Hierarchical Pathfinding A* In The Context of The Sims

Thy Nguyen, Daniel Chechelnitsky

12/17/2019

Contents

1 Introduction to Pathfinding AI and Motivation

1.1 Pathfinding in The Sims and Other (Arguably More Useful) Real-Life Applications

In the classic game franchise The Sims, Sims are supposedly more nuanced and believable after each iteration as the developers account for more aspects of human behavior in their artificial intelligence system. However, at the core of this system is how a Sim move from one point, or object or person, to another, and getting this to at least resemble real-life behavior is the first step in making it look believable.

Beyond the Sims, pathfinding is an essential component not just in video games but other fields such as transportation and robotics. Since agents in most systems move in a goal-directed manner, the program must be able to identify a path that avoids obstacles and is efficient. At the same time, the problem is twofold in that the path should not only be the most efficient in practice and also in terms of computing resources.

1.2 Graph Theory in Disguise

Although pathfinding can appear complicated, it is essentially a weighted graph search/traversal problem. Research on pathfinding is heavily based on Dijkstra's algorithm for finding the shortest path, and most pathfinding algorithms operate on principles similar to Dijkstra's.

1.3 Motivation

With a passion for algorithms, algorithms involving graphs, and video games, we decided to explore the pathfinding AI system in the Sims series. This research provides a comprehensive overview of how agents (Sims) move in their terrain and introduces the core algorithm – HPA*. It then simplifies the problem and

presents our implementation of HPA* as mapping rooms and entrances to nodes on a weighted graph and running A* on said graph to find the shortest path between two points.

2 Principles of HPA* and A*

Although there are several ways to implement pathfinding in games, A* and HPA* are the most popular and are used in the Sims. In this section, we provide an overview of these two algorithms.

2.1 A*

2.1.1 The Heuristic

The A* algorithm is one of the most popular choices for pathfinding in computer science due to its efficiency, optimality, and completeness. It is also fairly flexible and can be used in a wide range of contexts. A* is essentially an informed version of Dijkstra's as it knows in advance the target destination, and it factors in this knowledge with a heuristic function, the approximate cost from the current node to the goal. Each node x is assigned an f score, where $f(x) = g(x) + h(x)$, with $g(x)$ being the distance from the current node to the starting node, and $h(x)$ the heuristic distance from the current node to the destination. A* only expands on a node if its f score is the current smallest, therefore reducing the amount of nodes that are considered in regular Dijkstra's (Swift, 2017).

2.1.2 Pseudo-Pseudocode

The most basic implementation of A* given the start and end nodes is as follows:

- Place the start node on the Open list. This list keeps track of possible nodes to explore.
- While Open is not empty:
  - Pop the node off Open that has the lowest f score and put it in the Closed list of explored nodes.
  - If the node is the destination, terminate the program and return the path.
  - For each of its reachable neighbors: compute its f, g, and h based on the current node.
    - If it is not in Open, make the current node its parent, record its scores, and put it in Open.
    - If it is in Open, check to see if this path to it is better by comparing its current f score and the computed f score. If it is, change its parent to be the current node and update its scores.

2.2 Hierarchical Pathfinding A*

As the most basic implementation of A* is on a grid, merely overlaying a grid on top of the terrain and treating each square as a node would require a lot of computational power as the size of the terrain increases. Most games therefore adapt A* into HPA* by preprocessing the terrain into a weighted graph to reduce the number of nodes to explore (Le, Amandeep, & Narendra, 2008).

2.2.1 Pseudo-Pseudocode

1. Preprocessing a Terrain into a Weighted Graph

   1.1. Transform terrain into a square grid, with each cell in the grid being the same size.

   1.2. Determine the accessibility of each cell and only consider the cells that are accessible to the agent. Essentially, these are the cells that do not contain any obstacles.

   1.3. Divide the grid into subgrids, and find the entrances between the subgrids. In the Sims, a subgrid is considered a room, and an entrance a door.

   1.4. Use nodes and edges to assemble a connected graph. A node is an entrance, and an edge is constructed by connecting two nodes within the same subgrid. Since a node is right along the border between two subgrids, it is therefore in both of them.

2. Running the Pathfinder

   2.1. Add start node S and goal node G to the connected graph and run A*.

   2.2 Refine the path and apply smoothing as needed in order to get the most optimal path.

3 Implementation of HPA*

3.1 HPA* in The Sims

In the Sims, a plot is considered as a grid of same-sized squares. When a house is constructed, the program builds a room graph, with each room being a subgrid, and breaks down each room into smaller chunks, or sub-subgrids. The motivating idea is to build a path bottom-up in this hierarchical structure (Bourse, 2014).

3.2 HPA* by Dummies

3.2.1 Guiding Principles

We decided to implement a high-level version of HPA* in the Sims. In our code, we only have the plot of land as the grid and the rooms in the house the subgrids. Each subgrid contains nodes – the entrances to the room that it corresponds to. We add an edge between any two nodes, or entrances, in the same subgrid, with its weight being the Euclidean distance between them. Since an entrance leads from one room to another, its node representation is considered to be in both of those corresponding subgrids. To account for indoor and outdoor movements, we overlay a subgrid on top of the grid, and its nodes are the entrances that only belong to one subgrid. Since we do not account for obstacles in our implementation, the returned path is just the ordered list of nodes the agent should visit.

3.2.2 Object-Oriented Implementation in Java

3.2.2.1 Classes

3.2.2.1.1 Nodes, Entrances, and Positions

A `Node` has the following properties:

1. `(x, y)` coordinates in the grid.
2. f, g, h scores.
3. Its parent and goal `Nodes`.
4. Its Map of neighbors, with the neighboring node as the key and the distance between them the value.

Entrance and Position extend Node in the sense that they are also represented as nodes in the graph. However, an entrance represents a door, while a position represents a possible point of movement. The reason behind this distinction is that an entrance can only be along the border of two subgrids and can therefore be in both of them, while a position can be anywhere within a single subgrid.

3.2.2.1.2 Subgrid

A `Subgrid` is the representation of a room and keeps track of the following:

1. Its upper-left `(x1, y1)` and lower-right `(x2, y2)` coordinates.
2. A list of `Nodes` in the subgrid.

A `Subgrid` can add an `Entrance`, checking to see whether it is along its border, or a `Position`, checking whether it is within the subgrid. Anytime a node is added to it, it will automatically create the neighborly relationships between the new node and each of the existing nodes. A subgrid can also delete a node and thus disconnect it from all of its previous neighbors.

3.2.2.1.3 Grid

A `Grid` is the entire plot of land and has:

1. A list of indoors `Subgrids` (rooms in a house).
2. An outdoor `Subgrid` that is of equal size to the grid itself to account for moving inside and outside the house.
3. A list of `Nodes` anywhere in the grid.

The grid mediates all interactions between subgrids and nodes After adding subgrids to the grid, entrances are added to subgrids via the grid Based on an entrance's coordinates, the grid will find its corresponding subgrids and add it to them. Similarly, when a position is added, the grid will determine which subgrid it is in.

3.2.2.2 The Abstract Graph

The reason why the grid handles interactions between subgrids and nodes is that the program needs to find the subgrid(s) corresponding to a node in order to create neighbors for a node, and two nodes can only be neighbors if they are in the same subgrid. The weight of an edge is the Euclidean distance between the nodes it is joining. The resulting graph is thus weighted and undirected.

3.2.2.3 Writing A*

Following the pseudocode outlined above and a version of grid-based A* (SOURCE), we wrote our version of A* that runs a graph. The function takes in the start and end nodes, and assuming that their neighbors have been added, finds the nodes that would construct that path between them. Open is initialized as a priority queue, using f score to order the nodes. The heuristic h is computed as the Euclidean distance between the current node and the goal.

3.2.2.4 Running Everything

3.2.2.4.1 The Backend Version

For the backend version, it is helpful to draw out the grid, subgrids, and entrances and their coordinates before instantiating objects of the corresponding classes as the program does not handle collision nor obstacles. To run the program:

1. Create the `Grid`, supplying the upper-left and lower-right coordinates.
2. Create `Subgrid(s)`, again specifying the upper-left and lower-right coordinates of each and making sure that the subgrids do not intersect.
3. Add a subgrid `s` to the grid `g` by calling `g.addSubgrid(s)`.
4. Create `Entrance(s)` by providing the `(x, y)` coordinates.
5. Add an entrance `e` to `g` by calling `g.addEntrance(e)`.
6. Create any Position `p` and add it to `g` by calling `g.plopDownPosition(p)`.
7. To find the path from starting node `s` and destination node `d`, make sure they are added to the grid and call `AStar.findPath(s, d)`.

The class `HPA` has a main method to put code and there are examples provided inside the `AStarTest` class. Some minor notes to run the algorithm:

- Add `Subgrids`, `Entrances`, and `Positions` in that order and do not add any `Subgrids` after `Entrances/Positions` have been added. Create a new `Grid` and populate it if you want to create a new terrain.
- Make sure subgrids do not overlap or exceed the bounds of the grid – the program has not yet accounted for these behaviors.
- Can add however many positions before running A*, just make sure you specify which two nodes are the start and end.

3.2.2.4.2 The UI Version

Run this version by running the main method in the `Program` class. The steps to take are: 1. Build the subgrids by dragging from the upper-left to the lower-right corners of a subgrid to create one. Try not to overlap them as the program does not handle collision. 2. Add the entrances along the edges of subgrids. The program does have a checking mechanism for this (based on the backend implementation). 3. Add start and end positions anywhere in the grid. For simplicity's sake, you can only add one of each, though this is not a constraint in the backend version. 4. Build the graph if you want to visualize the abstract graph, but this step is not necessary. Run AStar by clicking on Find Path. The program will color all of the nodes in the path. 5. Clear the path before adding another start and end positions. This will remove the previous endpoints from the grid. 6. Clear the grid if you want to create a new grid with new subgrids and entrances as adding subgrids after entrances and positions have been added will cause unwanted behaviors.

4 Discussion

4.1 Conclusions

The project was an exciting attempt for us to gain a better understanding in graphs and video game development. Finding out that complex in-game behavior is built upon simple algorithms such as Dijkstra's or A* helps us realize that algorithms have infinitely many applications that we are not yet aware of. We also learned that it is much more efficient to separate the backend and GUI versions, because if our backend version runs smoothly then we will just have to map GUI components to what we already had. This reduces code coupling and makes it easier to debug.

4.2 Limitations of Implementation

Since we implemented the first step in finding the path and did not account for obstacles, the program returns the ordered list of nodes the agent should pass through and not the actual path the agent should take. Therefore, the program does not perform any path refinement or smoothing. Although it checks for valid placement of entrances and positions within a grid/subgrid, it does not verify that subgrids do not overlap, meaning that an entrance could be in more than two subgrids. As expected, this would lead to unpredictable and undesirable results when trying to find an optimal path.

Additionally, the Sims's implementation of HPA* deconstructs a room (subgrid) into sub-subgrids, and even smaller sub-sub-subgrids if necessary, it can only construct a room based on the provided coordinate inputs. The game optimizes this nested structure to find a path bottom-up, building the path from the smallest to the biggest structure, thus saving computing resources should an obstacle suddenly block the current path or the action is cancelled. Since our version can only create subgrids from the provided coordinate inputs and does not have active agents, it does not have this dynamic component of the algorithm.

5 References

Bourse, Y. (2014). Artificial Intelligence in The Sims series. Retrieved from https://team.inria.fr/imagine/files/2014/10/sims-slides.pdf.

Le, D. M., Amandeep, S. S., & Narendra S. C. (2008). Hierarchical Pathfinding and AI-Based Learning Approach in Strategy Game Design. International Journal of Computer Games Technology. https://doi.org/10.1155/2008/873913.

Swift, N. (2017). Easy A* (star) Pathfinding. Retrieved from https://medium.com/@nicholas.w.swift/easy-a-star-pathfinding-7e6689c7f7b2.