A* Pathfinding

NATHANIEL WATERWORTH U1763480,
Dept. of Computer Science, University of Huddersfield

1. INTRODUCTION

2. METHODS

3. RESULTS AND EVALUATION

4. DISCUSSION

5. FUTURE WORK

REFERENCES

List all relevant references and sources here using either APA, Harvard or IEEE style.

[1] GROWING WITH THE WEB. (2012). *A* PATHFINDING ALGORITHM*. RETRIEVED FROM HTTPS://WWW.GROWINGWITHTHEWEB.COM/.

[2] CUI, X.C. & SHI, H.S. (2011). A*-BASED PATHFINDING IN MODERN COMPUTER GAMES . *IJCSNS INTERNATIONAL JOURNAL OF COMPUTER SCIENCE AND NETWORK SECURITY, 11* (1), 125-130. RETRIEVED FROM HTTP://PAPER.IJCSNS.ORG/.

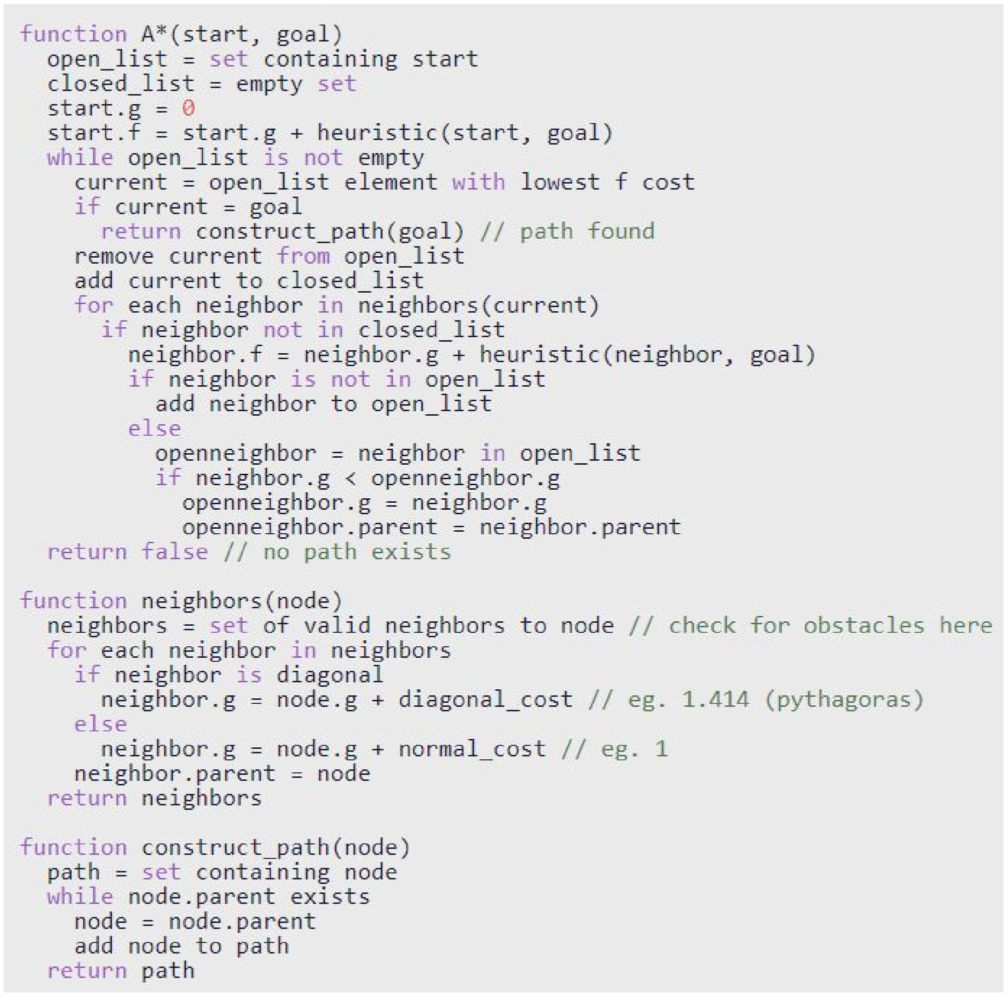
[3] SEBASTIAN LAGUE [SEBASTIAN LAGUE]. (2014, DEC 24). *A* PATHFINDING (E04: HEAP OPTIMIZATION)* [VIDEO FILE]. RETRIEVED FROM HTTPS://WWW.YOUTUBE.COM/WATCH?V=3DW5D7PLCTM&LIST=PLFT_AVWSXL0CQ5UMV3PMC9SPNKJFP9EGW&INDEX=4

[4] CANSIAN, M.C. (2017). *A* WITH NAVIGATION MESHES*. RETRIEVED FROM HTTPS://MEDIUM.COM/.

INTRODUCTION

Pathfinding is a well researched field as it has many real world and game based applications. There are many approaches to a solution to pathfinding using various algorithms such as Dijkstra's and Bread/Depth First. Though A* has been identified as the possible optimal solution for pathfinding. This report will illustrate the strengths and weaknesses of the approach I have taken to pathfinding with A* and possible A* optimisations in future work.

METHODS



```
function A*(start, goal)
  open_list = set containing start
  closed_list = empty set
  start.g = 0
  start.f = start.g + heuristic(start, goal)
  while open_list is not empty
    current = open_list element with lowest f cost
    if current = goal
      return construct_path(goal) // path found
    remove current from open_list
    add current to closed_list
    for each neighbor in neighbors(current)
      if neighbor not in closed_list
        neighbor.f = neighbor.g + heuristic(neighbor, goal)
        if neighbor is not in open_list
          add neighbor to open_list
        else
          openneighbor = neighbor in open_list
          if neighbor.g < openneighbor.g
            openneighbor.g = neighbor.g
            openneighbor.parent = neighbor.parent
  return false // no path exists

function neighbors(node)
  neighbors = set of valid neighbors to node // check for obstacles here
  for each neighbor in neighbors
    if neighbor is diagonal
      neighbor.g = node.g + diagonal_cost // eg. 1.414 (pythagoras)
    else
      neighbor.g = node.g + normal_cost // eg. 1
    neighbor.parent = node
  return neighbors

function construct_path(node)
  path = set containing node
  while node.parent exists
    node = node.parent
    add node to path
  return path
```

Fig. 1 Pseudocode for A* Algorithm [1]

1. Converted pseudocode [1] (Fig.1) of an A* algorithm
2. Used Manhattan Heuristic to calculate travel costs.
3. Costs: 10 - vertical and horizontal moves. 14 - diagonal moves.
4. Created grid class for the algorithm to operate on.
5. Created a pawn class to use the found path and navigate the grid.
6. Created a boundary class to instantiate obstacles to maneuver around.
7. Applied weights to certain tiles to represent slower paths due to terrain which is common in both real world and game applications.

I developed two types of pawn, a ground and air unit. The ground unit is affected by short/tall walls and muddy terrain (slows pawn movement by factor of 4). Though the air unit is only affected by tall walls. In a randomly generated grid, it composed in the following ratios: 50% open space, 13.3% muddy terrain, 13.3% low walls, 13.3% tall walls. This allows ground unit to have traversability of 66.6% of the grid whilst affected by terrain penalties. Whilst the air unit has traversability of 83.33% of the grid and unaffected by movement penalties.

I created a typical game environment to demonstrate the navigation of the pawns.

Controls for application:

1. Click and hold left/right/middle mouse button - rotate around pivot point
2. Space - cycle through observation points (pawns)
3. Enter - Set pawn walking to mouse destination taking the shortest route (if there is a path to that point)

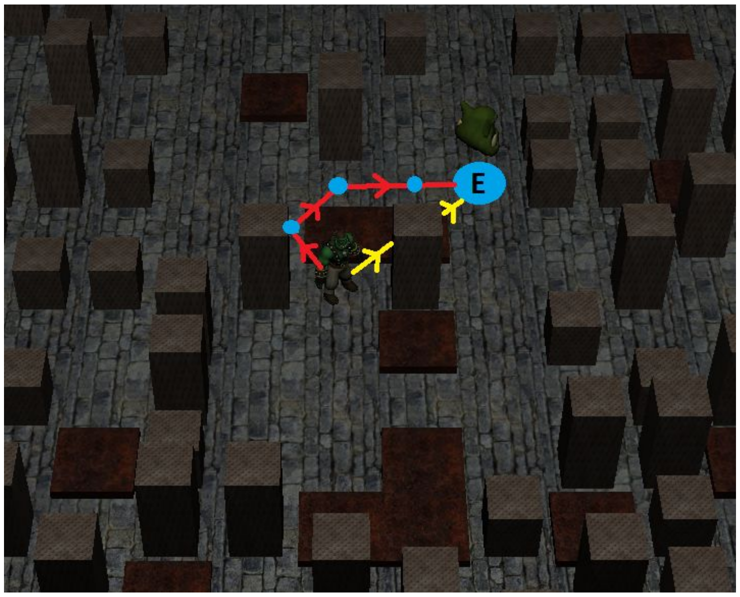


Fig. 2 Air and Ground Unit Path Representation.

Pawns show different behaviors due to the effect of terrain:

1. Red - shortest path for the ground unit.Cost: 48 (14+14+10+10).
2. Yellow - shortest path for air unit. Cost: 28 (14 + 14).

If red attempted to take the yellow path a penalty would increase the travel cost on the mud tiles by a factor of 4 resulting in cost: 70 ((4*14) + 14).

RESULTS AND EVALUATION

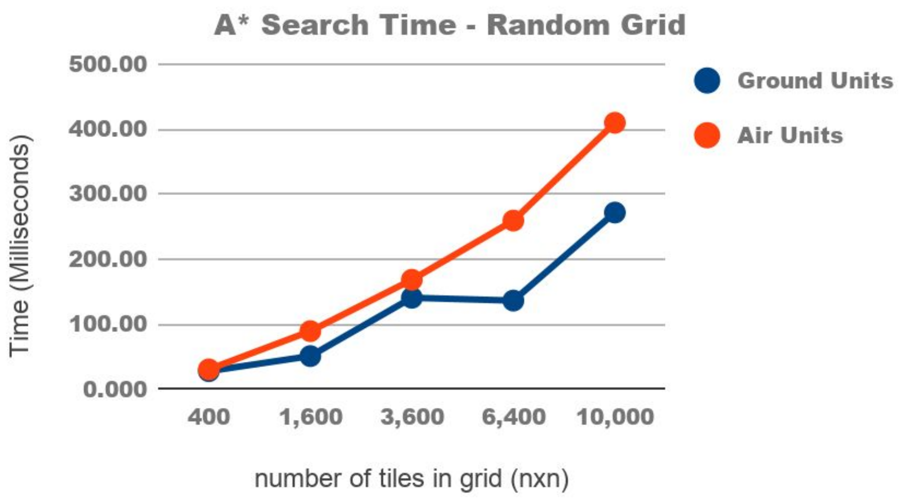


Fig. 3 Search Time comparison between different unit types on a random grid.

This graph demonstrates the average time to complete a search in finding the shortest path from one corner to the opposite corner in a grid populated with obstacles. It shows how the generally the time taken for a ground unit to find the destination is less despite more obstacles, primarily because there are less possible nodes to check so an optimal route becomes more apparent. However, there are odd cases where air units triumph in search time which can be identified as the ground units being led down promising routes that amount to nothing resulting in backtracking to a previous point.

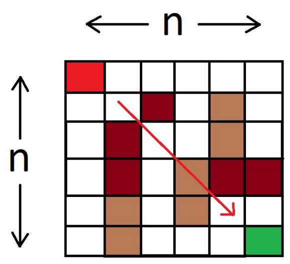


Fig. 4 A generic random grid representation

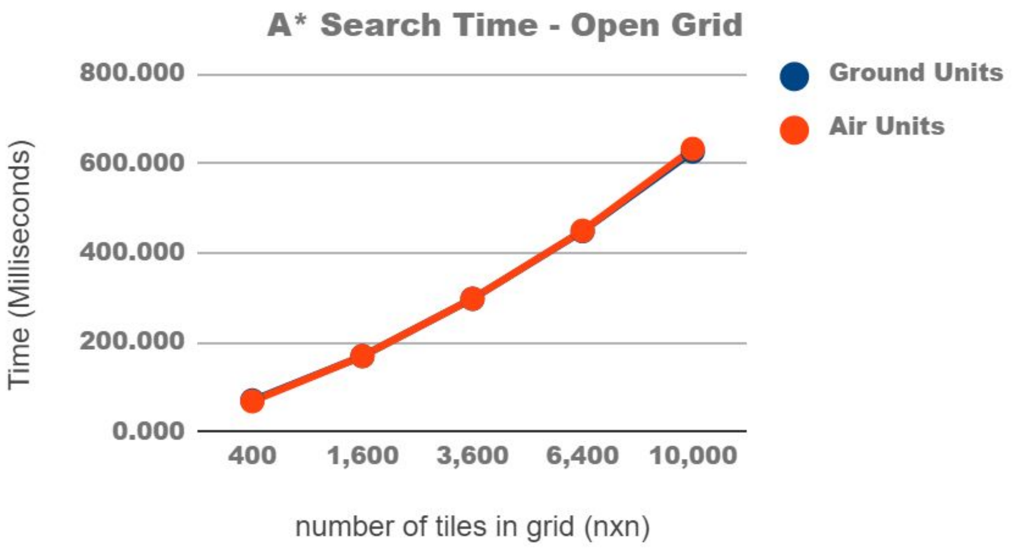


Fig. 5 Search Time comparison between different unit types on an empty grid

When analysing the time taken for both units to traverse an empty grid, as expected they have equal search times as they have an equal number of points to cover.

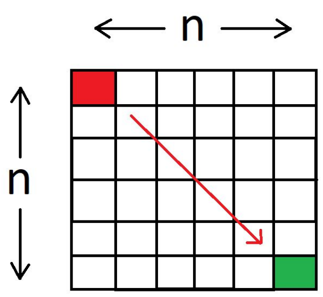


Fig. 6 A generic random grid representation

When comparing Fig.3 and Fig.5 it emphasises the effectiveness of my A* implementation in a clustered environment, but also highlights its weakness in an open environment as the number of traversable points increases.

DISCUSSION

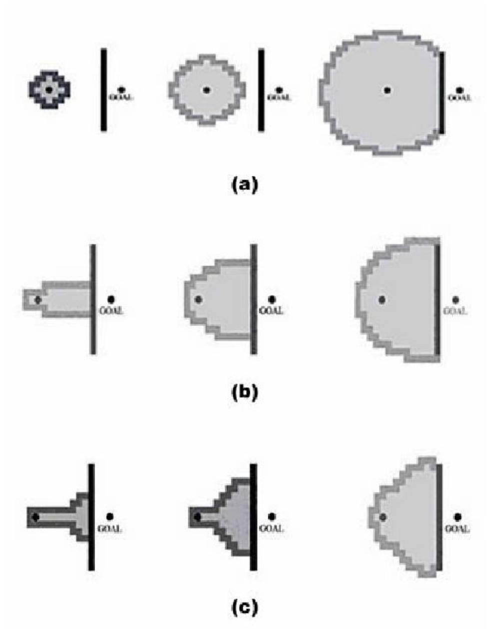


Fig. 7 Heuristics Comparison [2]

I approached using A* as it has a heuristic which allows it give priority to nodes that are suppose to be a option. Whereas Dijkstra's attempts to explore all possible paths which could be more expensive. Demonstrated in fig.7 are the nodes searched using different heuristics, where (a) the first set represents behaviour shown by Dijkstra's in a grid based graph.

Furthermore I chose to use 8 directional travel, despite having a greater search time due to the increased number of possible paths due to the search producing a more optimal and representable path. As shown below an 8-directional search produces a more direct path with a lower overall cost, whilst 4-directional search produces a less direct path with a greater cost.

Although number of directions of travel from a node could ultimately be dependant on the application of the search. For instance, some games may only require 4-directional movement.

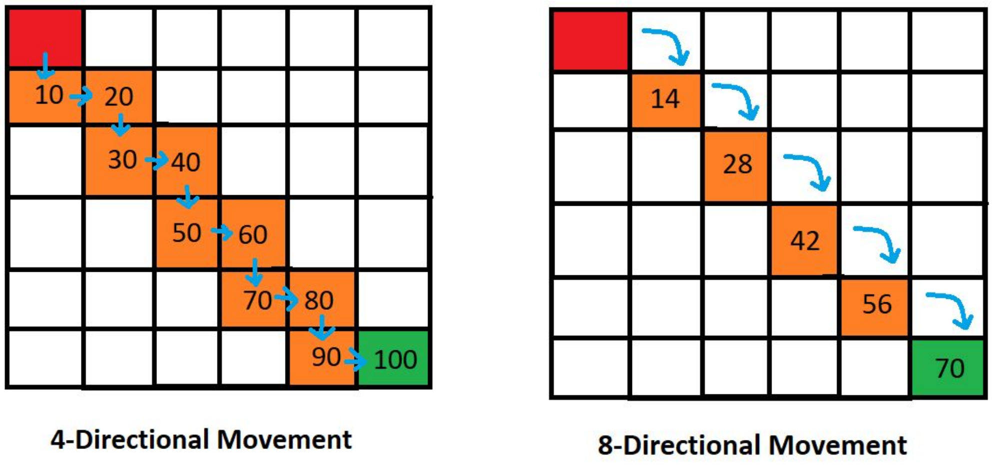


Fig. 8 Comparison of of multi-directional travel

FUTURE WORK

There is great potential for an improvement on overall search time. For instance, I could convert the open list into an array that can apply a heap sort [3] on the array to maintain order. The average search time will then be of order O(log(n)), whereas it currently is O(nlog(n)) where n is the size of the list.

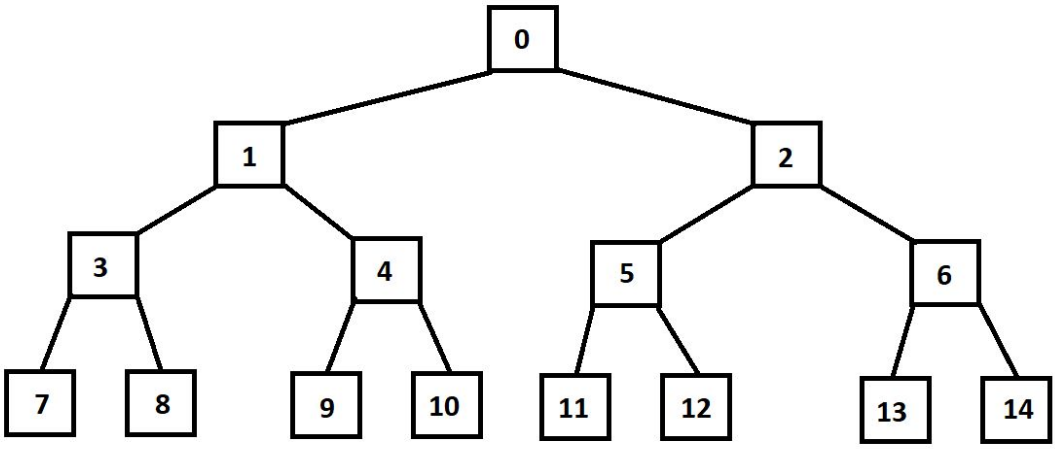


Fig. 9 Binary Tree

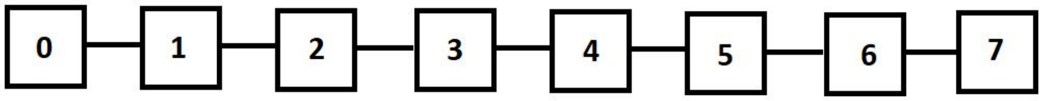


Fig. 10 List

When there is order in the tree such that each parent represents a smaller f cost than both its children. It then only requires one comparison between each tier when removing and replacing the node at index 0 since all nodes can be directly accessed. Whilst to access a node at index n in a list, all previous nodes must first be accessed.

Finally, in the scenarios which had an empty grid of size n^2. The search time was large considering that a pawn could travel from any point in the grid to another point in one straight line move.

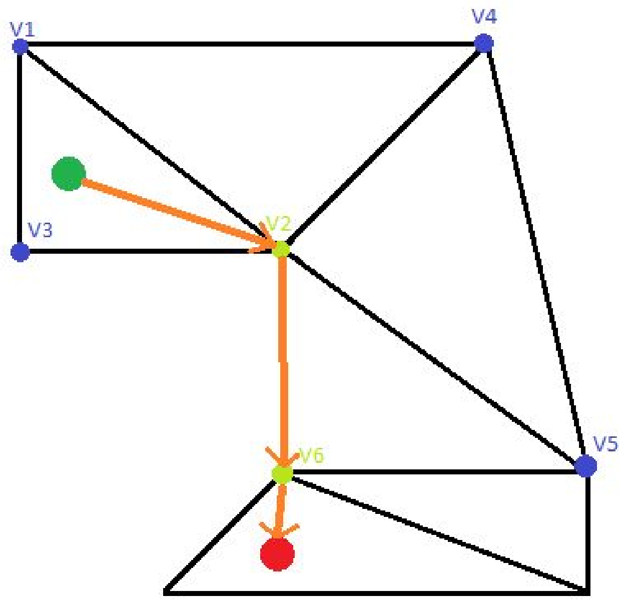


Fig.11 A* Pathfinding with Navmesh [4]

If the traversable area was broken down into triangles with vertices as nodes. Then we could execute a few checks to cover a much larger region. The logic could be broken down into the following:

1. Check if both points lie within a triangle, if so the point may be reachable.
2. Check if the point lies within the same triangle, if so it could calculate distance between points and at direction and distance to path.
3. Check if endpoint is in a different triangle, if so proceed with A* path calculation.

This calculation would be optimal if the triangles be calculated based on an existing non dynamic environment on startup. Despite each check being more costly in the navmesh, in an empty grid of size 100 x 100 there are 9,999 points to consider whilst in the same grid a navmesh would have 4 points to consider.