Reachability problem

Giacomo Bergami

Newcaslte University

Objectives

- Understanding in which way A* extends Dijkstra's Algorithm.
- Understanding the pros and cons for each proposed algorithm.

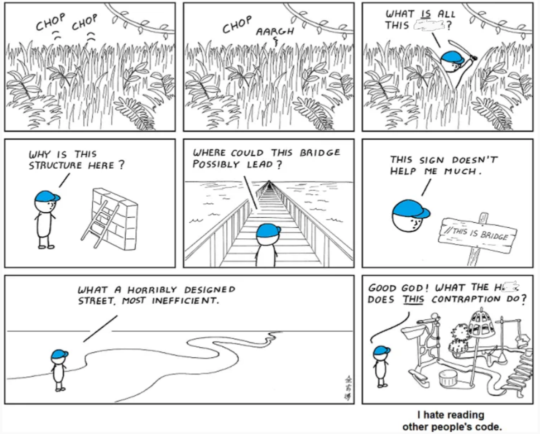
Reachability Problem

≪*Given a computational (potentially infinite state) system with a set of **allowed rules** or transformations, decide whether a certain state of a system is reachable from a given initial state of the system.*≫

1. **Set of states**: the (potentially infinite) states within a graph. We will consider such graph as a discretization of a 2D/3D scene into tiles.
2. **Allowed rules**: the set of the possible movements across tiles (e.g., avoiding obstacles and only considering adjacent tiles) via graph edges.
3. Rather than computing all the possible paths between two points, we will consider a **cost minimisation problem** ($\min\arg_{\texttt{src}\to\dots\to t\in\Pi}\delta(\texttt{src}\to\dots\to t)$) returning the path minimising such cost, where a cost of the path is the cost of all its associated edges: $g_{\texttt{src}}(t)=\delta(\pi)=\sum_{u\to v\in\texttt{src}\to\dots\to t}\delta(u,v)$. This requires associating a cost $\delta(u,v)$ to each edge $u\to v$.

The last lecture (Lecture №10) will consider a more general case, where each state is a resource of interest, and the allowed rules are the rules that allow the system to gain more resources. This will make no assumptions on the underlying data structure.

Navigating the Search Space


CHOP CHOP
CHOP AARGH
WHAT IS ALL THIS ?
WHY IS THIS STRUCTURE HERE ?
WHERE COULD THIS BRIDGE POSSIBLY LEAD ?
THIS SIGN DOESN'T HELP ME MUCH.
//THIS IS BRIDGE
WHAT A HORRIBLY DESIGNED STREET, MOST INEFFICIENT.
GOOD GOD! WHAT THE H DOES THIS CONTRAPTION DO?
I hate reading other people's code.

https://abstrusegoose.com/432

- State machines can be used require the machine to abide by some desired behaviour.
- To ensure correctness, they constraint the behaviour into a given protocol.
- On the other hand, this does not allow the machine to follow its own decisions given the navigation space.
- Use Case Scenario: **Scene Navigation**.
  1. If we had to encode this within a state machine, then we will need to hardcode all the possible paths!
  2. This cannot consider also if the path has to be re-computed at run-time given a moving obstacle.
  3. Then, we need to tell the machine how it can independently navigate the space given a *minimum* environment info.

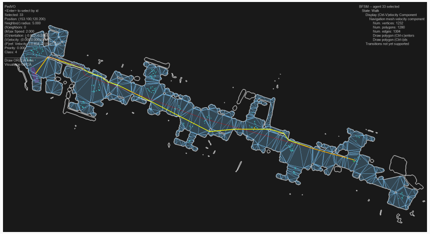
Scene Navigation, a.k.a. Pathfinding

Standard Grid in SimCity 3000

Hex Map in Civilization V

- We can represent the areas of the enviornment as a graph, where:
  1. nodes represent a *walkable tile* in the world, and
  2. edges represent walkable paths between tiles.
- By representing the scene as a graph, we are always forced to pursue a *greedy* approach, by considering for our navigation neighbouring nodes first.
- We need to design an algorithm to determine which node we can visit next given the one being currently visited, and to also allow a *backtracking* mechanism to re-calculate the path if, during the discovery phase, we found a less costly path.
- If we provide no additional assumption (e.g., *heuristic* or edge weight differentiation), all the neighbouring nodes will be treated as being the same $\rightarrow$ DFS search!

NavMeshing and suboptimal paths



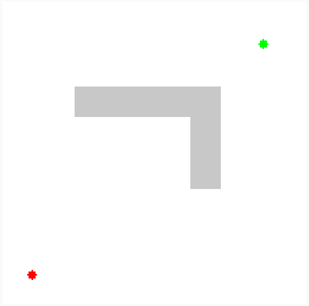
```
PedVO
<Enter> to select by id
Selected: 33
Position: (153.100,120.200)
Neighbo(r) radius: 5.000
(N)eighbors: 0
(M)ax Speed: 2.000
(O)rientation: (-0.8[illegible],-0.5[illegible])
(V)elocity: (0.000,0.000)([illegible])
(P)ref. Velocity: (-0.854,-0.5[illegible])(1.[illegible])
Priority: 0.000
Class: 4
Draw OR(C)A lines
Visualize ORCA
```

```
BFSM -- agent 33 selected
 State: Walk
   Display (Ctrl-V)elocity Component
     Navigation mesh velocity component
       Num. vertices: 1232
       Num. polygons: 1280
       Num. edges: 1304
       Draw polygon (Ctrl-c)enters
       Draw polygon (Ctrl-i)ds
   Transitions not yet supported
```

`https://github.com/MengeCrowdSim/Menge/issues/146`

- Complex 3D environments might use a navigation mesh to represent the moveable areas.
- Every node could be a convex polygon or a triangle, where (as always) you can traverse tiles sharing edges.
- Game arenas are often processed during development by dedicated tools that generate navigation meshes suitable for querying at runtime.
- This then poses the question: which position on the tile shall the agent occupy? Please observe that straightforward algorithms might generate suboptimal paths (*yellow*) significantly deviating from the expected outcome (*red*).

Dijkstra (1959)



1. Set `src` at distance zero from itself. All the other nodes, without a better guess, are at distance $+\infty$
2. Initialise Q with all the graph nodes sorted according to the distance to u (priority queue).
3. For all the elements in Q minimising the distance to u ($u := \min\arg_{u \in Q} g_{\texttt{src}}(u)$):
   3.1 Pop u from Q
   3.2 For all the neighbours v of u in Q:
      3.2.1 Let $\mathit{tmp}_{g(v)}$ be the temporary cost of arriving to v from `src` defined as the sum of the cost of arrivint to u from the source with $\delta(u, v)$ ($\mathit{tmp}_{g(v)} = g_{\texttt{src}}(u) + \delta(u, v)$).
      3.2.2 Retain u as v's direct ancestor and $\mathit{tmp}_{g(v)}$ as the final $g_{\texttt{src}}(v)$ (i.e., the cost for reaching v from `src`) if $\mathit{tmp}_{g(v)}$ is better that the previously computed value for $g_{\texttt{src}}(v)$.

Dijkstra (1959) – Discussion (1)

- The original algorithm was designed for computing the distance between a source node and all the remaining nodes on the graph.
- We can early-stop this computation as soon as we reach v as our target, and take its associated cost as the minimum cost for reaching v from the source.
- If we crawl backwards across all the parent states, we reconstruct the path associated to the minimum cost.
- Given Q as a list of nodes, we have a time complexity of $O(|E| + |V|^2)$. A better data structure (*Fibonacci heaps*) guarantees a $\Theta((|E| + |V|) \log |V|)$ time complexity (not in this implementation!).
- Uniform-cost search is a straightforward extension of this algorithm not requiring to possibly enumerate all nodes within the graph.
- Problem: given an open map with no obstacles, all edges will have an equal weight, and there is no preferred tile according to the position of the target on the scene.

Dijkstra (1959) – Algorithmic Correctness (Intuition)

A more rigorous proof requires a proof by contradiction+induction, which is not discussed here.

- When no other node is visited, we can only assume that the cost of reaching the source node is zero.
- The cost of traversing any neighbour u of the initial node matches the edge cost, thus $g_{\texttt{src}}(u) = \delta(\texttt{src}, u)$.
- In a graph, a node u might have multiple direct ancestors: as we ensure to iterate all the edges for all the nodes, we ensure to retain as its best ancestor the one being less distant to the overalls ource.
- Given distance additivity, this guarantees overall path optimality.

Dijkstra's limitations

1. The algorithm requires that all edges should have a zero or positive cost.
2. The algorithm assumes to have a finite graph (assumption for Line 2).
3. Prioritizing over the neighbouring nodes given their relative distance to the target:
   - Design a custom heuristic function $h_t(u) \approx \delta(u \to \cdots \to t)$ estimating the distance of each node u to the target t.
   - Thus, when deciding for u's best neighbour v, we will now estimate the total cost to reach destination by considering the edge traversal cost plus the heuristic, $\delta(u,v) + h_t(v)$, thus considering the best neighbouring node v to reach the target t via u:

$$f_{\texttt{src},t}(u) = g_{\texttt{src}}(u) + \delta(u,v) + h_t(v) \tag{1}$$

   - The best candidate will be the one minimising the score $\delta(u,v) + h_t(v)$, thus obtaining $f_{\texttt{src},t}(v) = g_{\texttt{src}}(v) + h_t(v)$.

Admissible and monotone *heuristics*

We say that an heuristic is admissible if it never overestimates the cost of reaching the goal, but it only provides a lower-bound.

- If the heuristics over-estimates the cost of reaching to destination, we might get a sub-optimal solution rather than the desired one.

We say that an heuristic is monotone if the difference between the heuristic values for each pair of connected nodes does not exceed the effective cost associated with the arc connecting them: $h_t(u) \leq \delta(u, v) + h_t(v)$.

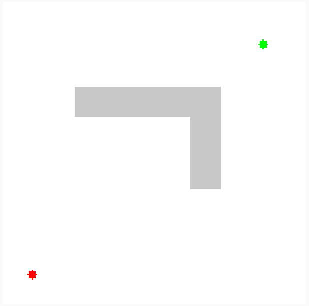
- Any monotone heuristic is also admissible.

We can show the last point by considering the straight-line distance between two points is a monotone heuristic that, beyond video-games, it is also used in calculating the optimal road route between cities on a map. It is:

admissible: no road between two points can be shorter than their straight-line distance.

monotone: Given triangular map $\triangle ABC$, the straight line distance $\overline{AC}$ is less that $\delta(A, B) + \overline{BC}$.

A* Search Algorithm (1968)



1. We extend the first point of Dijkstra by consdering the estimated distance of the source from the target $f_{\texttt{src},t}(\texttt{src})$ as $h_t(\texttt{src})$, as the source has a zero distance from itself. All the other nodes, without a better guess, are at distance $+\infty$
2. Consider Q as the set of nodes to be visited, starting from $\texttt{src}$ (*open set*) and P as the set of nodes that were already visited (*closed set*).
3. $u := \min \arg_{u \in Q} f_{\texttt{src},t}(u)$:
   - 3.1 Pop u from Q and add u to P
   - 3.2 For all the neighbours v of u in $Q \setminus P$:
     - 3.2.1 Let *tmp*$_{g(v)}$ be as in Dijkstra.
     - 3.2.2 If *tmp*$_{g(v)}$ is better that the previously computed value for $g_{\texttt{src}}(v)$ or $v \notin Q$, retain u as v's direct ancestor and *tmp*$_{g(v)}$ as the final $g_{\texttt{src}}(v)$.
     - 3.2.3 Push v in Q and set $f_{\texttt{src},t}(v)$ as per Eq.1

A*'s Limitations

The main culprit of this algorithm consists on the choice of the heuristic function, which cannot be arbitrary:

- When the heuristic being used is admissible but not monotone, a node might be visited many times, up to an exponential number of cases. In such circumstances, Dijkstra's algorithm could outperform A* by a large margin.

A* guarantees its termination on finite graphs with non negative weights:

- A* also assumes that a goal state exists at all, and is reachable from the start state; if it is not, and the state space is infinite, the algorithm will not terminate.
- If the graph contains negative weights, we are forced to consider other algorithms (Bellman-Ford's, Johnson's).