Search Algorithms, Intelligent Systems

Riccardo Andronache

December 4, 2019

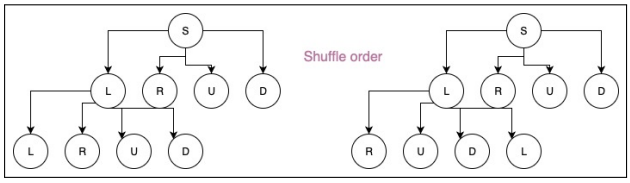
1 Approach

1.1 Depth-first search

A strategy followed by depth-first search is, as its name implies, to search "deeper" in the tree whenever possible. However, in this assignment "whenever possible" implies an infinite depth if we run into caveats like redundant paths or infinite loops.
Quote from Artificial Intelligence: A Modern Approach:

> The properties of depth-first search depend strongly on whether the graph-search or tree-search version is used. The graph-search version, which avoids repeated states are redundant paths, is complete in finite state spaces because it will eventually expand every node. The tree-search version, on the other hand, is not complete[...]

Since we are dealing with a tree[1] search we can neither optimize to the point where we do graph search instead nor avoid redundant paths. Therefore, before taking action we should think of a way that solves the infinite cycles problem. Since the branching factor[2] of the tree is $b = 4$, we can not be certain that there will be no cycles, but the following method makes sure to escape them.



Shuffling the order of the actions to be applied on the current node, thus creating new candidates for the fringe array would fix the problem.

1.2 Breadth-first search

Breadth-first search is one of the simplest algorithms for searching a graph or a tree and the archetype for many important algorithms. Breadth-first search is so named because it expands the frontier between discovered and undiscovered vertices uniformly across the breadth of the frontier, i.e. the algorithm discovers all vertices at distance K from starting vertex before discovering any vertices at distance K+1.

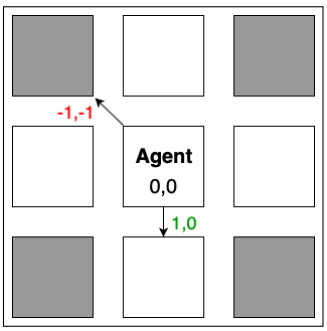
Used a non-recursive implementation of BFS: uses a queue Q, the frontier along which is currently searching. This method is defined to be complete, even for infinite graphs/trees, thus we are guaranteed to find a goal state, if one exists. Even though this provides the shortest path from source to destination, in the worst case scenario, suppose our solution is at depth D, and we expand all the nodes until we get to the last one on that depth, then the total number of generated nodes is $O(B^D)$. Hence, since it does not scale well for large inputs, especially trees, I had to make sure that the data structures I was using suited this algorithm well.

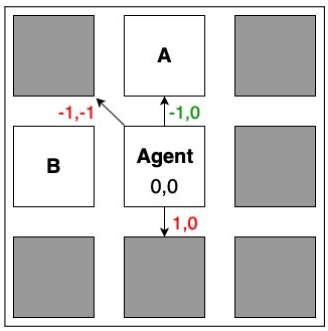
[1] b - branching factor, d - depth of solution.
[2] number of possible actions in this case.

I will try to give a brief insight in how I am applying the moves on the agent. I could have done this in the previous method, but since BFS is a sensitive type of search that requires more care it is better to do it in here.
Given position (x,y) of the agent I am creating a 3x3 searching block centered at the position of the agent.

```
public List<Grid> getNeighbors(Grid parent){
  List<Grid> neighbors = new ArrayList<Grid>();

  for(int x = -1; x <= 1; x ++) {
    for(int y = -1; y <= 1; y ++) {
      // Check for correctness of the move, i.e: can't go on diagonals.
      if( isCorrectMove(x, y) ) {
        int checkX = agentPosition.x + x;
        int checkY = agentPosition.y + y;
        // Check if the move is within the grid.
        if( isWithinGrid(checkX, checkY) ) {
          Grid candidate = new Grid(parent, checkX, checkY);
          neighbors.add(candidate);
        }
      }
    }
  }

  return neighbors;
}
```

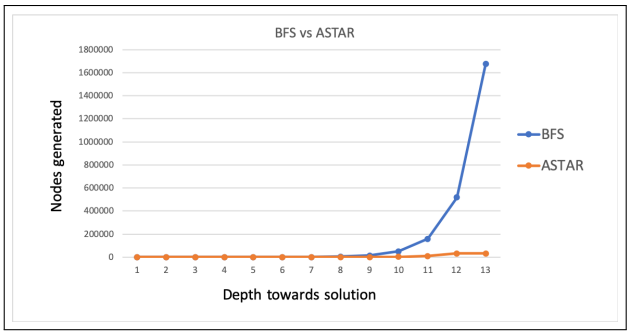
This will save us some boilerplate code by assuming 8 directions of movement. In our case 4 directions are enough so we can cross out the diagonals.





The pictures above are an intuitive idea of what I meant by create a 3x3 grid around the agent's position. The shaded areas represent unwalkable areas. We can notice that when x = 0 and y = 0 we apply no moves on the agent so we can skip that step. Also when x = -1 and y = -1 we can see from the picture above that is highlighted in red, meaning that is an illegal move. Or we can see another situation where the agent is in the bottom right corner and the only possible moves are on the left and down the grid. In the cases that a position of the grid contains valuable information, a non-empty tile, for example: A or B then we have to make sure that something keeps track of that. This is way more optimized than searching through the entire grid for the positions of A, B, C.

Further away I went with the problem in grid size, the running time growth of BFS looked like an exponential function compared to A^*.



1.3 A^*

A^* (pronounced "A-star") is a graph traversal and path search algorithm, which is often used in Computer Science due to its completeness, optimality, and optimal efficiency. -Wikipedia
A^*'s ability to vary its behaviour based on the heuristic and cost functions can be very useful in a game. The trade-off between speed and accuracy can be exploited to make the algorithm faster.
-theory.standford.edu
A^* selects the path that minimizes $f(n) = g(n) + h(n)\ \forall n \in \mathbb{D}$ [3].

The algorithm be manipulated in such ways that the number of nodes generated decrease significantly; provided something in extras regarding this matter. However, my solution relies on two lists, first is open list which represents the nodes generated after applying a move on the agent and second is closed list which represents the nodes already expanded with lowest f cost. Now, as you may have noticed, I could have used a min heap with respect to the f costs for the open list. That does not decrease the running time[4], but it increases the speed of the algorithm, which is not relevant for the purpose of this coursework, but useful to get a quick feedback from the program. The grid coordinates of A, B and C become crucial to get in constant time[5].

```
for(int i = 0; i < openList.size(); i ++)
{
    Grid candidate = openList.get(i);

    if(candidate.fCost < currentGrid.fCost || candidate.fCost == currentGrid.fCost &&
    candidate.hCost < currentGrid.hCost) {
            currentGrid = candidate;
            currentIndex = i;
        }
}
```

In the code above I am trying to highlight how nodes are picked for the closed list. Generally we loop through the open list - neighbours of the current grid - and try to find the neighbour with the lowest f cost. This is not the only condition we should take into consideration, because we can have neighbours with the same f costs. In that case we compare the neighbours with respect to their h cost and pick the one with lowest h cost.

[3] where $\mathbb{D}$ is the set of all nodes generated, n is the next node on the path, g(n) is the cost of the path from the start node to n, and h(n) is a heuristic function that estimates the cost of the cheapest path from n to the goal.
[4] in our case is expressed as nodes generated.
[5] again, trick that improves the speed of the algorithm.

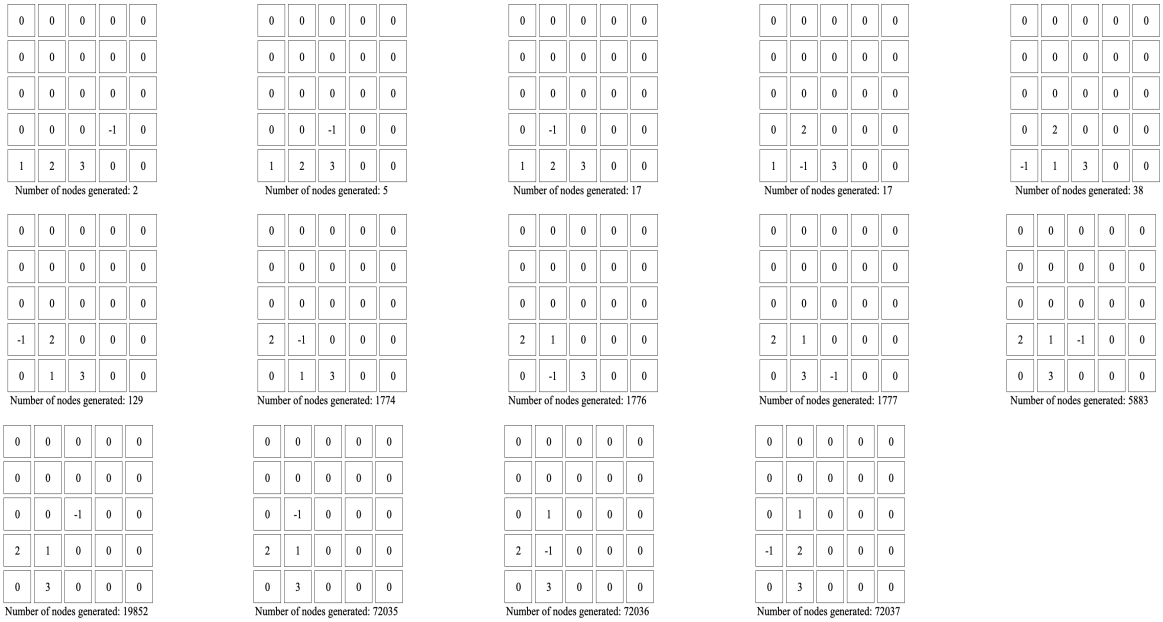
Heuristic used

I have chosen Euclidian Distance which works for any direction of movement. However, if this is the case, then I may have trouble using A* directly because the cost function g will not match the heuristic function h. If we don't assume diagonal move costs are the same as normal move costs and since Euclidian Distance is shorter than Manhattan or diagonal distance, I will still get the shortest path, but A* will take longer to run.

Euclidian distance heuristic

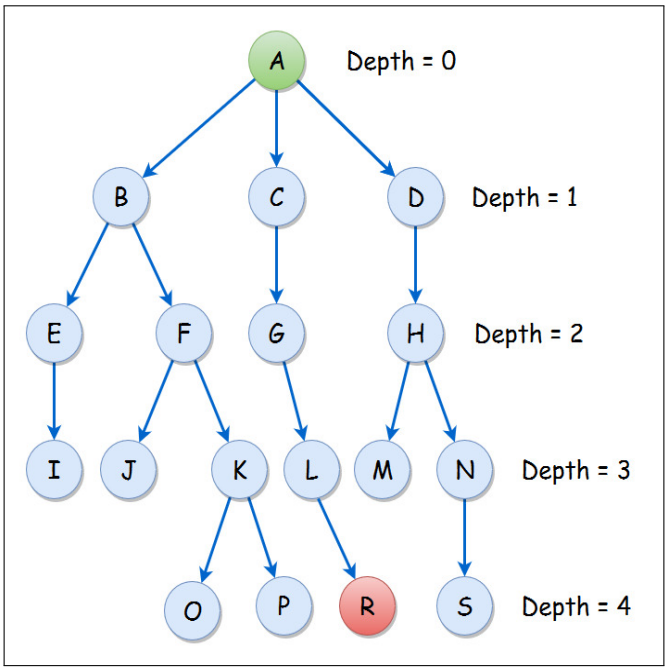
```
function heuristic(node):
    dx = abs(node.x - goal.x)
    dy = abs(node.y - goal.y)
    return sqrt(dx*dx + dy*dy)
```

Behaviour of A^* on a 5×5 grid



1.4 IDDFS

Iterative deepening depth first search[6] (IDDFS) is a hybrid of BFS and DFS. In IDDFS, we perform DFS up to a certain depth and keep increasing this depth after every iteration, thus preventing us search through an infinite branch of the tree. Even though I could have easily implemented this one using my DFS method, I tried to do it using recursion. I will display a picture and walk you through one example to understand the intuitive idea behind IDDFS.



Our starting node A is at a depth of 0. Our goal node R is at a depth of 4. The above example is a finite tree, but think of the above tree as an infinitely long tree and only up to depth = 4.
As stated earlier, in IDDFS, we perform DFS up to a certain depth and keep incrementing this allowed depth. Performing DFS upto a certain allowed depth is called Depth Limited Search (DLS).

```
public Grid DLS(Grid current, int depth, int running_time) {
  RUNNING_TIME = running_time;
  current.nodesGeneratedBefore = RUNNING_TIME;
  if(depth == 0 && current.checkGoalState()) {
    return current;
  }
  if(depth > 0) {
    for(Grid neighbor : current.getNeighbors(current)) {

      Grid found = DLS(neighbor, depth - 1, RUNNING_TIME + 1);
      if(found != null) {
        return found;
      }
    }
  }

  return null;
}
```

As Depth Limited Search (DLS) is important for IDDFS, let us take time to understand it first.
Let us understand DLS, by performing DLS on the above example. In Depth Limited Search, we first set a constraint on how deep we will go. Let's say our depth is 2. Now, in the above diagram, place your hand to cover the nodes at depth 3 and 4.

What is the order in which a normal DFS would visit them?

It would be as follows – ABEFCGDH

[6] http://theoryofprogramming.com/2018/01/14/iterative-deepening-depth-first-search-iddfs/

2 Evidence

I will take a trivial case and print the steps it takes to get to the goal state. To be rigorous with respect to this section I will try to use the debugger in the retrace path method which should take us from the goal node back to the start node.

Firstly, let's look at the method that retraces the path.

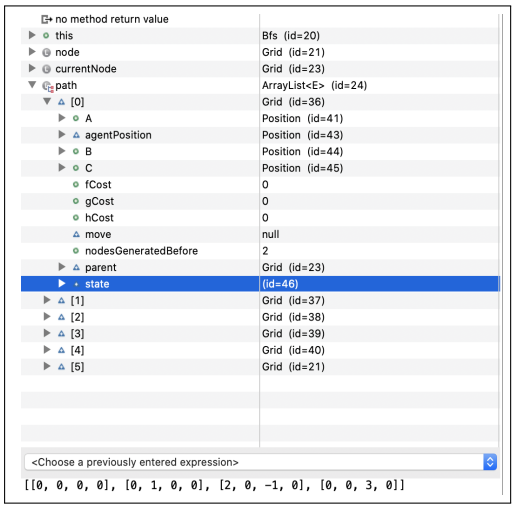
```
private void retracePath(Grid node) {
  Grid currentNode = node;

  List<Grid> path = new ArrayList<>();
  while(currentNode.parent != null) {
    path.add(currentNode);
    currentNode = currentNode.parent;
  }

  Collections.reverse(path);

  for(Grid g : path) {
    g.printState();
    System.out.println("Number of nodes generated by now " + g.nodesGeneratedBefore);
    System.out.println();
  }
}
```

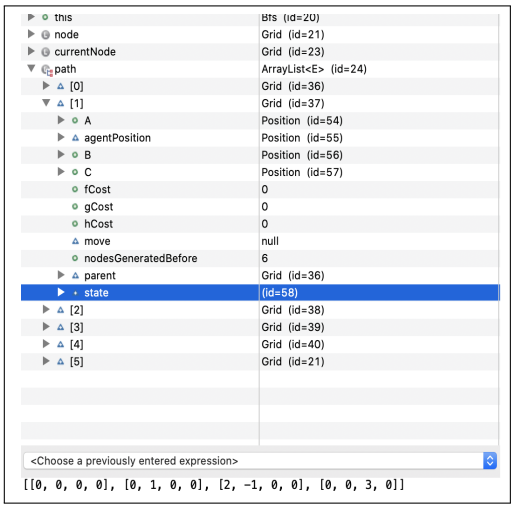
Secondly, Let's look at the method that checks a node against the goal node.

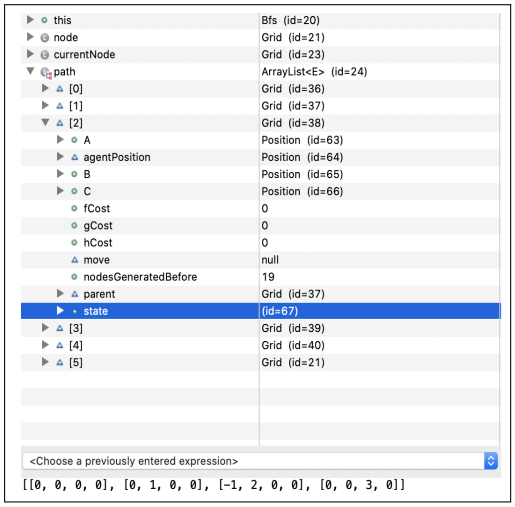
```
protected boolean checkGoalState() {
  if( A.x == Search.goalPositionA.x && A.y == Search.goalPositionA.y &&
      B.x == Search.goalPositionB.x && B.y == Search.goalPositionB.y &&
      C.x == Search.goalPositionC.x && C.y == Search.goalPositionC.y) {
    return true;
  }
  return false;
}
```

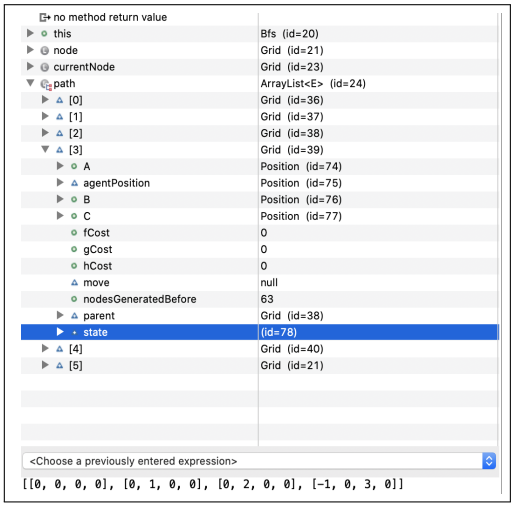
Remember at the beginning when I told you that keeping track of the positions of A, B, C will become handy at some point in the future, this is the case. Why this instead of just comparing the entire state of the current node with the state of the goal node? The answer is roughly easy, this is because that would take much more time overall considering how many nodes BFS generates for large grids. Furthermore, the position of the agent doesn't matter so the only things we need to worry about are A, B, C to be in the right positions.
I have started debugging in the for loop of the retrace path method. Down below you can see pictures of the states - in order - towards the goal state.
Even if the answers look similar for A*, BFS, IDDFS, that's because they all provide the shortest path towards the goal state. In the pictures below you can spot the difference by looking at the class' signature which lies next to `this` keyword in the pictures.
For DFS retracing the path would be a pain since it goes deeper and deeper into the tree until it finds the goal state. That means plenty of repeated states and probably sometimes cycles. Hence, I decided to print some of the beginning states and some of the end states.

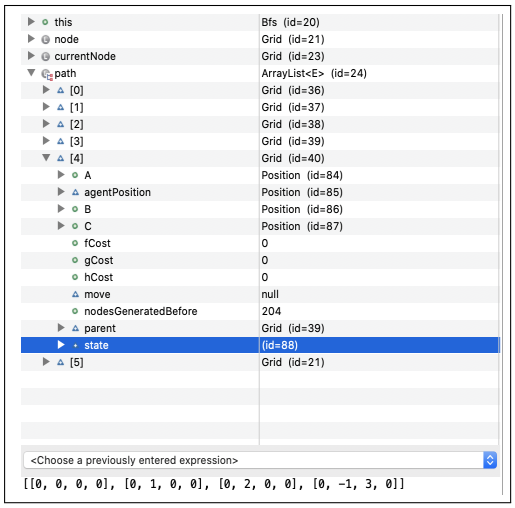
2.1 Breadth-first search

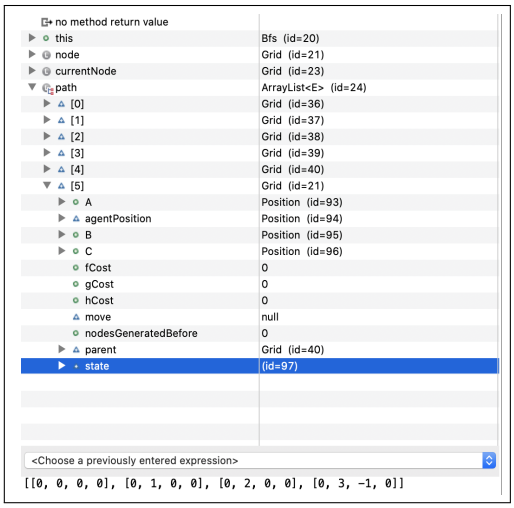




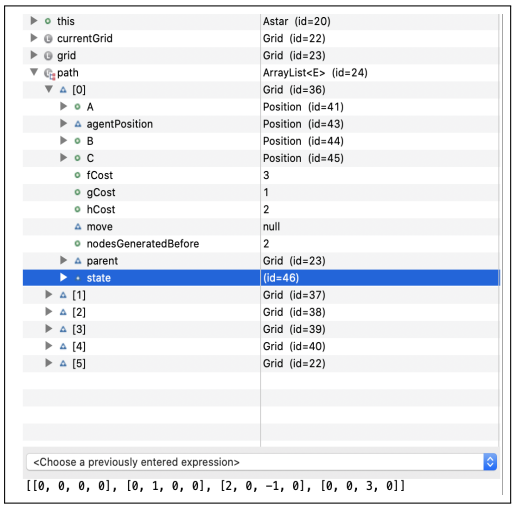


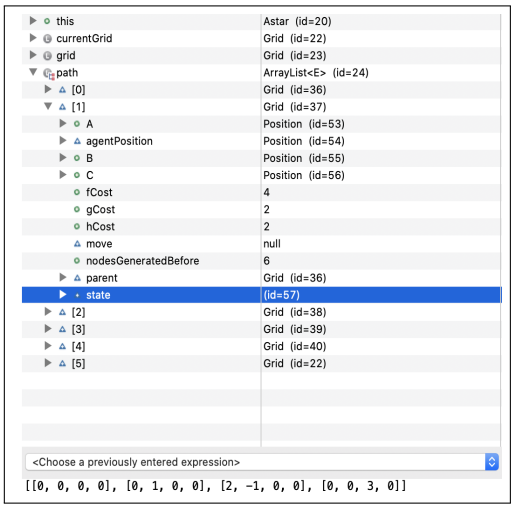


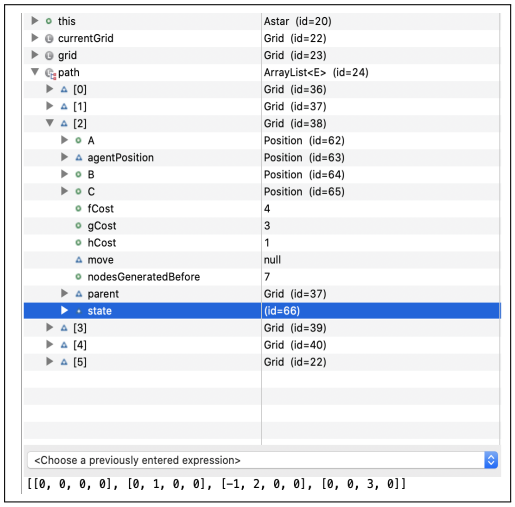


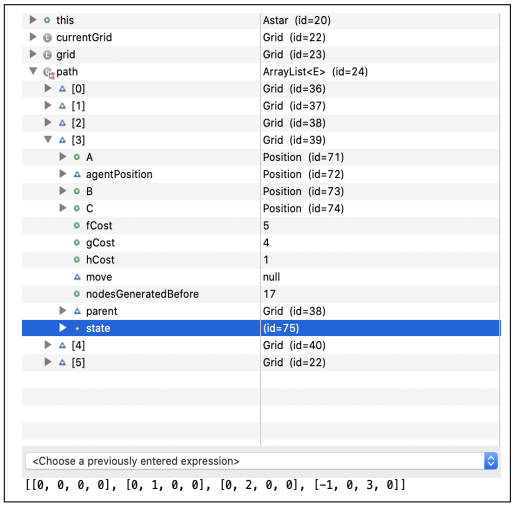


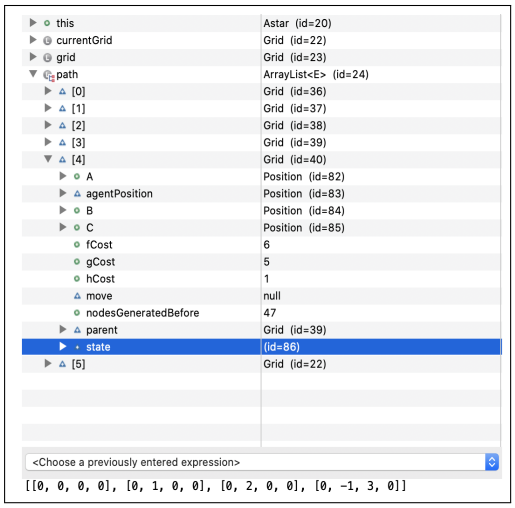
2.2 A^*

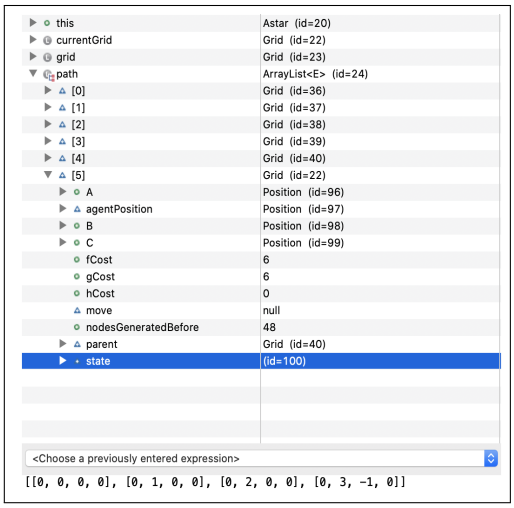

this Astar (id=20)
currentGrid Grid (id=22)
grid Grid (id=23)
path ArrayList<E> (id=24)
[0] Grid (id=36)
A Position (id=41)
agentPosition Position (id=43)
B Position (id=44)
C Position (id=45)
fCost 3
gCost 1
hCost 2
move null
nodesGeneratedBefore 2
parent Grid (id=23)
state (id=46)
[1] Grid (id=37)
[2] Grid (id=38)
[3] Grid (id=39)
[4] Grid (id=40)
[5] Grid (id=22)
<Choose a previously entered expression>
[[0, 0, 0, 0], [0, 1, 0, 0], [2, 0, -1, 0], [0, 0, 3, 0]]


this Astar (id=20)
currentGrid Grid (id=22)
grid Grid (id=23)
path ArrayList<E> (id=24)
[0] Grid (id=36)
[1] Grid (id=37)
A Position (id=53)
agentPosition Position (id=54)
B Position (id=55)
C Position (id=56)
fCost 4
gCost 2
hCost 2
move null
nodesGeneratedBefore 6
parent Grid (id=36)
state (id=57)
[2] Grid (id=38)
[3] Grid (id=39)
[4] Grid (id=40)
[5] Grid (id=22)
<Choose a previously entered expression>
[[0, 0, 0, 0], [0, 1, 0, 0], [2, -1, 0, 0], [0, 0, 3, 0]]

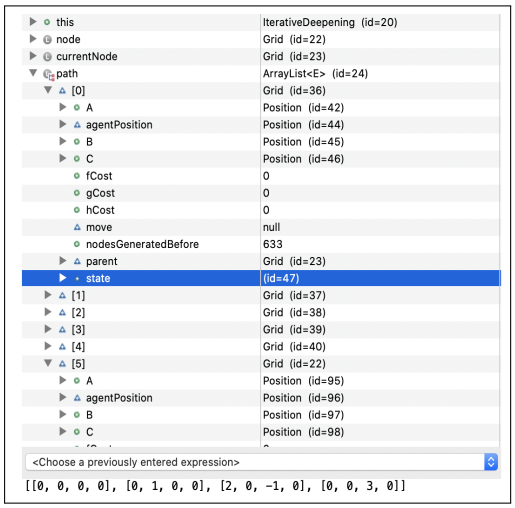

this Astar (id=20)
currentGrid Grid (id=22)
grid Grid (id=23)
path ArrayList<E> (id=24)
[0] Grid (id=36)
[1] Grid (id=37)
[2] Grid (id=38)
A Position (id=62)
agentPosition Position (id=63)
B Position (id=64)
C Position (id=65)
fCost 4
gCost 3
hCost 1
move null
nodesGeneratedBefore 7
parent Grid (id=37)
state (id=66)
[3] Grid (id=39)
[4] Grid (id=40)
[5] Grid (id=22)
<Choose a previously entered expression>
[[0, 0, 0, 0], [0, 1, 0, 0], [-1, 2, 0, 0], [0, 0, 3, 0]]

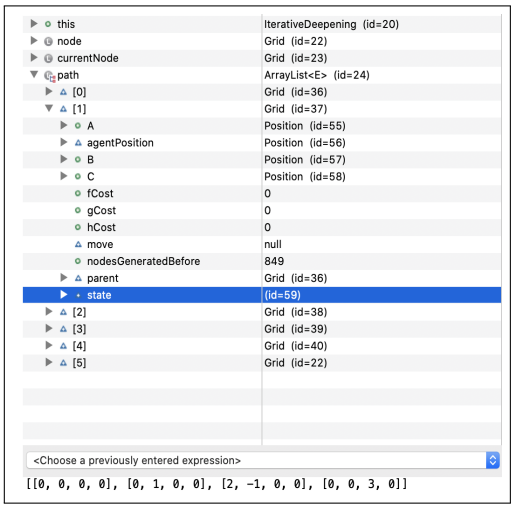

this Astar (id=20)
currentGrid Grid (id=22)
grid Grid (id=23)
path ArrayList<E> (id=24)
[0] Grid (id=36)
[1] Grid (id=37)
[2] Grid (id=38)
[3] Grid (id=39)
A Position (id=71)
agentPosition Position (id=72)
B Position (id=73)
C Position (id=74)
fCost 5
gCost 4
hCost 1
move null
nodesGeneratedBefore 17
parent Grid (id=38)
state (id=75)
[4] Grid (id=40)
[5] Grid (id=22)
<Choose a previously entered expression>
[[0, 0, 0, 0], [0, 1, 0, 0], [0, 2, 0, 0], [-1, 0, 3, 0]]

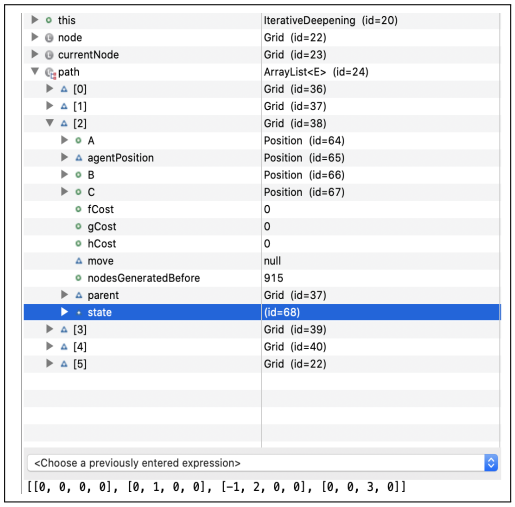

this Astar (id=20)
currentGrid Grid (id=22)
grid Grid (id=23)
path ArrayList<E> (id=24)
[0] Grid (id=36)
[1] Grid (id=37)
[2] Grid (id=38)
[3] Grid (id=39)
[4] Grid (id=40)
A Position (id=82)
agentPosition Position (id=83)
B Position (id=84)
C Position (id=85)
fCost 6
gCost 5
hCost 1
move null
nodesGeneratedBefore 47
parent Grid (id=39)
state (id=86)
[5] Grid (id=22)
<Choose a previously entered expression>
[[0, 0, 0, 0], [0, 1, 0, 0], [0, 2, 0, 0], [0, -1, 3, 0]]

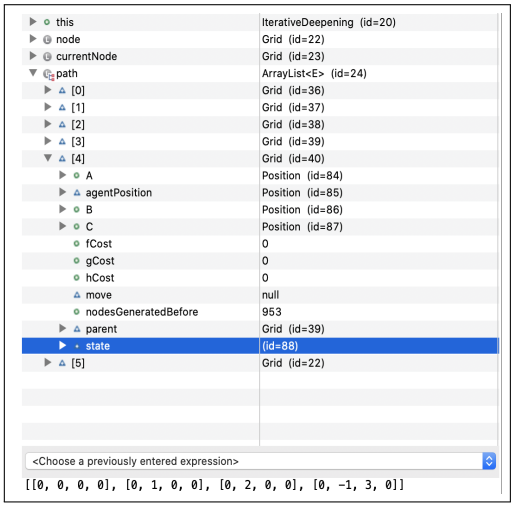

this Astar (id=20)
currentGrid Grid (id=22)
grid Grid (id=23)
path ArrayList<E> (id=24)
[0] Grid (id=36)
[1] Grid (id=37)
[2] Grid (id=38)
[3] Grid (id=39)
[4] Grid (id=40)
[5] Grid (id=22)
A Position (id=96)
agentPosition Position (id=97)
B Position (id=98)
C Position (id=99)
fCost 6
gCost 6
hCost 0
move null
nodesGeneratedBefore 48
parent Grid (id=40)
state (id=100)
<Choose a previously entered expression>
[[0, 0, 0, 0], [0, 1, 0, 0], [0, 2, 0, 0], [0, 3, -1, 0]]

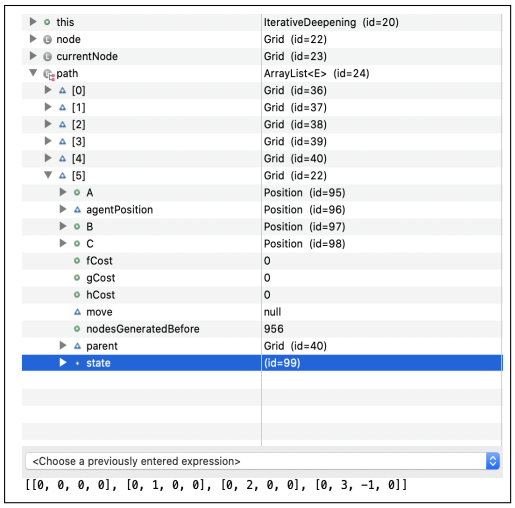
2.3 IDDFS

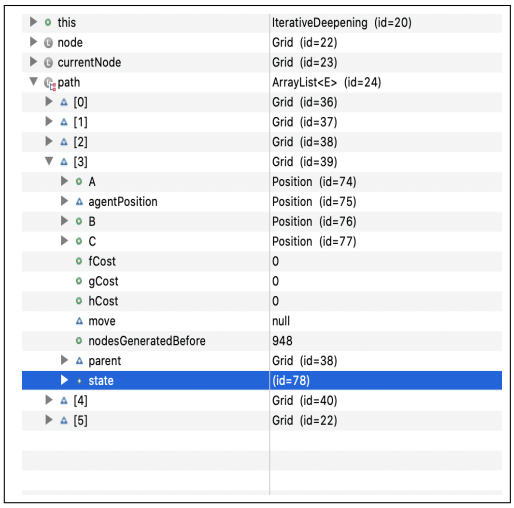

this IterativeDeepening (id=20)
node Grid (id=22)
currentNode Grid (id=23)
path ArrayList<E> (id=24)
[0] Grid (id=36)
A Position (id=42)
agentPosition Position (id=44)
B Position (id=45)
C Position (id=46)
fCost 0
gCost 0
hCost 0
move null
nodesGeneratedBefore 633
parent Grid (id=23)
state (id=47)
[1] Grid (id=37)
[2] Grid (id=38)
[3] Grid (id=39)
[4] Grid (id=40)
[5] Grid (id=22)
A Position (id=95)
agentPosition Position (id=96)
B Position (id=97)
C Position (id=98)
<Choose a previously entered expression>
[[0, 0, 0, 0], [0, 1, 0, 0], [2, 0, -1, 0], [0, 0, 3, 0]]


this IterativeDeepening (id=20)
node Grid (id=22)
currentNode Grid (id=23)
path ArrayList<E> (id=24)
[0] Grid (id=36)
[1] Grid (id=37)
A Position (id=55)
agentPosition Position (id=56)
B Position (id=57)
C Position (id=58)
fCost 0
gCost 0
hCost 0
move null
nodesGeneratedBefore 849
parent Grid (id=36)
state (id=59)
[2] Grid (id=38)
[3] Grid (id=39)
[4] Grid (id=40)
[5] Grid (id=22)
<Choose a previously entered expression>
[[0, 0, 0, 0], [0, 1, 0, 0], [2, -1, 0, 0], [0, 0, 3, 0]]

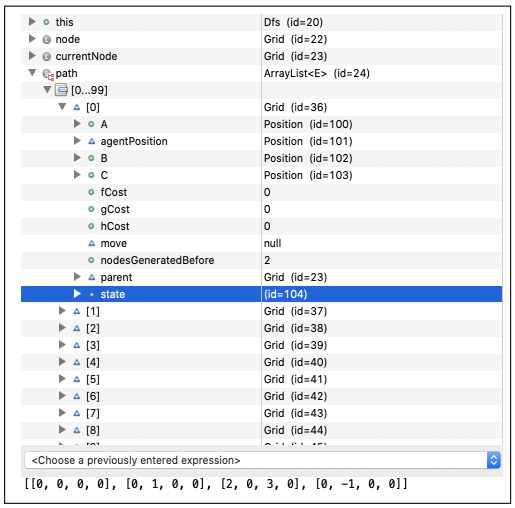

this IterativeDeepening (id=20)
node Grid (id=22)
currentNode Grid (id=23)
path ArrayList<E> (id=24)
[0] Grid (id=36)
[1] Grid (id=37)
[2] Grid (id=38)
A Position (id=64)
agentPosition Position (id=65)
B Position (id=66)
C Position (id=67)
fCost 0
gCost 0
hCost 0
move null
nodesGeneratedBefore 915
parent Grid (id=37)
state (id=68)
[3] Grid (id=39)
[4] Grid (id=40)
[5] Grid (id=22)
<Choose a previously entered expression>
[[0, 0, 0, 0], [0, 1, 0, 0], [-1, 2, 0, 0], [0, 0, 3, 0]]

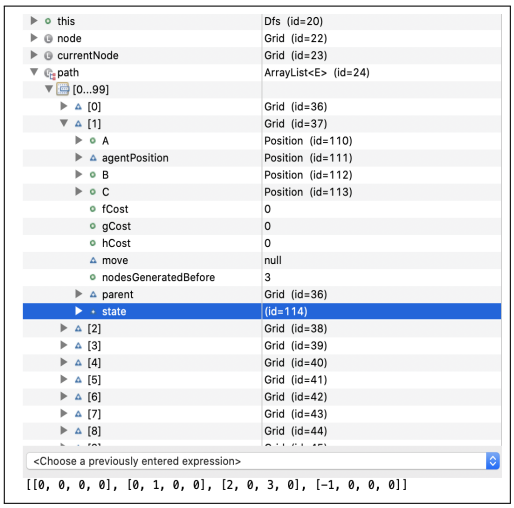

this IterativeDeepening (id=20)
node Grid (id=22)
currentNode Grid (id=23)
path ArrayList<E> (id=24)
[0] Grid (id=36)
[1] Grid (id=37)
[2] Grid (id=38)
[3] Grid (id=39)
[4] Grid (id=40)
A Position (id=84)
agentPosition Position (id=85)
B Position (id=86)
C Position (id=87)
fCost 0
gCost 0
hCost 0
move null
nodesGeneratedBefore 953
parent Grid (id=39)
state (id=88)
[5] Grid (id=22)
<Choose a previously entered expression>
[[0, 0, 0, 0], [0, 1, 0, 0], [0, 2, 0, 0], [0, -1, 3, 0]]

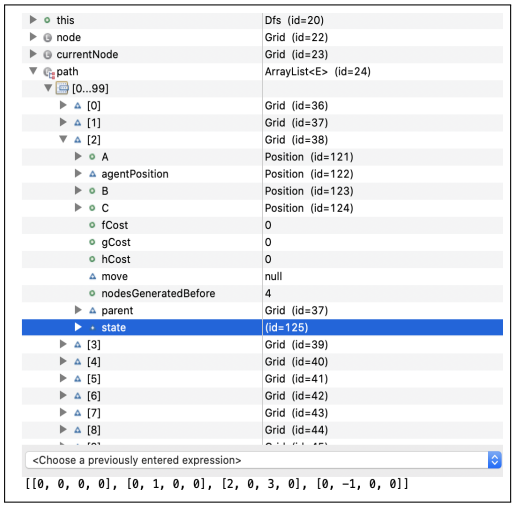

this IterativeDeepening (id=20)
node Grid (id=22)
currentNode Grid (id=23)
path ArrayList<E> (id=24)
[0] Grid (id=36)
[1] Grid (id=37)
[2] Grid (id=38)
[3] Grid (id=39)
[4] Grid (id=40)
[5] Grid (id=22)
A Position (id=95)
agentPosition Position (id=96)
B Position (id=97)
C Position (id=98)
fCost 0
gCost 0
hCost 0
move null
nodesGeneratedBefore 956
parent Grid (id=40)
state (id=99)
<Choose a previously entered expression>
[[0, 0, 0, 0], [0, 1, 0, 0], [0, 2, 0, 0], [0, 3, -1, 0]]

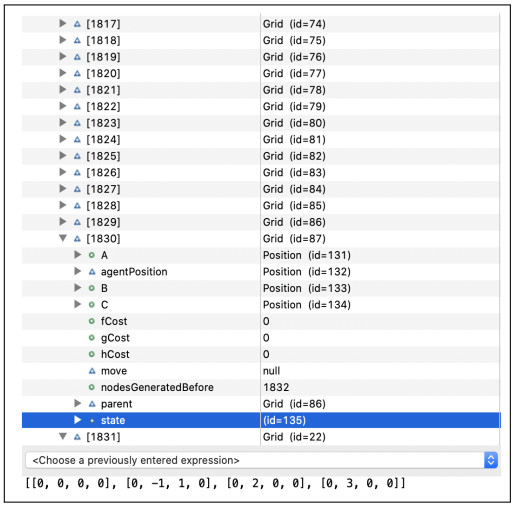

this IterativeDeepening (id=20)
node Grid (id=22)
currentNode Grid (id=23)
path ArrayList<E> (id=24)
[0] Grid (id=36)
[1] Grid (id=37)
[2] Grid (id=38)
[3] Grid (id=39)
A Position (id=74)
agentPosition Position (id=75)
B Position (id=76)
C Position (id=77)
fCost 0
gCost 0
hCost 0
move null
nodesGeneratedBefore 948
parent Grid (id=38)
state (id=78)
[4] Grid (id=40)
[5] Grid (id=22)

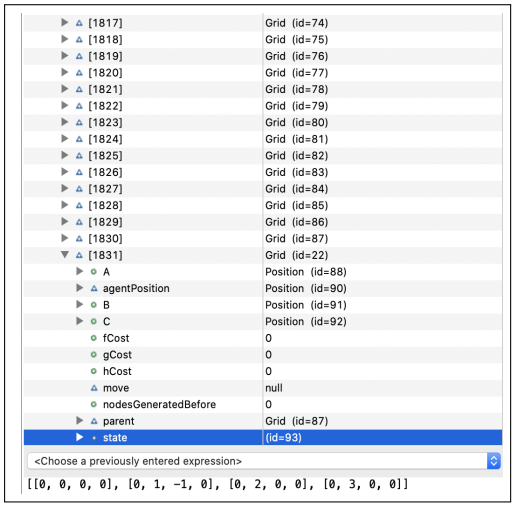
2.4 Depth-first search











2.5 Scalability

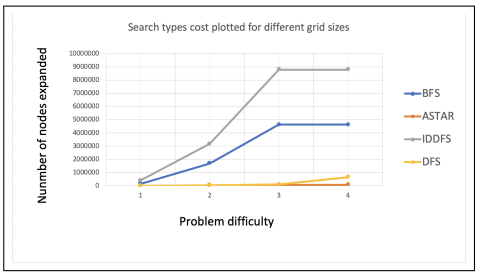
How do I test scalability?

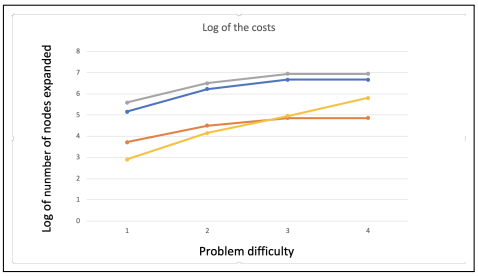
Consider a correct move towards the goal[7] as one level deeper in the tree. If we store a variable in the *Grid* [8] class that keeps track of the nodes generated before itself, then we can draw some conclusions.

	Number of nodes expanded			
Grid	BFS	ASTAR	IDDFS	DFS
3x3	144759	5267	390833	813
4x4	1675988	32008	3168808	14200.5
5x5	4622858	72037	8779809	89296
6x6	4622858	72037	8779809	646719.5

	Number of nodes expanded			
Grid	BFS	ASTAR	IDDFS	DFS
3x3	5.1606456	3.7215633	5.5919912	2.9100905
4x4	6.2242709	4.5052585	6.5008959	4.1523036
5x5	6.6649106	4.8575556	6.9434851	4.950832
6x6	6.6649106	4.8575556	6.9434851	5.810716

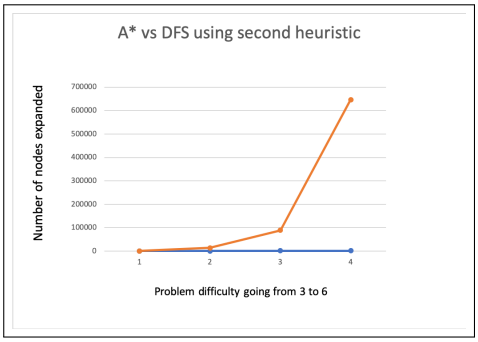
The first table represents the normal number of nodes expanded of the searches, whereas in the second table I took the log of the nodes to see the growth of the methods from different perspectives.
As you may have noticed already, there is a problem with IDDFS. That's because I used the recursive way of doing it, therefore I didn't know how to track the number of nodes expanded, BUT the search is working properly as you have seen above after debugging.

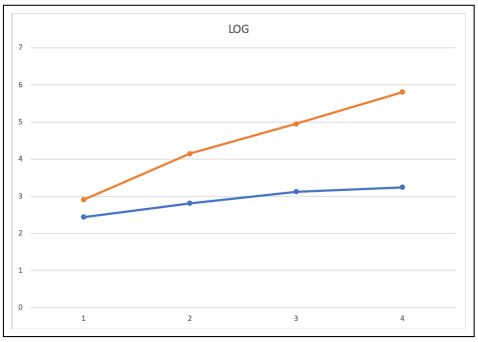




BFS uses a huge amount of memory, thus for really high inputs like 6x6 grids, with tricky paths between the source and destination, it may run out of heap memory. This occurred for 5x5 grid as well, but I have allocated 10Gb of RAM for the JVM and BFS managed to find a solution with those resources.
Although DFS beats A-star in the long run, it won't return the shortest path, which should usually be the key feature of a searching algorithm. However, DFS' speed could be used for its own purpose, i.e. finding whether there exists a path between player and target or not.

Bear in mind that I have tried to generalize the search and used Euclidean distance heuristic for ***A**** which works for 8 directions of movement. A recommended heuristic for ***A**** would have been Manhattan, which doesn't underestimate the distances, keeping g and h balanced. This is where I can bring my second heuristic (see in additional work) for ***A**** and prove that in the long run DFS is crushed!





[7] or a move from the path array in the code above.
[8] which represents a node in the tree.

3 Additional work and limitations

3.1 Another heuristic for A^*

I tried to optimise the heuristic for A^* and that's not by using the Manhattan distance, but using Euclidean distance with a subtle change in the formula.
My first thought to optimise the heuristic was to abstract out the square root function from the normal Euclidean distance heuristic.
Euclidean squared distance heuristic

```
function heuristic(node):
    dx = abs(node.x - goal.x)
    dy = abs(node.y - goal.y)
    return dx*dx + dy*dy
```

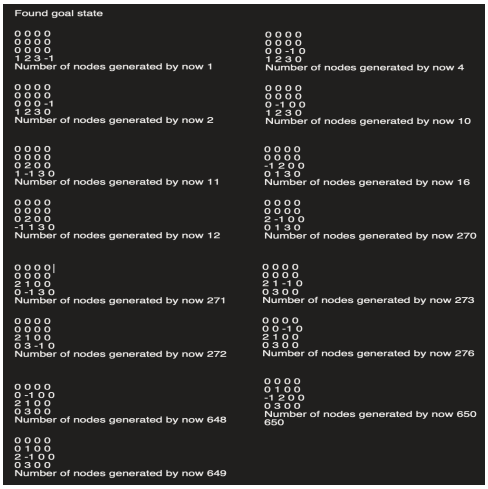
WHY?

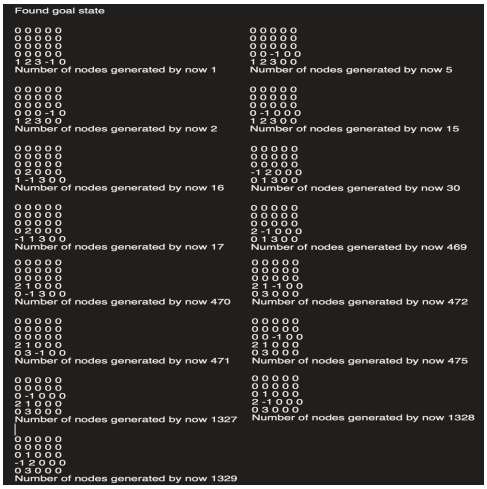
Because it makes the overall process slower proportionally to the states we have to check.

Even though the idea is not bad, this definitely runs into scaling problems. The scale of g and h need to match, because I am adding them together to form f. When A^* computes $f + h = g$, the squared distance (note we got rid of the square root) will be much higher than the cost g ending up with an overestimating heuristic. For longer distances this will approach the extreme of g not contributing to f and A^* will degrade into Greedy Best-First-Search.

However, in our case it works well since we don't have any unwalkable areas which may cause the algorithm to get "greedy".

To appreciate the improvement of the second heuristic I ran it on a 6x6 grid and expanded only 1744 nodes; I have provided pictures below following the step-by-step progression of the algorithm. This is a real WOW, the first one couldn't even handle a 6x6 grid.





3.2 Visualization game

I tried to get creative and made a little game in Unity3D visualizing how ***A**** works. In this case I made another optimisation, i.e. using a min heap for picking the least f costs. It turned out to be much more effective than using a normal list. If measured in CPU run-time I got from 300ms to 4ms for a 100x100units map where I considered the radius of a node to be 0.25 units in area.

```
List<Node> openSet = new List<Node>();
...
Node currentNode = openSet[0];
// find node with lowest f-cost; costly part of the algorithm.
for (int x = 1; x < openSet.Count; x++)
{
    if (openSet[x].fCost < currentNode.fCost || openSet[x].fCost == currentNode.fCost)
    {
        if (openSet[x].hCost < currentNode.hCost)
        {
            currentNode = openSet[x];
        }
    }
}
```

The code above could be easily replaced with 2-3 lines of code using a min heap as follows; full code of heap provided at the end of report.

```
Heap<Node> openSet = new Heap<Node>(grid.MaxHeapSize);
...
Node currentNode = openSet.RemoveFirst();
```

The *RemoveFirst* method takes care of removing the optimal candidate for our closed list using the following overridden method of *IComparable*.

```
@Override
public int CompareTo(Node other)
{
    int compare = fCost.CompareTo(other.fCost);
    if ( compare == 0)
    {
        compare = hCost.CompareTo(other.hCost);
    }

    // This return 1 if the value is higher, but we need the lowest value.
    return -compare;
}
```

Also, I have added unwalkable areas to see how A* performs in trying to avoid them. If the target is unreachable, i.e. if it's bounded by an obstacle the method stops updating.