CS4618: Artificial Intelligence I

Path Finding

Derek Bridge
School of Computer Science and Information Technology
University College Cork

Initialization

In [1]:

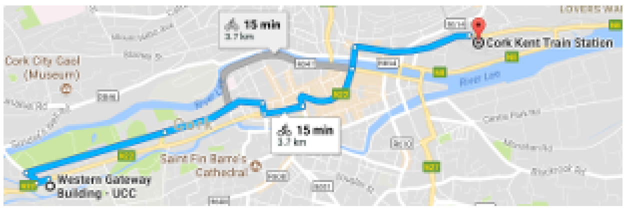
```
%reload_ext autoreload
%autoreload 2
%matplotlib inline
```

In [2]:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

Applications of route-planning and path-finding

- State space search has many application including cargo loading and automatic assembly, not just toy problems!
- But the most obvious of its applications are in route-planning and path-finding

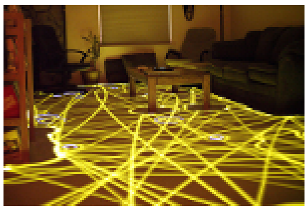


Directions

Warehouse picking

Delivery routes

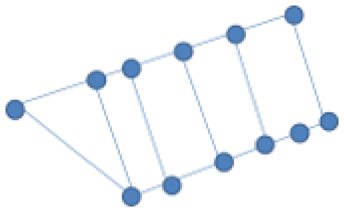
Search and rescue


Vacuuming & mowing

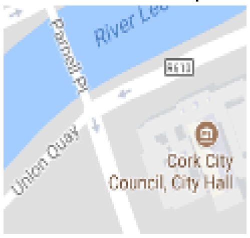
Pathfinding for NPCs

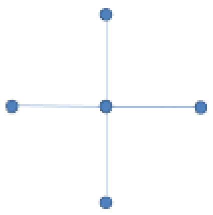
Route planning for road maps

- Most obvious graph representation for road maps:
  - intersections/junctions become nodes of the graph
  - stretches of road between intersections become its edges



- In fact, in addition to intersections/junctions, include extra nodes:
  - e.g. entrances to, and exits from, car parks, colleges, etc
  - e.g. any special points along a stretch of road such as start and end-points of bridges & tunnels, toll plazas, changes of road type or speed limits,…
- Node and edges can have extra data stored on them, e.g. distances, speed limits, road type, presence of a footpath, etc. on edges
- But this representation still has problems with turning restrictions such as these:



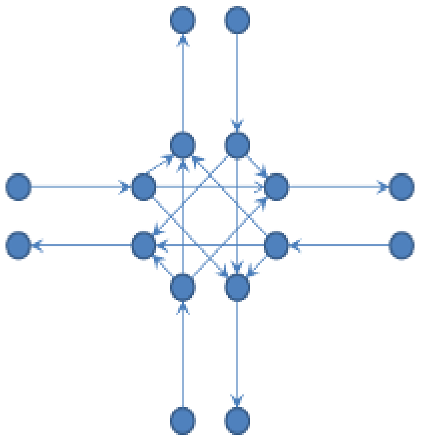
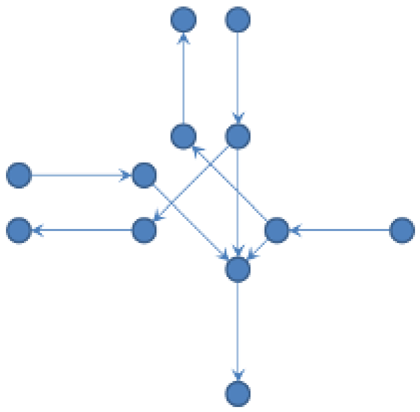


E.g. the graph fails to show that East-bound traffic cannot turn left

Route planning for road maps

- A more sophisticated representation introduces
  - separate edges for the lanes
  - additional nodes & edges to represent 'turns'

E.g.

- Now it is easy to represent one-way roads (how?) and turning restrictions (how?)

Route planning for road maps

- Edge costs
    - Costs are probably expressed in seconds
    - Calculated from, e.g.
        - static data about the edge: distance, road type, etc.
        - transport type (car, bike, foot)
        - dynamic data about the edge: traffic conditions, etc.
- Heuristics
    - Straight-line distance ('*as the crow flies*') never over-estimates
    - When distances are large enough, must take into account curvature of the Earth
        - The *haversine formula* calculates the distance between two points, given their longitude and latitude
    - But you must convert this to use same units as used for costs (seconds)

Route planning for road maps

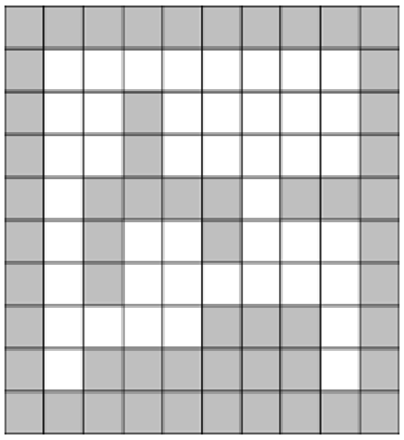
- In mobile apps, fast calculation of routes is important
  Also, fast *re-calculation*. Why?
- In fact, it might be more important than optimality:
    - Quickly find two or three good routes, rather than take longer to find the shortest route
- One good thing: branching factor is relatively low
- For speed-up:
    - obviously: optimized code, good choice of data structures, fast hardware, preprocessing,…
    - change the heuristic
        - one that underestimates but underestimates less, provided it is still cheap-to-compute
        - or, perhaps, an inadmissible heuristic, especially if it doesn't overestimate too much and is cheap-to-compute
    - alternative algorithms, e.g. hierarchical route planning, ALT algorithms, bidirectional ALT algorithms

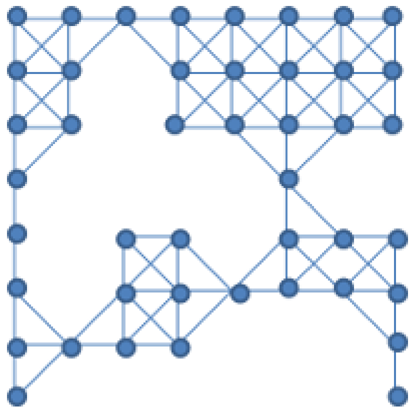
NPCs in games

- In many games, **non-player characters** (NPCs) are quite dumb
    - They appear, as if from nowhere, e.g. when the player enters a room
    - Their actions are scripted
- But, we are entering a new era of games where the NPCs will compete/cooperate with players and other NPCs in much more interesting ways
- One skill will be to navigate around the game world, which involves pathfinding and movement

Path finding for NPCs in games

- Consider a **grid-based game**:
  - the map comprises a **grid** of **tiles** (we'll assume 2D only and ignore wormholes, portals, etc.)
  - character movement is constrained to the grid, **axial** (horizontal & vertical) and sometimes also **diagonal**
- Most obvious graph representation for the grid:
  - obstacle-free tiles become nodes of the graph
  - edges connect to up to 8 neighbours
- E.g.





- Nodes can have extra data stored on them, e.g. type of terrain (grass, tarmac, marsh, …)

Edge costs for grids

- Simplest: all edges have a constant cost, e.g. 1 (although it might be different for different NPCs)
- More sophisticated: axial edges cost, e.g., 1; diagonal edges cost a bit more, e.g., 1.4
- Even more sophisticated is to add higher costs to certain edges, which will discourage paths that use these edges, e.g.:
  - If tiles have different altitudes, you can have higher edge costs for moving from lower altitude tiles to higher altitude tiles
  - If tiles have different terrain types, you can have higher edge costs for moving from, e.g., marsh to marsh tiles than grass to grass tiles
  - If tiles can contain dangers (e.g. enemies), you can have higher edge costs near these tiles (and even modify these costs dynamically if the enemies move) (so-called 'infuence maps')
  - …

Heuristics for grids

- Assume a 2D grid where current tile is $\langle x, y \rangle$ and the goal tile is $\langle x', y' \rangle$
  - Euclidean distance (Pythagoras!): straight-line distance
    $$h(\langle x, y \rangle) = \sqrt{(x - x')^2 + (y - y')^2}$$
  - Manhattan distance: counting axial movements
    $$h(\langle x, y \rangle) = |x - x'| + |y - y'|$$
  - Chebyshev distance: allows for diagonal movements too
    $$h(\langle x, y \rangle) = \max(|x - x'|, |y - y'|)$$
  - …
- Whether these are admissible (even in obstacle-free grids) is quite complicated
  - depends on whether only axial movement is allowed, or axial and diagonal
  - if both are allowed, depends on whether they have the same or different costs
- But, in any case, you might not care too much about optimality
  - Quickly calculating (and re-calculating)a good route might be better than taking longer to find an optimal route
  - Why else might you not want to always find optimal paths for NPCs?
  - You might even exit the search algorithm early (with only a partial path found) and start executing it. Why?

Speeding up the search

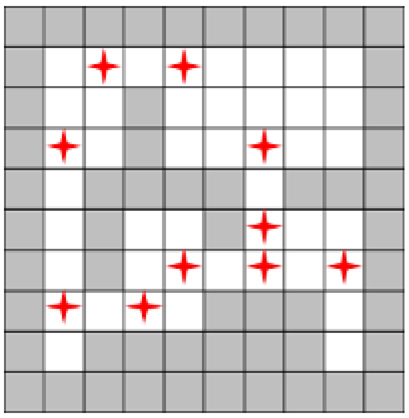
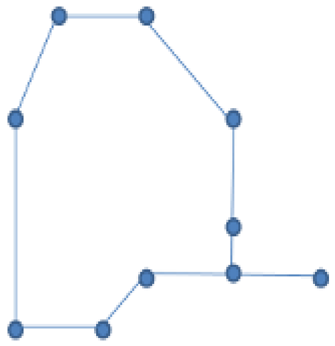
- Obviously: optimized code, good choice of data structures, fast hardware, preprocessing,…
- Change the heuristic
  - one that underestimates but underestimates less, provided it is still cheap-to-compute
  - or, perhaps, an inadmissible heuristic, especially if it doesn't overestimate too much and is cheap-to-compute
- Prune or carefully order the successors
  - E.g. in grids, there can be lots of re-exploration unless we include code to avoid it
  - E.g. in grids, we can devise algorithms that make use of symmetries to reduce work
  - E.g. assume axial and diagonal costs are the same (to simplify the example)
    And assume the previous state was tile A, and the current state is tile B

A	C	F
D	B	H
G	I	E

    In this situation, B's successors do not need to include C and D. Why not?
    A new algorithm, Jump Point Search, uses this idea recursively *during* A^* search to achieve massive speed-ups (especially when there are large obstacle-free areas)
  - E.g. in grids, there will be lots of ties: when inserting on the agenda an item with same f value as an existing item on the agenda, make sure the new one comes earlier in the queue than the existing one (Advanced question: Why?)
- Change the way you represent the grid as a graph (next slide)

Different graphs for grids

- Our representation has a lot of nodes and edges
- Consider instead a graph based only on **waypoints**
    - It's possible to define waypoints algorithmically, e.g. based on points that are visible from one another

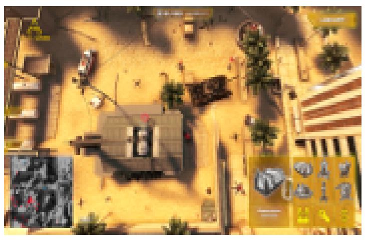



    - But you might define them manually instead
- ('Navigation meshes' are another possibility: nodes are polygons, each comprising several tiles)
- These graphs can be much smaller, but there is a problem:
    - at movement time, the path will need to be converted back into grid-based movement

Continuous maps

- In a **continuous map**, agents can move freely (as long as they avoid the obstacles)
    - not constrained to moving along roads
    - not constrained to moving between tiles

  E.g.



- How do we represent a continuous map as a graph?
- Answer: discretize it:
    - Place a (fine-grained?) grid on it
    - Place (lots of?) waypoints on it (or a 'navigation mesh')
- At movement time, the path (which will comprise linear moves) may need to be *smoothed* to make it more natural

Concluding remarks

- This lecture illustrates something about AI in general
  - Successful uses of AI don't just depend on clever algorithms
  - They still depend to a large extent on the human designer finding ways to incorporate **domain knowledge** into the solution

Some advice for journalists writing about AI (http://togelius.blogspot.ie/2017/07/some-advice-for-journalists-writing.html)
"Much of 'artificial intelligence' is actually human ingenuity…[W]hen building a system to solve a problem, lots of knowledge about the actual problem ('domain knowledge') is included in the system. This might take the role of providing special inputs to the system, using specially prepared training data, hand-coding parts of the system or even reformulating the problem so as to make it easier."

```
In [ ]:
```