Fast Trajectory Replanning

Using versions of A algorithm*

Dhiraj Deelip Gandhi

Harsh Bhatt

Supervised by Abdeslam Boularias

Department of Computer Science

Course - Artificial Intelligence (Spring'20)

Rutgers University

Contents

1 Technologies

1.1 | Maven 3

Advantages

- Better Debugging
- Reduced Code Duplication
- More Component Builds
- Better Collaboration

1.2 | Java8

Advantages

- Lambda Expression
- Type Annotations
- Object Oriented Programming
- Pipelines and Streams

1.3 | SpringBoot 1.5

Advantages

- Simplified
- Version Conflict Free
- Quick Setup and Run
- No XML Configurations

1.4 | JavaFX

Advantages

- Clean API
- Built-in animation system
- Media Support
- Better Property systems and styling

1.5 | Git(Version Control)

Advantages

- Distributed Model
- Branching and Merging is Easy
- Flexible Workflow
- Assured Data Intergrity

1.6 | Priority Queue

We have implemented priority queue using binary heap as mentioned in the assignment.

2

Installation Instructions

2.1 | Prerequisites

The prerequisites for this project to run is

1. Maven 3
2. Java 8

2.2 | Link

Download project from here : Source Code : https://github.com/Dhirajdgandhi/JavaProjects/blob/master/rutgers/src/main/java/AI/Assignment1

2.3 | Command

mvn spring:boot run

3

Understanding the methods

3.1 | Agent's First Move

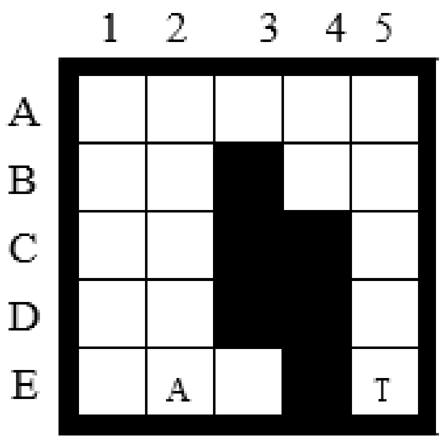


Figure 3.1: We can see the agent here at the position E2 and the target at position E5

In our problem, agent only knows about the neighboring cells or the cells that it has encountered and stored to be blocked. So, at the start, the agent's knowledge is that all the cells are unblocked. Then, it would start looking for the neighbor of the current cell to check if they are legal and unblocked. If the cell is **blocked**, the agent **stores this knowledge**, so that it is optimized in the subsequent A* searches and **prevents from running in an infinite loop**. Here, we have assumed that all moves take one time step for the agent.

Here, the heuristic also plays and important role in determining the move of the agent. The heuristic we are using here is the **Manhattan Distance**. The Manhattan distance of a cell is the sum of the absolute difference of the x coordinates and the absolute difference of the y coordinates of the cell and the cell of the target.

So in our current example, the agent is at the position *E*2 and knows that the move to the south is illegal. The other 3 cells are unblocked in the agent's grid-world. So, it calculates the f value of each of the neighbors and adds it to the open list (priority queue). We have, $f(n) = g(n) + h(n)$, where g is the step cost to move and h is the Heuristics. The f values :

1. ***EAST* is 1+2=3**
2. *NORTH* is 1+4=5
3. *WEST* is 1+4=5

Hence, open list has **EAST** as the highest priority, followed by the move to *NORTH* and *WEST*. Therefore, **the agent will move to east**.

3.2 | Does Agent always reach the target?

A* algorithm is a special case of finite graph search where we make the algorithm intelligent by using heuristics. To prove that A* is complete, we will prove that any finite graph search is complete which in turn will prove our cause.
The difference between the tree search and the graph search is that the graph search keeps track of the nodes that it has already searched and therefore would not get stuck in an infinite loop. In the case of a tree search, this kind of thing is not possible as we know that in DFS, the search could go into infinite loops sometimes.

Alternatively, we can use any heuristics until it is admissible and consistent. So, if we take a heuristic where the value of $h(n)$ is always 0. We can implement this h value in our algorithm as it is admissible and consistent. What would happen if we change the heuristics to this value? If we do that, the algorithm would search each and every node in the grid-world. And, if it does search the whole grid-world, it is bound to find the shortest path to the goal in finite time, provided the start is not separated by the goal by blocked cells. Thus, if we **use an admissible and consistent heuristics**, it will find the shortest path to the goal in a finite grid-world in shorter amounts of time.

Now, to get the bound for the number of moves.
Assume in a Grid World, all the cells are blocked except the initial cell. So the number

of unblocked cells here is $n = 1$. The maximum number of moves that the agent can make here is 0 as it can't move anywhere. That satisfies the equation of upper bound to $unblockedcells^2$ i.e. n^2. Consider Figure 4.5: Let's have only two unblocked cells, i.e., cell $E3$ and $E2$. The # of moves here is 4 which is bound by 2^2 i.e. 4. The 4 possible moves are :
1. Moving to the position $E2$
2. Moving to the position $E3$
3. Staying at the position $E2$
4. Staying at the position $E3$

Hence, at every possible cell, it has 2 options
1. Stay at the same position
2. Move to an unblocked position

And this happens till we have visited and explored all the $unblockedcells$ giving $unblockedcells^2$ moves as the most. This is the upper bound, as our agent won't move back and forth between two nodes and won't stay at it's place after an iteration. It rather announces that it is impossible to reach the target.

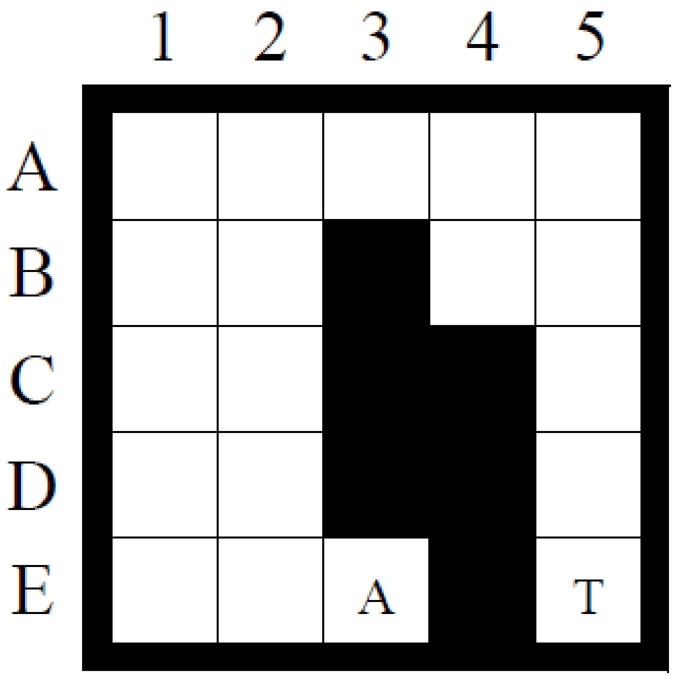


Figure 3.2: Original Problem

4 The Effect Of Ties

4.1 | How to break ties

In this section we will see what would happen if we have two nodes that have same f values. In that case, we can use the following equation to check which node should the algorithm select.

$$c * f(s) - g(s)$$

Let's see the below figure and consider the 2 black highlighted cells (*The don't mean block here*): Let's call the one closer to the start cell as A and the other one as B

A:

$$h(A) = 6$$

$$g(A) = 2$$

$$f(A) = g(A) + h(A) = 8$$

B:

$$h(B) = 2$$

$$g(B) = 6$$

$$f(B) = g(B) + h(B) = 8$$

There's a tie between the f values here. Clearly, B cell is closer to the goal and has a greater probability to reach it in less time. However, A cell is farther from the goal and has lesser probability to reach, as the probability of encountering blocked cells from *A->GOAL* is higher as the number of nodes from *A->GOAL* is higher, as $h(A) > h(B)$

Which says that B has travelled more than A and should be given chance to expand first.

And hence, B should win the tie to increase our probability of reaching the goal faster. Which generalizes to say that,

For any 2 cells A and B:

If $f(A) = f(B)$ Then, choose B if $g(B) > g(A)$ else choose A

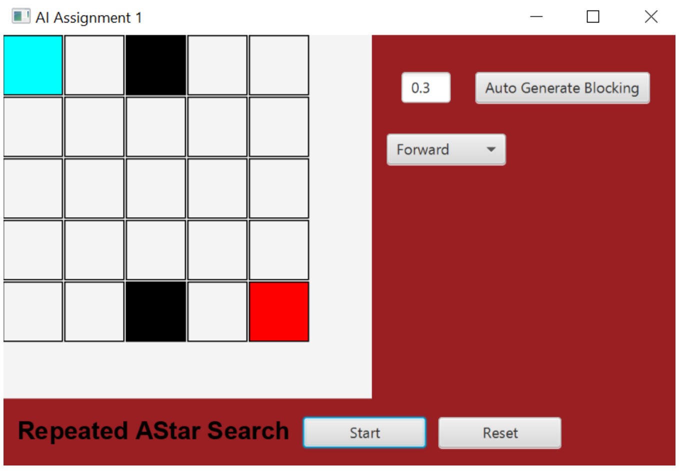


Figure 4.1: 5*5 matrix

5

Forward vs. Backward

5.1 | Implementation and comparison of both algorithms

The summary from the observations can be seen that the runtime and the nodes expanded of forward and backward A* is same if the maze does not have any blocked cell. The difference between the algorithms can be seen when there are blocked cells.

If we draw an imaginary diagonal in our matrix and check towards which side the clustering is more, near the initial state or near the final state. If the clustering of blocked cells is dense near the initial state as compared to the goal state, then Repeated forward A* works better in terms of runtime and number of nodes expanded. The reason for this is when we are in the dense area initially, it can quickly backtrack and find a good path to get out of the blocked area. After that, the pathfinding is quick as the number of blocked states are less and even if it encounters a blocked state, it can quickly backtrack to just some nodes and therefore, the runtime and nodes expanded is optimal. The same is the case when we use repeated backward A* and the clustering is dense near the goal state.

5.2 | Demo 1

Here, we will compare the runtime of both algorithms in a 10 * 10 matrix where the average blocked cells **density is higher near the initial state**. So, from our hypothesis above, repeated forward would be quicker than its counterpart.

From figure 5.2, we can see that the RunTime for repeated forward is 5385ms.

From figure 5.3, we can see that the RunTime for repeated backward is 8581ms.

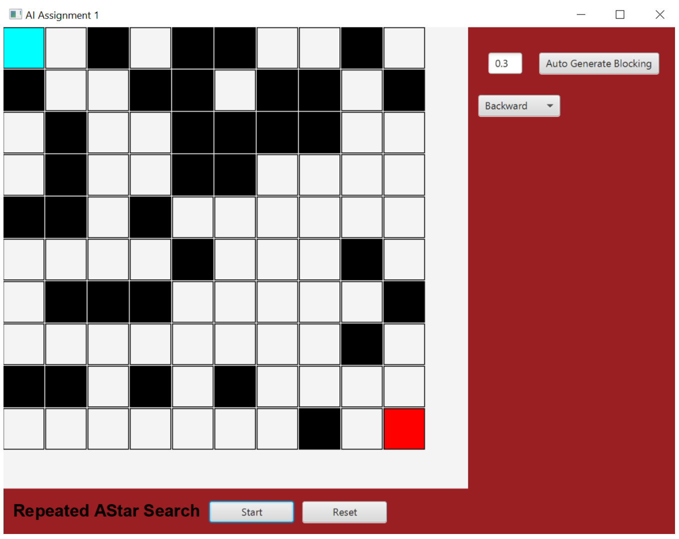
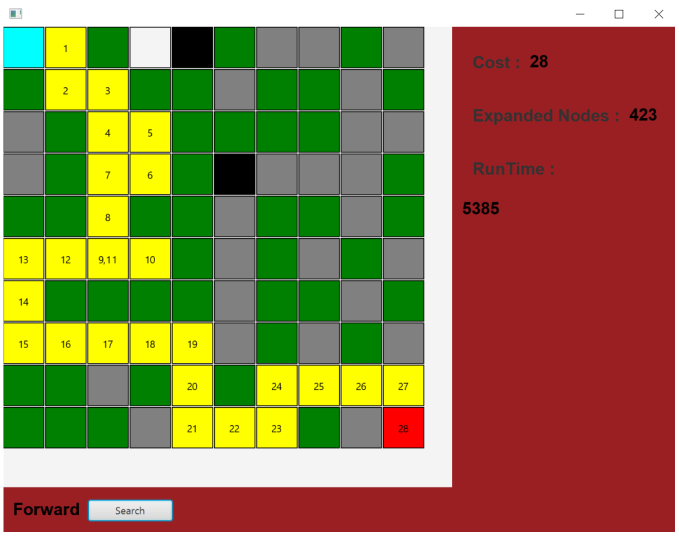


Figure 5.1: 10*10 problem



Cost : 28

Expanded Nodes : 423

RunTime :

5385

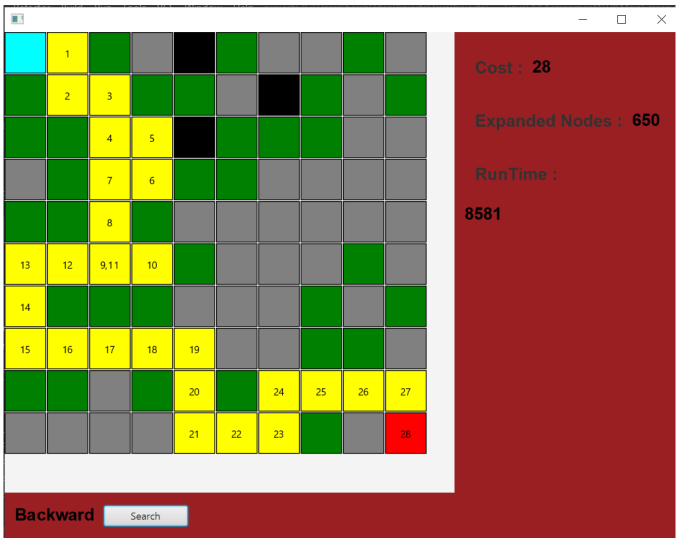
Figure 5.2: Repeated Forward

5.3 | Demo 2

Here, we will see the runtime of both algorithms in a 10 * 10 matrix where the average blocked cells **density is higher near the goal state**. So, from our hypothesis above, repeated backward would be quicker.

From figure 5.5, we can see that the RunTime for repeated forward is 11716ms.

From figure 5.6, we can see that the RunTime for repeated backward is 8189ms.



Cost : 28

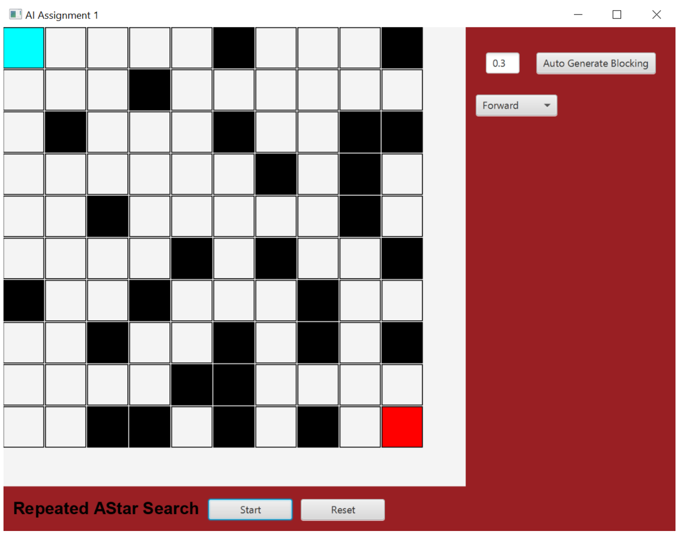
Expanded Nodes : 650

RunTime :

8581

Backward Search

Figure 5.3: Repeated Backward

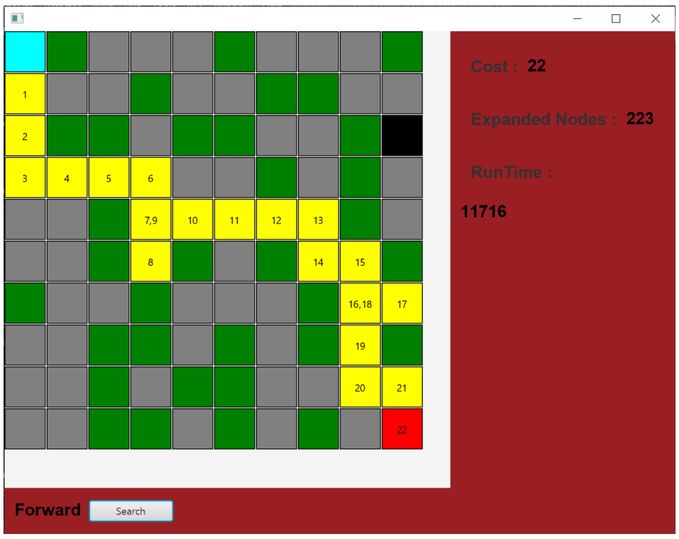


Repeated AStar Search

Figure 5.4: 10*10 problem

5.4 | Conclusion

So we can conclude that the runtime of the search depends on the amount of blocked nodes near the initial or goal cells. Whereas, in an unblocked environment, both algorithms take the same time.



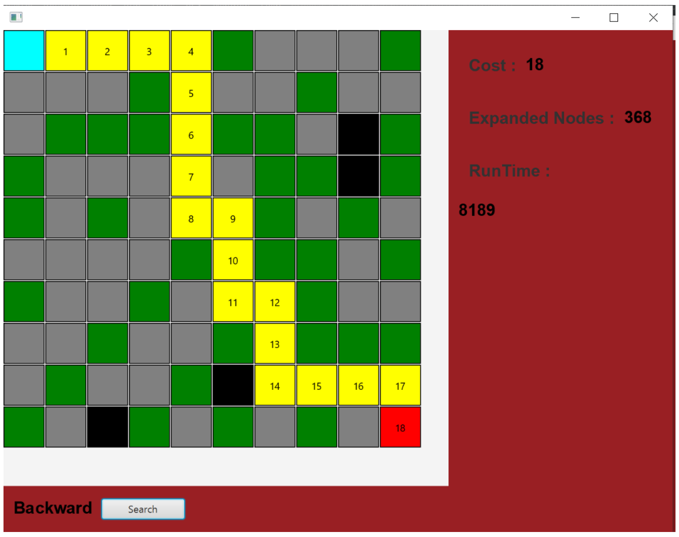
Cost : 22

Expanded Nodes : 223

RunTime :

11716

Figure 5.5: Repeated Forward



Cost : 18

Expanded Nodes : 368

RunTime :

8189

Figure 5.6: Repeated Backward

6

Heuristics in the Adaptive A*

6.1 | Why we use Manhattan distance

Firstly, in our grid-world, a move from the diagonals is not allowed. Now, two prove if an algorithm is admissible, we just have to prove that $h(n) \leq h^*(n)$ i.e. the value of our heuristics is less than or equal to the real value to go to n. And, to prove that an algorithm is consistent, we can prove this through the **Triangular Inequality**.

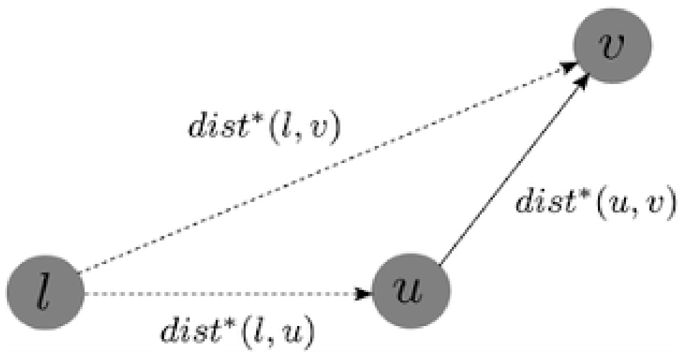


Figure 6.1: Triangular Inequality

The triangular inequality formula is :

$$h(l) \leq c(l,u) + h(u)$$

So, if we use **Manhattan Distance** as our heuristics *(when a move to diagonals are not allowed)*, the value to reach the goal node v should be less than the cost to move from l to u plus the manhattan heuristic from the node u. If we consider the 2*2 grid in figure 7.2, to move from $B1$ to $A2$, the agent has two ways to do that. Go up and then turn right or first go right and then go up. But, this is the exact definition of the **Manhattan Distance**. Thus, our triangular inequality would be satisfied when we use some bigger grids and therefore, the **Manhattan Distance** is consistent if our agent moves in the four main compass directions.

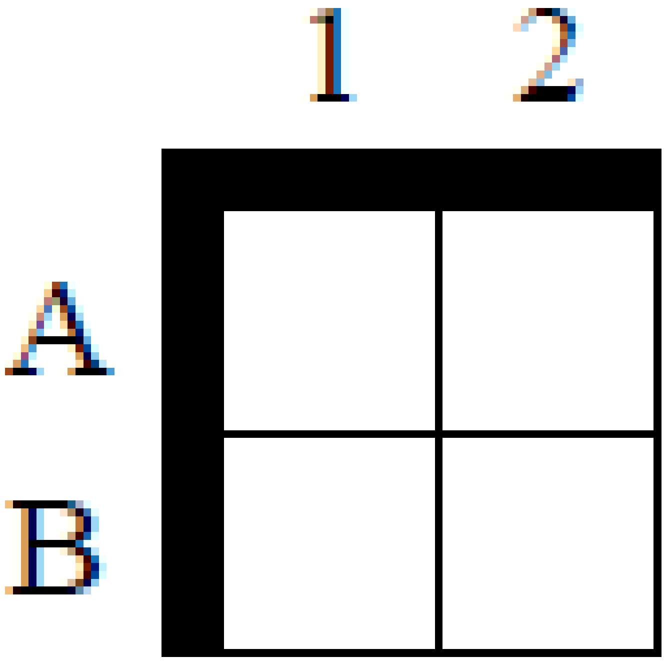


Figure 6.2: 2*2 grid

6.2 | $h_{new}(s)$ is consistent after cost increase

The consistency is proved by the triangular inequality by the formula :

$$h(l) \leq c(l,u) + h(u)$$

So, the h value for any cell can change when either the start node is changed or the goal node is changed, or the cost of an action is changed (we would only take the case when the cost is increased). In the case of Adaptive A* algorithm, we use the formula :

$$h(s) = MAX(H(s, s'_{goal}), h(s) - h(s'_{goal})$$

Here, we would check the difference between the two values of heuristics so that we have an accurate value of h to use in for the Adaptive A* algorithm. Case 1: When the start state is changed, the heuristics of any cell with respect to the goal would still remain the same as the heuristics is just the Manhattan distance between a cell and the goal cell.

Case 2: When the cost between 2 nodes increase, l and u increases, it would not effect our inequality as the value of $h(l)$ would still be less. Therefore, the consistency of the h values is maintained in the Adaptive A* algorithm. Case 3: The predicted cost between any cell to goal cell increases, on account when we encounter blocked cells while moving along the "shortest presumed-unblocked path" found by our A* search. In Adaptive A* we update our heuristics with this new cost found.

7

Forward A* vs Adaptive A*

7.1 | Runtime of both Forward and Adaptive A*

From the observations, we can see that overall, Adaptive A* has lower runtime and the nodes expanded is also lower than or equal the repeated forward A*. The reason for these observations is that in Adaptive A* algorithm, we use a better heuristics, which is admissible and consistent even after updating them.

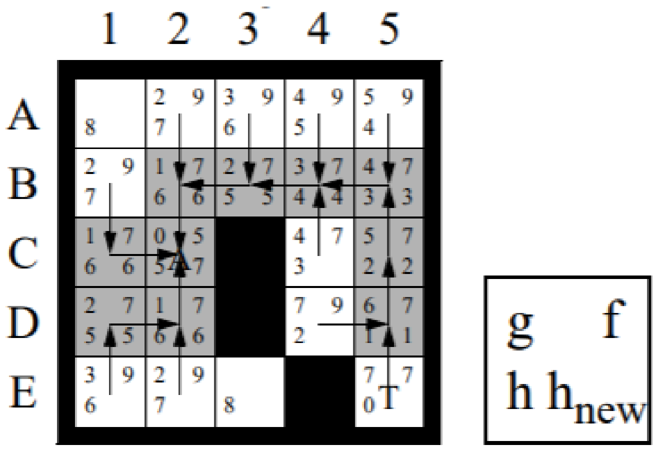


Figure 7.1: Explanation of h$_{new}$

Let us consider our agent currently at position $C2$. From the figure, the h value for this cell is 5. This value is due to the agent's knowledge of shortest presumed path. Now after backtracking, our updated h_{new} value would be 7 for the same cell as the agent now knows about the blocked neighbor cells.

Now consider the distance from $C2$ to $B2$ as y, distance from $B2$ to $B5$ as x, and distance from $B5$ to $E5$ as k, Mathematically, we can generalize as :

$$H^{\text{new}}(s) = (2 * y) + x + k$$

.

$$H^{\text{old}}(s) = x + k$$

Giving that we increased our heuristic by 2*y = 2*1 = 2 steps due to the C3 blocked cell we encountered while moving.

We will now show two of the observations we checked.

7.2 | Demo 1

First, we see the original problem of 5*5 matrix.

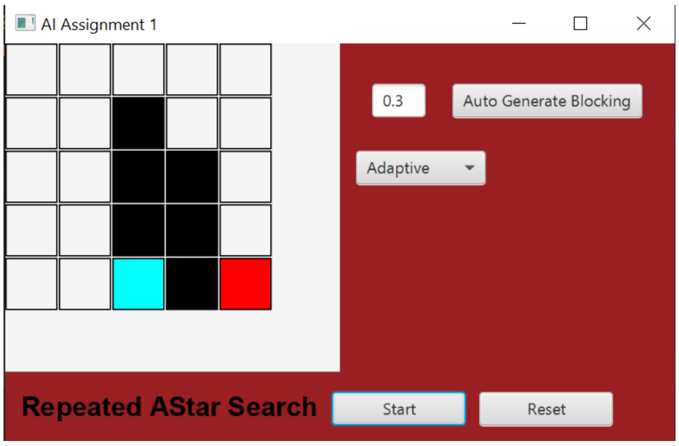


Figure 7.2: 5*5 original problem

From figure 7.3, we can see that the RunTime for repeated forward is 5725ms.

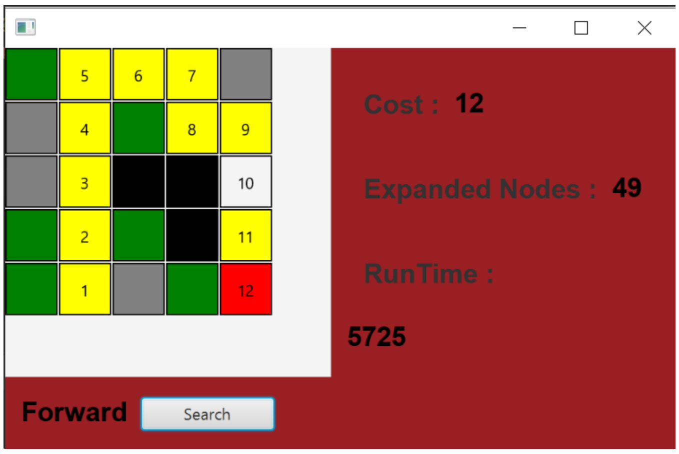
From figure 7.4, we can see that the RunTime for adaptive A* is 3054ms.

7.3 | Demo 2

Here, we will see the runtime of both algorithms in a bigger sized, 30*30 matrix.

From figure 7.6, we can see that the RunTime for repeated forward is 70744ms and # Expanded Nodes is 49.

From figure 7.7, we can see that the RunTime for adaptive A* is 51662ms and # Expanded Nodes is 45.



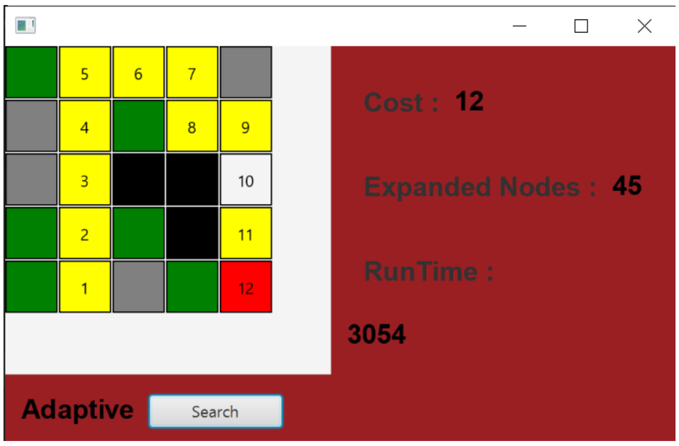
Cost : 12

Expanded Nodes : 49

RunTime :

5725

Figure 7.3: Repeated Forward



Cost : 12

Expanded Nodes : 45

RunTime :

3054

Figure 7.4: Adaptive

7.4 | Conclusion

From two demonstrations shown here, the optimal Cost and path found is the same. However, considering run time, the **Adaptive A* algorithm is faster as compared to the Repeated Forward A* algorithm**. This is due to the fact that we have changed our h values to more accurate ones.

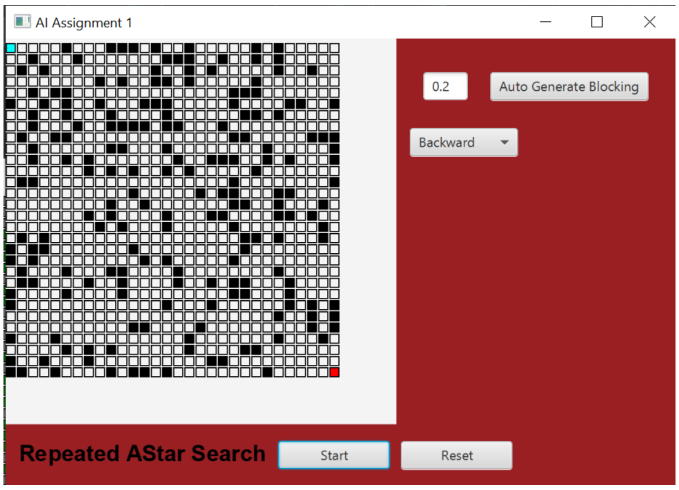
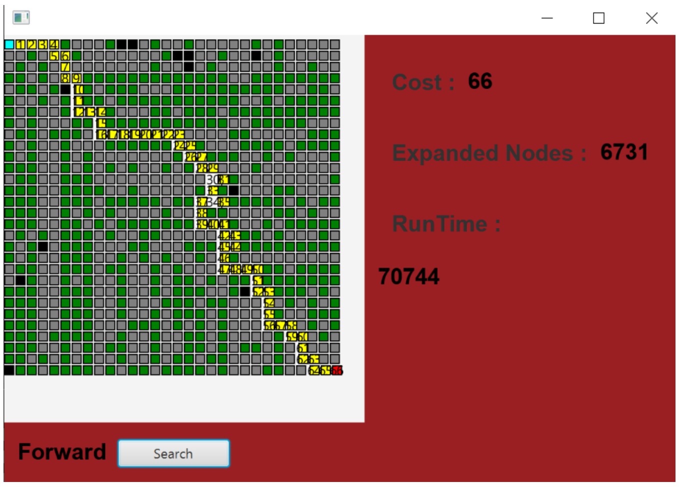


Figure 7.5: 30*30 problem



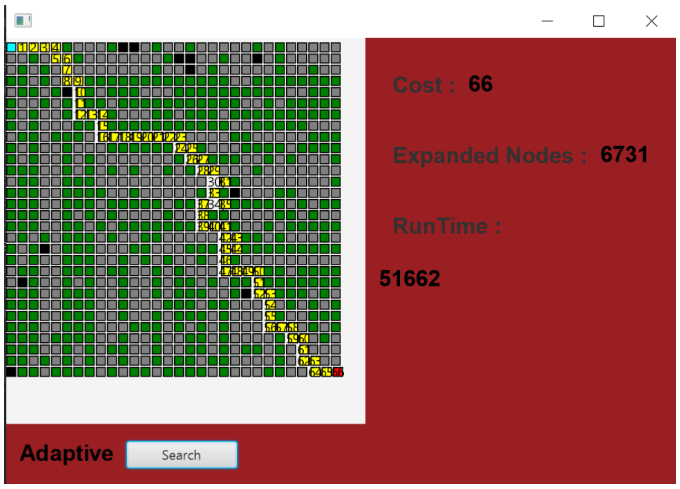
Cost : 66

Expanded Nodes : 6731

RunTime :

70744

Figure 7.6: Repeated Forward


Cost : 66

Expanded Nodes : 6731

RunTime :

51662

Figure 7.7: Adaptive

8

Statistical Significance

8.1 | Can two search algorithms be systematic?

From Chapter 5, we have a hypothesis that if the clustering of blocked cells is near initial cell, the Repeated Forward A* algorithm would work faster than the Repeated Backward A* and vice-versa.

From the 50 demonstrations for a matrix of 10*10 we took when the blocked cells were clustered near the initial cell, we found that the average time taken by the repeated forward A* algorithm was around **5400ms** while the average time for repeated backward was somewhere around **8500ms**.

So, by using these values, our repeated backward A* algorithm was around **57%** slower.

Now, when the blocked cells were clustered near the goal cell, we found the average time taken by the repeated backward A* algorithm was around **8100ms** while time taken for repeated forward A* algorithm was almost **12000ms**. Therefore, using these values, we found out that the repeated forward A* algorithm was **48%** slower.

8.2 | Conclusion

We can conclude from the observations found above that if the differences between two algorithms are systematic,i.e. in this case, related to cluster of blocked cells, our algorithm would return a systematic results accordingly. There would be some deviation from this result, though, when an algorithm in its first iteration find the correct shortest presumed unblocked path to be the final path. Then, the density of blocked cells would not matter as the other algorithm is already efficient in its first path chosen.

9 References

Empirical Methods for Artificial Intelligence by Paul Cohen(2006). http://www.eecs.harvard.edu/cs286r/courses/spring08/reading6/CohenTutorial.pdf

Coding Challenge 51.1: A* Pathfinding Algorithm - Part 1 by Daniel Shiffman (The Coding Train). https://www.youtube.com/watch?v=aKYlikFAV4k

JavaFX Java GUI Tutorial - 1 - Creating a Basic Window by thenewboston https://www.youtube.com/watch?v=FLkOX4Eez6o

Cormen, Leiserson and Rivest, Introduction to Algorithms, MIT Press, 2001.

Adaptive A* paper http://idm-lab.org/bib/abstracts/papers/aamas08b.pdf