COMP702: Classical Artificial Intelligence

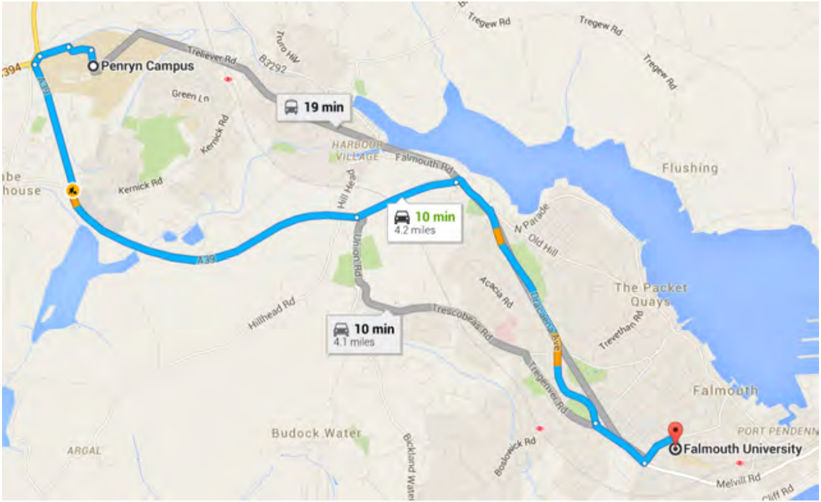
8: Pathfinding and Planning

Pathfinding

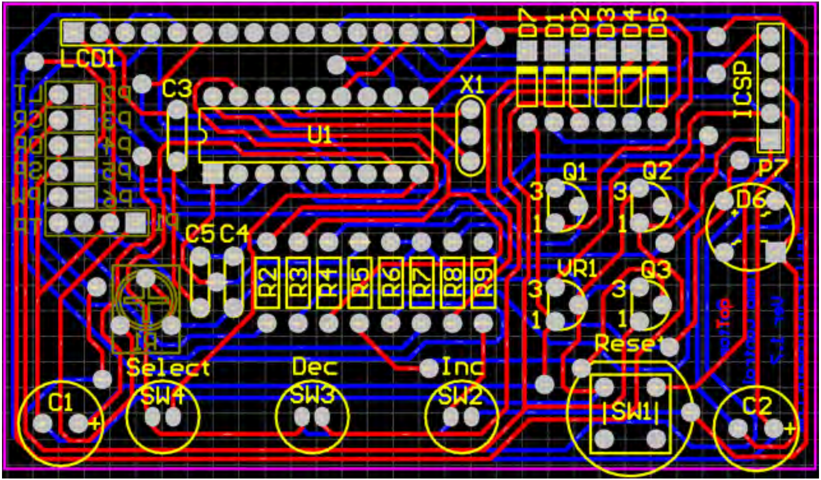
The problem

- We have a **graph**
  - **Nodes** (points)
  - **Edges** (lines between points, each with a **length**)
- E.g. a road map
  - Nodes = addresses
  - Edges = roads
- E.g. a tile-based 2D game
  - Nodes = grid squares
  - Edges = connections between adjacent squares
- Given two nodes A and B, find the **shortest path** from A to B
  - "Shortest" in terms of edge lengths — could be distance, time, fuel cost, ...

Applications of pathfinding



Applications of pathfinding



Applications of pathfinding

Applications of pathfinding

Applications of pathfinding

Pathfinding as search

- Basic idea: build a **spanning tree** for the graph
- Root node is A (the start node)
- Edges in the tree are a **subset** of edges of the graph
- Once the tree includes B, we can read off the path from A to B
- Need to keep track of two sets of nodes:
  - **Open set**: nodes within 1 edge of the tree, which could be added next
  - **Closed set**: nodes which have been added to the tree, and shouldn't be revisited (otherwise we could get stuck in an infinite loop)

Graph traversal

- **Depth-first** or **breadth-first**
- Can be implemented with the open set as a **stack** or a **queue** respectively
- Inefficient — generally has to explore the **entire map**
- Finds a path, but probably not the **shortest**
- Third type of traversal: **best-first**
  - "Best" according to some heuristic evaluation
  - Often implemented with the open set as a **priority queue** — a data structure optimised for finding the **highest priority** item

Greedy search

- Always try to move **closer** to the goal
- Visit the node whose **distance to the goal** is **minimal**
- E.g. **Euclidean** distance (straight line distance — Pythagoras' Theorem)
- Doesn't handle **dead ends** well
- Not guaranteed to find the **shortest** path

Dijkstra's algorithm

- Let $g(x)$ be the sum of edge weights of the path found from the start to x
- Choose a node that minimises $g(x)$
- Needs to handle cases where a shorter path to a node is discovered later in the search
- **Is** guaranteed to find the shortest path
- ... but is not the most efficient algorithm for doing so

A* search

- Let $h(x)$ be an estimate of the distance from x to the goal (as in greedy search)
- Let $g(x)$ be the distance of the path found from the start to x (as in Dijkstra's algorithm)
- Choose a node that minimises $g(x) + h(x)$

Properties of A* search

- A* is **guaranteed** to find the shortest path if the distance estimate $h(x)$ is **admissible**
- Essentially, **admissible** means it must be an **underestimate**
  - E.g. straight line Euclidean distance is clearly an underestimate for actual travel distance
- The more accurate $h(x)$ is, the more efficient the search
  - E.g. $h(x) = 0$ is admissible (and gives Dijkstra's algorithm), but not very helpful
- $h(x)$ is a **heuristic**
  - In AI, a heuristic is an estimate based on human intuition
  - Heuristics are often used to prioritise search, i.e. explore the most promising options first

Tweaking A*

- Can change how $g(x)$ is calculated
  - Increased movement cost for rough terrain, water, lava...
  - Penalty for changing direction
- Different $h(x)$ can lead to different paths (if there are multiple "shortest" paths)

String pulling

- Paths restricted to edges can look unnatural
- Intuition: visualise the path as a string, then pull both ends to make it taut
- Simple algorithm:
  - Found path is $p[0], p[1], \ldots, p[n]$
  - If the line from $p[i]$ to $p[i+2]$ is unobstructed, remove point $p[i+1]$
  - Repeat until there are no more points that can be removed

Hierarchical pathfinding in Factorio

https://factorio.com/blog/post/fff-317

What is the graph?

- In a tile-based game, the graph comes from the geometry of the tiles
- In a 3D environment, the graph can be built automatically from the level geometry (e.g. with a **navigation mesh**)
- Can get complex — dynamic obstacles, vaulting, jumping, ...
- Following the path can also be complex — **steering** behaviours

Planning

Planning

- An **agent** in an **environment**
- The environment has a **state**
- The agent can perform **actions** to change the state
- Actions have a **cost** associated with them
- The agent wants to change the state so as to achieve a **goal**
- Problem: find a low-cost sequence of actions that leads to the goal

Planning as search

- We can construct a **state-action graph**
- (Similar to a **game tree**, but may include **multiple paths** or **cycles**)
- Now the planning problem becomes very similar to the pathfinding problem (albeit possibly with multiple goals)
- We can use many of the same algorithms (DFS, BFS, Dijkstra)
- We can also use A* if we can come up with an admissible heuristic

Representing planning problems

- We can code the state-action representation manually
- Or we can use a more general representation...

STRIPS

STRIPS planning

- **St**anford **R**esearch **I**nstitute **P**roblem **S**olver
- Describes the state of the environment by a set of **predicates** which are true
- (A predicate is basically a function which returns a `bool`)
- Models a problem as:
  - The **initial state** (a set of predicates which are true)
  - The **goal state** (a set of predicates, specifying whether each should be true or false)
  - The set of **actions**, each specifying:
    - Preconditions (a set of predicates which must be satisfied for this action to be possible)
    - Postconditions (specifying what predicates are made true or false by this action)

STRIPS example

STRIPS framework

- STRIPS gives a common framework for defining planning problems
- Definitions in terms of **propositional logic**
- Easy to **enumerate** and **simulate** actions, and hence search the state-action graph
- Possible to write general-purpose STRIPS solvers

GOAP

GOAP

- **G**oal **O**riented **A**ction **P**lanning
- Originally developed for F.E.A.R. (2005), since used in several games
- A modified version of STRIPS specifically for real-time planning in video games

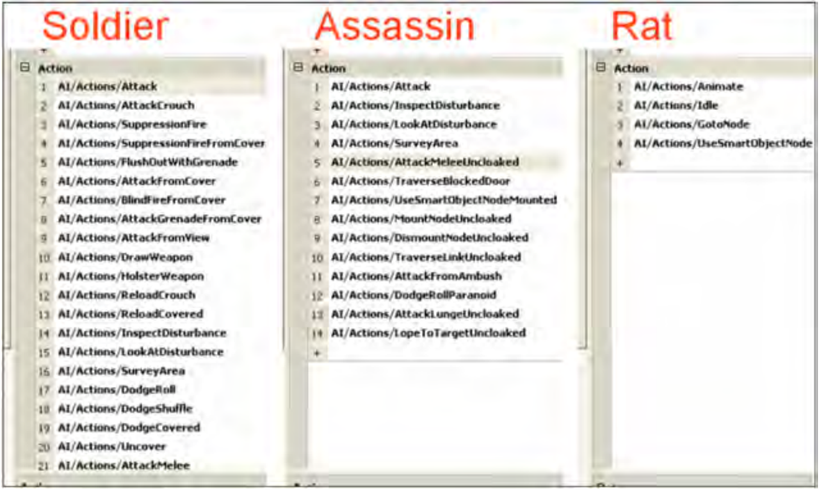
GOAP

- Each agent has a **goal set**
  - Multiple goals with differing **priority**
  - Goals are like in STRIPS — sets of predicates that the agent wants to satisfy
- Each agent also has a set of **actions**
  - Like in STRIPS — actions have preconditions and postconditions
  - Unlike STRIPS, each action also has a **cost**

Action sets

- Different types of agent could have the **same goals** but **different action sets**
- This will result in those agents achieving those goals in **different ways**
- NB this doesn't have to be explicitly coded — it **emerges** from the GOAP system
- E.g. this was used by the F.E.A.R. team to quickly add new enemy types

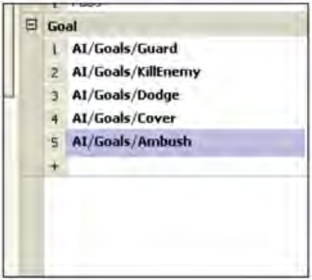
Action sets



Soldier
Action
1 AI/Actions/Attack
2 AI/Actions/AttackCrouch
3 AI/Actions/SuppressionFire
4 AI/Actions/SuppressionFireFromCover
5 AI/Actions/FlushOutWithGrenade
6 AI/Actions/AttackFromCover
7 AI/Actions/BlindFireFromCover
8 AI/Actions/AttackGrenadeFromCover
9 AI/Actions/AttackFromView
10 AI/Actions/DrawWeapon
11 AI/Actions/HolsterWeapon
12 AI/Actions/ReloadCrouch
13 AI/Actions/ReloadCovered
14 AI/Actions/InspectDisturbance
15 AI/Actions/LookAtDisturbance
16 AI/Actions/SurveyArea
17 AI/Actions/DodgeRoll
18 AI/Actions/DodgeShuffle
19 AI/Actions/DodgeCovered
20 AI/Actions/Uncover
21 AI/Actions/AttackMelee
Assassin
Action
1 AI/Actions/Attack
2 AI/Actions/InspectDisturbance
3 AI/Actions/LookAtDisturbance
4 AI/Actions/SurveyArea
5 AI/Actions/AttackMeleeUncloaked
6 AI/Actions/TraverseBlockedDoor
7 AI/Actions/UseSmartObjectNodeMounted
8 AI/Actions/MountNodeUncloaked
9 AI/Actions/DismountNodeUncloaked
10 AI/Actions/TraverseLinkUncloaked
11 AI/Actions/AttackFromAmbush
12 AI/Actions/DodgeRollParanoid
13 AI/Actions/AttackLungeUncloaked
14 AI/Actions/LopeToTargetUncloaked
Rat
Action
1 AI/Actions/Animate
2 AI/Actions/Idle
3 AI/Actions/GotoNode
4 AI/Actions/UseSmartObjectNode

Layering

- Goal set allows different behaviours with different priorities to be **layered**
- E.g. enemy AI in F.E.A.R.:



Implementing GOAP

- An **abstracted** view of the game world is used for planning
- Represented as a fixed-length array (or struct) of values
- Predicates (preconditions, postconditions, goals) represented in terms of this array representation
- Most implementations also allow for **programmatic** preconditions (e.g. calling the pathfinding system to check availability of a path)

Implementing GOAP

- Not difficult to implement
- Open-source implementations do exist
- Not built into Unity or Unreal, but asset store packages are available

Finding the plan

- As in STRIPS, we can build a **state-action graph**
- Since actions have costs, we can use **Dijkstra's algorithm** to find the lowest cost path to the goal
- (or A* if we can find a suitable heuristic)
- Plan is a **queue** of actions that the agent then executes
- If the plan is interrupted or fails then the agent can **replan**

Using GOAP

- Planning is suitable when achieving a goal requires a specific **sequence of actions**
- Especially when the plan is not obvious, or when you want to let the plan be **emergent**
- Does require **abstraction** in a real-time videogame setting, though STRIPS-like definitions give a useful framework for this

“AAA game AI” compared: GOAP vs behaviour trees

- BT: Designer specifies “how”
- GOAP: Designer specifies “what” — “how” is in whatever system is used to implement actions (FSMs in F.E.A.R.; could use BTs or hand coding)
- Both: actions (tasks in BT) are modular and reusable between agents
- GOAP: goals are also modular and reusable
- BT: goals are not represented explicitly
- BT can be classified as **authored behaviour**
- GOAP can be classified as **computational intelligence**

Workshop