Exploring A star and Star Craft

Jerlin Yuen and Mick Estiler

February 20, 2024

1 Introduction

To preface, we have decided to use DFS as our main method to generate our maze. We have used the repository, https://github.com/algoprog/Laby, to base our own implementation of the maze generation in our project. We mainly used Chris's implementation since it also included a start and end node within the 2d 101x101 maze matrix.

2 Question 1

Here, we will explore a bit about the A star algorithm.

Part A We want to explore the reasoning of why A* moves east instead of north in Figure 8 of the Homework 1 writeup.

First, we want to recognize that the algorithm does not know which nodes are blocked and which are unblocked. Only neighbouring nodes are visible (immediate north, south, west, east nodes). In figure 8, there are 3 possible nodes that can be traversed at the start, each of which is unblocked. They are: $(E, 1)$, $(E, 3)$, $(D, 2)$ The heuristic $h(n)$ that will be used will be Manhattan distances. $g(n)$ will simply be the total traveled distance.

At the start, $g(n)$ is obviously 0. $f(n) = h(n) + g(n)$, but $g(n) = 0$, so $f(n) = h(n)$. A* explores the path with the lowest $h(n)$. At $(E, 1)$, $h(n) = 4$. At $(D, 2)$, $h(n) = 4$. At $(E, 3)$, $h(n) = 2$. Obviously, $h(n) = f(n)$ at $(E, 3)$ is the smallest value so the agent moves to the east first.

Part B We want to give a convincing argument that the agent in finite gridworlds indeed either reaches the target or discovers that this is impossible in finite time. And also prove that the number of moves of the agent until it reaches the target or discovers that this is impossible is bounded from above by the number of unblocked cells squared.

First let's give a convincing argument that the agent in a finite girdworld will reach the target given that there is no blocked cell between the agent and the target.

Claim 1: If there are no blocked cells separating the agent and the target in a finite gridworld, then the agent is guaranteed to reach the target.

Proof: Let P be the set of all possible paths between the agent and the target. Since there are no blocked cells that barr agent from reaching target, P is non-empty since there has to exist at least one path. As the gridworld is finite, P is also finite. Therefore, the agent can explore each path in P one by one, ensuring it reaches the target or determines impossibility in finite time.

Now, we prove that an agent in a finite world will discover that it will not reach the given target in finite time given that there is no path from the agent to the target.

Claim 2: The exploration process of the agent in a finite gridworld either reaches the target or discovers impossibility in finite time.

Proof: Since there is a finite number of paths to explore in the gridworld, the agent can explore each path one by one. A* algorithm in this exploration uses a consistent manhattan heuristic, which maintains

a closed list ensuring that A* does not revisit any nodes. Since there is a finite amount of posible nodes to visit, we will explore all possible nodes, which will finish in finite time.

Now we prove that the number of max moves is less than the number of unblocked cells squared.

Claim 3: The number of moves of the agent until it reaches the target or discovers impossibility is bounded from above by the number of unblocked cells squared.

Proof: Let n be the number of unblocked cells. Because A* uses a consistent heuristic, the agent visits each unblocked cell at most once even in its worst case scenario. Therefore, the number of moves is at most n. Since there are n unblocked cells, the total number of moves is always less than $n \times n = n^2$.

3 Question 2 - Exploring Tie-Breaking Methods in Priority Queues

In this section, we delve into the concept of implementing different methods to resolve ties within a min priority queue. Our focus is on a queue that organizes nodes based on their f value, ensuring that the node with the smallest f value is retrieved when polling the queue. However, scenarios arise where two f values are exactly equal, necessitating a tie-breaking mechanism to maintain queue order.

To address this, we consider prioritizing nodes based on their g values as a tie-breaker. Given that our implementation is in Java, this tie-breaking logic is incorporated into the `compareTo` function. Specifically:

- **Prioritizing Smaller g Values:** If our goal is to prefer nodes with smaller g values in the event of a tie, the `compareTo` function can be implemented as follows: `return Double.compare(this.g, o.g);`.
- **Prioritizing Larger g Values:** Conversely, to prefer nodes with larger g values when a tie occurs, we would adjust the `compareTo` function to: `return -Double.compare(this.g, o.g);`.

This approach allows us to fine-tune our priority queue's behavior, ensuring that it aligns with the specific requirements of our algorithm or use case.

3.1 Runtime Analysis in a 101x101 Grid

In our 101x101 grid that was run 50 times, the average runtime for each grid search is as follows:

- For prioritizing smaller g values, the time each grid took was 9.384355004 seconds.
- For prioritizing larger g values, the time each grid took was 8.03633 seconds.

The difference in runtime can be attributed to the grid's design. Larger, more complex grids, such as our 101x101 scenario with over 10,000 squares, seem to favor algorithms that resemble depth-first search (DFS). When prioritizing larger g values, the algorithm explores nodes that are farther away, akin to how DFS operates by delving deeper into one path before backtracking.

3.2 Implications of Prioritizing Larger g Values

This method appears to run more efficiently because the goal state is likely very far from the start state. Such a strategy shares similarities with the depth-first search approach, which often runs faster in large, complex search spaces. DFS explores as far as possible along each branch before backtracking, which can be more efficient in scenarios where the goal state is distant. This minimizes the number of nodes visited by not fully exploring every level before moving to the next.

Consequently, prioritizing larger g values in our priority queue inadvertently adopts a strategy akin to DFS. This is particularly beneficial in our grid search problem, where the path to the goal is long and winding. It leverages the advantages of DFS, such as avoiding the overhead associated with maintaining a large frontier, which is common in breadth-first search (BFS). This tailored approach enhances our algorithm's efficiency in navigating through the complexities of the grid.

3.3 Further Explorations of Different Methods to Break Ties

In addition to direct comparisons between g values for tie-breaking, another approach involves assigning each node a single priority value. This method deviates from ordering nodes based purely on their smallest f value, relying instead on a pre-calculated priority that incorporates both f and g values. This strategy offers nuanced control over the prioritization process, particularly regarding how f and g values contribute to a node's overall priority.

3.3.1 Preferring Smaller g Values with a Single Priority Value

To implement an algorithm that prefers smaller g values using a single priority value, we adjust the formula to favor nodes with lower g values. The formula for preferring smaller g values is:

```
final double c = 20000; // Constant larger than any possible g-value in the grid.

// Calculate priority values incorporating both f and g, favoring smaller g-values.
double thisPriority = c * this.f + this.g;
```

This approach ensures that among nodes with the same f value, those with smaller g values receive higher priority.

3.3.2 Preferring Larger g Values with a Single Priority Value

Conversely, to prefer larger g values, we use a formula that inversely relates the g value's effect, rewarding higher g values with more prioritized positions. The formula for larger g values is:

```
final double c = 20000; // Constant larger than any possible g-value in the grid.

// Calculate priority values incorporating both f and g, favoring larger g-values.
double thisPriority = c * this.f - this.g;
```

This method prioritizes nodes with larger g values when f values are equal, aligning with strategies that benefit from deeper or costlier path explorations.

3.3.3 Impact of the Constant C on Priority Calculation

The constant C significantly influences the priority calculation for both smaller and larger g values. The choice of C allows for the fine-tuning of f and g values' relative impact:

- **Higher C Value:** Increasing C minimizes the relative impact of g value differences, emphasizing the f value's dominance in the priority calculation. This setting is more neutral regarding g value preferences, focusing on f value disparities.
- **Lower C Value:** Lowering C increases the significance of g value differences in the priority calculation. For smaller g values, this makes the algorithm more sensitive to g value reductions. For larger g values, it accentuates the benefit of higher g values in tie-breaking scenarios.

Adjusting C provides a mechanism to balance the influence of f and g values in determining node priority, allowing the algorithm to be customized to specific search problem requirements or optimization goals. In our example, of 101x101 grids, the C should be greater than 10,201.

3.3.4 Runtime Analysis and Algorithm Efficiency

The efficiency of algorithms using this priority calculation method varies based on grid design and implementation specifics. The analysis of runtime data for both preferences reveals how different C values and the emphasis on g values affect algorithm performance across various configurations. In our 101x101 grid, conducted over 50 runs, the average runtime for each grid search is as follows:

- For prioritizing smaller g values with a single priority value, the average time each grid took was 9.61443 seconds.
- For prioritizing larger g values with a single priority value, the average time each grid took was 8.584355004 seconds.

Interestingly, the trend of prioritizing larger g values leading to a faster result, as compared to prioritizing smaller g values, persists here. However, the performance times when using an integer as a priority value are slightly longer compared to direct comparison. This could be attributed to the extra CPU cycles required to calculate the priority value or it may be a result of the relatively small sample size.

4 Question 3 - Comparing Forwards vs. Backwards A*

We also used the tiebreak method to favor larger g values. If the g values are equal, we decided randomly on which one to explore.

- For Forwards A* the average time each grid took was 9.11443 seconds and explored 421,140.14 nodes.
- For Backwards A* the average time each grid took 10.072 seconds and explored 3,652,076.38 nodes.

It seems there there is a significant difference between nodes explored in backwards vs forwards A star. There could be a few explanations as to why.

4.1 Forwards A*

Forwards A* starts from the agents current position and aims to reach the goal. This may be a lot more intuitive as it simulates the perspective of an agent and is continually moving forward. It may also encounter obstacles A LOT sooner. Since when the maze is first starting, obstacles near the goal state are almost invisible whereas obstacles near the start state are likely to be seen. Remember that we can only see obstacles if the agent travels to there in run(), not through the hypothetical path computed in computePath().

4.2 Backwards A*

Backwards A* starts from the goal position and works back to the agent's current location. Can be less intuitive since it involves thinking in reverse but can be advantageous in certain scenarios, such as when the area near the goal is less obstructed and the initial path planning phase can proceed with fewer interruptions. Since the goal area is surrounded by obstacles we can't see yet, Repeated Backward A* might expand fewer nodes initially, as it can more freely establish a path before encountering denser obstacle distributions.

4.3 Comparison

While Backwards and Forwards A* algorithms show negligible efficiency differences on small grids, their performance diverges on larger grids. Backwards A* tends to lead to suboptimal efficiency as it explores nodes near the goal first, potentially ignoring obstacles near the start. Consequently, on large grids, it may find the quickest unblocked path from the goal to an undefined start area, requiring navigation around the entire grid for a suitable entrance into the start node. In contrast, Forwards A* prioritizes exploring paths away from obstacles near the start, resulting in a potentially more direct route but encountering more obstacles near the goal. Thus, the choice between the two algorithms should consider the grid's size and complexity, as well as the trade-offs between initial exploration strategies and overall pathfinding efficiency.

4.3.1 Goal Visibility

In some scenarios, the visibility of the goal from the start position or vice versa can impact the effectiveness of each algorithm. For instance, if the goal is highly visible from the start position, Forwards A* might have an advantage as it can quickly identify a path towards the goal. Conversely, if the start position is highly visible from the goal, Backwards A* might have an advantage as it can quickly identify a path towards the start.

5 Question 4 - Heuristics in the Adaptive A*

6 Question 5 - Adaptive A*

Runtime: 9.524 seconds Nodes Explored: 353434.02

7 Conclusion

Conclude your document here.