Games as a Context for AI *week 6*

Games AI should be quick (for real-time use), use predictable resources, and have understandable behaviour (giving the ability to design game play). They should be believable – not too stupid but not too clever. The player's experience is built on 'suspension of disbelief'. This, however, is not always the case with non-games AI.

Pathfinding or 'search' is a generic problem, not only applicable to games, e.g. robots, routing network packets, travel planning. It usually works on a node graph.

Breadth first is visiting each level, one by one. Depth first is visiting each child node to its maximum depth. The implementation is quite similar: depth first uses a stack to expand child nodes, while breadth first uses a queue. Both have the same big O complexity.

In games, breadth first is usually most suited as we often want to look a fixed number of moves ahead, however depth first uses less memory and is slightly easier to implement.

Dijkstra's Algorithm *week 7*

Key ideas with Dijkstra's: planning (it works out the complete route in advance), divide and conquer (decomposes the planning into an iterative process). It eliminates suboptimal search directions early in the search process. Dijkstra's explores all possible paths, including dead ends and works on general graphs (not just grids).

To perform Dijkstra's algorithm, we need a set of nodes (the map) and pair costs to traverse from one node to its neighbours. We obviously need to know the start and end nodes. For each node, we keep the total cost from the start node: the current shortest total distance from the start and a link to the neighbour that got us to that node with the lowest cost. The link could just be the neighbour's coordinates. We also need two lists of nodes, those that are open (not yet found their shortest route) and those that are closed (have found the shortest route).

Before starting the algorithm, each node's total costs should be set to either infinity or a very large number. The total cost of the start node should be zero. Links don't matter, but all nodes should be in the open list.

To find the path, do the following, iteratively:

1. Find the node, N, in the open list with the current lowest total cost.
2. Move that node to the closed list (take its current estimate as final).
3. Re-estimate the cost of each open neighbour.
4. If the new estimate is lower than the current total, update it and link to N.

When the end node is closed, we are finished. Follow the links from the end node back to the start node to get the shortest path.

The limitations of Dijkstra's are that it needs to work out the entire solution in advance, it doesn't adapt to changes in the environment, it doesn't adapt to changes in the target position, and it doesn't take anything else moving in the environment into account.

A* modifies Dijkstra's to search more likely paths first in a fairly simple way: adding a heuristic function (a guess as to how far each node is from the target) to the open node total costs when choosing the next node to close.

The new cost function is the cost from the start for a node (determined as in Dijkstra's) plus the heuristic function for that node (an estimate of the minimum cost to get to the target). The heuristic function can be defined in many ways.

A* and Further Optimisations *week 8*

The heuristic function should be admissible – it should not over-estimate the actual shortest distance to the target from the node, otherwise it isn't guaranteed to get the actual shortest path.

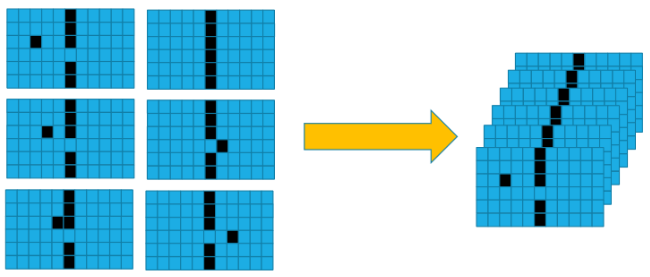
Manhattan distance: difference in x + difference in y.

Diagonal distance: (dx + dy) – 0.6 * smallest of (dx, dy), for a square grid

Euclidian distance: sqrt($(dx)^2 + (dy)^2$)

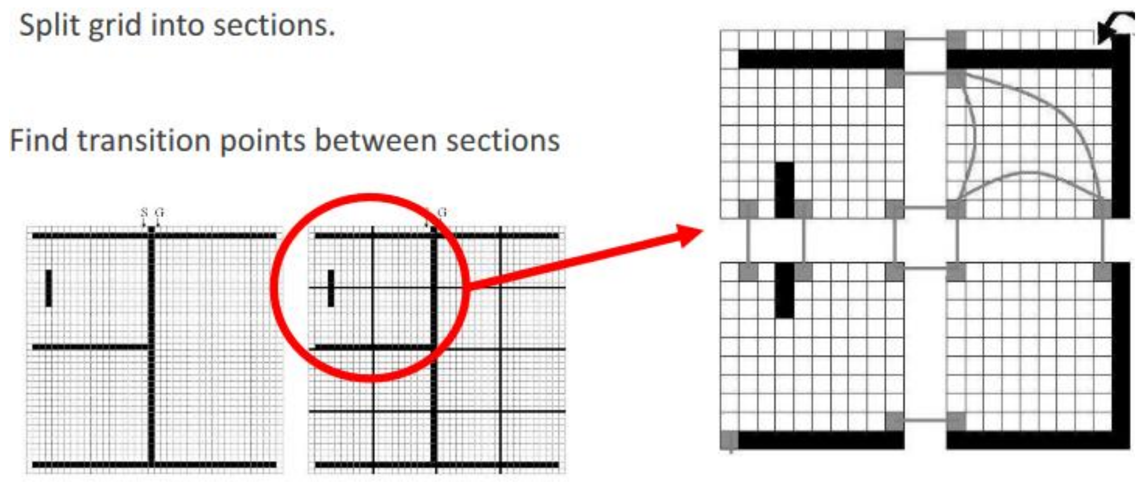
As A* is an optimisation of Dijkstra, it has the same limitations. If the environment changes, we may have to re-run the algorithm, but that could mean lots of recalculations. We can path patch, only rerunning the algorithm from where the environment change has happened. This can save some computational overhead, but it doesn't look as believable.

With two A* agents at the same time, we can convert a 2D map to a 3D map. Each layer of the 3D map includes the standard environment, with the first agent's position on a different layer per time-step. This then allows us to shift the second agent to a different layer after each step, meaning it sees the first agent's position as an obstacle and avoids it based on its position at that specific time. This works if the first agent runs its algorithm first, then a 3D map is created, then the second agent runs its algorithm.



The problems with a 3D approach are that optimal paths are not guaranteed, but this makes the agents somewhat believable. If the agents move at different speeds, this could be an issue. Paths generally get longer and can't be easily patched.

A* is still not very fast. We can pre-calculate routes on the map (but this doesn't respond to any environmental changes) or we can break the problem down hierarchically. This splits maps into sections, each with set inter-sectional routes. These routes can be pre-calculated, with only the final section needing a real-time calculation, as the target is within that section, rather than in a neighbour. As this is pre-calculated, it is quicker and can be patched more easily. It is also faster (10x in some cases) and nearly as optimal (99%).


Split grid into sections.
Find transition points between sections
S G
G

Another way of improving the search's speed is symmetry. Rectangular Symmetry Reduction is a simple algorithm to help optimise searches. It identifies rectangular blocks of clear nodes, leaving edge nodes in place. RSR then creates direct connections (bridges) between nodes on opposite sides of the clear rectangles and searches using the edges and bridges only.