CITS3001

Daniel Paparo

1 Lecture 1

Agent: Essentially an actor in a scene. In this case it will be a computer agent.

1.1 Algorithm design

It is important to differentiate the problem, the algorithm and the implementation.

Computational problem: A good general description of a question to be answered.

Most algorithms cannot be described without the definition of an acompanying datastructure. The algorithm is essentially just modifying the datastructure until the solution is found. We also need to become more mindful of the complexity of a program or algorithm.

1.2 Design Process

1. what is the problem
2. Does a solution exist
3. Otherwise can we find a solution/are there multiple
4. Is the algorithm efficient or efficient enough
5. Does the implementation of the problem have any issues?

Some key considerations when designing an algorithm include:

- Running time
- Resources
- Development time

Invariant: True at the start of every loop and at the end.

Proof by contradiction: Assume that a property is true, and then show that this property will lead to a contradiction.

Numerical stability: Essentially the accumulation of error. For example:

$$x + y = z$$

is actually:

$$(x + e) + (y + e) = (z + 2e)$$

1.3 Complexity of insertion sort

Simply look at the number of loops and how many accesses occurs of the data structure. Important to remember that constant factors do not effect the *Big O* complexity as CPUs increase by constant factors. The important part of the complexity is the factor that dominates all other factors. The only really feasible ones are polynomial, but often others are being used for more difficult problems, if the solution is efficient enough for the dataset.

1.4 Finite state machines

Will essentially take a state, read an input, then give some state back. This is a state machine. This does not work well withd data structures as it can only ever know what is directly under the read head, and where it has come from.

2 Lecture 2

2.1 Optimization

What we are typically dealing with in artificial intelligence is optimization. Optimization algorithms deal with situations in which there may be multiple correct answers, but we are searching for a better one. An example of an optimization problem is a bottling machine: If we start running it too fast then it will wear down (Costing money to repair). If we run it too slow then it wont make enough bottles and thus wont sell as many (Lose money as it is not producting what is required). We must decide apon a balance of these factors, and the solution we come to will not necessarily be the ideal solution, but this creates the concept of the feasible sollution.

Feasible solution: A solution which will fulfil the given requirements, but not necessarily optimally. Some feasible solutions are better than others, and in different ways (this may be: cost, time etc.)

We must make a distinction between *Feasibility* and *Optimization*. Sometiemes the feasible and optimal solution are only slightly apart.

2.1.1 Complications of optimization

- The criteria can be ill defined and can change over time, requiring different solutions and defining a new feasible solution
- The solution space or search space may be extremely large

2.2 P versus NP

The *vertex cover* Solution cannot be run in polynomial time, and is an **NP - Hard** problem. A computational problem is **P** if it is a deterministic algorithm that can run in polynomial time. This "polynomial time" is said in terms of a turing machine. A computational problem which is NP is a non deterministic problem that runs in polynomial time. An NP complete problem is one that runs in non-deterministic polynomial time, and an NP hard problem is atleast as difficult as these. The real issue here is whether NP problems can run in polynomial time in a deterministic manor.

Turing machine: A theoretical machine which contains infinite tape memory with a single read head, and is a state machine.

Deterministic: Will always produce the same result if given the same starting condition with no wiggle room.

Non-deterministic: Can give many answers, but only one needs to be correct (there is no true "only answer").

All NP-Hard problems have the following properties:

- There is no known polynomial time algorithm
- The only known algorithms take exponential time
- If you could solve one in polynomial time, then you can solve all in polynomial time.

2.3 Solution strategies

2.3.1 Greedy Algorithms

They will take the single easiest answer available at each step, constructing a solution using some kind of heuristic to help make the "easiest" choice. This answer will be feasible, however in most cases will not be optimal. These will not verify the answers once they have been completed.

Example:

The following is an example of the "activity selection problem". The following rules can apply for the greedy solution:

- Choose the shortest rask
- Choose the task with the fewest overlaps
- Choose the task that starts earliest
- Choose the task that starts latest
- Choose the task that finishes earliest
- Choose the task that finishes latest

The greedy solution does not pay attention to how the solution will be affected by local decisions. This solition requires a sort which is of $O(lgn)$ and a linear search, which is of $O(n)$, leaving us with an overall complexity og $O(nlgn)$. While the greedy approach works in a short amount of time, and gives alright answers, it isn't always optimal, and can often be caught up by small mistakes. This solution does not consider any other solutions to what is immediately best.

Matroid: A problem in which the greedy solution (with the correct heuristic) will be the correct solution every time. This is the case in the fractional knapsack problem, in which the weight to value ratio can be used to fill up the fictional knapsack.

Local rules can affect an entire solution (such as in a graph problem). The greedy algorithm will not look at the greater effects of a local choice, and the locally optimal solution can result in reduced, and less optimal decisions later on in the search space.

2.3.2 Dynamic programming

Essentially we are finding many sub problems and adding them together. This is expressed as one or more reccurance relations. the time complexity of the algorithm is $O(nW)$ - this is because W is unbounded, and can be large. n is the sample size - what we need to look at is the size of the input, not the value.

An example of this is in the discrete knapsack problem.

2.3.3 Linear programming

These probles are everywhere. This is a well balanced optimization problem. For us to get to the optimal solution we essentially have to traverse a **3D simplex**. This is a hill climbing algorithm - in the worst case it is $O(2^n)$. This solution is good for functional solutions, but these are often not good for integer solutions. The integer problems are far more difficult.

Important: Recognise linear and mixed integer linear problems.

3 Lecture 3

Approximation algorithms will give a feasible solution to something which is NP ard. We must compare the returned value to the actual optimal value - this will give us the efficiency value. This is the kind of thing that will be done in testing to validate the efficiency of the algorithm.

3.1 Travelling salesman problem

A lot harder than a normal graph domination problem. What we are concerned here is with the shorted distances and geometric distances. There are a few ways that we can approach this problem.

3.1.1 Greedy solution

Using two different heuristics we will get two different complexities:

- The nearest neighbor heuristic will give us $O(n^2)$
- Starting at all cities will give us $O(n^3)$

3.1.2 Minimal spanning tree

Building a minimal spanning tree will give us an optimal solution. To do this:

- Find a minimal spanning tree.
- Do a DFS
- Visit cities in order of discovery time (essentially creating a table of shortcuts).

This is guarenteed to give an approximation of $\leq 2optimal(1)$

3.1.3 Coalesce simple paths

- We can sort the edges by weight
- Take the next best edge, but make sure we dont make a cycle

At the end of this we will be given a tour, with an alright path.

3.1.4 Insertion methods

There are a few heuristics we can use for this method:

- Nearsest insertion (which works in $O(N^2)$)
- Shortest insertion (which works in $O(N^2)$)
- Random insertion (which works in $O(N^2)$)
- Cheapest insertion (which works in $O(N^2)$)

3.1.5 Iterative improvements

A common feature of tours produced by a greedy heuristic is that it usually is easy to see how they can be improved immediately by changing a few edges. This leads to the idea of iterative improvement:

- This creates a feasibly solution quickly
- Modifying it slightly (and repeatedly) to improve it
- Will require a rule for changing one feasibly solution to another
- A schedule for deciding which changes to make

One basic improvement involves deleting two edges and replacing them with two new edges. This will maintain the tour but might offer a more optimal route.

Another improvement we can make is a **2-optimal** tour. This is an algorithm that, in every iteration, examines every pair of edges ina tour and perfoms an exchange that improves the tour. There are more complicated schemes for this which involve deleting more edges, such is in the **k-optimal** tours.

3.2 Local optima

In the search space: there will be many local optima, but only one global optimal solution. Gradient based searching can be used to help escaping the local optima, and there are a few techniques to do this:

3.2.1 Simulated annealing

This occurs by allowing the search process to make backward moves. Each iteration takes the form:

```
Randomly generate a neighbor T' of the current T,

If c(T') <= c(T):
    Accept T'

If c(T') > c(T):
    Accept T' with probability p
```

3.2.2 Tabu search

This is related to the english term "taboo" which is something that is prohibited or forbidden. The tabu search tries to avoid weaknesses, and spends the most time exploring local optima, whilst retaining the ability to escape them. The fundamental idea is that we maintain a tabu list detailing the last vertices which have been visited at each iteration. This is a very aggresive method, and each attempt moves in the best possible direction. THis avoids spending too much time near one local optimum, forcing it to visit other parts of the search space. This is quite difficult to implement and also extremely computationally expensive.

3.2.3 Genetic algorithms

Genetic algorithms will try to avoid getting stuck in local optima by maintaining a population of solutions. These solutions will be distributed accross the search space, meaning that it is unlikely for a single local optima to trap all of them. At each generation the population of size n is used to create n new solutions, and the best n of these "survives" to the next generation. These are extremely successful on a wide range of problems, and they work well in difficult domains that are often beyond simpler methods. These are forms of adaptive systems, and often fall into one of the following techniques (computationally intelligence):

- Genetic algorithms
- Swarm optimisation
- Ant colony optimisation
- Learning classifiers
- Neural networks
- Artifical immune systems

4 Lecture 4

In optimization we are effectively just running an algorithm, and in artificial intelligence we are just running optimization. THe question must be asked: at what point do algorithms end and artifiial intelligence start? Most AI advancements are mostly outside of research, but are mainly due to extremely large neural networks found on graphics cards.

Problem solving: coming up with ones own solution to a problem that it has never encounted before.

4.1 Ways of thinking

There are different ways of thinking and acting. In a simple form: humans will *act and think humanly*. We often want to avoid this in AI and the following metrics can be used:

4.1.1 Thinking humanly

We see how humans think and attempt to replicate it in software/hardware. To do this we need a lot of cofnative sciency, using psychologically based experiments, or brain imaging. This brings about another question:

Do we attempt to model the mind? Or do we model the brain?

Intelligence: The ability to be indistinguishable to a human.

Turing test: Essentially determining if the "person" that a human is talking to is either a human or a computer. If the computer manages to "fool" the human listener, then the computer has "passed" the turing test.

4.1.2 Thinking rationally

These will act "better" than humans. This will essentially require the codifications of thoughts. By essentially using these rules, we are finding things which are possible, but we may find impossible.

4.1.3 Acting rationally

An agent will be constructed in order to carry out the work of the artifical intelligence. An agent must be defined:

Agent: this will have a perception of its surroundings, and the ability to perform actions in order to complete some goal.

4.2 Agents

An agent is broken down into **percepts** (or sensors) and effectors. A rational agent will always try to do the right thing, based on a set of goals. This rational action is one that is theorized to get the agent closer to its goal, but doesnt necessariy mean that it is going to be successful. Success of an agent is somewhat relative.

An agents function is to map its percepts sequences to actions.

4.2.1 A simple reflex agent

- Will choose a condition using condition action rules
- No history is stored
- This is how some simple organisms work

This will be useful for most control systems. Simply just if/then/else statements all the way through, and can only deal with what is happening in the immediate present.

4.2.2 Model based reflex agents

These are basically consider the context of the percepts to see how it should respond. Still looking at the immediate present, but will use a model to decide on the optimal decision.

4.2.3 Goal based agents

These will still have a very good model of the world, but will have far more autonomy. This now has a concept of how its actions will change the world around it.

4.2.4 Utility based agents

Note: This will be useful in the project

The goal is binary, we either win or we dont. Tihs agent must maintain its expected utility. The goal here is to optimize the function.

5 Lecture 5

Our searches are mostly going to based around finding some goal state. Artifical intelligence is based entirely on doing effective and efficient searches. Searches are what differentiates AI from normal machine learning, which dont rely on them. AI ultimately always comes down to a DFS.

We must define a:

- State (state of our world)
- Goal

- Cost
- Solution

In these problems we are building up a graph representation, removing any which break our constraints. These problems will always result in a tree.

The fundamental idea is:

- At all times we are in one state s
- s will offer several actions
- We must choose one action and explore it
- We must keep all of s to be explored later

```
GeneralSearch (problem, strategy):
    initialise the tree with the initial state of problem
    while no solution found
        if there are no possible expansions left
            then return failure
        else use strategy to choose a leaf node x to expand
            if x is a goal state
                then return success
            else expand x
                add the new nodes to the tree
```

Often it is important to check for cycles - not always though - in a path finding algorithm it will be, but in sodoku: not so much. To build the this, we must have:

- A set of possible states s
- A start state s_0
- A goal function
- A terminal condition
- a successor function

```
U = {s0} -- unvisited nodes
V = {} -- visited nodes
while U ≠ {}
    s = select a node from U
    if s Î V -- occurs check
        then discard s
    else if g(s)
        then return success
    else if t(s) -- cut-off check
        then discard s
    else
        V = V + {s}
        U = U - {s} + successors(s)
```

The performance of these search stratergies is defined by:

- Its completeness

- Optimality
- Time complexity
- Space complexity

Its important to note that infinite trees can occur in which there is no solution - but the algorithm cannot possible know this when beginning the search.

5.1 Uninformed search strategies

- Breadth-first search - Expanded the shallowest node next
- Uniform-cost search - Expanded the lowest-cost node next
- Depth first search - Expand the deepest node next
- Depth limited search - Depth-first, but with a cut off
- Iterative depening depth-first search - Repeated depth-limited, but with increasing cut offs
- Bidirectional search - search from both ends concurrently

Critereon	BFS	Uniform Cost	DFS	Depth-limted	Iterative	Bi-directional
Complete	Yes	Yes	No	No	Yes	Yes
Time	$O(b^d)$	$O(b^{1+(C/\epsilon)})$	$O(b^m)$	$O(b^l)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+(C/\epsilon)})$	$O(bm)$	$O(bl)$	$O(bd)$	$O(b^{d/2})$
Optimal	Tes	Tes	No	No	Yes	Yes

6 Lecture 6

6.1 Informed search algorithms

What we really want to do is weigh the options which get us closed to our goal. To do this we will need to know about the goal.

6.1.1 The greedy approach

The limitations of the greedy approach is that: we might end up not at our destination, and not optimal. We bring about the idea of a heuristic function. These greedy models have the concept of the visited node.

6.1.2 A* search

The A* search is not much better than the greedy approach as far as asymptopic complexity is concered. The A* search can simply be seen as an extension of Dijkstra's shortest path algorithm, and uses many of the same techniques. This is an example of a **best-first search** algorithm, and will essentially select the percieved optimal solution, which will get us closer to our goal - if that doesnt work out, then we will try some other avenue. The algorithm must make use of an `open` and `closed` list of nodes. The general algorithm for this is as follows:

```
function A*(start, goal)
  open_list = set containing start
```

```
closed_list = empty set
start.g = 0
start.f = start.g + heuristic(start, goal)
while open_list is not empty
  current = open_list element with lowest f cost
  if current = goal
    return construct_path(goal) // path found
  remove current from open_list
  add current to closed_list
  for each neighbor in neighbors(current)
    if neighbor not in closed_list
      neighbor.f = neighbor.g + heuristic(neighbor, goal)
      if neighbor is not in open_list
        add neighbor to open_list
      else
        openneighbor = neighbor in open_list
        if neighbor.g < openneighbor.g
          openneighbor.g = neighbor.g
          openneighbor.parent = neighbor.parent
return false // no path exists
```

Quite simply: This will choose the vertext which is getting closest to the goal, which will minimise the effects of a dead end. To do this, the algorithm will need to estimate the costs to extend the current path from the goal, using the function:

$$f(n) = g(n) + h(n)$$

in which $f(n)$ is the overall estimated const function,$g(n)$ is the cost of the function so far from origin to node n, and $h(n)$ is the *heuristic function* from node n to the destination.

This heuristic function must be:

- Always an underestimate - never an over estimate
- Must be monatomic. That is: as we get closer to the goal, the cost must get larger.

A* has the following properties:

Complete - Yes, unless infinite *Optimal* - Yes *Time* - $O(x)$ *Space* - $O(x)$ as it keeps all nodes in memory, typically a priority queue.

Clearly, the x value will depend on the quality of the heuristic.

6.1.3 Determining the quality of a heuristic

There are a few obvious heuristics we can use for a pathfinding algorithm:

- Straight line heuristic is an obvious one
- A heuristic can be anything which gets us closer to our goal

The best heuristic is one that givse you the highest estimate, that is still an underestimate.

6.1.4 Deriving a heuristic

We can derive a heuristic by using a reduced restriction - this is: throwing away some of the restrictions.

6.1.5 Memory bounded A*

The key limiting factor in the A* search is normally the space availability - similar to a breadth-first search. We solved the space problem for uninformed strategies by iterative deepening, which basically trades space for time, in the form of repeated calculation of some nodes. We can do the same here, in which we impose a depth cut off if the f-cost is too high. This suffers from 3 main problems:

- By how much should we increase the k value?
- It doesnt use all the space available
- The only information communicated between iterations is the f-cost limit

6.1.6 Simplified memory of A*

Implements the A* algorithm, but uses all of the memory available, but will keep a little bit of information about the heuristic at any node in particular. This is best for NP problems. This will basically expand the most promising node until the memory is full - once this occurs it will drop the least promising nodes. If all other nodes in the current tree are not promising then we will need to rebuild this subtree - this is a trade off between space and time.

7 Lecture 7

Games offer a good platform for testing intelligence of beings and agents.

Zero-sum game: In which one player must win, and the other loses

In chess, there is no uncertainty - every possible move in the current state can be realised simply by looking at the board. In poker there is a great deal of uncertainty - what cards do which players have, and are those combinations a winning combination. We can use hints from the other players to somewhat work around this.

7.1 Incompleteness vs. non-determinism

Incompleteness: Not all information for the search is currently known - this can include things such as sensor limits, or intractability (the full description of potential states is too large to store).

Non-determinism: We dont yet have all of the informatio required to make an optimal decision. This can include anything outside of the AI's control, such as which cards that another player may or may not have. We can make a model which can attempt to mitivate this.

Both of these imply uncertainty, and are often *almost* interchangable. The methods to counteracting these are very similar.

7.2 Approaching uncertainty

7.2.1 Contingency planning

All possibilities are stored in memory and can simply be called once the right conditions to rise. This requires a huge planning tree, and will thus require a large amount of memory.

7.2.2 Adaptive planning

Alternate between planning, acting and sensing. This requires lots of execution time between moves, but will cut down on the amount of memory required.

7.2.3 Strategy learning

Uses approximations, abstractions and generalisation to choose a feasible solution on the fly.

7.3 Planning for opponents

Just knowing which branch will results in a success is not good enough, however. Random agents are a relatively good approximation for an apponent. If we can cope with a random agent, then we can approximate a real, but dumb opponents. A smart oponent is quite difficult to deal with. What we really need is to look at our goal.

7.3.1 Minimax

This brings about the idea of a utility function. Both players are going to choose the tree which maximises their own score. From our perspective: our player is trying to maximise our score (this is the *max*), and the opposition is trying to minimise our score (this is the *min*). Minimax is based on the idea that both players are going to try to maximise their own scores, so we need to work within this constraing to minimize our opponents score, and maximize our own.

```
function minimax(node, depth, maximizingPlayer) is
    if depth = 0 or node is a terminal node then
        return the heuristic value of node
    if maximizingPlayer then
        value := −∞
        for each child of node do
            value := max(value, minimax(child, depth − 1, FALSE))
        return value
    else (* minimizing player *)
        value := +∞
        for each child of node do
            value := min(value, minimax(child, depth − 1, TRUE))
        return value

minimax(origin, depth, TRUE)
```

The biggest problem with minimax is that it needs to get the terminal nodes, otherwise it is incomplete. A cut off test (a depth limit) will not get to a terminal node, but can have some function to figure out if this position is a good place to be in. This is called an **Evaluation function**, or a **Heuristic function**, and can be then used by the minimax function in place of a terminating factor.

Quiesence: is a situation in which the values are unlikely to change in the future. This term literally means "quietness" and is where the game is unlikely to change outcome in the next few plays, and can be used to decide whether to investigate a subtree (of a move more) for "interesting" plays, as apposed to a "quiet play", which would have many more children to search.

The horizon effect: This is based on the fact that, for many games, we cannot search the entire search space. The horizon effect is brought about by: if we only search to a depth of k, might a decision that looks optimal at k actually be a really poor decision for k+1? How do we know at what depth we should stop the search to try to mitigate this? The quiesence search can help with this.

7.3.2 Alpha beta pruning

The most general strategy of gaming playing. This can be used to compliment minimax. In this situation we can essentially remove whole sections of a tree. This can allow us to ignore some branches at a high level and allow for more appealing branches to be explored more deeply. We need to know what other parts of the tree are up to at any ply. The pseudocode for this is as follows:

```
function alphabeta(node, depth, α, β, maximizingPlayer) is
    if depth = 0 or node is a terminal node then
        return the heuristic value of node
    if maximizingPlayer then
        value := −∞
        for each child of node do
            value := max(value, alphabeta(child, depth − 1, α, β, FALSE))
            α := max(α, value)
            if α ≥ β then
                break (* β cut-off *)
        return value
    else
        value := +∞
        for each child of node do
            value := min(value, alphabeta(child, depth − 1, α, β, TRUE))
            β := min(β, value)
            if α ≥ β then
                break (* α cut-off *)
        return value

alphabeta(origin, depth, −∞, +∞, TRUE)
```

The alpha-beta pruning method has the following features:

- it does not affect the result.
- good move ordering means we can prune more.
- We want to test expected good moves first.
- Perfect ordering douobles our search depth.
- Sometimes we can learn good orderings, known as speed up learning.

8 Lecture 8

The actions that the agent make will determine the utility the agent can provide, and ultimately the ability of the agent to reach a goal.

Sequential decision problems (SPDs): A series of actions and rewards, but the agent doesnt know what maps to what. We must learn the mapping as we go along, this is not an optimization problem.

Value iteration: Estimates the reqard, refines reqards, repeatedly, then uses rewards to generate a plan.

Policy iteration: Make an initial plan, calculate rewards and then remake the plan, repeatedly.

An imporant point to make is that we cannot know the procudure continuously - this defines AI. These agents are of known, accessible and deterministic domains.

Utilities: Essentially the end game. Each state can have a utility.

Policy: A set of state actions. What should we do at any particular state.

A policy can considered to be a proxy for intelligence. At that point it is simply a reflex agent.

8.1 Optimal policies

In order to determine the optimal function we must determine the step cost. For instance, it might be worth copping a fail rather than taking a long route if the cost per step is sufficiently high.

8.1.1 Bellman equation

$$U_i = R_i + max\Sigma M^a_{ji} U_j$$

The bellman equation is one of the key underpinnings factors of the SDP. We are starting with one utility and finding our next one.

8.2 Value iteration

This is essentially determining the time utility at any state. This is locally true to the bellman equation There are also approximation functions that we can use instead of a true value. This will work by essentially creating a convergence. The new utility is always made based on the assumption that the old utility is true.

The estimated utilities are initially used by the agent to explore the space at the cost of a step cost. Then it will increase as the path is found. Once we know the utilities we can use our policy to find our way back.

8.3 Policy iteration

This uses iterative approximation to make the policies better. It is important to realise that the policy space is smaller than the utility space. As such, it is more efficient (typically) to use policy iteration over value iteration. This uses a similar learning enviroment to value iteration. We keep updating the policy until it stops changing. It has been shown that this converges quicker than the value iteration.

Next Present Value: This is derived from economic theory. Typically we can wait for a better offer to come around, and as such if we wait forever the potential gains are infinite. We cannot do this in a game. In this case we will simply look at the rewards that are currently available.

8.4 External agents

There are cases in which an agent has no terminal state. Introducting simple randomness or a genetic algorithm can turn the algorithm into a more accurate state.

9 Lecture 9

A learning agent can experience an event and then use what it has observed from this event and make its policies better from this. At the moment: much of our AI's intelligence comes from its creators. A learning agent can act autonomously and adapt autonomously.

Learning agents are broken into four main components:

- **A performance element**: Chooses the actions that are known to have good outcomes.
- **A learning element**: Chooses the actions that are known to have good outcomes.
- **A critic element**: Responsible for generating feedback on actions.
- **A problem generator**: Responsible for generating new experiences.

9.1 The learning element

This has two key goals:

- Wants to improve the performance element.
- Imrpove the time performance of the performance element.

The design of these elements is impacted by:

- The components of the performance element being improved
- The representation of the components
- The feedback available
- The prior information

9.2 Performance element

These may have many components relating to mapping the current state of play to an action, and the design will be impacted by the context - these are highly context dependent.

9.3 Providing feedback

There are three broad categories of learning:

- **Supervised learning**: Corresponds to being taught by an expert and is given a good example of the correct way to play.
- **Reinforcement learning**: Basically learning from experience. We are really trying something new and seeing how it goes.
- **Unsupervised learning**: Usually in the absence of feedback. This wont be told what is good or bad. This is really learning pathways from the input.

Sometimes we will have prior knowledge. Sometimes not. We need to make a comparison between exploration and exploitation. Do we stick with what we know? Or do we try to find a new or better way?

9.4 Function approximation

We can ultimately reduce all problems down to a function. What we can say is: *"If i do this action, what will the effects be on the world around me"*.

We have a learning function that maps events to an outcome in the world.

9.5 Inductive learning

We are essentially trying to build a mechanical function that maps our inputs/outputs to real world outcomes. We will typically be dealing with a great deal of noise in our function and with our observations and as such we need to have an understanding of acceptable errors.

Discretise: Breaking up outcomes so that the solution space is relatively small. This makes the algorithms much quicker.

9.6 Decision trees

A tree has:

- Limited inputs
- Limited outputs
- Well formed propositions
- Can be huge in size

We are really trying to minimize the size of the decision tree but need to arrive at the same (or an acceptably similar) outcome.

9.7 Assessing performance

Having intelligence is to learn from the data available quicky. We must determine what a "good" tree is. We can do this by:

- Collecting large sets of data
- Dividing them into a training set and test set
- Use the training set to learn the process
- Use the test set to assess the resulting agent.

10 Lecture 10

10.1 Reinforcment learning

Supervised learning is perfect when we are supplied with the input and output of some actions. These can be used to base our learning off. Often times we do not have this for trainingn our agents. Typically the learning example is not perfect:

- No examples provided
- No model of the enviroment
- No utility function at all

These types of agents rely on feedback to imporve its performance. Some key considerations of this type of learning is:

- Is the enviroment known?
- Is the enviroment accessible?
- Are rewards given at terminal states or all states?
- Are rewards given in bulk? Are they given for all components of the utility?

- All feedback should be utilized. This is often very hard.

There is a tradeoff between how much we want to learn and how much we want to do for ourselves

10.2 Learning

Learning is broken up into passive learning and active learning

10.2.1 Passive learning

This is often much safer and is essentially watching the world go by and figuring out what moves are good and what moves are bad. These learners are typically considered to be associatted with a more intelligent entity (such as a chess master or similar).

Active learning These will havce no fixed function. The agent must select actions using what it has learnt so far. This is to say that it must learn the function that it will be using This could be done by usnig a problem generator to learn what options exist.

Utility learning These agents learn state utilities and subsequently use these to maximise the overall utility. This is to say that they know where they need to go, therefore must have a model for the enviroment. This"deep" knowledge can mean that they learn faster (value iteration).

Q-Learning These agents learn an "action-value" function. That is that they have a model of the expected utility of making a certain decision. This means that they do not need to have a sense of how an action will go, just how "good" that individual action is. This is described as "shallow knowledge" and can restrict learning (policy iteration).

10.2.2 Updating

An important part of learning is the update function. Once again we can tell from bellmans equations that the state utilities are not independent.

Adaptive dynamic programming Essentially we are using the bellmans equation at the same time as learning. This needs to learn what the probability of any action will be. This then reduces to the value determination process, once the learning is completed. We are just adaptively learning the rewards. The problem with this is that the algorithm is quite slow. This is good description of learning.

Temporal difference learning This will cut down on processing time and is essentially a slimmed down version of ADP. The key idea here is that the localised utility will be conistant with the Bellman equation. We are really simply trying to get close to ADP. It is important to note that the average U_i will converge on the bellman equation eventually.

These two models are very close and TDL can be seen as a cheap version of ADP. TDL converges on ADP far slower but will run faster. These are some of the considerations we must make when optimizing our agents.

10.2.3 Active learning

In active learning we must not only learn utilities but also select actions. The agent will need to explore the options in order to improve its utility. We must balance the present with future rewards.

10.2.4 Exploration vs. Exploitation

We must choose to either perform well in the current situation, or learn the enviroment. This is an extremely important trade off. Typically we will start off by doing some exploration and then trying to work towards just doing the task (similar how a human learns their job then becomes a master at it). We want to make sure that we dont make the same mistakes over and over again.

At the beginning all actions are considered good. As we learn more we update our knowledge about anyy given situation.

10.2.5 Generalisation in learning

We can never expose our agent to all possible states of play. We need to generalize and learn about all states of play. We need to trade off between exploration and exploitation. A really large hypothesis space will mean:

- More chance it includes a suitable unction
- The space is more sparse
- The function requires more memory
- More examples are needed for learning
- Convergence will be slower
- Its harder to learn online vs. offline

11 Lecture 11

11.1 Logical agents

What we have have been working with previously has been agents whom respond to stimuli. THese agents have no higher level logic, or an understanding of what is really happening. We can add this higher level logic in the form of propositional logic. From this we can develop a concept of an inference. THis is really giving relationships in the world to our agents and allowing the agent to relate one thing to another. This is what we would be running under the hood of a smart assistant such as alexa.

11.1.1 Knowledge

We already have a concept of "knowledge". A knowledgebase, as apposed to a database, has data with relationshios and the concept of negative knowledge - that is "what can i deduce from what i know about this to not be true".

We can build this knowledgebase from:

- Tell the agent what it needs to know initially
- Then the agent can ask itself questions and make its own decisions
- The agent can tell itself facts as these facts develop

This is a higher level look at the way that we must implement an agent.
When we are designing the knowledgebase of an agent we require:

- A language that can represent our knowledge
- A method of processing this knowledge so that we can make decisions and infer from logic

The former can be done through propositional logic and first order logic.
We can break down the world into different "facts". These are statements in propsotional logic. Our agent should beb able to work out from these facts something about the world.

11.1.2 Propositional logic

- A sentence is valid if it is true for all possible models
- A sentence is satisfiable if it is true in some models
- A sentence is unsatisfiable if it is true in no models
- Two sentences are logically equivilant if they are true in the same set of models

11.1.3 Entailment

We need to ask the question: *Does everything we know about the world allow us to solve the problem?*

$$\alpha \models \beta$$

This is pronounced as: *"Alpha entails Beta"*.
We could use truth tables to check entailment. To check if $\alpha \models \beta$ we:

1. Determine rows in which $\alpha = True$
2. If β is true in all of these rows then $\alpha \models \beta$

We can use an inferencing system to infer these truth tables to save on complexity (as a truth table has 2^n rows in which n is the number of propositions).
Complete: The propositions can deal with every situation correctly
Soundness: If it does the correct derivation
An agent with a knowledgebase and a sound and complete inference system can be used to derive all possible outcomes.

Pros and cons Pros:

- Syntax corresponds to facts (Declarative)
- Allows partial / disjunctive / negated information
- Meanings are context independent

Cons:

- Limited expressive power

To fix this we need a language with variables.

11.1.4 First order logic

Propositional logic provides things which can sometimes be true and sometimes be false. This introduces variables. These will include:

- Predicates
- Functions
- Variables
- Connectives
- Equalities
- Quantifies

We can use these properties to aptly express what the agent knows and what it doesn't know.

11.1.5 Situation calculus

This captures the situation of all variable facts at any particular point in time.

Frame problem: We want a representation in which we dont have to keep stating.

Qualification problem: Actions dont always work

Ramification problem: Actions may have secondary effects

Effect axioms: What actions do change

Frame axioms: What actions dont change

Successor state axioms: These solve the frame problem in situation calculus. This enumuerates the affects of the axioms.

11.1.6 Proof methodologies

Forward chaining: Working forward from the known facts and then trying to derive the original query.

Backward chanining: Working backwards from a known query and see if it can be related to the known facts

Resolution: Using the known facts to try to disprive the negation of the query - that is to "prove by contradiction".

Proof by resolution This proof has 3 steps:

1. Convert the agent's database to a conjunctive normal form (CNF)
2. Negate the query and add it to the database (this is only done notionally, as it will be removed)
3. Repeatedly apply the resolution principal to try to demonstrate a contradiction

Resolution is sound and complete for propositional logic. What we are really trying to do is find if there is any exadmple in the database that is a contradiction to our negation.

12 Lecture 11.5

12.1 Agent planning

We are interested in building up a model made of complex formula. The world is made up of a big conjunction of logical facts. A plan is simply a sequence of actions. We want to build then plan

using knowledge that is known about the world. A plan is simply a sequence of actions: We want to build then plan using existing knowledge. In principal we can use entailment in determining the plan, however more feasibly we can use:

- Propositionalism
- Use inference rules

12.1.1 Propositionalism

We can replace the first order logic for propositional logic. That is that we can convert first-order sentences to propositional sentences. This can simply be done with:

- Universal instatiation
- Existential instantiation

The application of these allows for a propositional database to be created for which we can apply our previous techniques.

12.1.2 Unification

We can follow from logical reasoning that something followsing from something else. This allows us to make an inference of the state of the world.

The resolution principal: This is simply using proof by contradiction to prove that a propsition follows

13 Lecture 12

What we are really interested in is the building up of a knowledge of the world around us, and to use this knowledge base to attempt to reach our goal.

13.1 Planning

What we need is a context independent solution to the problem which will allow for us to break up (factor) the problem allowing for it to be solved in a piecewise manor. This is in many ways similar to a search algorithm, however we are typically dealing with these problems in a divide and concur technique.

	Search	Planning
States	Internal values of data structures	Logical sentences
Actions	Coded methods	Preconditions and effects
Goal	Coded methods	Logical sentences
Plan	Sequence from S_0	Constants on actions
	Implicit and hard to decompose	Explicit and easy to decompose

Any action is defined by:

- A precondition
- An effect

13.1.1 Partial order planning

A partial order will place steps on a plan only when it is necessary to do so. A final result will desult in a directed acyclical graph (DAG). By using a topologicy sorting method we can turn this into a linear plan.

Topology sort: Simply turns a graph of a total order into a graph of a partial order, that respoects the same dependencies of the original DAG. There can be multiple valid outcomes of this sort.

13.1.2 Clobbering

The notion that our actions have consequences when we commit them comes up. The preconditions of the states change as we make some action. We say that these actions "clobber" orther actions when they undo those actions

Sussmans anomoly: A problem that cannot be solved by addressing one goal first and then the other.

13.1.3 Real world consideratinos

Agents in the real world will always need to deal with uncertainty. The information may be incomplete - the information might be incorrect. These are both instances of the qualification problem: In the real world there are a lot of unknowns and unanticipated consequences. These issues are amplified when there are many agents working on the same issue.

There are two general approaches to this problem:

- **Conditional planning**: We can anticipate problems and then build contingencies. This is expensive because it deals with unlikely cases.
- **Monitoring and replanning**: Assumes "normal" execution and continually checks on the plan. It will replan if necessary.

13.1.4 Conditional planning

For some state p, check p against our current database and proceed accordingly.

13.1.5 Monitoring and replanning

We can check the progress of the plan in three ways:

- **Action monitoring**: Checks if the preconditions of the next actions are met.
- **Plan monitoring**: Checks whether the preconditions of the remaining plan have been met
- **Goal monitoring**: Checks whether changing the goal is appropriate

In all cases failiure requires replanning.

13.1.6 Situated planning

The most general approach to planning is situated planning. The agent interleaves planning and execution. The agent always views itself as "part of the way through the plan".

These activities include:

1. Executing a plan step
2. Monitor the state of the world
3. Fix deficiencies in the plan
4. Refine the plan in light of new information

13.2 Replanning

We can just simply replan from scratch, however it is more efficient to replan to restore the exprected state to a point where the original plan can resume execution. This is harder than the former option, but is often worth it and more closely matches a human rection and is useful when time is of the essence.