Shortest Path Planning Algorithms Documentation

Dávid Blaško

December 2024

1 Introduction

This paper contributes to the semester project in the Algorithms of Artificial Intelligence at the Brno University of Technology, Faculty of Mechanical Engineering, Department of Automation and Computer Science. My motivation for his project was the visualization and implementation of a few shortest-path planning algorithms discussed during the semester.

2 Shortest Path Planning Algorithms

Shortest path planning algorithms are fundamental in computer science and operations research, addressing the problem of finding the most efficient route between two points in a graph or network. These algorithms have applications in transportation, telecommunications, robotics, and many other fields.

2.1 Brief history

The concept of shortest-path planning can be traced back to the mid-20th century. One of the earliest algorithms was developed by Edsger W. Dijkstra in 1956, specifically for non-negative weighted graphs. Over time, other algorithms like Bellman-Ford (1958), A*, and Floyd-Warshall emerged to address different constraints and graph types, such as graphs with negative weights or requiring all-pairs shortest paths.

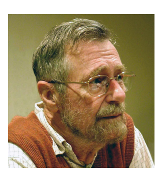

Figure 1: Dijsktra in 2002.

2.2 Categories of shortest-path planning algorithms

Shortest path algorithms can be classified by their approach and purpose:

1. **Single-Source Shortest Path Algorithms** - these find the shortest paths from a given source to all other nodes. For example *Dijkstra's, Bellman-Ford's* algorithms.
2. **All-Pairs Shortest Path Algorithms** - calculate shortest path between all pairs of nodes. Examples are *Floyd-Warshall, Johnson's* algorithms.
3. **Heuristic-Based Algorithms** - designed for specific scenarios like grid-based pathfinding (with different metrics like Manhattan, Euclidean, Octile), often used in games, robotics etc. For example *A*, D** aglortihms.
4. **Dynamic or Incremental Algorithms** - efficiently handle changes in the graph, such as adding or removing edges. Examples are *Lifelong Planning A** and Dynamic Shortest Path algorithms in general e.g *Shahrokhi's* algorithms.
5. **Specialized Algorithms** - designed for specific graph structures or constraints, such as *Bidirecitonal Dijkstra* for faster search or Yen's algorithm for *K-shortest paths.*

3 Dijkstra's algorithm

It computes the shortest paths from a single source node to all other nodes in a graph with non-negative edge weights. For negatively weighted edges is used Bellman-Ford's algorithm. Dijkstra's algorithm uses a greedy approach, progressively exploring the least-cost paths first.

Key Steps:

1. **Initialization** - starts with a source node, assigning it a distance of 0 and all other nodes a distance of infinity.
2. **Exploration** - repeatedly select the unvisited node with the smallest known distance, update it's neighbors distances, and mark it as visited.
3. **Termination** - continue until all nodes have been visited or the shortest path to the target nodes is found.

Time Complexity

For graphs utilizing adjacency matrices it's $O(V^2)$, and for priority queue (e.g binary heap) implementation it's $O((V + E)logV)$.

Algorithm 1: Pseudocode of Dijkstra's Algorithm in Python

Input: *grid*, *start*, *end*
Output: *True* if a path is found; raises error otherwise

```
// Initialize data structures
open_set ← priority queue with (0, id(start), start)
came_from ← empty dictionary // Tracks the path
g_score[node] ← ∞ for all nodes in grid
g_score[start] ← 0
visited ← empty set
while open_set ≠ ∅ do
    current ← pop_lowest_cost(open_set) // Pop the node with the
        lowest cost
    if current ∈ visited then
        // Skip already processed nodes
    visited ← visited ∪ {current}
    if current = end then
        reconstruct_path(came_from, end)
        return True
    foreach neighbor ∈ current.neighbors do
        if neighbor ∉ visited and ¬neighbor.is_obstacle() then
            temp_g_score ← g_score[current] + 1 // Calculate
                tentative cost
            if temp_g_score < g_score[neighbor] then
                came_from[neighbor] ← current
                g_score[neighbor] ← temp_g_score
                push(open_set, (g_score[neighbor], id(neighbor),
                    neighbor)) // Add to queue
                neighbor.set_open()
    draw_grid() // Visualize the grid
    update_display()
    if current ≠ start then
        current.set_closed() // Mark node as explored
    delay(30) // Optional delay for visualization
ValueError("No path found!")
```

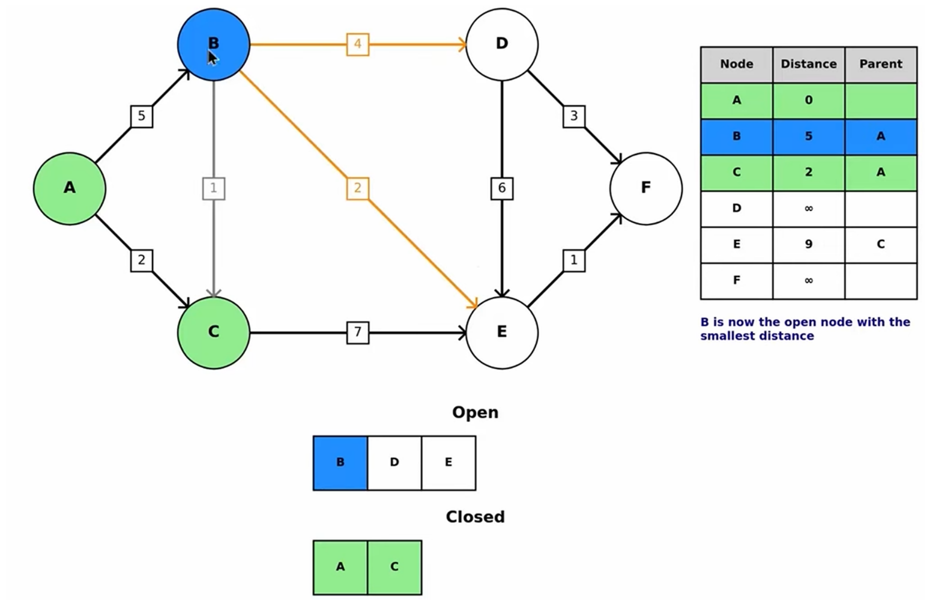

Figure 2: Dijsktra's algorithm scheme.

4 A* algorithm

It's a widely used heuristic-based pathfinding algorithm designed to find the shortest path from a start node to a goal node in a weighted graph. Developed in the late 1960s, it extends Dijkstra's algorithm by incorporating heuristics to guide the search, making it more efficient in many cases.

Key steps:

1. **Initialization** - starts with a source node, assigning it a cost function $f(n) = g(n) + h(n)$, where $g(n)$ is *actual cost from start node to the current node* and $h(n)$ is *heuristic estimate of the cost from the current node to the goal*.

2. **Exploration** - expands the node with lowest $f(n)$ value, updating the cost of its neighbors, and add them to the open list - *priority queue*.

3. **Termination** - continue until all nodes have been visited or the shortest path to the target nodes is found.

Time complexity

In the worst case it is $O(E)$, but often much faster in practise due to the heuristic function.

Algorithm 2: Pseudocode of A* algorithm in Python

```
Input: grid, start, end
Output: True if a path is found; raises an error otherwise
// Initialize priority queue and other data structures
open_set ← [] // Priority queue for exploration
heapq.heappush(open_set, (0, id(start), start)) // Push starting
    node with cost 0 into the queue
came_from ← empty dictionary // Track path information
g_score ← {node : ∞ for all nodes in the grid} // Cost from the
    start to all nodes, initialized to infinity
f_score ← {node : ∞ for all nodes in the grid} // Estimated total
    cost from the start through a node to the goal
g_score[start] ← 0 // Starting node has 0 cost
f_score[start] ← heuristic(start, end) // Initial heuristic
    estimate for starting node
while open_set ≠ [] do
    current ← heapq.heappop(open_set)[2] // Pop the node with
        the lowest f_scorefromthepriorityqueue
    if current = end then
        reconstruct_path(came_from, end) // Reconstruct the
            found path
        return True
    foreach neighbor ∈ current.neighbors do
        temp_g_score ← g_score[current] + 1 // Tentative cost for
            neighbor
        if temp_g_score < g_score[neighbor] then
            came_from[neighbor] ← current // Update path
                information
            g_score[neighbor] ← temp_g_score // Update actual cost
                to neighbor
            f_score[neighbor] ←
                g_score[neighbor] + heuristic(neighbor, end) // Update
                estimated total cost
            heapq.heappush(open_set, (f_score[neighbor], id(neighbor),
                neighbor)) // Push neighbor into the priority
                queue
            neighbor.set_open() // Mark neighbor as open for
                exploration
    draw_grid() // Visualize the current state of the grid
    pygame.display.update()
    // if current ≠ start then
        current.set_closed() // Mark as explored in
            visualization
    pygame.time.delay(30) // Pause for visualization
ValueError("No path found!")
```

5 Greedy Best First Search algorithm

It is a heuristic-based graph search algorithm that uses an informed approach to prioritize paths based on a heuristic function $h(n)$, which estimates the cost from a given node to the goal (usually using *Euclidean* or *Manhattan* distance). It is greedy because it always expands the most promising path first, based solely on the heuristic, without considering the actual cost incurred so far. Compared to A* doesn't guarantee the most optimal path.

Key steps:

1. **Initialization** - starts at the initial node (root) and adds it to the open list - *priority queue.*
2. **Exploration** - choose the node with the lowest heuristic value of $h(n)$ from the open list. Explore the neighbors of the current node and calculate their heurstic values. Add them to the open list if they haven't already been explored.
3. **Termination** - repeat until the goal node is reached or the open list is empty.

Time complexity

Depends on the actual size of maze, grid and its structure. At worst case may need to explore all nodes $O(E+V*logV)$ or traversing all edges (E) or extracting all nodes from priority queue using heap operations $O(V * logV)$.

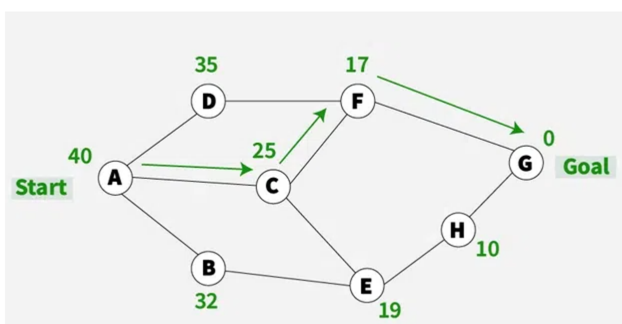

Figure 3: Greedy Best First Search scheme.

Algorithm 3: Pseudocode of Greedy BFS algorithm

```
Input: grid, start, end
Output: True if a path is found; raises an error otherwise
// Initialize priority queue and other data structures
open_set ← [] // Priority queue for exploration
heapq.heappush(open_set, (heuristic(start, end), id(start), start))
    // Push starting node with heuristic into the queue
came_from ← empty dictionary // Dictionary to track path
    information
visited ← empty set // Set to track explored nodes
while open_set ≠ [] do
    current ← heapq.heappop(open_set)[2] // Pop the node with
        lowest heuristic value from the priority queue
    if current = end then
        reconstruct_path(came_from, end) // Reconstruct the
            found path
        return True
    if current ∈ visited then
        // Skip if already visited
    visited ← visited ∪ {current}
    foreach neighbor ∈ current.neighbors do
        if neighbor ∉ visited then
            came_from[neighbor] ← current // Track the path
                information
            heapq.heappush(open_set, (heuristic(neighbor, end),
                id(neighbor), neighbor)) // Push neighbor into the
                priority queue with its heuristic value
            neighbor.set_open() // Mark neighbor as open for
                exploration
    draw_grid() // Visualize the current state of the grid
    pygame.display.update()
    if current ≠ start then
        current.set_closed() // Mark the explored nodes in
            visualization
    pygame.time.delay(30) // Pause to allow visualization
ValueError("No path found!")
```

6 Conclusion

Shortest path planning algorithms are fundamental tools for solving a variety of real-world problems, including robotics navigation, transportation planning, logistics, and network routing. This document explored the primary algorithms used for shortest path computation, focusing on Dijkstra's algorithm, the A* algorithm, and Greedy Best First Search.

Dijkstra's algorithm provides a systematic and guaranteed way to find the shortest path by exploring the least-cost paths in a graph with non-negative weights. While reliable, it can be computationally intensive, especially for large graphs. The A* algorithm improves on Dijkstra's by using heuristics to prioritize exploration toward the goal, which often leads to faster computation in practice while still ensuring the shortest path when implemented correctly. Greedy Best First Search, on the other hand, relies solely on heuristic estimates to prioritize nodes, making it faster but without the guarantee of finding the optimal path.

The comparison of these algorithms highlights that each has its unique strengths and trade-offs, depending on the problem's constraints, graph structure, and computational resources. Through pseudocode, key steps, and visualization examples, this document illustrated how these algorithms function and how their processes differ.

In conclusion, understanding the principles and differences among these shortest path planning algorithms is essential for optimizing pathfinding across diverse application areas. They are foundational to fields such as robotics, AI, logistics, and telecommunications, and they provide opportunities for further exploration, adaptation, and innovation.