Optimal Path Finding using Dijkstra's Algorithm with Dynamic Programming

Name: Matthew Trang
Student PID: mattluutrang

Date: February 21, 2022
Project: 1

Abstract

Finding optimal paths through a two-dimensional (2D) maze is a common problem which can be represented as a graph. Using Dijkstra's Algorithm, we can find an optimal solution for the shortest path though the maze. In this paper, a dynamic programming approach is applied to Dijkstra's Algorithm, in both top-down and bottom-up approaches. The performance of both are compared with respect to accuracy and efficiency, and contrasted with a recursive approach.

1 Introduction

In this project, I address the topic of finding the optimal path through a 2D Maze using Dijkstra's Algorithm and Dynamic Programming. Dijkstra's Algorithm is a graph-based algorithm that finds the shortest path through a series of connected nodes with various weighted, directed edges. The optimal solution of the problem is the path which has the lowest cumulative distance between the start and end points. For solving 2D mazes, the algorithm can be used to apply a grid of nodes to the maze, and assign weights to each node, based on if they are traversable or not [2].

1.1 Motivation

2-D mazes are common and the task of finding the optimal path through them is a hard problem with many different solutions. There are a number of navigation tasks which maze exploration can be used to solve, such as city navigation, subway planning, mapping tasks, and others. Finding an optimal minimal path solution would improve the functionality of all of these tasks.

1.2 Formulation

Let G be a weighted, directed graph consisting of V vertices and E edges, such that $G = (V, E)$, where all edge weights are positive and non-zero, or $w(u, v) \geq 0$ for all edges $u, v \in E$. Dijkstra's algorithm maintains a set of vertices S which represent the set of vertices whose final shortest-path distance has already been determined. The algorithm then repeatedly selects the vertex $u \in V - S$ with the smallest final shortest-path distance, then relaxes all edges from u to update the distances of the vertices reachable from u, then adds the vertex u to S. The algorithm continues the selection, relaxation, and exclusion process until the selected vertex d is the desired destination. Then, it is proven that the path taken by tracing the vertices back from the vertex d is the smallest distance [1]. Thus, the algorithm optimizes the minimal distance from the start to the end vertices. In this problem, we consider a 2D maze of size $n \times n$, where n is the length in pixels of a side of the maze, and each pixel corresponds to a vertex, thus resulting in a graph of size n^2. Each pixel is connected with an edge to a pixel which it shares a side with, meaning each vertex has at most 4 edges.

1.3 Sample Input

The input to the algorithm is the image of the map that the algorithm must calculate the optimal path for. The image consists of a map made of black walls and white paths, with a green pixel for the start position and a red pixel for the end position, as shown in Figure 1. The maze images can be any number of rows and columns, with any number of paths to the start and end positions, as long as there is at least one valid path. Different sample inputs could be created using a maze maker. In this project, 23 mazes were created as sample test inputs.

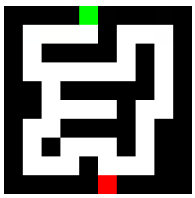


Figure 1: Sample 10 by 10 maze input for the algorithm

2 Applying Dynamic Programming

2D Maze solving, and more generally, graph path solving, is suitable for a Dynamic Programming approach as it can be broken down into optimal subproblems of the minimum distance to get to each point in the maze, with each point building upon the distance of a different point. Dynamic Programming is already being applied to the graph path solving problem through Dijkstra's Algorithm. While Dijkstra's Algorithm is more commonly known as a greedy algorithm, it can also be considered as a Dynamic Programming successive approximation procedure, inspired by the Bellman's Principle of Optimality [3].

2.1 Structure of an Optimal Solution

The algorithm uses the computation of the distances for each node during previous iterations and saves that value for each node, instead of recomputing the value of nodes at each step. Additionally, when the algorithm moves to a new vertex u in $V - S$, if the vertex u shares an edge with a vertex in S, the algorithm can skip the edge calculation for that vertex, as it has already calculated the optimal minimum distance from the source to that vertex. Thus, the algorithm exhibits optimal substructure, the optimal solution to finding the path to the final node encompasses the optimal solution of finding the path to each of the nodes before the final node on the optimal path. Dijkstra's Algorithm can be solved using a dynamic programming approach by the following.

2.2 Computing an Optimal Solution

After performing Dijkstra's Algorithm on the graph, the optimal solution for the minimum path problem can be found by the following:

Algorithm 1: GetSolution(G, d)

Data: G is a graph, d is the destination node, which has been reached
Result: shortest path

```
1 begin
2     path is initialized as an empty vector
3     while d.parent exists:
4         path appended with d.parent
5         d ← d.parent
6     return path
```

2.2.1 Naive Recursive Algorithm

While Dijkstra's Algorithm is innately a Dynamic Programming approach, it is possible to write a much more inefficient purely recursive method to solve the optimal shortest-path problem. This method would not use precalculated values, but instead would calculate the distances recursively for each node in the set S. The algorithm for the recursive approach to the maze solving problem is as follows:

Algorithm 2: MazeRecur(G, s, d, p)

```
Data: G = (V, E) is the graph, s is the
      source node, d is the destination
      node, p is the current path
Result: shortest path
1  begin
2      if s == d:
3          return p
4      p is appended with s
5      n is initialized as empty
6      for v ∈ V[s]:
7          if v ∉ p:
8              np = MazeRecur(G, v, d, p)
9              if np.length() < n.length():
10                 n = np
11     return n
```

The for loop inside each call is $\Theta(n^2)$, where n^2 is the number of vertices in the graph and n is the length of one of the sides of the maze, as it must check if the vertex it checks is in the path list, which could be as large as n^2, and the recursion itself is $\Theta(n^4)$ as it could potentially have to check for every node until it finds the finish node, and this could occur for every node. Thus, for the whole algorithm, the efficiencies are $\Theta(n^6)$, Big-O(n^6), and $\Omega(n^2)$ in the best case if the maze is linear with no branches.

2.2.2 Top-Down Dynamic Programming

The Top-Down/Memoized Algorithm for 2D Maze Solving aims to use Dijkstra's Algorithm in a way that writes the procedure recursively in a natural way, and then uses the results of the saved subproblems if they have previously been computed, or calculates them if they haven't been computed. The pseudocode for the Top-Down approach to Dijkstra's Algorithm is as follows:

Algorithm 3: GetMinDist(R)

```
Data: R = V − S is the set of nodes
      that have yet to be visited
Result: the node with the shortest
        distance from the source
1  begin
2      d = ∞
3      n = None
4      for u ∈ V:
5          if u.dist < d:
6              d = u.dist
7              n = u
8      return n
```

Algorithm 4: UpdateNeighbors(R, u)

```
Data: R is the set of nodes that have
      yet to be visited, u is the node to
      update the neighbors of
1  begin
2      for v ∈ R[u]:
3          if v.dist > u.dist + E[v, u]:
4              v.dist = u.dist + E[v, u]
5              v.parent = u
```

Algorithm 5: DijkRecurTD(G, s, d)

```
Data: G = (V, E) is a graph, s is the
      start node, d is the destination
      node
Result: shortest path
1  begin
2      G = G - s
3      if s == d:
4          return [s]
5      UpdateNeighbors(V − S, u)
6      u = GetMinDist(G)
7      if u is None:
8          return None
9      path += DijkRecurTD(u, d)
10     return path
```

Algorithm 6: DijkstraTD(G, s, d)

Data: $G = (V, E)$ is a graph, s is the start node, d is the destination node
Result: shortest path

```
1  begin
2      S ← ∅
3      initialize u.dist = ∞ : ∀u ∈ V
4      s.dist = 0
5      DijkRecurTD(G, s, d)
6      return GetSolution(G, d)
```

The runtime is $\Theta(|E|*T_{update} + |V|*T_{getmin})$ as it must run an update for each edge, and get the minimum for each vertex. The time to update the edges is constant, as each vertex only has 4 edges, and the GetMinDist function is $\Theta(|V|)$ as it is must search to find the minimum value in the set. Thus, for the whole algorithm, the efficiencies are $\Theta(4|V|+|V|^2) = \Theta(n^4)$, Big-O$(n^4)$, and $\Omega(n^2)$ in the best case if the maze is simple.

2.2.3 Bottom-Up Dynamic Programming

The Bottom-Up approach is similar to the Top-Down approach, but instead of recursively updating the graph, it would instead update the graph in a bottom-up fashion.

Algorithm 7: DijkstraBU(G, s, d)

Data: $G = (V, E)$, s, d
Result: shortest path

```
1  begin
2      S ← ∅
3      initialize u.dist = ∞ : ∀u ∈ V
4      s.dist = 0
5      for i = 1 to V.size:
6          u = GetMinDist(V − S)
7          S ← S ∪ u
8          if u == d:
9              break
10         UpdateNeighbors(V − S, u)
11     return GetSolution(G, d)
```

The runtime of the Bottom-Up approach is exactly the same as the Top-Down approach, with potentially more time saved due to less function calls, but still the same asymptotic behavior. Thus, the efficiencies are $\Theta(n^4)$, Big-O(n^4), and $\Omega(n^2)$.

2.3 Sample Output

Figure 2 shows the output of the Dijkstra's Algorithm for the 10 by 10 maze in Figure 1. The optimal shortest path from the green start pixel to the red finish pixel follows the blue path. In this case, the optimal path was a length of 19 pixels, which can be verified manually. Each algorithm was able to correctly come to this result, thus all three are equally accurate.

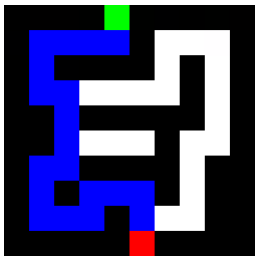
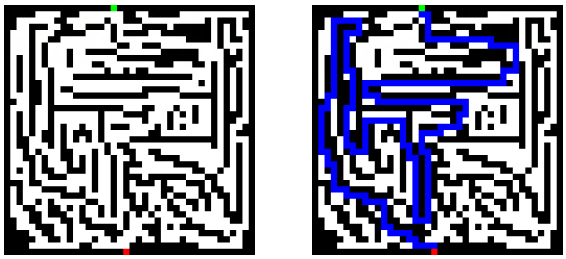


Figure 2: Output of a optimally navigated maze, where the blue line represents the shortest path.

Another example of the program output is shown in Figure 3, which uses a 40 by 40 pixel maze as input.



(a) Input (b) Output

Figure 3: Input and Output for a 40x40 maze

3 Benchmarks

In all cases, the algorithms were benchmarked using the time library in Python, which allows for precise time values in fractional seconds from a performance counter. The timing results do not include time spent loading in or annotating the images. It can be seen that the time taken for the recursive algorithm is exponentially larger than that for the Dynamic Programming approaches, with the 30 by 30 maze taking 3.19 seconds for the recursive approach compared to 0.05 seconds for both DP approaches. In fact, when moving to the 40 by 40 maze, the recursive approach takes over 15 minutes, and is thus not considered in the graph. The runtimes of the two DP algorithms are very similar, following expectations.

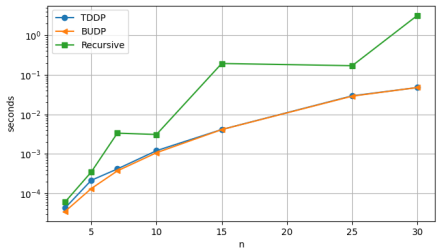


Figure 4: Runtimes of the 3 different algorithms for different n values on a log scale.

4 Parameters and Extensions

The program parameters that can affect the running time include the choice of input maze dimensions, with the choice of n maze length size having a squared effect on the number of nodes. This has a large impact on the runtime of the algorithms, as the more nodes, the more iterations of distance calculations must be done. The impact is shown in Figure 4. The difficulty of the maze path is also a factor in the algorithm, as the deeper the path is from the start to the end node, and the more branching paths which also lead towards the destination, the more iterations the algorithm has to perform to compare between the paths. The effect of an easy, medium, and hard difficulty maze is shown in Figure 5.

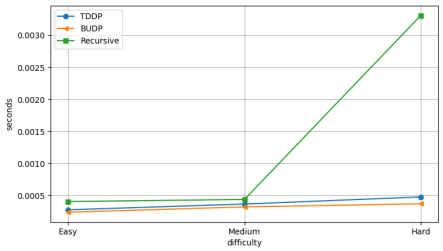


Figure 5: Runtimes of the three methods on varying levels of difficulties for a 5 by 5 maze.

However, the accuracy of the path finding algorithm is not affected by the difficulty of the maze, as Dijkstra's Algorithm has been proven to be optimal [1], and indeed, the output of the algorithms all follow the same minimal path which has been manually verified as optimal.

5 Conclusion

In conclusion, the optimal path through a 2D maze can be solved using Dijkstra's Algorithm with Dynamic Programming as an efficient and effective method as an alternative to some other heuristic based maze algorithm, or a recursive implementation.

References

[1] Thomas H. Cormen, editor. *Introduction to algorithms*. MIT Press, Cambridge, Mass, 3rd ed edition, 2009. OCLC: ocn311310321.

[2] Maxwell Reynolds. Python Learning Project: Dijkstra, OpenCV, and UI Algorithm (Part 1) - Prog.World, October 2020.

[3] Moshe Sniedovich. Dijkstra's algorithm revisited: the dynamic programming connexion. page 22.