DFS(G, src, goal)
2    new stack Q, new set V
3    add src to V, add src to Q
4      while (Q is not empty) loop {
5      t <= Q.remove()
6      if t is goal {
7        return t
8        }
9      for all edges e in G adjacent to t loop {
10     u <= G.adjacentVertex(t,e)
11       if u is not in V {
12         add u to V, add u to Q
13         }
14       }
15     }
16   return none
17 end DFS
```

Refer to Figure 2.2 below for screenshots of the DFS algorithm in varying scenarios.



(a) DFS in action, no wall. **(b)** Path found by DFS, with walls. **(c)** A worst-case scenario for DFS.

Figure 2.2: Screenshots of running the depth-first search algorithm in varying scenarios.

We note that depending on the location of the goal and the size of the graph, it is very easy for DFS to find an *extremely* subpar solution! The final reconstructed path, as shown in Figure 2.2c is very long, and that's even after Pacman searches a good number of the nodes on the graph.

2.3 Depth-Limited Search

Depth-Limited Search is a variant on DFS, or depth-first search. Iterative-deepening depth-first search also uses this particular algorithm as a part of its implementation. It is a useful variant of depth-first search since, as discussed previously, DFS is not always guaranteed to terminate on an infinite graph or one that is too large to visit in its entirety. It differs from regular DFS in that there is a limit imposed on the depth of the search, but still searches throughout the graph in a similar manner as DFS would.

Now, we discuss a few characteristics of depth-limited search. Though we have the advantage of terminating the search at a certain maximum depth to avoid the completeness issue encountered, the goal might be within a reasonable distance and wind up being "unreachable" because the algorithm terminated already. Thus, we note that in the absence of knowledge of the approximate distance to the goal node, depth-limited might be a rather unreliable.

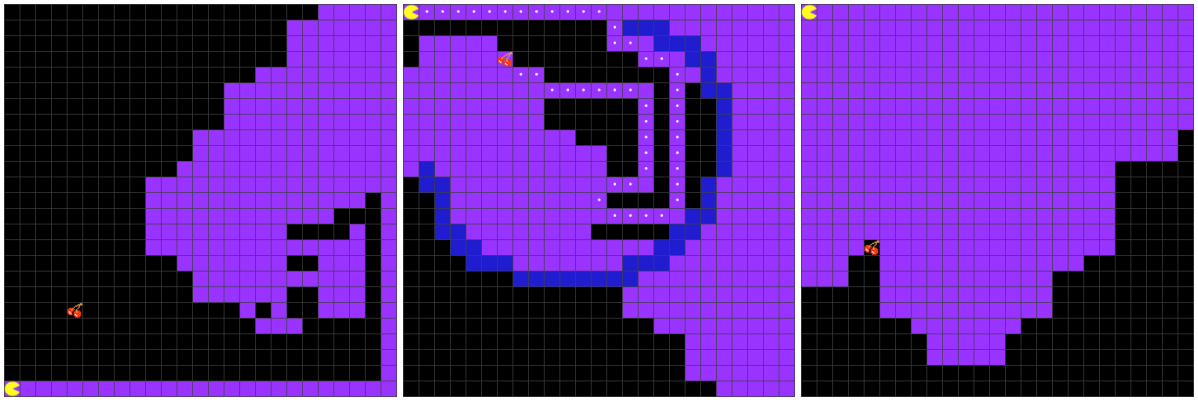
Since depth-limited is a variant of DFS, the *time complexity* and *space complexity* is the same as it is for regular DFS; that is, $O(b^{d+1})$ and $O(b^{d+1})$, respectively.

Finally, we note that just like DFS, depth-limited search is not guaranteed to find an optimal path since it still explores one path to its end.

The pseudocode for depth-limited search is given below.

```
1  public double DLS(src, goal, depth)
2    if (depth >= 0) {
3      if (src == goal) {
4      return node
5        }
6      for each child in src.getNeighbors() loop {
7      DLS(child, goal, depth-1)
8        }
9      }
10   return none
11 end DLS
```

Refer to Figure 2.3 below for screenshots of the depth-limited search algorithm running under various scenarios. Note that the dimensions of the graph is 25x25 squares; for Figures 2.3a and 2.3b, the depth was set to 50, which is sufficient to reach every node on the graph. However, in Figure 2.3c, the depth was reduced to 30.



(a) DLS in action, no wall. **(b)** Path found by DLS, with walls. **(c)** A worst-case scenario for DLS.

Figure 2.3: Screenshots of running the depth-limited search algorithm in varying scenarios.

As noted previously, if the maximum depth is not set to a sufficient value, then the algorithm may not find a path to a goal that is very reachable, as in Figure 2.3c. This is because the algorithm terminates before getting a chance to explore further.

2.4 Dijkstra's Algorithm

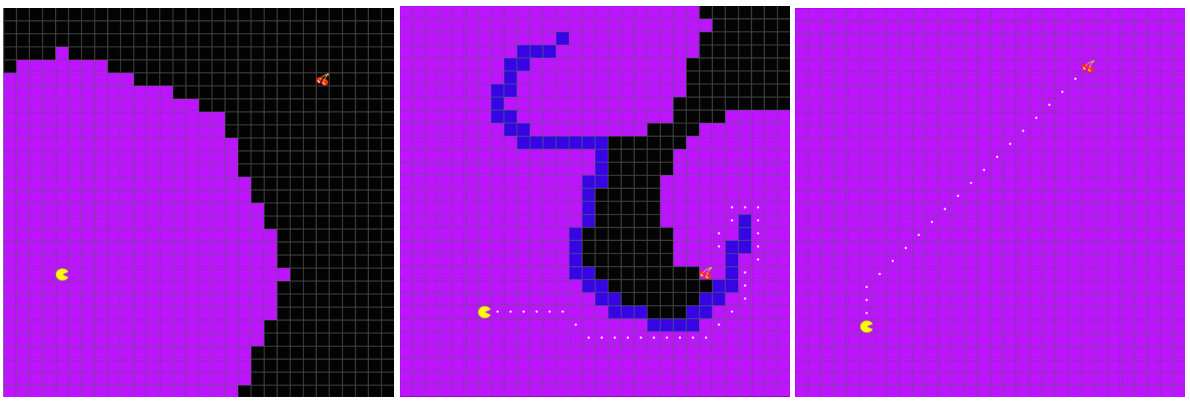
Dijkstra's Algorithm is specifically used for graph searches and is implemented to find the shortest path from every node to the node of origin, creating a shortest-path tree. Dijkstra's can be considered a heuristic search and is similar to a greedy search if there's a known goal node. We also have to make sure that for every node, we store a distance value, report the node in increasing order of distance from the start node, and mark the node as visited/explored. We then are essentially constructing the shortest-path tree edge by edge. At each step, we add one new edge, corresponding to the construction of the shortest path to the current new node. In our implementation of Dijkstra's, we decided to use a min-heap as our priority queue.

Now, we examine a few attributes of Dijkstra's Algorithm. In the worst case scenario, where Dijkstra's is running on a graph with V vertices and E edges, the *time complexity* is $O(|V|^2)$. Note that in this case, we are implementing a simpler version of Dijkstra's, in which we store nodes in a min heap (which serves as our priority queue) such that extrating minimum from it is just a linear search. Also, Dijkstra's even finds the shortest paths from a start node to all other nodes in the graph. Additionally, the *space complexity* of Dijkstra's is $O(|V|)$ to keep track of all the vertices of the graph.

The pseudocode we used to implement Dijkstra's is given below.

```
1  public double Dijkstra(G, src, goal)
2      new set distances, which holds distance estimates
3      new set explored
4      new set path
5      new priorityQueue PQ
6        x prioritized on d[x]
7      while (PQ is not empty) loop {
8        x <= PQ.removeMin()
9          add x to explored
10         if (x == goal){
11           return x
12           }
13       for all successors y of node loop {
14           newDist <= distances[y] + weight[x,y]
15             if newDist < distances[y]{
16               distances[y] <= newDist
17                 path[y] <= x
18                 PQ.promote(y, distances[y])
19               }
20           }
21       }
22     return none
23 end Dijkstra
```

Refer to Figure 2.4 below for screenshots of the Dijkstra's Algorithm running under various scenarios.



(a) Dijkstra in action, no wall. **(b)** Path found by Dijkstra, with walls. **(c)** A worst-case scenario for Dijkstra's Algorithm.

Figure 2.4: Screenshots of running Dijkstra's search algorithm in varying scenarios.

As seen in Figure 2.4c, the algorithm searches almost the entire graph in the worst-case scenario. Note how Dijkstra's Algorithm will search the entire graph in order to find the shortest path to *every* node - this can also be observed in Figure 2.4c. Therefore, the algorithm doesn't terminate until the entire graph is searched, whereas uniform cost search, a variant of Dijkstra's that we will discuess next, is essentially the same but terminates when the goal is found.

2.5 Uniform Cost Search

Uniform-Cost Search is a type of tree-search algorithm that is most often used to traverse a graph/tree with weighted edges, since its purpose is to find a path to a destination with the least cost. The search works by starting at the root node and expanding one of its neighboring nodes based on which has the least total cost from the root. Nodes are continually visited one after another in this manner until the destination is found. We note that this algorithm is sometimes called Dijkstra's algorithm, especially in the context of finding a shortest path from a starting node to a goal node; however, a true implementation of Dijkstra's algorithm would be such that the algorithm visits every accessible node from the starting node in order to find the shortest path to every reachable node on the graph.

The implementation of this search algorithm typically involves the use of a priority queue, which keeps track of all of the "frontier" (unexplored) nodes of visited nodes. In the queue, each node has an associated total path cost from the starting point, such that the node on the frontier with the lowest cost is given the highest priority. Thus, the node at the front of the queue is always expanded first, after which the path costs of the frontier nodes of this expanded node are also added to the priority queue.

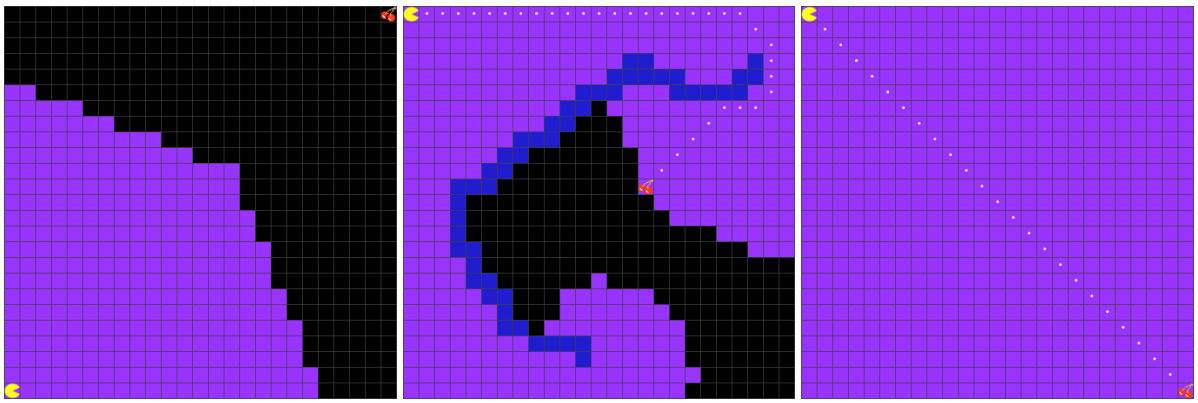
Now, we discuss a few characteristics of uniform-cost search. The *time complexity*, in the worst-case scenario, is $O(b^{1+C^*/\epsilon})$, where C^* is the cost of the optimal solution and ϵ is the positive step cost. The *space complexity* is the same as the time complexity. If it so happens that all step costs are the same, then the space & time complexity are reduced to $O(B^{d+1})$ and $O(b^{d+1})$, respectively. And, as we have alluded already, uniform-cost search is guaranteed to find an optimal solution, if one exists.

The pseudocode for uniform-cost search is given below.

```
1 public double UCS(G, src, goal)
2   new int cost <= 0
3   new priorityQueue frontier
```

```
4    new set explored
5    while (frontier is not empty) loop {
6      node <= frontier.pop()
7      if (node == goal) {
8        return node
9        }
10     add node to explored
11     for each child in node.getNeighbors() loop {
12       if child is not in explored {
13         if child is not in frontier {
14           add child to frontier
15         } else if child is in frontier && has higher cost {
16           replace existing node with child
17           }
18         }
19       }
20     }
21   return none
22 end UCS
```

Refer to Figure 2.5 below for screenshots of the uniform-cost search algorithm running under various scenarios.



(a) UCS in action, no wall. **(b)** Path found by UCS, with walls. **(c)** A worst-case scenario for UCS.

Figure 2.5: Screenshots of running the uniform-cost search algorithm in varying scenarios.

As shown by Figure 2.5b above, the search algorithm terminates and reconstructs a path as soon as the goal node is found. In the worst-case scenario as shown in Figure 2.5c, the algorithm searches the entire graph.

2.6 Bidirectional Search

Bidirectional Search is a graph searching algorithm that works by simultaneously running two different searches: one from the start node and one from the goal node. When the two searches meet at any point, the search terminates and a path is constructed from the start to the end. We note that the type of search being run from either end is a design decision made depending on the structure of the graph to be searched; for our implementation, we chose to use BFS (breadth-first search) in either direction.

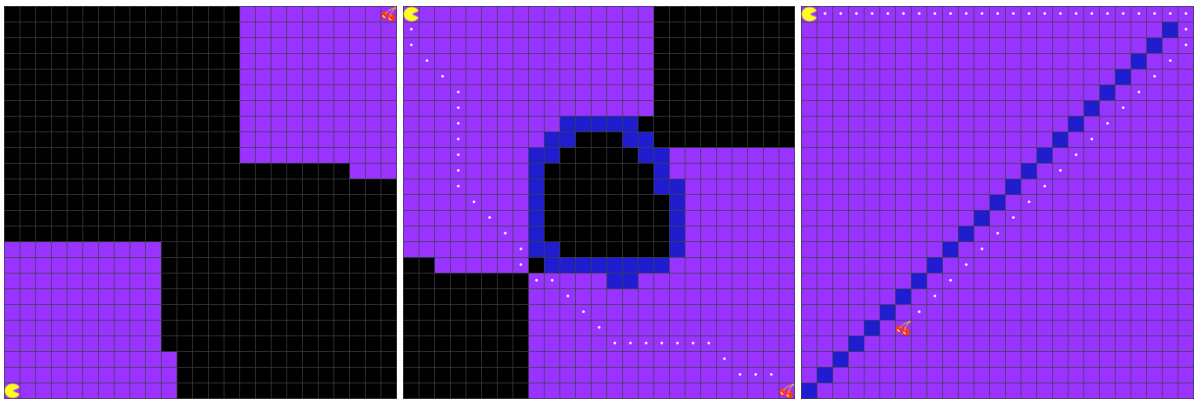
Now, we'll discuss a few characteristics of bidirectional search. Firstly, we note that running this search is, in most cases, faster than simply running one search from the start to the end. As a result, the *time complexity* is affected; with branching factor b, the time complexity turns out to be $O(b^{d/2})$, which is much less than the time complexity of one breadth-first search alone. Similarly, the *space complexity* is also $O(b^{d/2})$. Checking if a node exists in both sets of visited nodes can be done in constant time, so doing so does not affect the overall time or space complexity.

The pseudocode for bidirectional search is shown below.

```
1  public double BDS(G, src, goal)
2    new queue Q1, new queue Q2
3    new set V1, new set V2
4    add src to V1, add src to Q1
5    add goal to V2, add goal to Q2
6      while (Q1 not empty && Q2 not empty) loop {
7        if (Q1 not empty) {
8          t <= Q1.remove()
9          if (t is goal || t is in Q2) {
10           return t
11           }
12         for all edges e in G adjacent to t loop {
13           u <= G.adjacentVertex(t,e)
14           if u is not in V1 {
15             add u to V1 and to Q1
16             }
17           }
18         }
19       if (Q2 not empty) {
20         s <= Q2.remove()
21         if (s is goal || s is in Q1) {
22           return s
23           }
24         for all edges e in G adjacent to s loop {
25           u <= G.adjacentVertex(s,e)
26           if u is not in V2 {
27             add u to V2 and to Q2
28             }
29           }
30         }
31       }
32   return none
33 end BFS
```

Refer to Figure 2.6 below for screenshots of the bidirectional search algorithm running under various scenarios.

As we can see in Figure 2.6a, running BFS from both the start and end nodes reduces the number of nodes needed to search the entire graph (in the absence of obstacles) by almost two times! Refer to Figure 2.1a for comparison. As with regular BFS, a worst-case scenario for using bidirectional search would be when the entire graph must be searched to find a solution, as shown in Figure 2.6c.



(a) BDS in action, no wall. **(b)** Path found by BDS, with walls. **(c)** A worst-case scenario for BDS.

Figure 2.6: Screenshots of running the bidirectional search algorithm in varying scenarios.

2.7 Iterative Deepening Depth-First Search

Iterative Deepening Depth-First Search builds upon the depth-first tree search and finds the best depth limit. It does this by gradually incrementing the depth limit until a goal is found. This continues to occur until the depth limit reaches the depth of the shallowest goal node.

This algorithm essentially combines both the benefits of depth-first and breadth-first search. Similar to depth-first search, the *space complexity* is only $O(bd)$. Similar to breadth-first search, the search is complete when the branching factor is finite and optimal when the path cost is a nondecreasing function of the depth node. As for the *time complexity*, it ends up being $O(b^d)$, which is asymptotically the same as breadth-first search. This is due to the idea that in the search, the nodes on the bottom level (depth d) are generated once, those on the second to bottom level are generated twice, and so on, up until the children of the root are generated d times.

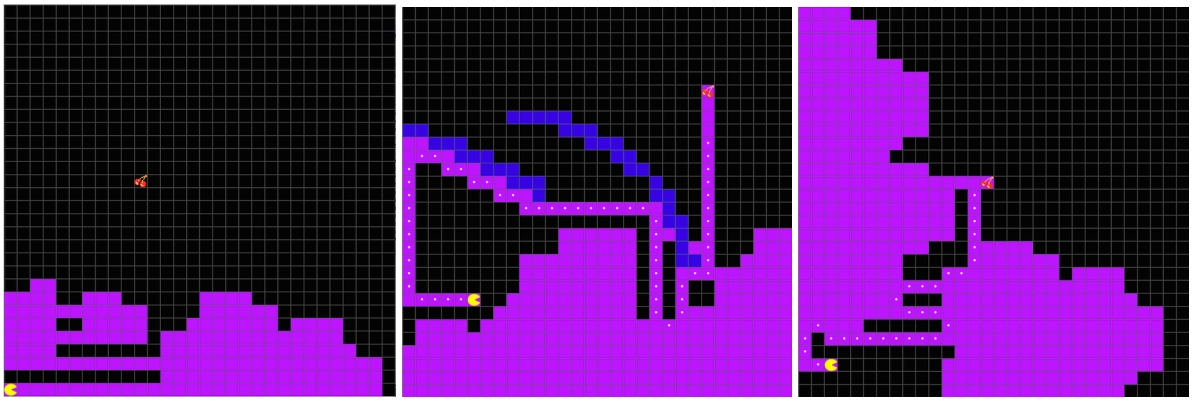
It is also interesting to note how iterative deepening search might at first glance seem very wasteful since states are generated multiple times. However, this turns out not to be a huge issue since almost all the work is done at the deepest level of the search.

The pseudocode for iterative deepening depth-first search is shown below.

```
1 public double IDDFS(src, goal, depth)
2     for(depth=0, depth < infinity, depth++)
3     {
4       double result = DLS(src, goal, depth)
5         if (result found)
6           return result
7     }
8     return none
9 end IDDFS
```

Refer to Figure 2.7 below for screenshots of the IDDFS algorithm running under varying scenarios.

We can conclude that iterative deepening depth first search is a hybrid search strategy between breadth first search and depth first search, inheriting their advantages. It is generally faster than both of those other algorithms. Most importantly, iterative deepening search is preferred when the search space is large and the depth of the solution is unknown.



(a) IDDFS in action, no wall. **(b)** Path with IDDFS, with walls. **(c)** A worst-case for IDDFS.

Figure 2.7: Screenshots of running the iterative deepening depth-first search algorithm in varying scenarios.

2.8 A* Search

A* search uses a best-search first strategy when searching. Therefore, a node is selected for expansion based on an evaluation function, f(n). This function essentially serves as a cost estimate so that the node with the lowerst evaluation is expanded first. The overall implementation is very similar to uniform-cost search, except the priority queue is ordered by f(n) instead of g(n). A* search includes a heuristic component function h(n) which represents the estimated cost of the cheapest path from the state a node n to a goal state. Specifically, A* evaluates nodes by combining g(n) and h(n) in the following way: f(n) = g(n) + h(n). As a result, f(n) is the estimated cost of the cheapest solution through n and as the search is executed, it tries the node with the lowest value of f(n). In our case, we decided to use a simple straight-line heuristic for our A* implementation, which is physically the smallest distance between any two nodes.

A* search is both complete and optimal, provided that the heuristic satisfies certain conditions. First, h(n) has to be admissible, meaning that it never overestimates the cost to reach the goal. Secondly, h(n) must have consistency, meaning that for every node n and every successor n' of n generated by an action a, the following must be true: h(n) ≤ c(n,a,n')+ h(n'). In this case, A* can be implemented more efficiently as no node needs to be processed more than once.

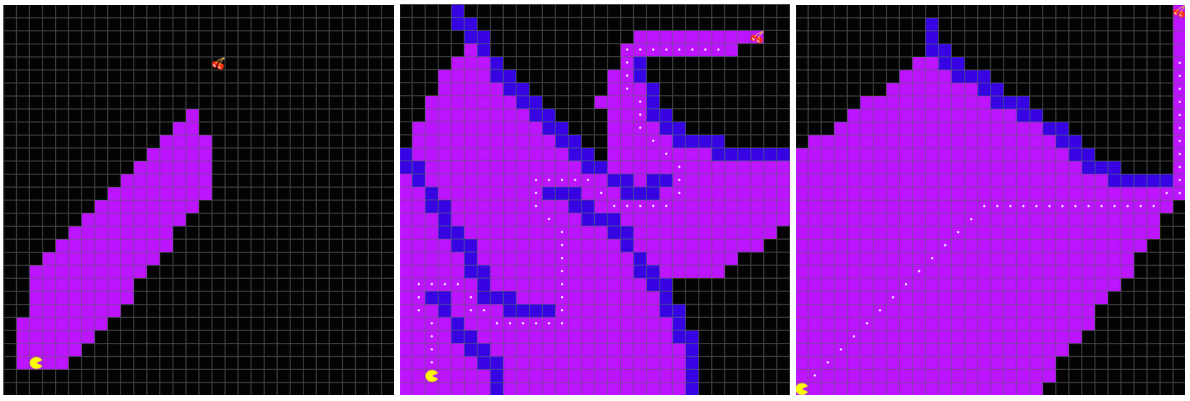
The time complexity of A* depends on the implemented heuristic. In the worst case scenario, the number of nodes expanded is exponential in the length of the solution d such that the *time complexity* ends up being $O(b^d)$. This is with respect to the assumption that there is a reachable goal from the start state, otherwise the algorithm will not terminate and the state space is infinte. In the worst case, the *space complexity* is also $O(b^d)$ since a queue must still contain all unexpanded nodes.

The detailed pseudocode for the A* search implemented in our project is included below.

```
1 public double AStar(G, src, goal)
2     precompute h[n], which holds the heuristic distances
3       h[x] = distance from x to goal
4     new set distances, which holds distance estimates
5     new set explored
6     new set path
7     new priorityQueue PQ
8       x prioritized on d[x] + h[x]
9     while (PQ is not empty) loop {
```

```
10        x <= PQ.removeMin()
11          add x to explored
12          if (x == goal){
13            return x
14            }
15        for all successors y of node loop {
16            newDist <= distances[y] + weight[x,y]
17              if newDist < distances[y]{
18                distances[y] <= newDist
19                  path[y] <= x
20                  PQ.promote(y, distances[y] + h[y])
21                }
22            }
23        }
24      return none
25  end AStar
```

Refer to Figure 2.8 below for screenshots of the A* algorithm running under varying scenarios.



(a) A* in action, no wall. **(b)** Path found by A*, with walls. **(c)** A worst-case scenario for A*.

Figure 2.8: Screenshots of running the A* search algorithm in varying scenarios.

As we can see in Figure 2.8a, since A* uses a heuristic to "guess" which search nodes are considered more likely to lead to the goal, it allows us to often find the best path without having to search the entire map. As a result, this makes the algorithm much faster. Also note the example worst-case scenario in Figure 2.8c. We find that sometimes if A* reaches a node where two equal paths exist to the goal, it will end up exploring both of them, which results in unnecessary parts of the graph to be searched. To save time, it may be better to have an A* implementation that explored only one of these paths if possible.

2.9 Greedy Best-First Search

Greedy Best-First Search is a graph-searching algorithm that is closely related to A* search, which is discussed in the previous section. It is a type of informed search in that it uses a heuristic, like A* search, and this heuristic is usually a distance estimate h(n). However, it differs from A* search in that it does *not* take into consideration the cost of the path traveled so far – so it turns out that greedy best-first search will continue down a path until it reaches a "dead-end" of sorts, which is when it will backtrack to try to find another path to the goal. In this respect, greedy

best-first search and DFS are quite similar, since both search algorithms will try a single path until a goal is reached or the path comes to a dead end. Our implementation of greedy best-first search also uses the straight-line heuristic that is used in our implementation of A*.

This slightly differing quality of greedy best-first search results in drastically different characteristics as compared to A*. Given the finite graph that we use and repeated state elimination, the algorithm is complete – but it is possible for the algorithm to get stuck in loops, since the cost of reaching the current node is not taken into consideration.

The *time complexity* of greedy best-first search is $O(b^m)$, where b is again the maximum branching factor and m is the maximum depth of the state space. We note, however, that just as with A* search, a good heuristic can lead to dramatic improvement in the time complexity. In essence, greedy best-first search can potentially become more like DFS, but with reduced branching. The *space complexity* is also $O(b^m)$, since we note that all nodes visited are kept in memory.

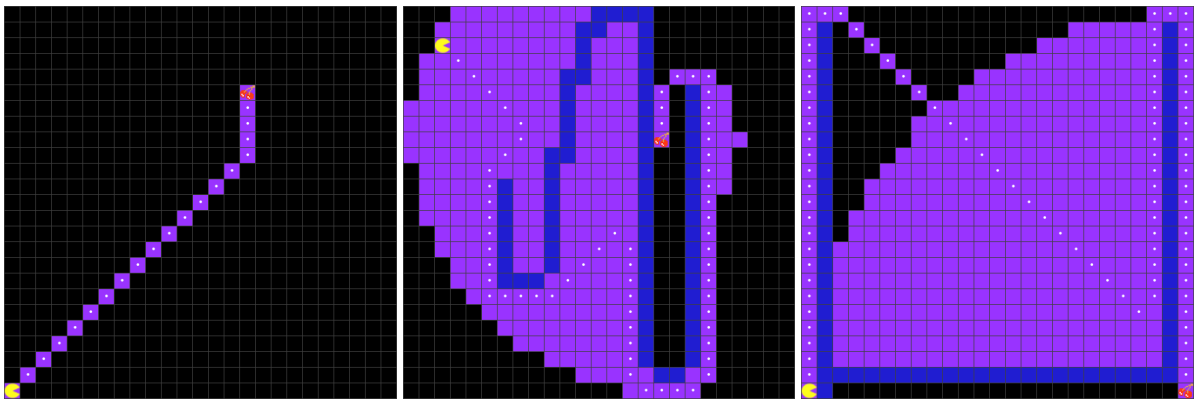
Finally, we note that greedy best-first search, unlike A*, is *not* optimal. Since the algorithm prioritizes time to completion over finding a shortest/least cost path, it is likely that in many instances, greedy best-first search returns a suboptimal solution. However, in general, greedy-best first searches fewer nodes than A*.

The detailed pseudocode for our implementation of greedy best-first search is given below.

```
1  public double greedyBestFirst(G, src, goal)
2      precompute h[n], the heuristic distances
3        h[x] <= distance from x to goal
4      new set distances
5      new set explored
6      new set path
7      new priorityQueue PQ
8      while (PQ is not empty) loop {
9        x <= PQ.removeMin()
10         add x to explored
11         if (x == goal){
12           return x
13           }
14       for all successors y of x loop {
15           newDist = distances[y] + weight[x,y]
16             if newDist < distances[y] {
17               distances[y] = newDist
18                 path[y] = x
19                 PQ.promote(y, h[y])
20               }
21           }
22       }
23     return none
24 end greedyBestFirst
```

Refer to Figure 2.9 below for screenshots of the greedy best-first search algorithm running under varying scenarios.

As we can see, greedy best-first search, on average, explores much fewer nodes than A* (compare with Figure 2.8 for reference) – but this happens at the cost of not always finding the shortest path. This phenomena is quite apparent in Figure 2.9b, when the search extends into a "pocket" and reconstructs a path that unnecesssarily visits this pocket. A worst-case scenario



(a) Greedy best-first in action, no wall.

(b) Path found by greedy best-first, with walls.

(c) A worst-case scenario for greedy best-first.

Figure 2.9: Screenshots of running the greedy best-first algorithm in varying scenarios.

is shown in Figure 2.9c, where the algorithm winds up reconstructing a very suboptimal path, even having explored enough nodes to reconstruct one that is shorter.

3 Code Implementation

3.1 Code Architecture

Our code base was initially adapted from Cornell's CS 2110: Object-Oriented Programming and Data Structures, offered in fall 2012 with Professor Doug James, but was heavily adapted to extend for algorithms, add a Pacman theme, and better highlight key differences between the algorithms. The main Java code architecture is mainly made of four class files: GUI.java, MapPanel.java, Map.java, and DigraphW.java. Four other classes implement auxiliary data structures; IntPair.java contains the code for (x, y) coordinates, AdjListEntry.java implements an adjacency list, and MinHeap.java and HeapEntry.java implement a minimum heap. Finally, our two classes DigraphWTest.java and HeapTest.java provide test cases using the JUnit framework.

For a more comprehensive look, our code is available on GitHub at *www.github.com/jjc297/InteractivePathFinding*.

3.2 Graphical User Interface

The graphical user interface (GUI) code is primarily located in GUI.java and partitions the window into the lefthand toolbar panel, bottom message console, and main map panel or the mazeâĂŹs grid. It uses the standard Swing framework for buttons, labels, borders, colors, images, and layout. GUI.java places the algorithm and tool buttons (source, destination, obstacle, erase, reset map) using Swing's GridLayout. Additionally, GUI.java initializes the message console, which communicates game status to the user; for example, whether the game is playable, if a source and destination have been set, potential errors, and final time and distance. Finally, the grid code resides in MapPanel.java, which paints the graph grid and creates the Pacman eating animation. MapPanel is mainly used for the overall look of the grid and calls the path functions, while Map.java implements the actual grid functions, such as setting and getting obstacles, setting and getting grid dimensions, reporting coordinate indexes, and resetting the graph functions. We created the icons such as Pacman, the cherry, and the tool buttons in Photoshop.

3.3 Path Algorithms

DigraphW.java houses the majority of the actual path finding code, as explained earlier in this paper. This code includes depth first search, breadth first search, depth limited, uniform cost, bidirectional, Dijkstra, iterative deepening depth first search, greedy best, A*, and iterative deepening A*. When a user clicks a path algorithm button, GUI.java calls the run function for that algorithm with MapPanel.java; MapPanel.java then calls the function in DigraphW.java, passing in the source, destination, visited cells, and path cells as parameters.

3.4 Design Choices

When creating our project, we centered our design choices around a Pacman theme because we believe it makes our project more interactive and adds new variety to frequently studied algorithms. We chose the black background, blue walls, uppercase font, and used a cherry as a goal to simulate a game experience. We added a small animation to draw out the path in white dots and have Pacman eat the cherry, reminiscent of the Pacman game. As a final note, the GUI used for the paper and for the presentation differ; the presentation version paints checked grid cells in gray; for our paper, we painted these cells in light purple and compacted the grid size for better visibility.

4 Conclusion & Summary of Results

After implementing all of our algorithms, we are then able to evaluate the results for each implementation. Our main measure of efficiency is based on the time is takes for the algorithm to complete. See Table 4.1 for a summary of our runtime results for each algorithm implemented.

Algorithm	Runtime (with no walls)	Runtime (with walls)
Breadth-First Search	1 ms	1 ms
Depth-First Search	2 ms	2 ms
Depth-Limited Search	1 ms	3 ms
Dijkstra's Algorithm	28 ms	19 ms
Uniform Cost Search	18 ms	7 ms
Bidirectional Search	3 ms	4 ms
Iterative Deepening Depth-First Search	40 ms	55 ms
A* Search	4 ms	17ms
Greedy Best-First Search	5 ms	11 ms

Table 4.1: Table representing runtime evaluations for each algorithm implemented.

An important note is that the runtime results presented above is not very accurate or representative of the actual time it takes for each algorithm to run. This is due to the idea that our grid space is not that large and therefore, the differences between runtimes of each algorithm may not be as noticeable. Most of the times evaluated are also directly related to the computer's own processing speed and so this has to be taken into account when analyzing the times noted in the above table.

Through the charting of our pathfinding algorithm results, we can make sense of why some algorithms are more efficient implementations than others. For example, we note that our two informed searches, greedy best-first search and A* search, tend to perform much better than some of the other implemented uninformed searched. Informed search methods tend

to perform quite well since they are given some guidance on where to look for solutions. We also noted before that a good heuristic algorithm can lead to dramatic improvements in time complexity so it makes sense as to why this would be the case and we can see these noticeable differences in the table above.

Overall, through our pacman GUI, we were able to execute a collection of graph search algorithms and visually determine the differences between each. By noting the time complexity, space complexity, and worst-case scenarios for each algorithm, we conducted an in-depth evaluation of how each method chose to construct a path from the start to goal node. With the addition of custom obstacles and start/goal nodes, we were also able to determine certain scenarios that are more beneficial or detrimental to the runtime of each algorithm. Finally, through the presentation of an overall visual representation of our agent, pacman, and his path to food, we were able to demonstrate all our search strategies in a unique and interactive way.