COMP250: Artificial Intelligence

4: Utility-Based AI

Utility

Utility theory

- How does an AI agent measure "goodness" or "usefulness" of the outcomes of its actions?
- **Utility theory** supposes that a given outcome can be assigned a **single number** measuring its **utility**
- Actions can then be **ranked** by utility, and one with the **highest** utility chosen
- Utility is sometimes called **reward**, **payoff**, **fitness**
- Multiply by -1 and we have **cost**

Utility — example

- A laptop costs £500 from Amazon or £450 from Bob's Computers
- Assuming everything else equal, where do you buy it from?

Utility — example

- A laptop costs £500 from Amazon or £450 from Bob's Computers
- Amazon offers next-day delivery, but delivery from Bob's Computers takes 4 weeks
- Assuming everything else equal, where do you buy it from?
- Utility is a **single number** — we essentially have to put a monetary value on the longer wait time

Weighting

- Utility is often formed of several **decision factors**
- In the laptop buying example: price and delivery time
- To get a utility value we can take a **weighted sum**

```
utility = WEIGHT_PRICE * price + WEIGHT_TIME * time;
```

- Here `WEIGHT_PRICE` and `WEIGHT_TIME` are constant values
- The values used will influence the agent's behaviour and so must be carefully tuned by the designer

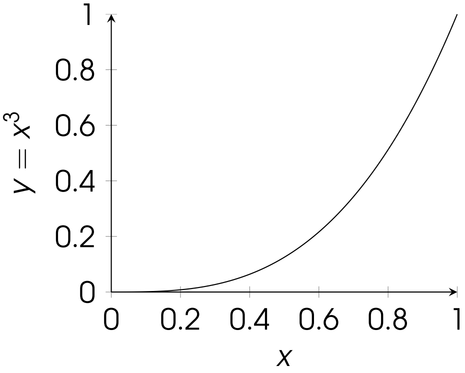
Utility — example

- A GPU costs £400 from Amazon or £500 from Bob's Computers
- Both offer next day delivery
- You have a moral objection to supporting large corporations, and would rather support local businesses
- Assuming everything else equal, where do you buy it from?
- To apply utility theory, we need to **quantify** everything — which may mean putting a numerical value on **intangible** things

Nonlinearity

- Decision factors are not always linear
- E.g. a difference of £1 is more significant for something that costs £5 than something that costs £500
- E.g. a difference of 1 HP is more significant if the agent is close to death than if it is at full health
- Therefore we may want to apply a **curve** mapping to decision factors

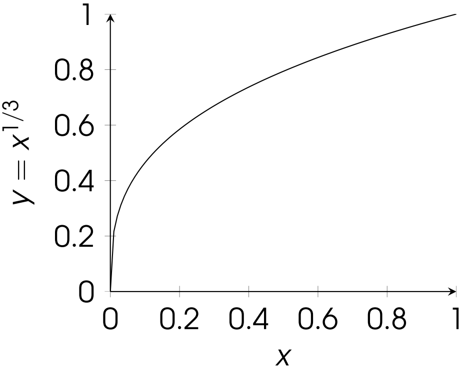
Polynomial curve



- Formula: $y = x^k$
- C#:

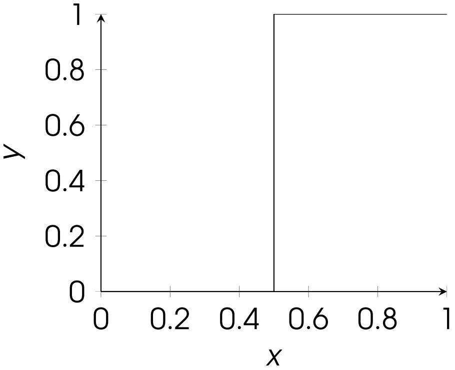
  `Mathf.Pow(x, k)`
- k is a constant: bigger k gives a steeper curve

Inverse polynomial curve



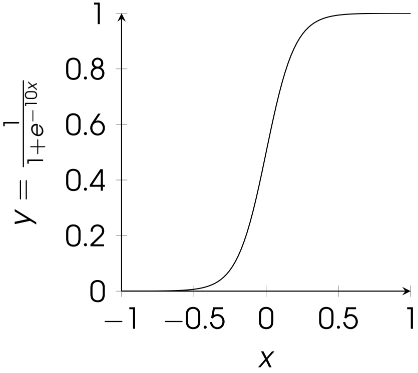
- Same formula as polynomial curve, but k is between 0 and 1
- k closer to 0 gives a steeper curve

Step function



- Formula:
  $$y = \begin{cases} 0 & \text{if } x < 0.5 \\ 1 & \text{if } x \geq 0.5 \end{cases}$$
- C#:
  ```
  (x < 0.5f) ? 0 : 1
  ```
- Models a **threshold** or **if-then** rule

Logistic function



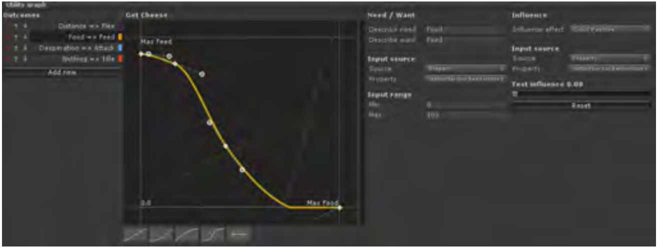
- Formula: $y = \frac{1}{1+e^{-kx}}$
- C#:

```
1 / (1 + Mathf.Exp(-k*x))
```

- Similar to step function but with a softer transition from “off” to “on”
- k is a constant: bigger k gives a steeper curve

Tweaking curves

- Adjusting utility curves is **more art than science**
- It's worth **experimenting** with different curve types to get the desired result
- **Graphical curve editors** are worth investigating
- E.g. `AnimationCurve` in Unity, "Curve Float" in UE4
- E.g. InstinctAI asset for Unity:



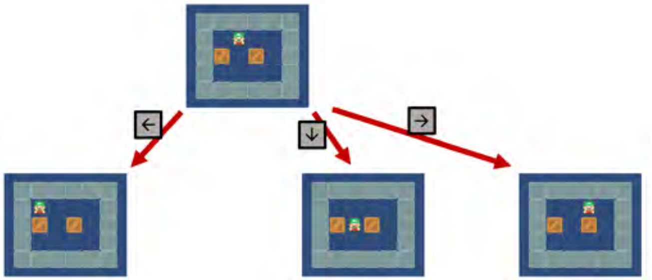
Inertia

- Utilities will generally change rapidly as the state of the environment changes
- This could result in the agent **changing its mind** as to which action has the best utility
- Can force the agent to finish its current action before evaluating and choosing another
- Can give a utility bonus to sticking to the current action — this still allows the agent to change its mind if the current action's utility becomes very bad

1-ply Search

1-ply tree

- Recall from last time: in discrete planning problems or games, we can build a **state-action tree**
- Consider a tree with only one level or **ply**



1-ply search

- Suppose we have a utility measure for states
- I.e. a function which, given a state, assigns it a utility score
- Then we can "search" for the action leading to the best utility score

1-ply search

```
procedure OnePlySearch(state)
    bestAction ← null
    bestUtility ← −∞
    for each valid action from state do
        nextState ← copy of state
        apply action to state
        utility ← Utility(nextState)
        if utility > bestUtility then
            bestAction ← action
            bestUtility ← utility
        end if
    end for
    return bestAction
end procedure
```

Why 1-ply search?

- It is often easier to come up with a utility measure for states than for actions
  - E.g. distance to goal — greedy search
  - E.g. material evaluation in chess
- Much faster than a full-blown BFS or game tree search, if depth of forward planning is not required

Expectation

Expected utility

- If the environment is **stochastic**, the outcome of an action (and hence its utility) may not be known with certainty
- Let $p(x)$ be the probability that a given action has utility x
- Then the **expected utility** is

$$\sum_x x \cdot p(x)$$

- That is, the sum of utility values weighted by their probabilities

Expected utility — example

- A slot machine pays out:
  - £1 with probability 0.05
  - £5 with probability 0.03
  - £10 with probability 0.02
  - Nothing with probability 0.9
- The expected payout is

  $$1 \times 0.05 + 5 \times 0.03 + 10 \times 0.02 + 0 \times 0.9 = 0.4$$

  i.e. £0.40
- If it costs £1 to play the slot machine, the expected utility overall is $-$£0.60
- (Although the actual utility can range from $-$£1 to $+$£9)

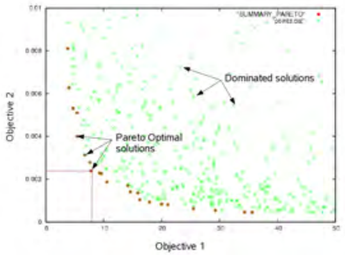
Multi-Objective Optimisation

Multi-objective optimisation

- Utility-based AI is **single-objective**: all decision factors must be combined into a single number
- An alternative is **multi-objective**: treat all decision factors as separate, and find an action that optimises all of them at once

Pareto optimality

- Consider the space of all possible solutions (e.g. actions or plans)
- A solution is **Pareto dominated** if there is some other solution that is better than it on **all** decision factors at once
- A solution is **Pareto optimal** if it is not dominated by any other solution



Single-objective vs multi-objective

- Multi-objective optimisation gets around the problem of having to tune weights for decision factors
- However, there are generally a large number of Pareto optimal solutions so we need some other method to tie-break between them
- ... which may boil down to weights (or at least priorities) anyway