Ant Nest Building with AI integrated

COMP4200 – Artificial Intelligence

Phong Vo
University of Massachusetts – Lowell
Department of Computer Science
Phong_Vo@student.uml.edu

Giang Tran
University of Massachusetts – Lowell
Department of Computer Science
Giang_Tran2@student.uml.edu

I. Abstract:

Our project uses A* algorithm to make a game called Ant's Nest Building. The project is initialized with our interest in exploring the way an automatic vacuum cleaner works by going around to collect the trash and put it in the basket. We use the algorithm to help the ant find the shortest path to avoid all the bricks, collect all the leaves in the map, then bring them back to the nest. We do this project in order to build an entertaining game as well as a system that solves a real problem based on the heuristic search strategy.

II. Introduction:

In computer science, A* is a search algorithm in graphs. This algorithm finds a path from the originating node to a given destination node (or to a node that satisfies the target condition). This algorithm uses a heuristic rating to rank each node according to an estimate of the best route through that node. This algorithm browses nodes in the order of this heuristic evaluation. Therefore, the algorithm A* is an example of a best-first search. [2]

Consider the problem of finding the way - the problem that A* is often used to solve. A* builds ascending all the routes from the starting point until it finds a path that reaches the destination. However, just as all search algorithms have information, it only builds routes that appear to be gradually reaching their destination. [3]

To determine which routes are likely to lead to the destination, A* uses a heuristic evaluation function of the distance from any point to the destination. In the case of finding a route, this rating can be the distance the crow flies - a rough rating that is used for the distance of a road. [2]

The difference of A* from the best option search is that it also takes into account the distance traveled. That makes A* complete and optimal, meaning that A* will always find the shortest path if such a path exists. A* does not guarantee to run faster than simpler search algorithms. In a labyrinth-like environment, the only way to get there is to first go to the far end and finally return. In that case, trying the buttons in the order "closer to the destination, first tried" can be time-consuming. [2]

III. Literature Review:

We tried to apply the A* algorithm to the agent (an ant) so that it can find the shortest ways to go to collect the leaves and avoid the bricks. The agent needs to know how many ways it could reach the target (a leaf) without obstacles (the bricks) and choose the shortest path from those. When the agent reaches the current target, it will try to carry out the algorithm again with the new target (the nest). The targets will be switched between the leaves and the nest whenever the agent reaches one of them. Vo (*author*) built up the complete map by using the matrix and make the animation for the game. Tran (*author*) developed the utilization of the algorithm to make the agent move:

Suppose n is an attainable state (there is a path from the initial state 0 to n). We define the evaluation function: $f(n) = g(n) + h(n)$

g (n) is the cost from the original node n0 to the current node n
h (n) estimated cost from current node n to destination
f (n) the estimated total cost of the path through the current node n to the destination
A heuristic estimate of h (n) is considered to be acceptable if for every node n:

$0 < h(n) < h^*(n)$

Where h * (n) is the actual (actual) cost to go from node n to destination.

After that, both of us have tried to make the agent recognize the targets and switch among those when being reached.

IV. Methodology:

Approach: [1]

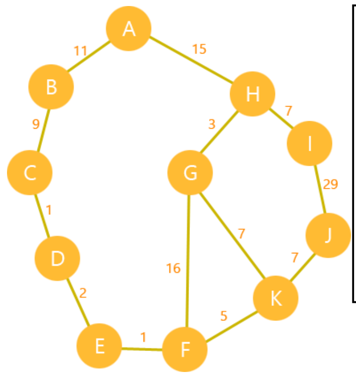
- Open: a set of states that have been born but not taken into account.
- Close: set of states that are considered.

Cost (p, q): is the distance between p, q.
g (p): distance from the initial state to current state p.
h (p): value evaluated from the current state to the target state.
$f(p) = g(p) + h(p)$.

```
Step 1:
        Open: = {s}
        Close: = {};
Step 2: while (Open! = {})

        2.1 Choose the best state (top) p in Open (remove p from Open).

        2.2 If p is the end state then exit.

        2.3 Switch p to Close and create successive states q after p

            a) If q is already in Open
                If (g (q)> g (p) + cost (p, q))
                g (q) = g (p) + cost (p, q)
                f (q) = g (q) + h (q);
                prev (q) = p          // vertex of q is p
            b) If q is not already in Open
                g (q) = g (p) + cost (p, q);
                f (q) = g (q) + h (q);
                prev (q) = p;
                Add q to Open.
            c) If q is in Close
                If (g (q)> g (p) + cost (p, q))
                Remove q from Close;
                Add q to Open.
```

<u>Step 3:</u> Can't find it.

<u>Graph:</u>



h(A) = 60	h(B) = 53	h(C) = 36
h(D) = 35	h(E) = 35	h(F) = 19
h(G) = 16	h(H) = 38	h(I) = 23
h(J) = 0	h(K) = 7	

Peak begins A.
The ending ends K.

Fig. 4.1: Hypothesis of the graph search algorithm

Estimate the distance from the current vertex to the ending vertex f (x) = g (x) + h (x) where g is the shortest distance from the current vertex to the destination.

Example f (A) = 0 + 60.

Step	P	The vertices connected to P	Open	Close
0			A60	
1	A	B, H	B64, H53	A
2	H	G, I, A	B64, G34, I45	A, H
3	G	H, K, F	B64, I45, K32, F53	A, H, G
4	K	G, F, J	B64, J32, F49, I45	A, H, G
5	K (stop)			

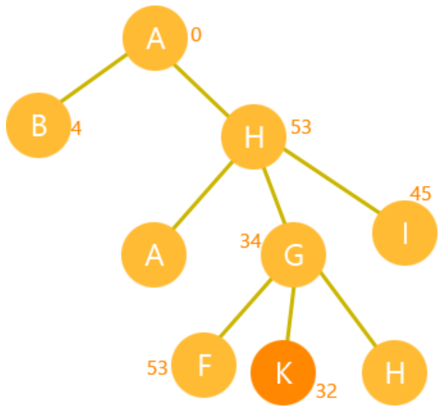


Fig 4.2: Search tree corresponding to the above graph.

Set up:

A map will be created based on user's choice (width and height can be customized) with customizable numbers of leaves and bricks. The agent will be set to the ant, the obstacles will be set to the bricks, the targets will be set to the leaves at the beginning and can be switched to the nest in between. The main parameters should be the locations X and Y (coordinates of x-y axis) of the agent, number of targets, number of obstacles, the list of nodes that can be reached, list of moves that can be achieved, maximum width and height of the map. [3]

Hypothesis: the agent will collect all the leaves (one per turn) and bring them back to the nest. Due to the location of the items in the map are randomly generated, sometimes the agent will be covered around by the bricks and cannot find the way out. Therefore, the game is unresolvable with the announcement "Can't find path!". The program will be returned to stop then.

V. Result:

Game' conception:

The project is developed with A* algorithm which is optimized for pathfinding in order to gain the reaching out of the leaves. There is an ant going to build its nest, literally marked by the UML logo, which is located at the top-left corner of the field. The purpose of the program is to optimize the path of the ant to be least walks as it can. The material for building up the nest are initialized with ten (10) leaves which are laying around the map. Every leaf might be blocked by ten (10) bricks. Hence, the ant must wisely reach out to the leaves without hitting the walls. Once it gets a leaf, it brings it back to the nest and then continues heading to the other remaining leaves. The loop of collecting the leaves is finished once all of the leaves are gathered in the nest. The size of the field can be changed rather than 10x10. The number of leaves in the field can be also changed.

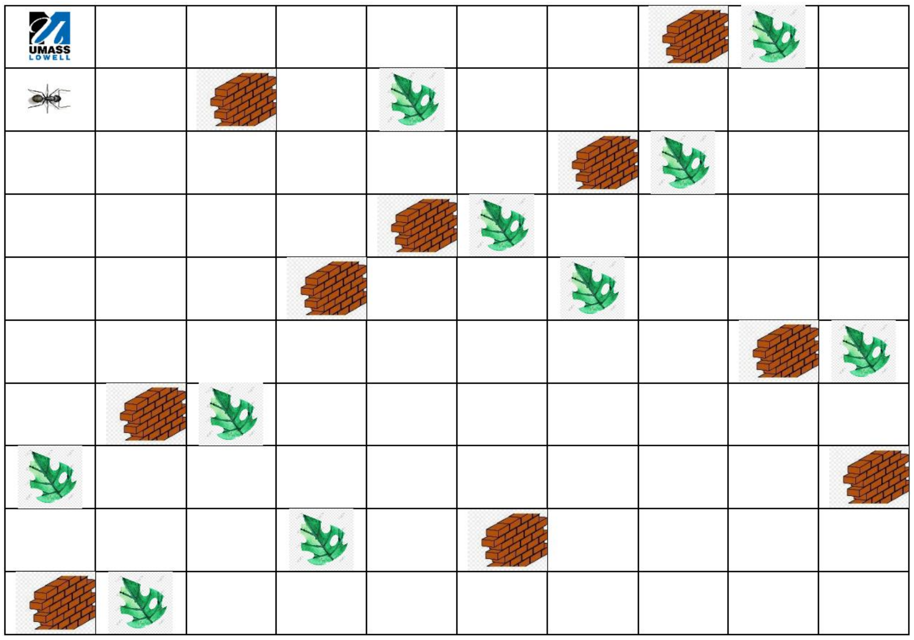


Fig. 5.1: The initial state of the program

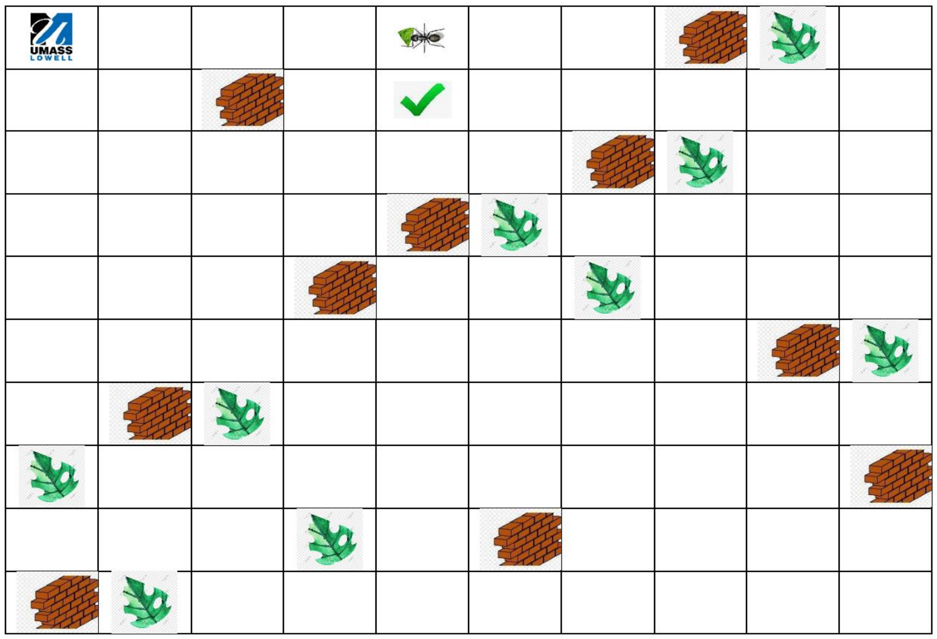


Fig. 5.2: The ant has successfully collected a leave, then bring back to the nest.

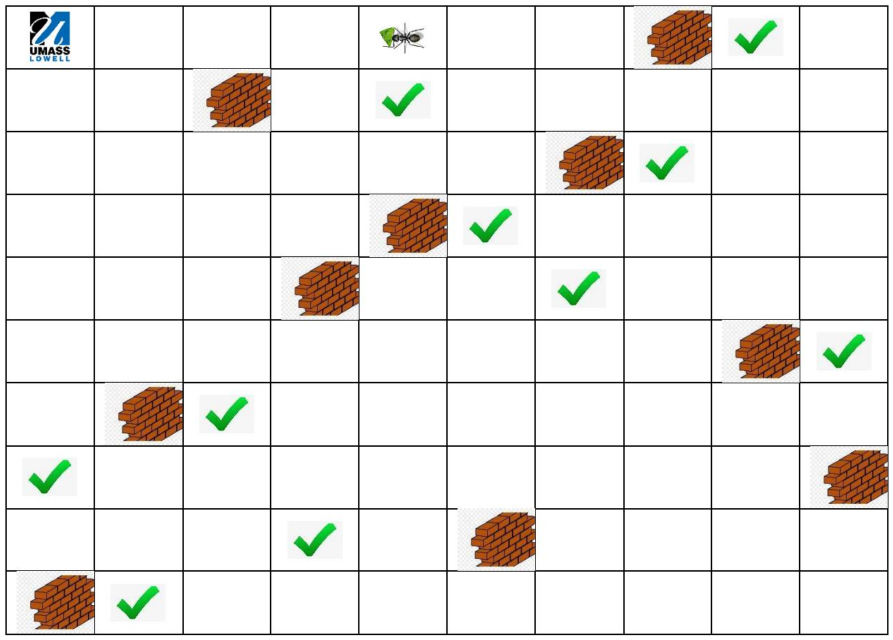


Fig. 5.3: The ant has successfully collected all of the leaves.

The game would finish when the ant collects all the leaves on the map (final state reached) => successful. The path that the ant goes to collect the leaves would be the shortest path (takes the least cost with both heuristic and expected costs) to reach the final state.

VI. Discussions:

The optimization of A* algorithm is the best fit for the small-sized system. If there is no need for repeating back-checking the past nodes, then A* algorithm is the best choice. If the map is big enough in the quantity of nodes, then IDA* can do better. A* algorithm can perform with its best solution but it would not be optimized. [2]

Our group had been offered for using IDA* for this project. Unfortunately, after a comparison between A* and IDA*, we decided to apply A* because our project is small-sized and optimization is preceded. As the thesis implementation has mention above, in all of the perfect cases, the agent can collect all of the leaves without hitting any brick. But in some unexpectedly weak cases, the agent cannot go toward the first leaf just because it is covered around by bricks whose locations were randomly generated by the function.

VII. Conclusion:

Although our project has finished, we need to improve the approach of the problem by using IDA* in the case of the field expands to a bigger size with many more nodes allocated. The reason was IDA* algorithm is the better way rather than A* in memory usage, complexity and time. After IDA* would have used, we can compare and clearly confirm the *pros* and *cons* of each algorithm.

REFERENCES

[1] Russell, Stuart J., et al. Artificial Intelligence: A Modern Approach. Prentice Hall, Boston, 2010.

[2] Lester, Patrick (2005): A* Path-finding for beginners.

[3] Sharma, Shrawan and Pal, B.L. (2015): Shortest Path Searching for Road Network using A* Algorithm.