CS3243: Introduction to Artificial Intelligence **Fall 2020**

Lecture 2: August 19

Lecturers: Prof. Kuldeep S. Meel *Scribe: Ang Zheng Yong*

2.1 Administrative details

2.1.1 Consultation hours

1. Prof. Kuldeep: Tuesdays 3-4pm
2. Dr. Ler: Fridays 10-11am

2.1.2 Miscellaneous

1. Submissions due on Week 3:
   (a) Lecture notes scribing
   (b) Assignment 1
2. Projects:
   - Project 1 description and grouping will be released next week.
   - There will be a peer review component to ensure fairness. However, teams should have the following to assist in the peer review process:
     (a) Open communication channel (partners should respond to each other within 24 hours).
     (b) List of tasks to be done by each person.

2.2 Recap of Week 1

In week 1, we have covered:

1. Reflex agents
   - Passive.
   - Next state depends only on percept.
2. Model-based reflex agents
   - Passive.
   - Next state depends on the percept, and the model of the world.
3. Goal-based reflex agents
   - Active.

- Next state depends on the current state, the percept, and the model of the world.
- Typically only have a single goal.

4. Utility-based agents
   - Might have multiple conflicting goals, and the agent might not be able to achieve all of its goals.
   - Agent may assign a value (i.e. weight) to each goal, and attempt to balance them.

2.3 More on... agents

Definition 2.1 (Autonomous agents)**.**

Autonomous agents *are agents which can update its agent program based on its experiences.*

The goal of this module is to design *rational autonomous agents.*

2.3.1 How do we model a problem?

Abstraction: we can abstract away the unnecessary details in the problem, and attempt to capture the basic properties of the problem. Then, we can model these properties as a graph.

Let's assume that we have a goal-based agent, in a deterministic and fully-observable task environment, e.g. a mopbot. What will be some useful abstractions in this case?

1. State: $< L, A, B >$
   - L represents the location of the mopbot.
   - A and B represent the status of locations A and B respectively, i.e. whether they are clean or dirty.
2. Actions: {left, right, clean, idle}
   - The list of actions that can be performed by the mopbot.
3. Transition model: $g : \text{State} \times \text{Action} \to \text{State}$.
   - Difference between a transition model and an agent function:
     - Agent function: given the current state and percept, what should I do?
     - Transition model: given the current state, what will happen if I perform a certain action?
4. Performance measure (a.k.a. cost function): $h : \text{State} \times \text{Action} \to \mathbb{R}$.
5. Goal state: $< *, C, C >$, where C indicates that the location is clean.
   - The goal state may not necessarily only consist a single state.
   - In the case of the mopbot, the goal state can be both $< A, C, C >$ and $< B, C, C >$.
6. Start state

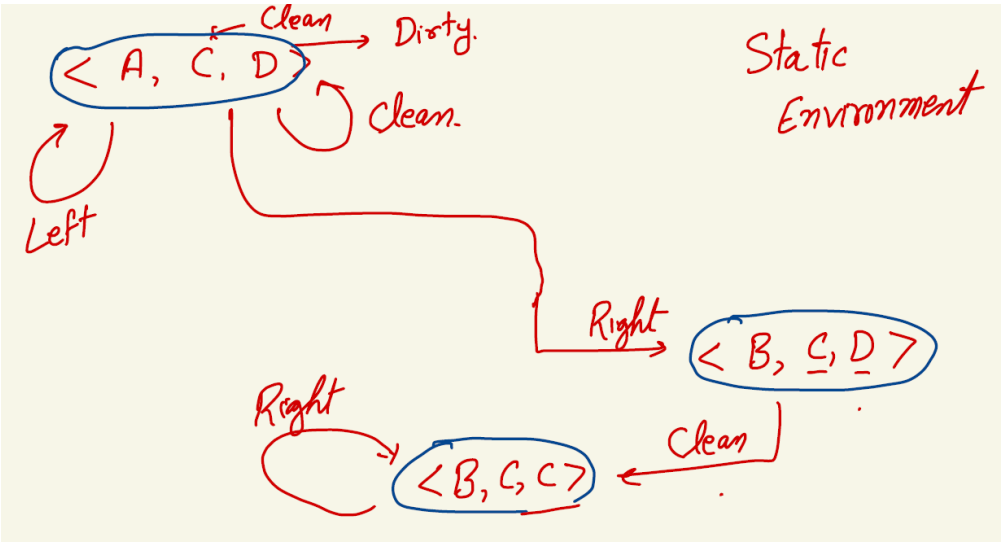

Figure 2.1: Transition model of the mopbot (incomplete)

We can illustrate the transition model using a graph, as shown above. However, the graph representing the transition model can potentially be infinitely large, even for seemingly small problems.[1]

After modelling our problem as a graph, we can now begin exploring the techniques used in graph search.

Remark: In fact, mathematics can be seen as a form of goal-based learning.

- When proving mathematical statements, we often start from an initial statement, and try to reach our final statement (complete our proof).

$$\text{Initial statement} \to \quad \to \quad \ldots \quad \to \quad \to \text{Final statement}$$

[1] Examples include the $(3n + 1)$ problem, and the Four Fours problem

2.4 Uninformed search

Let us assume that we have obtained the following graph: How can we find a path from s_0 to s_2?

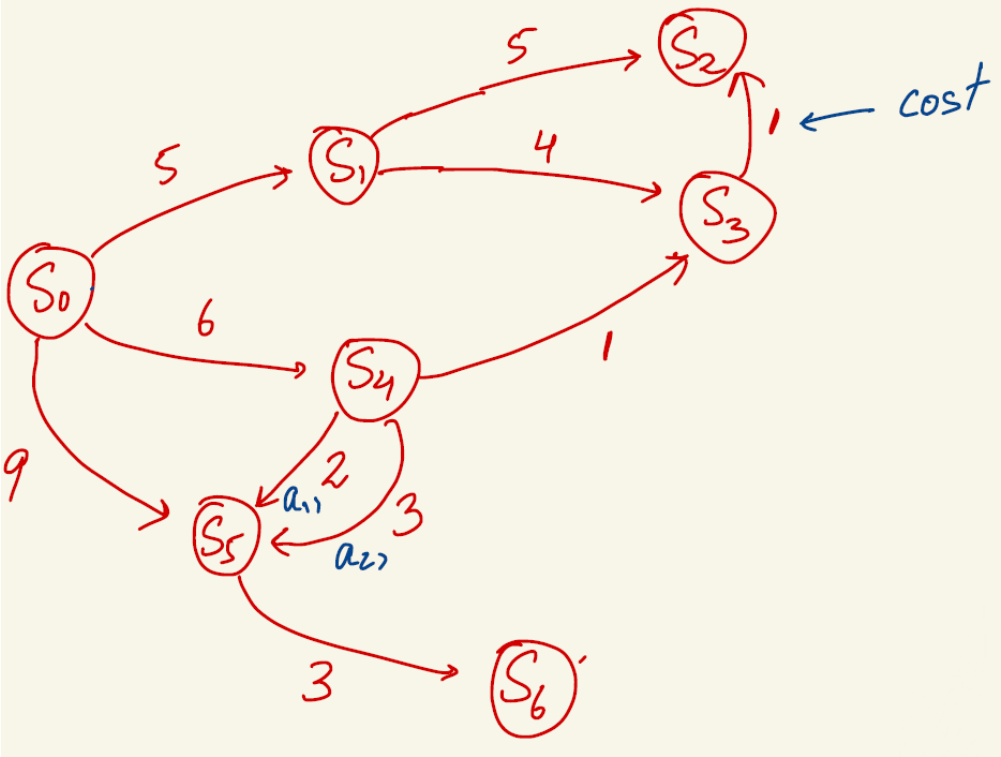

Figure 2.2: A graph with start state s_0 and end state s_2

1. Tree search:
   (a) At our current state, we review our list of performable actions.
   (b) If there is at least one performable action, we will execute the action.
   (c) If we hit a dead end, we will go back and try out another action which hasn't been executed.

   - Limitation: this algorithm makes many unnecessary repeated moves.
     – E.g. after exploring s_6 through $s_0 \to s_5 \to s_6$, we will hit a dead end and start exploring another route from s_0. Let's say we take the route $s_0 \to s_4 \to s_5 \to \ldots$. Then, we will continue traversing s_5 for a second time, even though we have already traversed it before.

2. Graph search:

   - The algorithm is similar to that of tree search, but we will now memoize the nodes that we have traverse.
     – What do we memoize?
       (a) State
       (b) Parent

(c) Action (that can be carried out at the node)
(d) Path cost

- Limitation: we will require additional memory to store our memo table (i.e. space-time tradeoff).

3. Breadth-first search

Algorithm 1 BFS

```
1: procedure BFS(u)
2:     if GoalTest(u) then
3:         return path(u)
4:     end if
5:     F ← queue(u)
6:     E ← {u}
7:     while F is not empty do
8:         u ← F.pop()
9:         for all children v in u do
10:            if GoalTest(v) then
11:                return path(v)
12:            else
13:                if v not in E then
14:                    E.add(v)
15:                    F.push(v)
16:                end if
17:            end if
18:        end for
19:    end while
20:    return FAIL
21: end procedure
```

- Limitation: not optimal.

4. Dijkstra.

2.4.1 Algorithmic analysis

How do we determine whether an algorithm is good?

1. Completeness: the algorithm will find a path to the goal, if the goal is reachable from the starting state.
2. Optimality: if the algorithm finds a path, the path will be of minimum cost.
3. Time complexity.
4. Space complexity.

Exercise: Is BFS complete?

Proof:
Assume that there is a path from the start state s_0 to the goal state s_g, i.e. $\pi : s_0, s_{i_1}, \ldots, s_{i_k}, s_g$, of length $k+1$.

Base case: when $k+1 = 0$, it means that $s_0 = s_g$.

IH: we assume that the algorithm is able to reach nodes at distance $\leq k$.

When we have reached s_{i_k}, we should have either:

- already explored s_g, i.e. s_g is along the path from s_0 to s_{i_k}, or
- s_g will be explored next.

Thus, BFS is complete.

□

Exercise: Is BFS optimal?

Proof:
No. We will proof this by contradiction.

Referring to Figure 2.2, BFS will return $s_0 \to s_1 \to s_2$, instead of $s_0 \to s_4 \to s_3 \to s_2$, which is the actual optimal path.

□

Exercise: What are the time and space complexities of BFS?

Taking b to be the branching factor of our graph, and d to be the depth (i.e. the minimum path length from the start to our goal),

- Time complexity: $b + b^2 + \ldots + b^d = O(b^{d+1})$
- Space complexity: $O(b^{d+1})$

So BFS is not optimal. How can we do better?

1. We can replace the queue used by our frontier F in Algorithm 1 with a priority queue.
   - The nodes will be ordered according to their cost from the starting node.
   - When node u is popped from F, it will be the minimum cost node that is unexplored.
2. We should not mark the child nodes as "explored" too early in Algorithm 1.
   - By marking them as "explored" early on, we will not update/relax their weights after going through them once more from another path.

CS3243: Introduction to Artificial Intelligence **Fall 2020**

Lecture 3: August 26

Lecturers: Prof. Kuldeep S. Meel *Scribe: Ang Zheng Yong*

3.1 Recap of Week 2

3.1.1 BFS

BFS is not optimal. How do we fix it?

1. We can replace the queue used by our frontier F in our BFS algorithm with a priority queue.
   - The nodes will be ordered according to their cost from the starting node.
   - When node u is popped from F, it will be the minimum cost node that is unexplored.
2. We should not mark the child nodes as "explored" to early in our BFS algorithm.
   - By marking them as "explored" early on, we will not update/relax their weights after going through them once more from another path.

3.2 Uninformed Search (continued)

3.2.1 Uniform Cost Search

After going through these modifications, we obtain a new algorithm, which we call **uniform cost search (UCS)**. Our modified algorithm will look like:

Algorithm 1 Uniform Cost Search

```
1: procedure UniformCostSearch(u)
2:     F ← PriorityQueue(u)
3:     E ← [u]                                                    ▷ a dict
4:     ĝ[u] = 0                ▷ keeps track of min. path cost of reaching u discovered so far
5:     while F is not empty do
6:         u ← F.pop()
7:         if GoalTest(u) then
8:             return path(u)
9:         end if
10:        E.add(u)
11:        for all children v of u do
12:            if v ∉ E then
13:                if v ∈ F then
14:                    ĝ[v] ← min(ĝ[v], ĝ[u] + c(u, v))
15:                else
16:                    F.push(v)
17:                    ĝ[v] ← ĝ[v] + c(u, v)
18:                end if
19:            end if
20:        end for
21:    end while
22:    return FAILURE
23: end procedure
```

Remark: UCS is different from Dijkstra in the sense that Dijkstra initializes $\hat{g}$ for all the nodes.

- However, we may have infinitely many nodes in our graph.
- This would cause us to be stuck in the initialization step, and our algorithm would not work.

3.2.2 Properties of UCS

Exercise: Is UCS complete?

Proof:
UCS is only complete when there does not exist an infinitely long chain of edges with weights 0.

Suppose we have a graph containing edges with weight 0, as seen in the following image:

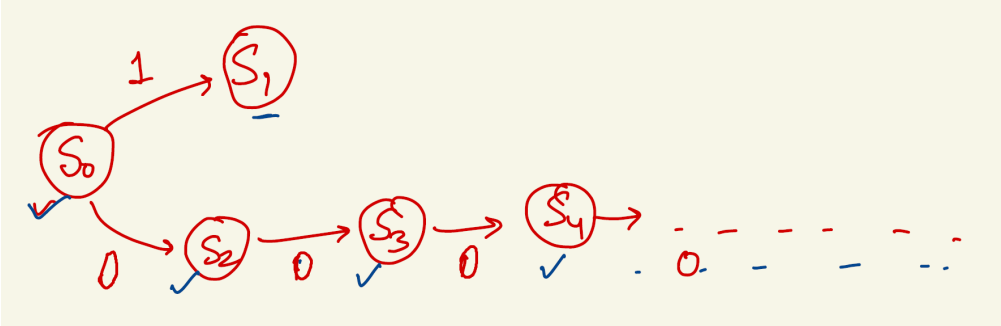

Figure 3.1: Graph with edges of weight 0

UCS would continue down this chain $(s_2, s_3, \ldots)$ containing edges of weight 0. If this chain is infinitely long, UCS would never explore s_1 in finite time.

However, suppose that all of the edges in this chain $c(s_0, s_2), c(s_2, s_3), \ldots$ have nonzero weights. That is, all the edges $c(s_0, s_2), c(s_2, s_3), \ldots$ have some weight $\epsilon > 0$. Then, at some point down the chain, the sum of the edge weights down this chain would exceed $c(s_0, s_1)$. This would then cause UCS to be able to explore s_1 and reach the goal state. Hence, UCS is complete in this case. □

Remark: An algorithm can still be complete even though there are infinite states. This is because we do not necessarily need to traverse through all the states in order to get to our goal.

Exercise: Is UCS optimal?

UCS is optimal if we can ensure that when we are performing the goal test, we have found the optimal path.

Proof:
We will prove a more general claim: when we pop a node u from F, we would have found the optimal path to u. We shall first define our notations as such:

- $g(u)$: actual min. path cost from initial node to node u.
- $\hat{g}(u)$: min. path cost from initial node to node u, based on our current knowledge of the graph.
- $\hat{g}_{\text{pop}}(u)$: the value of $\hat{g}(u)$ when we pop u from F.

Thus, we can remodel our claim into a mathematical equation, i.e. $\hat{g}_{\text{pop}}(u) = g(u)$.

Let us now consider the optimal path from the initial node s_0 to node u, say $(s_0, s_1, \ldots, s_k, u)$. We will prove our claim by induction on this path.

- Base case: $\hat{g}_{\text{pop}}(s_0) = g(s_0) = 0$. This is clearly true.

- Inductive hypothesis: For all $\{s_0, s_1, \ldots, s_k\}$, we assume that $\hat{g}_{\text{pop}}(s_i) = g(s_i)$.
- Finally,
  1. We have $\hat{g}_{\text{pop}}(u) \geq g(u)$ for all $u \in$ state space. This is clearly true since we will have $\hat{g}_{\text{pop}}(u) = g(u)$ if we have already found the optimal path to u while u is still in the priority queue, and $\hat{g}_{\text{pop}}(u) > g(u)$ if the optimal path to u has yet to be found.
  2. Furthermore, when we pop s_k,

$$\begin{aligned}
\hat{g}_{\text{pop}}(u) &= \min\{\hat{g}(u), \hat{g}_{\text{pop}}(s_k) + c(s_k, u)\} \\
&\leq \hat{g}_{\text{pop}}(s_k) + c(s_k, u) \\
&= g(s_k) + c(s_k, u) && \text{(from IH)} \\
&= g(u)
\end{aligned}$$

  3. From (1) and (2), we get $\hat{g}_{\text{pop}}(u) = g(u)$. This proves our initial claim, and thus UCS is optimal.

□

Exercise: Time and space complexities of UCS?

Solution:
Time complexity: $O(b^{d+1})$, where

- b is the branching factor of the recursion tree, and
- d is the depth of the recursion tree (from root to goal state).

Space complexity: $O(b^{d+1})$ □

Remark: Is there any way to use less space than $(O(b^{d+1}))$?
Yes, by using UCS with tree search.

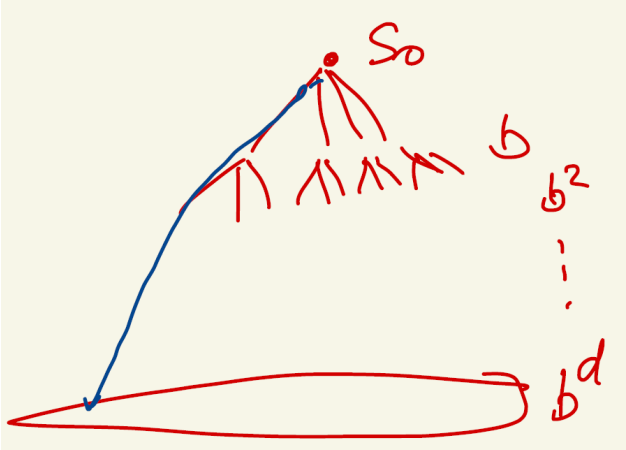

Figure 3.2: Visual representation of tree search

This way, we will only have to use $O(bd)$ space to keep track of the frontier. We can implement this by replacing the queue in Algorithm 1 with a stack. However, there will be tradeoffs if we decide to use UCS with tree search instead of UCS with graph search. For instance, we will lose optimality and completeness.

- With tree search, the graph may keep going down a single path, and never get to the goal.

Remark: There will always be tradeoffs.

- If we want to ensure completeness, the space complexity of our algorithm will be high.
- If we want to minimize our space consumption, we would have to sacrifice completeness.
  - However, on some occasions, it is perfectly fine to implement an algorithm that is not complete (e.g. when there are finite states).

3.2.3 BFS, DFS, and UCS

So far, we have gone through various algorithms which explore our state space without any additional information provided to them (i.e. uninformed search algorithms). Sometimes, these algorithms may perform redundant moves.

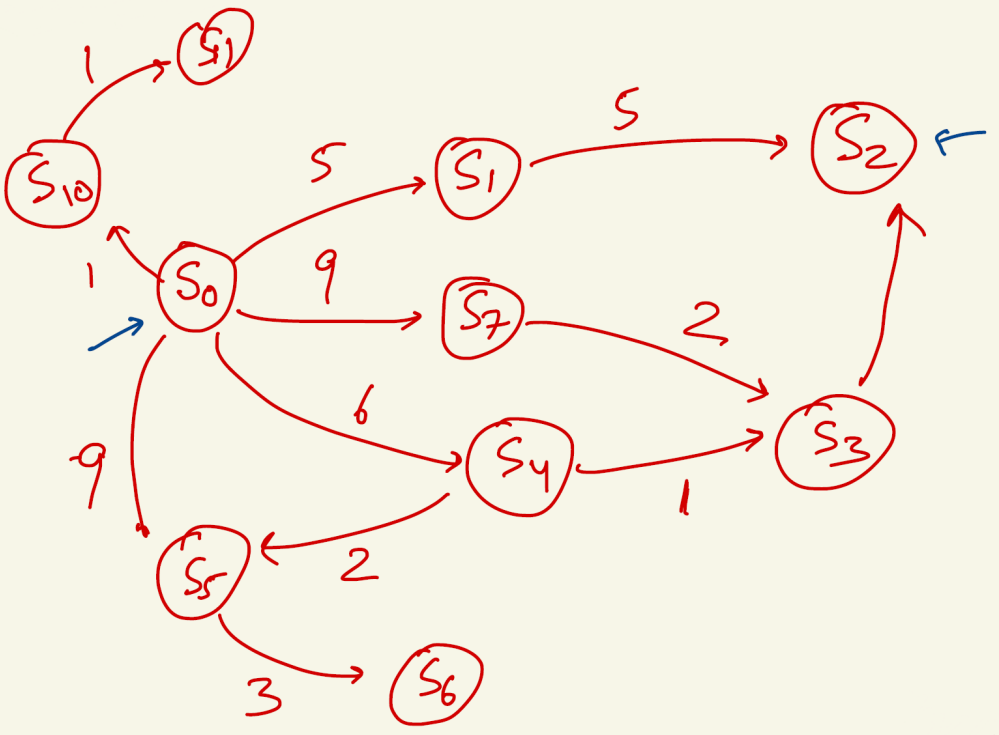

Figure 3.3: Graph with initial state s_0 and goal state s_2

For instance, when we run UCS on this graph, UCS will first traverse $s_0 \to s_{10} \to s_1$, even though the state s_1 is not along the path from s_0 to s_2. If we have some information (e.g. heuristics), our algorithms might be able to make better decisions.

3.3 Informed Search

3.3.1 Heuristics

We shall begin by defining what a heuristic is. Heuristics will be useful in helping us make more informed decisions in our search process.

Definition 3.1 (Heuristic)**.**

A heuristic, denoted as $h(u)$, is the estimated cost from a state u to the goal state.

3.3.2 A^* search

A^* search is basically UCS, but at each iteration, we will pop the node with the smallest $\hat{f}(u) = \hat{g}(u) + h(u)$ instead of the node with the smallest $\hat{g}(u)$. The algorithm for A^* is as follows:

Algorithm 2 A^* Search

```
1: procedure A*SEARCH(u)
2:     F ← PriorityQueue(u)                ▷ Prioirity queue should be implemented with f̂
3:     E ← [u]                             ▷ a dict
4:     ĝ[u] = 0
5:     while F is not empty do
6:         u ← F.pop()
7:         if GOALTEST(u) then
8:             return path(u)
9:         end if
10:        E.add(u)
11:        for all children v of u do
12:            if v ∉ E then
13:                if v ∈ F then
14:                    ĝ[v] ← min(ĝ[v], ĝ[u] + c(u, v))
15:                    f̂[v] ← ĝ[v] + h(v)
16:                else
17:                    F.push(v)
18:                    ĝ[v] ← ĝ[v] + c(u, v)
19:                    f̂[v] ← ĝ[v] + h(v)
20:                end if
21:            end if
22:        end for
23:    end while
24:    return FAILURE
25: end procedure
```

Exercise: Is A^* optimal?

If we can ensure that for an optimal path $s_0, s_1, \ldots, s_k, u$:

$$\hat{f}_{\text{pop}}(s_0) \leq \hat{f}_{\text{pop}}(s_1) \leq \ldots \leq \hat{f}_{\text{pop}}(u) \tag{3.1}$$

and

$$\hat{f}_{\text{pop}}(s_i) = f(s_i) \tag{3.2}$$

where $f(s_i) = g(s_i) + h(s_i)$, then we would have shown that A^* is optimal.

Proof:
If $f(s_0) \leq f(s_1) \leq \ldots \leq f(s_k) \leq f(u)$ and equation 3.2 both hold, then equation 3.1 would hold, and thus we will have shown that A^* is optimal.

Let us pick some arbitrary s_i, where $f(s_i) \leq f(s_{i+1})$. This would imply that:

$$\begin{aligned}
& g(s_i) + h(s_i) \leq g(s_{i+1}) + h(s_{i+1}) \\
\implies\ & h(s_i) \leq g(s_{i+1}) - g(s_i) + h(s_{i+1}) \\
\implies\ & h(s_i) \leq c(s_i, s_{i+1}) + h(s_{i+1}) && \text{(this implies a } \textbf{consistent heuristic}\text{)} \\
& \qquad \leq c(s_i, s_{i+1}) + c(s_{i+1}, s_{i+2}) + h(s_{i+2}) \\
& \qquad \vdots \\
& \qquad \leq c(s_i, s_{i+1}) + \ldots + c(s_k, u) + h(u) && \text{(where } h(u) = 0\text{)} \\
\implies\ & h(s_i) \leq OPT(s_i) && \text{(this implies a } \textbf{admissible heuristic}\text{)}
\end{aligned}$$

where $OPT(s_i)$ refers to the optimal path cost from s_i to the goal state.

Hence, if we are able to obtain a heuristic function h that satisfies the properties of **consistency** and **admissibility**, then our algorithm will be optimal.

□

Exercise: The value of $f(s_i)$ along *any path* from s_0 to u is:

Solution:
Neither increasing nor decreasing. However, the *optimal path* from s_0 to u will have nondecreasing values of $f(s_i)$. □

CS3243: Introduction to Artificial Intelligence **Fall 2020**

Lecture 4: September 2

Lecturers: Prof. Kuldeep S. Meel *Scribe: Ang Zheng Yong*

4.1 Recap of Week 3

4.1.1 BFS, UCS, and A^*

Last week, we began with discussing BFS and UCS, and we came to the conclusion that we will be able to do better than these uninformed search algorithms if we are able to get some information about the goal. This led us to the discussion of A^* search.

4.1.2 Consistency and admissibility

We have shown that if $f(s_i) \leq f(s_{i+1})$, where:

- $f(s_i) = g(s_i) + h(s_i)$,
- $g(s_i)$ represents the minimum path cost from the starting node s_0 to node s_i, and
- $h(s_i)$ represents the estimated cost from node s_i to the goal node.

then A^* will be optimal.

In our proof, we have also discussed the conditions for consistent and admissible heuristics. Assuming that the optimal path from the starting node s_0 to some goal node s_g is $(s_0, s_1, \ldots, s_i, s_{i+1}, \ldots, s_g)$,

- A heuristic is **consistent** if $h(s_i) \leq c(s_i, s_{i+1}) + h(s_{i+1})$, where $c(s_i, s_{i+1})$ represents the cost to get from s_i to s_{i+1}.
- Additionally, if $h(s_g) = 0$, then $h(s_i) \leq OPT(s_i)$, where $OPT(s_i)$ represents the optimal path cost from s_i to s_g. This would imply that the heuristic is **admissible**.
- Consistency implies admissibility, but admissibility does not necessarily imply consistency.

We then concluded with the following theorem.

Theorem 4.1. *If h is consistent, then A^* with graph search is optimal.*

4.2 Optimal Heuristics

Currently, we do not have a sure-fire method of coming up with consistent heuristics. However, it is possible to design admissible heuristics. Admissible heuristics are sufficient for optimality if we are using A^* with tree search.

Theorem 4.2. *If h is admissible, then A^* with tree search is optimal.*

4.2.1 Designing admissible heuristics

Idea: find an optimal path from s_i to s_g that respects the restrictions of the problem (e.g. obstacles), and relax some of the restrictions so that $h(s_i)$ is **computable** and **informative**.

- Computable: the heuristic can be computed in finite time.
- Informative: the heuristic should provide us with information which would assist in speeding up our search.
  - A heuristic which gives $h(s_i) = 0$ for all $i = 0, 1, 2, ...$ is considered to be uninformative, as it does not provide any information that helps to speed up our search.

4.2.1.1 Heuristic 1: Euclidean/Straight-line distance

Problem: find the shortest path from point s_0 to point s_g in NUS.

By ignoring the presence of obstacles/unwalkable terrain in our path (relaxing the restrictions), we can set $h(s_i) = d(s_i, s_g)$, where $d(s_i, s_g)$ represents the Euclidean distance from s_i to s_g.

This heuristic is admissible since $h(s_i) \leq OPT(s_i)$.

- If the path from s_i to s_g is a straight line, then $h(s_i) = OPT(s_i)$.
- Else, if the path is not a straight line, then $h(s_i) < OPT(s_i)$.

This heuristic is also consistent.

1. By the triangle inequality, $d(s_i, s_g) \leq d(s_i, s_{i+1}) + d(s_{i+1}, s_g)$.
2. Also, $d(s_i, s_{i+1}) \leq c(s_i, s_{i+1})$, where $c(s_i, s_{i+1})$ represents the cost of getting from s_i to s_{i+1}. This is the case since $d(s_i, s_{i+1}) = c(s_i, s_{i+1})$ when the path from s_i to s_{i+1} is straight, and $d(s_i, s_{i+1}) < c(s_i, s_{i+1})$ when the path is curved.
3. From the definition of $h(s_i) = d(s_i, s_g)$, it follows that $h(s_i) \leq c(s_i, s_{i+1}) + h(s_{i+1})$. This implies that the heuristic is consistent.

4.2.1.2 Heuristic 2: Manhattan distance

Problem: find the shortest path from point s_0 to point s_g in a matrix with obstacles.

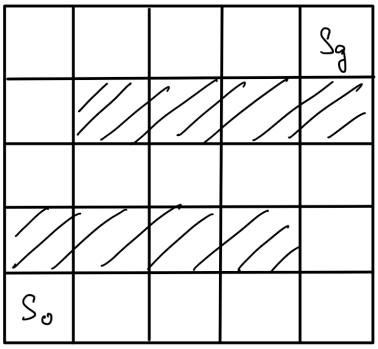

Figure 4.1: A matrix with start state s_0 and goal state s_g. Shaded boxes indicate obstacles.

With the additional restriction of only being able to move vertically and horizontally, the Manhattan distance would be a better heuristic than the Euclidean distance. This is because more restrictions of the original problem are being respected, thus the heuristic would give a more accurate representation of the cost from s_i to s_g.

This heuristic is admissible because:

- If the path from s_i to s_g does not have an obstacle in between, then $h(s_i) = OPT(s_i)$.
- If the path from s_i to s_g has an obstacle in between, then $h(s_i) < OPT(s_i)$.

4.3 Greedy Breadth First Search

How is it different from the other search algorithms?

- UCS uses $\hat{g}(u)$ to make decisions on which node to visit next, where $\hat{g}(u)$ represents the minimum path cost of reaching u (discovered so far).
- A^* search uses $\hat{g}(u) + h(u)$ to make decisions on which node to visit next.
- Greedy BFS uses only $h(u)$ to make decisions on which node to visit next.

Exercise: Greedy BFS is not optimal.

Proof:
We will prove by contradiction. Let us assume that we have a graph as shown below:

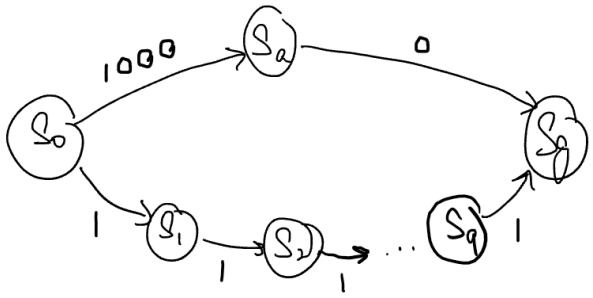

Figure 4.2: Graph with start state s_0 and goal state s_g

Since $c(s_a, s_g) = 0$ while $c(s_9, s_g) = 1$, consistent or admissible heuristic functions would likely produce a value of $h(s_i)$ such that $h(s_9) > h(s_a)$, even though $c(s_0, s_a) + c(s_a, s_g) = 1000 > c(s_0, s_1) + c(s_1, s_2) + \ldots + c(s_9, s_g) = 10$. However, since greedy BFS does not take $\hat{g}(s_i)$ into consideration, it will pick the path $s_0 \rightarrow s_a \rightarrow s_g$. Hence, greedy BFS is not optimal. □

4.4 Adversarial Search

In this section, we will be visiting problems involving multiple agents which will compete with each other, in the form of games.

4.4.1 Game 1: Bags and balls

Suppose that we have a game involving two bags, containing two balls each. We will be denoting the bags and balls as such (my notation differs slightly from what was delivered in lecture):

- B_1: bag 1.
- B_2: bag 2.
- $b_{1,1}$: ball 1 in bag 1.
- $b_{1,2}$: ball 2 in bag 1.
- $b_{2,1}$: ball 1 in bag 2.
- $b_{2,2}$: ball 2 in bag 2.

We will be playing as player *MAX*, and our goal will be to maximize our score. Our opponent will be playing as player *MIN*, and their goal will be to minimize our score (and thus maximizing their score).[1]

We can construct our game tree as such:

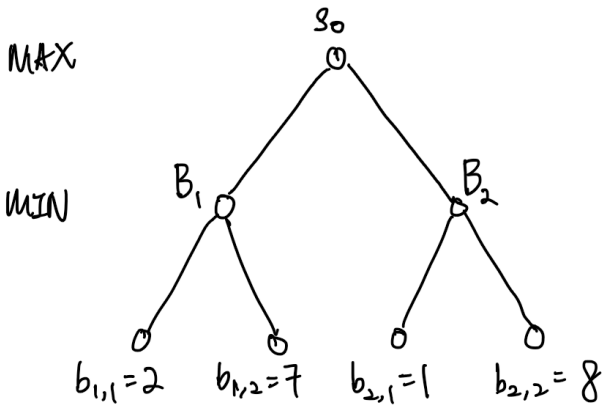

Figure 4.3: Game tree for bags and balls

In this graph, s_0 is our start state, whereas $b_{i,j}$ are the terminal states. We will also denote the branching factor of the graph by b, and depth by d. The number of possible states will be b^d.

We will examine what happens when both agents *MAX* and *MIN* plays optimally,

[1] This is commonly known as a *zero-sum game*.

- If B_1 is selected, *MIN* will pick $b_{1,1} = 2$ to minimize *MAX*'s score. If B_2 is selected instead, *MIN* will pick $b_{2,1} = 1$.
- Anticipating what *MIN* would play in both cases, *MAX* will select B_1 in order to maximize our score.

4.4.2 Game 2: Chess

Chess can also be modelled as a game. Here, our utility can be defined as follows:

- 1, if white wins.
- -1, if black wins.
- 0, if the game ends in a draw.

The objective of chess is the same as in Game 1, where we are aiming to maximize our utility, and our opponent is aiming to minimize our utility (and thus maximizing theirs).

However, the number of possible states is approximately 10^{120}, which cannot possibly be computed by modern computers.

4.4.3 Minimax function

In games, we can use the minimax function to help us get the optimal solution (the process is similar to that described in Game 1). The minimax function is recursively defined as such:

$$Minimax(s) = \begin{cases} Utility(s) & \text{if } Terminal(s) = 1 \\ \max\limits_{a \in \text{actions}} Minimax(Result(s, a)) & \text{if player} = MAX \\ \min\limits_{a \in \text{actions}} Minimax(Result(s, a)) & \text{if player} = MIN \end{cases}$$

- $Utility(s)$ represents the utility gained at state s.
- $Terminal(s)$ is a test that returns 1 if the node is a terminal node, and 0 otherwise.
- $Result(s, a)$ defines the resulting state when we perform an action a from state s.

4.4.4 Alpha-beta pruning

Since some game trees may be very large (such as the game tree for chess), is there any way to reduce the number of branches that we need to search?

Let us consider an example:

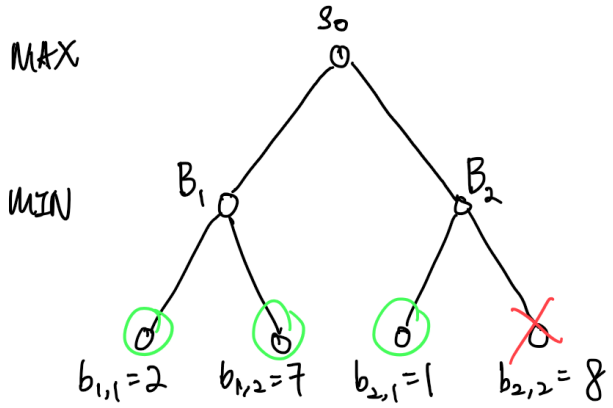

Figure 4.4: Game tree for bags and balls, with alpha-beta pruning

- We will explore this tree from left to right.
- We will first visit B_1.
  - Starting from the left, we will explore $b_{1,1} = 2$. Hence, MIN will be able to secure a score of ≤ 2.
  - MIN will then explore $b_{1,2} = 7$. However, since $b_{1,1} < b_{1,2}$, MIN will pick $b_{1,1} = 2$.
- After exploring the left subtree, MAX is able to guarantee a score of ≥ 2.
- We will then explore B_2 next.
  - Starting from the left again, we will first explore $b_{2,1} = 1$. hence, MIN will be able to secure a score of ≤ 1.
  - Now, if $b_{2,2} > b_{2,1}$, we know that MIN will select $b_{2,1} = 1$. On the other hand, if $b_{2,2} < b_{2,1} < b_{1,1} = 2$, we know that MAX will not pick from the right subtree. Hence, there is no need to check the value of $b_{2,2}$.

Remark: The order of evaluation of the nodes matter for alpha-beta pruning.

If the nodes were evaluated in the order $b_{1,1} \to b_{1,2} \to b_{2,2} \to b_{2,1}$ instead, then we will need to check the value of the last node. This is because $b_{2,2} = 8 > b_{1,1}$, and thus, if the last node has a value of > 2, MAX will pick from the right subtree instead.

4.5 Exercise

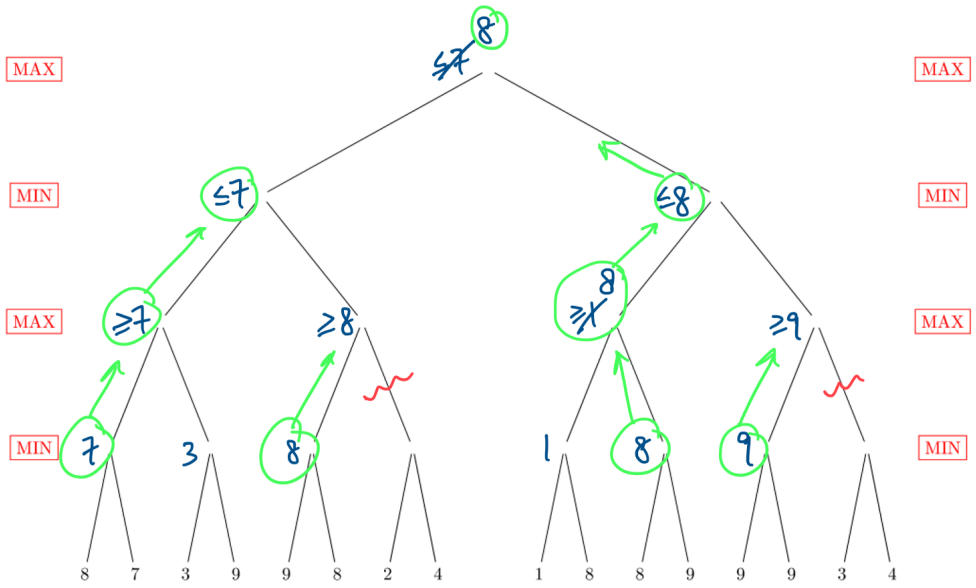

Figure 4.5: Exercise for Lecture 4

Using alpha-beta pruning, we do not need to explore the subtree containing the nodes (2, 4) and the subtree containing the nodes (3, 4).

CS3243: Introduction to Artificial Intelligence **Fall 2020**

Lecture 5: September 9

Lecturers: Prof. Kuldeep S. Meel *Scribe: Ang Zheng Yong*

5.1 Recap of Week 4

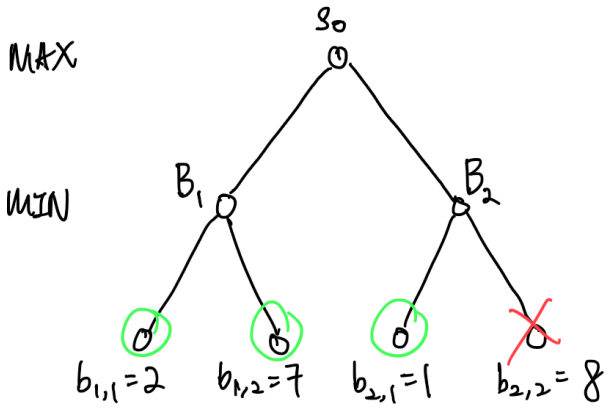

Figure 5.1: A game of bags and balls

We evaluated the balls from left to right in an orderly manner.

1. When the first ball $b_{1,1}$ is evaluated, player *MIN* knows that he is able to get a value of ≤ 2.
2. When the second ball $b_{1,2}$ is evaluated, player *MIN* would definitely select $b_{1,1} = 2$, and here player *MAX* knows that he is able to get a value of ≥ 2.
3. When the third ball $b_{2,1}$ is evaluated, player *MIN* knows that he is able to get a value of ≤ 1. Here, player *MAX* knows that it definitely does not want to be taking the right branch, since the left branch would give a greater utility ($2 \geq 1$).
    - Hence, we can ignore the last ball, and from the perspective of player *MAX*, we can treat it as if the right branch did not exist.

5.2 Alpha-Beta Pruning (Continued)

We begin this section by discussing the exercise provided last week.

5.2.1 Sample game tree

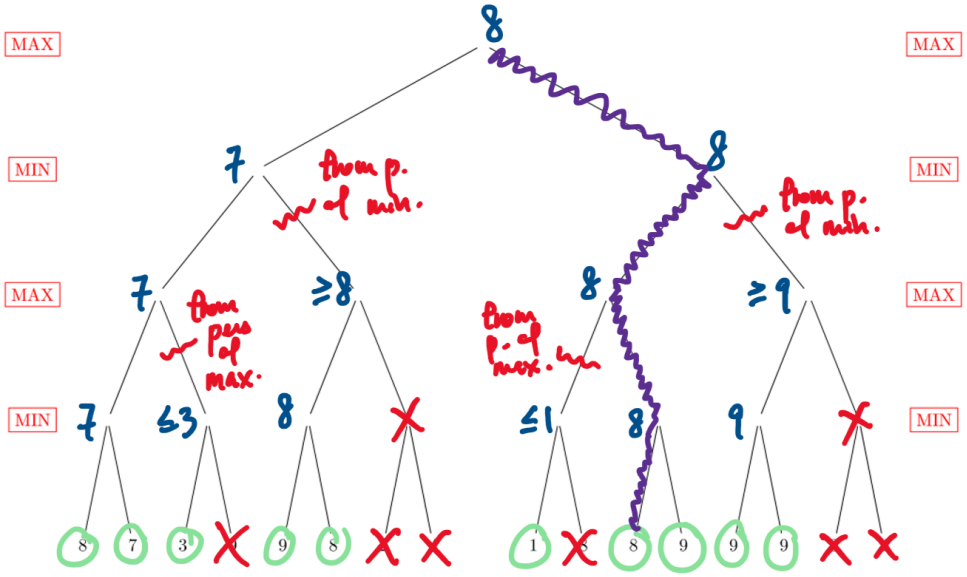

Figure 5.2: A game tree with 16 terminal nodes

Similar to the game tree for bags and balls, we will be evaluating the nodes of this tree from left to right, in an orderly fashion.

1. When the first leaf node (with *Utility* $= 8$) is evaluated, *MIN* knows that it is able to get a value of ≤ 8. After evaluating the second leaf node (with *Utility* $= 7$), *MIN* will select 7 instead.
2. When the third leaf node (with *Utility* $= 3$) is evaluated, *MIN* is able to get a value of ≤ 3 from this branch, and *MAX* knows that there is no point picking from this branch. Hence, we do not need to check the value of the fourth leaf node, and we will treat it as if this branch never existed.
3. Moving up the tree, *MIN* on the layer 2 knows that it is able to secure a value of ≤ 7. Following the reasoning in (2), we will be able to prune nodes 7 and 8 from our tree as well.
4. After repeated exploring and pruning, we will be able to end up with the game tree as shown above. The path marked in purple indicates our **strategy**, that is, it specifies *MAX*'s move in the initial state, followed by *MIN*'s move in the following state, followed by *MAX*'s subsequent move, etc.
    - More generally, a strategy is a function that maps from a set of states to a set of actions, i.e. $f:$ states $\rightarrow$ actions.

5.2.2 Minimax algorithm with alpha-beta pruning

Recall our minimax algorithm prior to introducing alpha-beta pruning:

Algorithm 1 Minimax

```
1: procedure MaxVal(s)
2:     if Terminal(s) == True then
3:         return Utility(s)
4:     end if
5:     v ← −∞
6:     for each a in Actions(s) do
7:         result ← Result(s, a)
8:         v ← max(v, MinVal(result))
9:     end for
10:    return v
11: end procedure
```

So, what is the meaning of α and β in the context of alpha-beta pruning?

- α: best value from the perspective of *MAX*.
- β: best value from the perspective of *MIN*.

By incorporating α and β into our minimax algorithm, we will be able to update our minimax algorithm (as shown on the next page).

We can then call the algorithm on our game tree as shown in Figure 5.2 with $MAXVAL(\text{root}, -\infty, \infty)$.

- The best value for *MAX* is initialized to $-\infty$.
- The best value for *MIN* is initialized to ∞.

Algorithm 2 Minimax with $\alpha - \beta$ pruning

```
1: procedure MaxVal(s, α, β)
2:     if Terminal(s) == True then
3:         return Utility(s)
4:     end if
5:     v ← −∞
6:     for each a in Actions(s) do
7:         result ← Result(s, a)
8:         v ← max(v, MinVal(result, α, β))
9:         if v ≥ β then                          ▷ condition for pruning
10:            return v
11:        end if
12:        α ← max(α, v)
13:    end for
14:    return v
15: end procedure
16:
17: procedure MinVal(s, α, β)
18:    if Terminal(s) == True then
19:        return Utility(s)
20:    end if
21:    v ← +∞
22:    for each a in Actions(s) do
23:        result ← Result(s, a)
24:        v ← min(v, MaxVal(result, α, β))
25:        if v ≤ α then                          ▷ condition for pruning
26:            return v
27:        end if
28:        β ← min(β, v)
29:    end for
30:    return v
31: end procedure
```

5.2.3 Space complexity of minimax

Considering a game tree with depth d,

- The original minimax algorithm would require a space of $O(b^d)$.
- Minimax with alpha-beta pruning would require a space of $O(b^{d/2})$.
  - This space consumption depends heavily on the order in which the nodes are visited.
  - However, under most circumstances, we would be able to get a space comsumption of just $O(b^{d/2})$.

Even though we have managed to reduce the branching factor of our tree from b to $\sqrt{b}$, we would still need to go through $35^{100/2} = 35^{50}$ distinct states for a simple game of chess. Is there any way for us to do better than this?

5.2.4 Improvements on minimax

Idea: At every iteration of our algorithm, we only explore nodes up to a depth l instead of exploring the entire tree (which has depth d). We will also need to address the following issues:

1. How do we pick l?
   - We start with $l = 1$, then $l = 2$, and so on.
   - At every depth, we will keep track of the best value known as far.
     - If we have more time to perform additional computations, we can then evaluate deeper levels.
   - This is also known as **iterative deepening**.
2. How do we know the cost of going down this path?
   - Only terminal nodes have a utility associated with them; the utility of intermediate nodes was previously derived from the utility of their child nodes (e.g. refer to Figure 5.1 and 5.2).
   - Hence, we use an **evaluation function** as a heuristic for guiding our branch selection.
     - An evaluation function is defined as: $f_{\text{eval}}(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s) = \sum_{i=1}^{n} w_i f_i(s)$, where:
       - f_i is a feature of the position, and
       - w_i is a weight associated with f_i.
         - w_i can be calculated through learning, or defined by humans.
       - For example, in chess, we *may* define $f_1(s)$ to be the number of bishops present, $f_2(s)$ to be the number of knights present, etc.
     - In our minimax algorithm, we will substitute $\mathit{Utility}(s)$ with $f_{\text{eval}}(s)$.

5.3 Local Search

Let us now head back to problems involving single agents. Particularly, we will be examining the n-queens puzzle.

5.3.1 n-queens puzzle

The objective of this problem is to arrange n queens on an n-by-n chessboard, such that the n queens won't attack each other. We can model our problem as a local search problem with:

1. State s.
2. $N(s)$: the neighbours of state s.
3. $Val(s)$: the value associated with state s.
   - As far as possible, $Val(s)$ should reflect the notion of "quality".
   - If s_g is a goal state, then $Val(s_g) = 0$.

For the n-queens puzzle, we can define $Val(s)$ to be the number of pairs of queens that are attacking each other in state s.

5.3.2 Hill climbing algorithm

After modelling the puzzle as a local search problem, how can we solve the problem? One way would be to use the hill climbing algorithm:

Algorithm 3 Single iteration of hill climbing

```
1:  procedure HillClimbing(s)
2:      minVal ← Val(s)
3:      minState ← ∅
4:      for u ∈ N(s) do
5:          if Val(u) < minVal then
6:              minState = u
7:              minVal = Val(u)
8:          end if
9:      end for
10:     return minState
11: end procedure
```

To get from the goal state from the initial state, we can recursively call $HILLCLIMBING(s)$ on *minState*, and terminate when *minState* is the goal state.

However, the hill climbing algorithm may not always lead to the goal state (it may get stuck in some local optima or a plateau, where neighbouring states have equally or less optimal $Val(s)$ than the current state). One example of such as state in shown below:

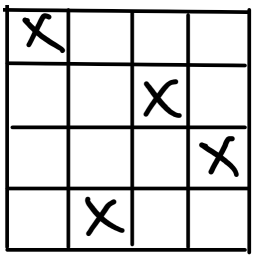

Figure 5.3: A state in the 4-queens puzzle, where "×" denotes the locations of the queens

In the state above, it can be observed that none of its neighbouring states would give a $Val(s)$ which is better than the current $Val(s)$, and so we are stuck in a plateau. How do we deal with cases such as this?

- Idea: Our algorithm should be able to permit minor mistakes once in a while. Then, we might be able to get to the goal state (i.e. global optima).
  - For instance, in the figure above, even though moving the queen in the upper-right corner down by 1 step would result in $Val(s)$ remaining as 1, it would actually lead us closer to the goal state.

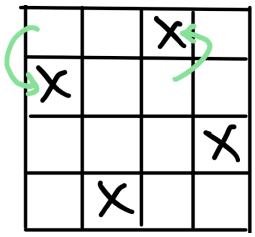

Figure 5.4: A goal state for the 4-queens puzzle

However, our algorithm shouldn't tolerate mistakes that are too bad (e.g. a step that results in $Val(s)$ increasing by more than 2). How could we design an algorithm which accepts some mistakes which are not too bad, while rejecting others?

- Suggestion: After the for loop in Algorithm 3, we can loop through all the neighbours of s for a second time, and return all the states with $Val(minState) + \alpha$ instead of just $minState$, where α is a constant that represents the amount of increase in $Val(s)$ that is still acceptable. This would result in a longer runtime of our algorithm, but prevent our algorithm from getting stuck.

CS3243: Introduction to Artificial Intelligence **Fall 2020**

Lecture 6: September 16

Lecturers: Prof. Kuldeep S. Meel *Scribe: Ang Zheng Yong*

6.1 Recap of Week 5

6.1.1 n-queens puzzle

We used local search, where we take a state and then move towards the neighbours of this state with a more optimal $Val(s)$. However, this algorithm might get stuck in some local optimum.

6.2 Simulated Annealing

This idea came from metallurgy, where in high temperatures, particles are able to move from a location C_1 to C_2 easily when $E(C_2) > E(C_1)$, i.e.

$$Pr\{C_1 \to C_2\} \propto e^{-\frac{E(C_2)-E(C_1)}{k_B T}}$$

6.2.1 Generic algorithm for simulated annealing

Algorithm 1 Simulated Annealing

```
1: procedure SimulatedAnnealing(s)
2:     for t = 1 to ∞ do
3:         s' ← a random neighbour of s
4:         if Val(s') = 0 then
5:             return s'
6:         end if
7:         s ← s' with Pr ∝ e^{-(Val(s')-Val(s))/(k_B T(t))}   ▷ k_B is the Boltzmann constant, and T(t) is a decreasing function on t
8:     end for
9: end procedure
```

The general idea is to allow the algorithm to make mistakes early on, and punish it later (lower probability of moving to its neighbours when t is large).

The algorithm can be modified in a number of ways to give rise to various variants:

1. The constant factor used for the proportionality in line 7 may vary (the sigmoid function $\frac{1}{1+e^{-x}}$ is most commonly used).
2. The mechanism of selecting s' in line 3 may vary in different implementations as well.
3. Line 7 may also be replaced with the following:

- If $Val(s') < Val(s)$, $s \leftarrow s'$.
- Else, $s \leftarrow s'$ with $Pr \propto e^{-\frac{Val(s')-Val(s)}{k_B T}}$.

6.2.2 Back to the n-queens problem

In the n-queens problem, $Val(s)$ represented the number of queens that are attacking each other.

By using Algorithm 1, we will explore neighbouring states even when $Val(s_{i+1}) > Val(s_i)$ with a probability as defined in the algorithm. Thus, we might be able to get out of the local optima (as shown in the figure below) and approach the global optima (i.e. the state where none of the queens are attacking each other).

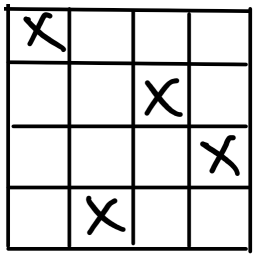

Figure 6.1: A state in the 4-queens puzzle, where "×" denotes the locations of the queens

However, there is still no guarantee of finding the global optima, since the simulated annealing algorithm is probabilistic. Is there any other approach which we could take?

6.3 Constraint Satisfaction Problems (CSPs)

We could define the 4-queens problem as a CSP:

1. Variables: $\{x_1, x_2, x_3, x_4\}$
2. Values/Domain for each of the variables: $\{D_1, D_2, D_3, D_4\}$, where each D_i represents the domain of variable x_i.
3. Constraints: NoAttack(x_1, x_2), NoAttack(x_1, x_3), ..., NoAttack(x_3, x_4).
    - We can represent the outcomes of NoAttack(x_i, x_j) in a truth table.

6.3.1 Solving a CSP

After modelling the problem as a CSP, we shall attempt to solve it. The key steps in solving a CSP include:

1. Select a value for the first variable.
2. Select a value for that next variable, that is consistent with the choices so far.
3. If we hit a dead end, backtrack. Else, repeat step 2.

For instance, let us consider the 4-queens problem:

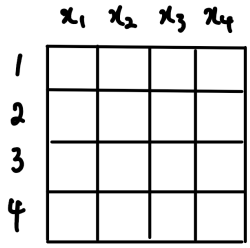

Figure 6.2: Start state of the 4-queens problem

1. We start by setting $x_1 = 1$.
   (a) Then, the domain of x_2 will be restricted to $\{3, 4\}$.
   (b) Assume we set $x_2 = 3$.
      i. Then, the domain of x_3 will be $\{\varnothing\}$. We have hit a dead end, so we backtrack.
   (c) Now, we set $x_2 = 4$.
      i. Then, the domain of x_3 will be restricted to $\{2\}$.
      ii. We set $x_3 = 2$.
         A. Then, the domain of x_4 will be $\{\varnothing\}$. We hit another dead end, so we backtrack.
2. Next, we set $x_1 = 2$.
   (a) Since the domain of x_2 is restricted to $\{4\}$, we set $x_2 = 4$.
      i. The domain of x_3 is restricted to $\{1\}$, so we set $x_3 = 1$.
         A. Finally, the domain of x_4 is $\{3\}$, and we set $x_4 = 3$ and obtain a valid solution.

The algorithm for solving the CSP is shown below.

Algorithm 2 Backtrack Search

```
1: procedure BacktrackSearch(prob, assign)
2:     if all var in (prob, assign) are assigned then
3:         return assign
4:     end if
5:     var ← unassigned variable in (prob, assign)
6:     for val ∈ OrderDomainValue(var, prob, assign) do
7:         if val is consistent with assign then
8:             add {var = val} to assign
9:             result ← BacktrackSearch(prob, assign)
10:            if result ! = failure then
11:                return result
12:            end if
13:            remove {var = val} from assign                ▷ Backtracking step
14:        end if
15:    end for
16:    return failure
17: end procedure
```

6.3.2 Inference

Is it possible to avoid making the unnecessary moves that the algorithm shown in the previous page would have made?

Idea: Once we have assigned a value to a variable x_i, we look at all the constraints that x_i appears in.

- From there, we will infer the restrictions on the remaining variables.
- If one of the variables cannot have a value, then we should immediately backtrack.

We can improve our algorithm via inference.

Algorithm 3 Backtrack Search

```
1:  procedure BacktrackSearch(prob, assign)
2:      if all var in (prob, assign) are assigned then
3:          return assign
4:      end if
5:      var ← unassigned variable in (prob, assign)
6:      for val ∈ OrderDomainValue(var, prob, assign) do
7:          if val is consistent with assign then
8:              add {var = val} to assign
9:              inference ← Infer(var, prob, assign)                ▷ Inference step
10:             add inference to assign
11:             if inference != failure then
12:                 result ← BacktrackSearch(prob, assign)
13:                 if result ! = failure then
14:                     return result
15:                 end if
16:             end if
17:             remove {var = val} and inference from assign        ▷ Backtracking step
18:         end if
19:     end for
20:     return failure
21: end procedure
```

Remark: Whenever the algorithm is doing inference, it is inferring under the current *assign*. Hence, whenever it backtracks, it will need to remove *inference* from *assign*.

CS3243: Introduction to Artificial Intelligence **Fall 2020**

Lecture 7: October 7

Lecturer: Prof. Kuldeep S. Meel *Scribe: Ang Zheng Yong*

7.1 Recap of Week 6

In week 6, we discussed backtracking search, where we first pick a variable and assign a value to it, and then check if the assigned value satisfies the constraints. The algorithm was given as:

Algorithm 1 Backtrack Search

```
1:  procedure BacktrackSearch(prob, assign)
2:      if all var in (prob, assign) are assigned then
3:          return assign
4:      end if
5:      var ← unassigned variable in (prob, assign)
6:      for val ∈ OrderDomainValue(var, prob, assign) do
7:          if val is consistent with assign then
8:              add {var = val} to assign
9:              result ← BacktrackSearch(prob, assign)
10:             if result != Failure then
11:                 return result
12:             end if
13:             remove {var = val} from assign                ▷ Backtracking step
14:         end if
15:     end for
16:     return Failure
17: end procedure
```

Then, we moved further to discuss the possibility of inference on future moves, so that we could potentially avoid making unnecessary moves (i.e. moves that lead to a dead end) that the naïve backtracking algorithm would have made. The algorithm for backtracking with inference is shown on the next page.

Algorithm 2 Backtracking with Inference

```
1: procedure BacktrackSearch(prob, assign)
2:     if all var in (prob, assign) are assigned then
3:         return assign
4:     end if
5:     var ← unassigned variable in (prob, assign)
6:     for val ∈ OrderDomainValue(var, prob, assign) do
7:         if val is consistent with assign then
8:             add {var = val} to assign
9:             inference ← Infer(var, prob, assign)                ▷ Inference step
10:            add inference to assign
11:            if inference != Failure then
12:                result ← BacktrackSearch(prob, assign)
13:                if result != Failure then
14:                    return result
15:                end if
16:            end if
17:            remove {var = val} and inference from assign        ▷ Backtracking step
18:        end if
19:    end for
20:    return Failure
21: end procedure
```

This week, we will be discussing how we could implement the inference component of our algorithm.

7.2 Inference

Before implementing the inference component, we will need to first decide on a data structure which we could use to store our inferences.

7.2.1 Representing inferences

In lecture, we discussed the possibility of representing our inferences as a set of values $\{X \notin S\}$, where S is a set of values which the variable X cannot possibly have due to the constraints of the problem.

Notation: in CS3243, we will sometimes use $X \neq 1$ to mean $X \notin \{1\}$, and vice versa.

7.2.2 Domain computation

We also introduced a helper function *ComputeDomain(X, assign, inference)*, which returns the set of values of X that X can take under the current assignment and inference.

For example, if $Dom(X) = \{1, 2, 3, 4\}$, where $Dom(X)$ represents the domain of X:

- If *assign* $= [\{X = 1\}]$ and *inference* $= [\{X \notin \{2, 3\}\}]$, then *ComputeDomain* will return $X \in \{1\}$.
- If *assign* $= [\{Y = 1\}]$ and *inference* $= [\{X \notin \{2, 3\}\}]$, then *ComputeDomain* will return $X \in \{1, 4\}$.

7.2.3 Formalizing inference

Now, we will formalize the ideas discussed in Week 6 and discuss the algorithm for inference. The algorithm for inference is shown below:

Algorithm 3 AC-3

```
1: procedure INFER(prob, var, assign)
2:     varQueue ← [var]
3:     while varQueue is not empty do
4:         Y ← varQueue.pop()                          ▷ We pick a variable from the queue
5:         for each constraint C where Y ∈ Vars(C) do   ▷ Vars(C) rep. the set of variables of C
6:             for all X ∈ Vars(C)\Y do                 ▷ All the variables of C other than Y
7:                 S ← COMPUTEDOMAIN(X, assign, inference)
8:                 for each v ∈ S do
9:                     if ∀var ∈ Vars(C)\X, C[X ↦ v] is not satisfied for all var = val assignments then
10:                        inference.add(X ∉ {v})
11:                    end if
12:                end for
13:                T ← COMPUTEDOMAIN(X, assign, inference)
14:                if T = {∅} then                      ▷ Inference concludes that there are no possible values for X
15:                    return Failure
16:                end if
17:                if S ≠ T then                        ▷ We were able to infer something about X
18:                    varQueue.add(X)
19:                end if
20:            end for
21:        end for
22:    end while
23:    return inference
24: end procedure
```

Remark: The order in which the variables X are explored (line 6 of the algorithm) may have an impact on the speed of inference, and they will be explored in future modules on CSPs.

We will now revisit the n-queens problem from previous lectures, and attempt to arrive at a solution using Algorithm 3.

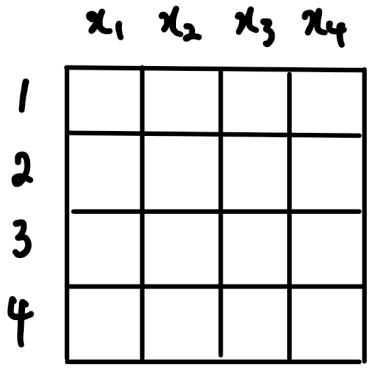

Figure 7.1: A starting state of the n-queens problem, where $n = 4$

1. Initially, we assign $x_1 = 1$.
   (a) If $x_2 = 1$, then C will not be satisfied, so we learn that $x_2 \neq 1$ (i.e. we add $x_2 \notin \{1\}$ to our inference). We will also learn that $x_2 \neq 2$.
   (b) Similarly, for x_3, we will infer that $x_3 \notin \{1, 3\}$.
   (c) For x_4, we will also infer that $x_4 \notin \{1, 4\}$.
2. Following our initial assignment, we have learnt that $Dom(x_2) = \{3, 4\}$. Regardless of whether we assign $x_2 = 3$ or $x_2 = 4$, we will be able to infer that $x_3 \neq 4$, and so we add $x_3 \notin \{4\}$ into our inference.
3. Eventually, we will hit a dead end and move on to assign $x_1 = 2$, and the process of inference will repeat.

7.2.4 Reducing the cost of inference

As discussed in the previous lecture, inference can be expensive, and sometimes it may not be worth checking all possible future states. Is there any way for us to eliminate unnecessary inference?

One way of reducing the cost of inference is by adopting an alternative implementation of inference. Some possible alternative implementations include:

1. *Forward checking*: we remove lines 17 to 19 from Algorithm 3. Thus, we are effectively only inferring one step ahead of us, and not too much into the future.
2. For line 17 of Algorithm 3, we replace "**if** $S \neq T$" with "**if** $|T| = 1$", i.e. we only infer further when there is only 1 value that variable X can take.
   - This implementation is more expensive that the former implementation, but it enables us to infer more about future states.

In practice, implementation 2 is usually used more than implementation 1. However, the choice of implementation mainly depends on the nature of the problem that we are trying to solve.

7.2.5 Speeding up inference

There are some ways which we could use to speed up inference. We have discussed two possible ways in lecture.

1. **Picking unassigned variables:**
   - One heuristic which we could adopt to pick the variable Y is the *Minimum Remaining Values (MRV)* heuristic. With this heuristic, we will always choose the variables with the smallest domain first.
     - For example, if *ComputeDomain(X_1, assign, inference)* $= \{1\}$ whereas *ComputeDomain(X_2, assign, inference)* $= \{2, 3\}$, then we will pick X_1 before X_2.
2. **Ordering domain values:**
   - We can also adopt a heuristic for picking domain values in line 8 of Algorithm 3: we can start from the values which are most likely to succeed, before going to the values which are less likely to give a successful assignment for subsequent variables.
     - One such heuristic is the *Least-Constraining-Value* heuristic. With this heuristic, we will always select the value that tries to leave the most flexibility for the assignment of subsequent variables.

7.2.6 Storing constraints

In the previous lecture, we have also briefly discussed on some ways of storing our constraints.

1. Truth table
   - For example, for the n-queens problem, we can store our constraints in the following truth table:

x_1	x_2	$NoAttack(x_1, x_2)$
1	1	*False*
1	2	*False*
1	3	*True*
$\vdots$	$\vdots$	$\vdots$

   - However, the limitation of using truth tables to store constraints is that our truth tables may be very large and expensive to store/read.
2. Mathematical methods
   - We can also express our constraints as functions/relations, or using arithmetic.

7.3 Back to CSPs

7.3.1 Local search on CSPs

After backtracking search, CSP researchers found that local search had a better performance than backtracking search, before they moved on to backjumping-enabled search.

To carry out local search on constraint satisfaction problems, we can simply replace the *BacktrackSearch* recursive call in Algorithm 2 with a recursive call on local search. The algorithm is shown below:

Algorithm 4 Local Search on CSPs

```
1: procedure BacktrackSearch(prob, assign)
2:     if all var in (prob, assign) are assigned then
3:         return assign
4:     end if
5:     var ← unassigned variable in (prob, assign)
6:     for val ∈ OrderDomainValue(var, prob, assign) do
7:         if val is consistent with assign then
8:             add {var = val} to assign
9:             inference ← Infer(var, prob, assign)                ▷ Inference step
10:            add inference to assign
11:            if inference != Failure then
12:                result ← LocalSearch(prob, assign)
13:                if result != Failure then
14:                    return result
15:                end if
16:            end if
17:            remove {var = val} and inference from assign        ▷ Backtracking step
18:        end if
19:    end for
20:    return Failure
21: end procedure
```

7.3.2 CSP Complexity

How hard are CSPs? CSPs are generally *NP*-complete, where *NP* refers to the class of algorithms that can be solved in non-deterministic polynomial time. What about other variants of CSPs?

1. *Binary CSPs*: where every constraint is defined over 2 variables.
   - Binary CSPs are also *NP*-complete.
2. *Boolean CSPs (SAT)*: where the domain of every variable is $\{0, 1\}$.
   - SAT is also *NP*-complete.
3. *2-SAT*: a binary and boolean CSP.
   - 2-SAT $\in P$. However, 3-SAT is *NP*-complete.

A fundamental problem in computer science is: is *P=NP*?

7.4 Midterm Discussion

7.4.1 Question 6

Taking the optimal path from s_0 to u to be

$$\pi^* : s_0, s_1, \ldots, s_k, s_{k+1}, \ldots, u$$

If we can prove that:

1. $f(s_0) \leq f(s_1) \leq \ldots \leq f(u)$ (required only along the optimal path), and
2. $\hat{f}_{\text{pop}}(s_i) = f(s_i)$.

then we could use these conditions to show that an A^* graph search with the heuristic will be optimal.

Hence, it suffices to show that condition 1 holds for the given heuristic *along the optimal path*. This can be shown by

$$\begin{aligned}
\forall s_i \in \pi^*, h(s_i) &= \frac{OPT(s_i)}{2} \\
&= \frac{c(s_i, s_{i+1}) + OPT(s_{i+1})}{2} \\
&< c(s_i, s_{i+1}) + \frac{OPT(s_{i+1})}{2} \\
&= c(s_i, s_{i+1}) + h(s_{i+1})
\end{aligned}$$

By showing the the heuristic is consistent along the optimal path, we would have shown that $\hat{f}_{\text{pop}}(s_g) = f(s_g)$, and thus the algorithm is optimal.

Remark: Consistency is a sufficient property for optimality, but it is not necessary. Consistency implies condition 1, but it suffices to show a property weaker than condition 1.

CS3243: Introduction to Artificial Intelligence **Fall 2020**

Lecture 8: October 14

Lecturer: Prof. Kuldeep S. Meel *Scribe: Ang Zheng Yong*

8.1 Markov Decision Processes

Back in Lecture 2, we defined a problem with:

1. States: S
2. Actions: A
3. Transition model: $T(s, a) : State \times Action \rightarrow State$
4. Performance measure: $h : State \times Action \rightarrow \mathbb{R}$
5. Initial state: s_0
6. Goal state: s_g

Today, we will consider a stochastic model, and we will redefine some of the aforementioned parameters to fit into our stochastic model.

8.1.1 Transition model

Recap: the transition model reflects the state that the agent will transition to from state s after executing the action a. For instance, given the following state space:

s_3	s_4	s_5	s_{10}
s_2	$\times$	s_6	s_9
s_1	s_{11}	s_7	s_8

If the transition model is $T(s_1, UP)$, the agent will get to the state s_2.

Now, let us consider a model where the agent moves in the direction that we want most of the time, and moves in another direction with a small probability. For example, if we request the agent to go "UP", it will move:

- up with a probability of 0.7,
- down, left, or right, each with a probability of 0.1.

The previous transition model is unable to encapsulate the idea of stochasticity, so we will need to extend the definition of the transition model to include the stochasticity of the model. We revise $T(s, a)$ (also written

as $P(s'|s,a)$) to represent the probability distribution over the states that the agent will transition to, upon performing action a in state s. Thus, for the above example:

$$T(s_1, UP) = P(s'|s_1, UP) = \begin{cases} 0.7 & \text{transitioning to } s_2 \\ 0.2 & \text{transitioning to } s_1 \\ 0.1 & \text{transitioning to } s_{11} \end{cases}$$

How effective is this model? If nondeterministic elements exist in a problem, this transition model would be able to accurately model the problem.

8.1.2 Reward function

The performance measure which we have previously introduced will be replaced by a reward function, which serves a similar purpose.

The reward function R can be expressed as $R : States \to \mathbb{R}$ or $R : States \times Actions \to \mathbb{R}$ (both expressions are equivalent to each other and can be used interchangeably). However, we will be continue our discussion with the model $R : States \to \mathbb{R}$ for the sake of being consistent with AIMA.

8.1.3 Goal state

Similar to previous lectures, we also aim to find a path from the initial state to the goal state. However, in today's discussion, we will not consider the goal to be a permanent entity; instead, we will consider the goal to be flexible.

8.1.4 Terminal states

In addition to the six parameters in our orignal problem definition, we will also introduce the idea of a terminal state. Terminal states are states where no further actions are taken after we reach them.

8.1.5 Example

Let us consider the example which we have previously mentioned.

s_3	s_4	s_5	s_{10}
s_2	$\times$	s_6	s_9
s_1	s_{11}	s_7	s_8

We will consider the set of states to be s_i, where an $\times$ indicates an unreachable state (i.e. a wall), and $\{s_9, s_{10}\}$ to be the set of terminal states. We will also assign rewards to each state:

- $R(s_9) = -1$,
- $R(s_{10}) = +1$,
- $\forall s_i \in States \backslash \{s_9, s_{10}\}, R(s_i) = -0.4$.

If the model was deterministic, a possible plan (i.e. sequence of actions) might have been:

$$\text{UP}, \text{UP}, \text{RIGHT}, \text{RIGHT}, \text{RIGHT}$$

and the states that we would've visited would be $s_1 \to s_2 \to s_3 \to s_4 \to s_5 \to s_{10}$.

Unfortunately, if the model was probabilistic/stochastic, our plan may not always work. Assuming a probabilistic model (with the probabilities listed in Section 8.1.1), the probability of our plan successfully bringing the agent to s_{10} is merely $0.7^5 = 0.16807 < 0.2$. Let us take a closer look at one possible circumstance under this probabilistic model.

- Initially, when we take the action "UP" in s_1, we could have landed in either s_2, s_1, or s_{11}, with probabilities 0.7, 0.2, and 0.1 respectively.
- If our agent (is extremely unlucky and) lands in s_{11} after executing the first action, it will continue to execute the action "UP" on the next iteration if it followed our plan, despite the cell above s_{11} being a wall.

Thus, our analysis shows that our plan may not always work for a stochastic model. Instead, we will require a more general idea, which is more adaptive than a plan. For instance, we could use a **policy**, i.e. a function which returns the action to be taken in every single state which we could be in. A policy is modelled as

$$\pi : \textit{States} \to \textit{Actions}$$

8.2 Utility

How do we put the reward function into our calculations? We would like our agent to maximize the amount of rewards gained, but how do we show this?

Assuming that our agent performs a sequence of actions, and the actions gives rise to a path (i.e. sequence of states) $[s_0, s_1, \ldots, s_n]$. We can define the utility of the sequence of states as:

$$U_h([s_0, s_1, \ldots, s_n]) = R(s_0) + R(s_1) + R(s_2) + \ldots + R(s_n) \tag{8.1}$$

$$\text{or } U_h([s_0, s_1, \ldots, s_n]) = R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \ldots + \gamma^n R(s_n) \qquad \text{where } \gamma \in [0, 1) \tag{8.2}$$

Equation 8.1 is known as the **additive model**, whereas equation 8.2 is known as the **discounted model**, which reduces (or discounts) the magnitude of future rewards. These are just 2 ways of representing the utility, and there are many other ways to model the utility. Note that the utility function is defined over a sequence of states instead of just a single state ($U_h([s_i]) = R(s_i)$, and it disregards the probability distribution of the model).

Why might we need to discount future rewards? We may want our agent to favor immediate rewards over future rewards, and it would also help us to navigate infinite state spaces. The state space would be infinitely large if there are no terminal states.

- If we use the discounted model, then $\forall s_i, R(s_i) \leq R_{max}, U_h([s_i, s_{i+1}, \ldots]) = \sum_i \gamma^i R(s_i) \leq \frac{R_{max}}{1-\gamma}$. This means that the maximum utility is upper bounded by a value $\frac{R_{max}}{1-\gamma}$ in an infinite state space.

8.3 Optimal Policies

Before discussing the idea of an optimal policy, we will need to first define the following:

- s_i represents the i^{th} state in the state space, whereas
- S_i represents the state reached at time i; i.e. it refers to a *random variable*.

For instance, given the following state space:

s_3	s_4	s_5	s_{10}
s_2	$\times$	s_6	s_9
s_1	s_{11}	s_7	s_8

we might have the following policy:

$$\pi(s_i) = \begin{cases} \text{UP} & \text{if } i \in \{1, 2, 6, 7, 8\} \\ \text{RIGHT} & \text{if } i \in \{3, 4, 5, 11\} \end{cases}$$

What are some possible sequences of states which we might observe? In fact, there are (infinitely) many possible combinations, such as:

$$[s_1, s_2, s_3, s_4, s_5, s_{10}]$$
$$[s_1, s_{11}, s_7, s_6, s_5, s_{10}]$$
$$[s_1, s_2, s_4, s_5, s_6, s_7, s_{11}, s_1, \ldots, s_{10}]$$
$$\vdots$$

As such, we might want to define the utility of a state s for a policy π, instead of defining the utility over a sequence of states. The utility of a state s for a policy π is given by:

$$\begin{aligned} U^\pi(s) &= E[U_h(\tau)] && \text{where } \tau \text{ is the seq. of states observed by policy } \pi \\ &= E\left[\sum_{t=0}^{\infty} \gamma^t R(S_t)\right] && \text{summation to } \infty \text{ indicates an infinite horizon} \end{aligned}$$

With this, we can define the **optimal policy** as:

$$\pi^* = \arg\max_\pi \{U^\pi(State)\}$$

and thus the action executed by an agent adopting the optimal policy at state s is

$$\pi^*(s) = \left(\arg\max_\pi \{U^\pi(s)\}\right)(s)$$

Note that the optimal policy is *independent of the start state*. The above equation does not care about where the agent originally started; it only looks at the utilities of all the policies from the *current state* s, finds out the policy that maximizes the utility, and then uses the corresponding action that should be taken.

This is why we could afford to define policies mapping from states to actions, instead of a *sequence of* states to actions. When we are looking at the discounted notion of utility under an infinite horizon, a policy would only depend on the current state which we are in.

8.3.1 Finding the optimal policy

How do we decide what action to take at state s following the optimal policy?

- We first look at all the states s' that are reachable from state s, i.e. $P(s'|s,a)$.
- Then, for all states s' that are reachable, we would want to compute the utility that we would get if we had followed the optimal policy π^* from there onwards, i.e. $U^{\pi^*}(s')$.
- Thus, the action that we should that is the action that maximizes the expected utility, i.e.

$$\arg\max_{a \in A(s)} \left\{ P(s'|s,a) U^{\pi^*}(s') \right\}$$

Since the optimal policy does not depend on the initial state, we can also define $U(s) = U^{\pi^*}(s)$. Then, we could rewrite our optimal policy as:

$$\pi^*(s) = \arg\max_{a \in A(s)} \{ P(s'|s,a) U(s') \}$$

8.3.2 Bellman equation

Remember that previously, we have discussed that

$$\begin{aligned} U(s) &= E[U_h(\tau)] \\ &= E\left[\sum_{t=0}^{\infty} \gamma^t R(S_t) \right] \end{aligned}$$

For $t = 0$, we know that $\gamma^t R(S_t) = R(S_0)$, so

$$\begin{aligned} U(s) &= E\left[\sum_{t=0}^{\infty} \gamma^t R(S_t) \right] \\ &= E\left[R(S_0) + \sum_{t=1}^{\infty} \gamma^t R(S_t) \right] \\ &= R(s) + E\left[\sum_{t=1}^{\infty} \gamma^t R(S_t | S_0 = s) \right] \qquad \text{when } t = 0, \text{ we were in state } s \end{aligned} \tag{8.3}$$

If we try to expand $E\left[\sum_{t=1}^{\infty}\gamma^t R(S_t|S_0=s)\right]$ by looking at all the states that we might end up in, we would get

$$\begin{aligned}
E\left[\sum_{t=1}^{\infty}\gamma^t R(S_t|S_0=s)\right] &= \sum_{s'} P(s'|s,\pi^*(s))\left(\gamma R(s') + E\left[\sum_{t=2}^{\infty}\gamma^t R(S_t|S_1=s')\right]\right) \\
&= \sum_{s'} P(s'|s,\pi^*(s))\gamma\left(R(s') + E\left[\sum_{t=2}^{\infty}\gamma^{t-1} R(S_t|S_1=s')\right]\right) \\
&= \sum_{s'} P(s'|s,\pi^*(s))\gamma\left(R(s') + E\left[\sum_{t'=1}^{\infty}\gamma^{t'} R(S'_t|S_0=s')\right]\right) \qquad \text{by letting } t'=t-1 \\
&= \sum_{s'} P(s'|s,\pi^*(s))\gamma U(s') \qquad \text{from equation 8.3}
\end{aligned}$$

Therefore,

$$\begin{aligned}
U(s) &= R(s) + E\left[\sum_{t=1}^{\infty}\gamma^t R(S_t|S_0=s)\right] \\
&= R(s) + \gamma\sum_{s'} P(s'|s,\pi^*(s))U(s') \\
&= R(s) + \gamma \max_{a\in A(s)}\left\{\sum_{s'} P(s'|s,a)U(s')\right\} \qquad \text{from how } a=\pi^*(s) \text{ is selected} \qquad (8.4)
\end{aligned}$$

Equation 8.4 is also known as the **Bellman equation**.

CS3243: Introduction to Artificial Intelligence **Fall 2020**

Lecture 9: October 21

Lecturer: Prof. Kuldeep S. Meel *Scribe: Ang Zheng Yong*

9.1 Markov Decision Processes

Recall that in our last lecture, we have discussed the Bellman equation:

$$U(s) = R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s'|s,a)U(s')$$

where

1. $R(s)$ is the reward that we would receive in the current iteration.
2. γ is the discounting factor (i.e. reduces the size of future rewards).
3. $\max_{a \in A(s)}$ indicates that we are adopting the optimal policy.
4. $\sum_{s'} P(s'|s,a)U(s')$ is the expected utility that we would receive in future.

However, max is a nonlinear operator, and thus we cannot use linear algebra techniques to solve the Bellman equation.

Remark: Why is max nonlinear?

Algorithm 1 max

```
1: procedure MAX(x_1, x_2)
2:     if x_1 > x_2 then
3:         return x_1
4:     else
5:         return x_2
6:     end if
7: end procedure
```

The above algorithm clearly can't be represented using a sequence of linear equations.

9.1.1 Value iteration

How then could we efficiently calculate $U(s)$? We could apply the following algorithm:

1. Initialize all utilities $U(s)$ to some value, e.g. 0.
2. Update $U_i(s)$ according to the following equation:

$$U_{i+1}(s) \leftarrow R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s'|s,a) U_i(s')$$

   - Here, $U_i(s)$ represents the utility value at iteration i, and we will continuously update the utility via a while loop.

3. We will stop the algorithm when $|U_{i+1}(s) - U_i(s)| < \epsilon$, where ϵ is some arbitrarily small threshold.
   - In other words, our algorithm will converge to some optimal value $U(s)$ after a number of iterations, i.e. $\forall s, \lim_{i \to \infty} U_i(s) = U(s)$.

9.1.2 Policy iteration

There is another way of solving the Bellman equation. For instance, wouldn't it be great if we did not have the max operator? Should we know the optimal policy, then we could easily remove the max operator from the Bellman equation. The updated Bellman equation would be

$$U^{\pi^*}(s) = R(s) + \gamma \sum_{s'} P(s'|s, \pi^*(s)) U^{\pi^*}(s')$$

However, we often do not know what the optimal policy is. The idea to start with some policy π_0 and attempt to find π^*; this is known as **policy iteration**. The algorithm is:

1. Initialize policy $\pi_0 : \textit{States} \rightarrow \textit{Actions}$ (some arbitrary mapping).
2. Compute $U^{\pi_i}(s)$ according to the following equation:

$$U^{\pi_i}(s) = R(s) + \gamma \sum_{s'} P(s'|s, \pi_i(s)) U^{\pi_i}(s')$$

   - Since the equation is linear, $U^{\pi_i}(s)$ can be computed for all s using Gaussian Elimination in $O(n^3)$ time.

3. Update $\pi_{i+1}(s)$ according to the following equation:

$$\pi_{i+1}(s) \leftarrow \arg \max_{a \in A(s)} \sum_{s'} P(s'|s,a) U^{\pi_i}(s')$$

4. Eventually, the algorithm will converge to the optimal policy, i.e. $\forall s, \lim_{i \to \infty} \pi_i(s) = \pi^*(s)$.
   - How long would it take for the algorithm to converge? At the moment, this is still an open research question. However, the algorithm has worked well in practice so far.

9.2 Reinforcement Learning

So far, everything we have discussed requires us to specify the transition model $T(s, a)$, which could be potentially very big. For instance, a mere game of chess could easily comprise of approximately 10^{40} states. In other real-world applications, such as autonomous vehicles, there are many more parameters to specify, and thus the number of states would be much bigger.

The key ideas of this section include **exploration** and **exploitation**.

9.2.1 Q-learning

For every state s, we would want to learn the action a. More specifically, we want our algorithm to learn the tuples (s, a). We could achieve this by modifying our Bellman equation accordingly:

$$\begin{aligned} U(s) &= R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s'|s, a)U(s') \\ &= \max_{a \in A(s)} \left\{ R(s) + \gamma \sum_{s'} P(s'|s, a)U(s') \right\} \\ &= \max_{a \in A(s)} Q(s, a) \end{aligned} \tag{9.1}$$

where $Q(s, a)$ can be represented as the following recurrence:

$$\begin{aligned} Q(s, a) &= R(s) + \gamma \sum_{s'} P(s'|s, a)U(s') \\ &= R(s) + \gamma \sum_{s'} P(s'|s, a) \max_{a' \in A(s)} Q(s', a') \qquad \text{from equation 9.1} \end{aligned} \tag{9.2}$$

In equation 9.2, $Q(s, a)$ can be interpreted as the reward in the current iteration + the expected future utility from taking action a. Now, we will be able to obtain $Q(s, a)$ with the following algorithm:

1. Initialize $\hat{Q}(s, a)$.
2. Choose an action a, get into a new state s', and update $\hat{Q}(s, a)$.
   - We will update $\hat{Q}(s, a)$ according to the following equation:

     $$\hat{Q}(s, a) \leftarrow \alpha(R(s) + \gamma \max_{a'} Q(s', a')) + (1 - \alpha)\hat{Q}(s, a)$$

     In other words, we will choose the action $\hat{a} = \arg\max_{a \in A(s)} \hat{Q}(s, a)$ with probability α, and choose randomly or based on $\hat{Q}(s, a)$ with probability $1 - \alpha$.
   - α is also known as the **learning rate**. Initially, we would want α to be high (i.e. exploration), but with time, α should tend towards 0 (i.e. exploitation). Thus, α can be defined as:
     - $\alpha(t) = \frac{1}{t}$, or some function that decreases with t, where t is the time (e.g. number of iterations) since the start of the algorithm, or
     - $\alpha(N(s, a))$, some function that decreases with $N(s, a)$, where $N(s, a)$ represents the number of times that we have taken action a at state s.
   - Notice that this idea is similar to that of simulated annealing, where we will allow the algorithm to "make mistakes" with a certain probability $1 - \alpha$.

9.2.2 Approximate Q-learning

The space of $\{(s, a)\}$ can be very large, and we might want to reduce it. This can be achieved by adopting an alternate formula for $Q(s, a)$,

$$Q(s, a) = \sum_{i=1}^{n} f_i(s, a) w_i$$

where

- $f_i(s, a)$ represents the features of the model, and
- w_i represents the weight attributed to each feature.

For example, in chess, features may consist of:

- the number of pawns on the board,
- whether the Queen is in the upper half of the board,
- etc.

By expressing $Q(s, a)$ as a linear combination of weights w_i, we will be able to reduce the dimension[1]of the problem. Hence, we would only need to worry about learning the weights w_i.

From the updating of $\hat{Q}(s, a)$, we had

$$\hat{Q}(s, a) \leftarrow \alpha(R(s) + \gamma \max_{a' \in A(s)} \hat{Q}(s', a')) + (1 - \alpha)\hat{Q}(s, a)$$
$$\implies \hat{Q}(s, a) \leftarrow \hat{Q}(s, a) + \alpha(R(s) + \gamma \max_{a' \in A(s)} \hat{Q}(s', a') - \hat{Q}(s, a))$$

Since $\hat{Q}(s, a) = \sum_{i=1}^{n} f_i(s, a)\hat{w}_i$, we can thus update $\hat{w}_i$ with the following equation:

$$\hat{w}_i \leftarrow \hat{w}_i + \alpha(R(s) + \gamma \max_{a' \in A(s)} \hat{Q}(s', a') - \hat{Q}(s, a)) \frac{\delta \hat{Q}(s, a)}{\delta w_i}$$

In fact, for linear combinations, $\frac{\delta \hat{Q}(s,a)}{\delta w_i}$ is indeed $f_i(s, a)$.

[1]see dimensionality reduction

CS3243: Introduction to Artificial Intelligence **Fall 2020**

Lecture 10: October 28

Lecturer: Prof. Kuldeep S. Meel *Scribe: Ang Zheng Yong*

10.1 Partially Observable Environments

So far, the environments which we have discussed are fully observable for the agent. However, this may not always be the case, and our model may only have a partially observable environment.

We have also discussed about stochastic models, where the actions taken by the agent would lead to outcomes determined by a probability distribution. For instance, consider the following state space:

s_3	s_4	s_5	s_{10}
s_2	$\times$	s_6	s_9
s_1	s_{11}	s_7	s_8

Previously, we have noted that given an action $A = \textit{direction}$, our agent would go in the specified direction with probability 0.7, and go in the other directions with probabilities 0.1 each. For instance, if we tell our agent to head *RIGHT* from state s_1, it will reach state s_{11} with probability 0.7, s_2 with probability 0.1, and s_1 with probability 0.2.

Today, we will be mainly discussing about partially observable environments where variables can take on binary values.

10.1.1 Example 1: Jar with 2 coins

Suppose that there is a jar containing 2 coins, c_{50} and c_{90}, where $P(\textit{heads}) = 0.5$ for c_{50} and $P(\textit{heads}) = 0.9$ for c_{90} respectively. In this problem, the agent is unable to see which coin is being picked, but it is able to observe our actions (i.e. tossing coins) and the outcome of the coin tosses.

Suppose further that we picked a coin, tossed it 6 times, and obtained the sequence $HTTHTH$, where H represents that the outcome is *heads* and T represents that the outcome is *tails*. The goal of the agent is to know which coin are we holding onto.

Before the coin was tossed, the agent perceives that the probability of us having c_{50} and c_{90} is 0.5 each. After the coin was tossed, the agent would update its perceived probability according to Bayes' Rule:

$$\begin{aligned} P(c_{50}|\textit{toss}) &= \frac{P(c_{50} \cap \textit{toss})}{P(\textit{toss})} \\ &= \frac{P(c_{50})P(\textit{toss}|c_{50})}{P(\textit{toss})} \\ &= \frac{0.5 \times 0.5^6}{0.5 \times 0.5^6 + 0.5 \times 0.9^3 \times 0.1^3} \\ &= 0.95542 \end{aligned}$$

$$\begin{aligned} P(c_{90}|toss) &= \frac{P(c_{90} \cap toss)}{P(toss)} \\ &= \frac{P(c_{90})P(toss|c_{90})}{P(toss)} \\ &= \frac{0.5 \times 0.9^3 \times 0.1^3}{0.5 \times 0.5^6 + 0.5 \times 0.9^3 \times 0.1^3} \\ &= 0.04458 \end{aligned}$$

10.1.2 Model classification

Initially, the agent has 2 models of the world:

- Model 1: c_{50} is chosen.
- Model 2: c_{90} is chosen.

After the coins were tossed, the agent could make use of the evidence (i.e. results of the coin tosses) to choose one of the models. This is known as **model classification**. The goal of the agent is to choose the model that is most capable of explaining the evidence.

In the aforementioned example, since $P(c_{50}|toss) > P(c_{90}|toss)$, the agent would likely conclude that c_{50} was picked instead of c_{90}.

10.1.3 Example 2: Jar with 100 coins

Now, suppose instead that our jar contains 100 coins, where:

$$\begin{aligned} c_1 &: P(H) = 0.01 \\ c_2 &: P(H) = 0.02 \\ &\vdots \\ c_{99} &: P(H) = 0.99 \\ c_{100} &: P(H) = 1 \end{aligned}$$

Similarly, we will pick one of the coins, toss the coin 6 times, and get the sequence $HTTHTH$. The goal of the agent is to figure out between c_{50} and c_{90}, which of them is able to better explain the evidence (i.e. based outcome of the coin tosses, is $P(c_{50}|toss) > P(c_{90}|toss)$ or $P(c_{50}|toss) < P(c_{90}|toss)$)?

Recall from the previous example that

$$P(c_{50}|toss) = \frac{P(c_{50})P(toss|c_{50})}{P(toss)} \text{ and } P(c_{90}|toss) = \frac{P(c_{90})P(toss|c_{90})}{P(toss)}$$

Since the denominators are equivalent, we only need to compute the numerators of the fractions in order to compare the 2 probabilities. This would save us a significant amount of time, since

$$P(toss) = \sum_{i=1}^{100} \frac{1}{100} P(toss|c_i)P(c_i)$$

and as $i \to \infty$, the calculation will become cumbersome.

10.2 Revision for Probability

10.2.1 Axioms of probability

For some events A and B,

1. $0 \leq P(A) \leq 1$
2. $P(A \cup B) = P(A) + P(B) - P(A \cap B)$
3. $P(\mathit{True}) = 1;\ P(\mathit{False}) = 0$

10.2.2 Conditional probability

$P(A|B)$ is described as: "given the occurence of event B, what is the probability of event A happening?" Mathematically,

$$\begin{aligned} P(A|B) &= \frac{P(A,B)}{P(B)} \\ P(A|B,C) &= \frac{P(A,B,C)}{P(B,C)} \\ &\vdots \end{aligned}$$

10.2.3 Independence of events

Two events A and B are said to be **independent** if $P(A|B) = P(A)$.

Given event B, event A is said to be **conditionally independent** of event C if $P(A|B,C) = P(A|B)$.

Remark: Prof has also briefly discussed the use of Venn diagrams to help visualize independent events, but it is currently Week 11 and it would take a rather large amount of time to draw them with the TikZ package D: This link provides an overview of Venn diagrams.

10.2.4 Chain rule

The chain rule is given by:

$$\begin{aligned} P(x_1, x_2, \ldots, x_n) &= P(x_1|x_2, x_3, \ldots, x_n)P(x_2, x_3, \ldots, x_n) \\ &= P(x_1|x_2, x_3, \ldots, x_n)P(x_2|x_3, \ldots, x_n)P(x_3, \ldots, x_n) \\ &\vdots \\ &= P(x_1|x_2, x_3, \ldots, x_n) \ldots P(x_{n-1}|x_n)P(x_n) \end{aligned}$$

10.3 Bayesian Networks

We will begin the discussion of Bayesian networks based on a simple example.

10.3.1 Example 3: Getting a job

It is tough to get our dream jobs. We will consider some of the factors which may influence our chance of getting into our dream job, including:

1. Grades: G indicates that our grades are high, whereas $\bar{G}$ indicates that our grades are low.
2. Job interview: I indicates that our interview went well, whereas $\bar{I}$ indiactes that our interview went horribly.
3. ERP: E indicates that we had to pay high ERP charges, whereas $barE$ indicates that we did not have to pay ERP charges.

In order to come up with a model of the world, the agent would require data for these variables. We can use a survey for our data collection. The table below shows a sample dataset which we might have collected:

G	E	I	freq.	Pr
T	T	T	160	$P(G,E,I) = 160/600$
T	T	F	60	$P(G,E,\bar{I}) = 60/600$
T	F	T	240	$P(G,\bar{E},I) = 240/600$
T	F	F	40	$P(G,\bar{E},\bar{I}) = 40/600$
F	T	T	10	$P(\bar{G},E,I) = 10/600$
F	T	F	60	$P(\bar{G},E,\bar{I}) = 60/600$
F	F	T	10	$P(\bar{G},\bar{E},I) = 10/600$
F	F	F	20	$P(\bar{G},\bar{E},\bar{I}) = 20/600$

From this table, the agent would be able to figure out all kinds of probabilities involving the three variables, which may be of its interest. For example,

$$\begin{aligned} P(G) &= P(G,E,I) + P(G,E,\bar{I}) + P(G,\bar{E},I) + P(G,\bar{E},\bar{I}) \\ &= \frac{160+60+240+40}{600} \\ &= \frac{500}{600} \end{aligned}$$

However, if there are many variables, the table size may become very large. For instance, if we consider 2 additional variables:

- Job offer: J indicates that we received an offer, $\bar{J}$ indicates that we did not receive an offer.
- Mood of the driver: D indicates that the driver is happy, $\bar{D}$ indicates that the driver is unhappy.

Then, our table will need to have 32 entries (or 31, if we choose to compute one of the entries mathematically instead of storing it). In general, if we have n variables, we would require a table with $O(2^n)$ entries. Is there any way for us to reduce the amount of space needed to store the entries?

10.3.2 Networks and Conditional Probability Tables (CPTs)

If we insist on taking note of the frequencies and probabiities of all the events, our table will definitely be large. However, we could aim for our agent to have a causal model of the world instead, i.e. if the outcome of an event X is independent of the outcome of an event Y, then we do not need to store $P(X|..., Y)$, where "..." may represent any number of variables. This is because if events X and Y are conditionally independent of each other, then $P(X|..., Y) = P(X|...)$.

Hence, our agent may have the following model of the world:

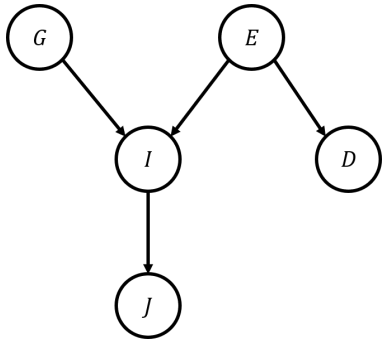

Figure 10.1: Bayesian network model for example 3

This directed acyclic graph (DAG) is also known as a **Bayesian/Inference/Belief Network**. In this graph, we are effectively making the following statements:

- The behavior of I is solely determined by G and E.
- The behavior of J is solely determined by I.
- The behavior of D is solely determined by E.

At each node, we can then construct the respective CPTs. For instance, at node I, our CPT would be:

G	E	Pr
T	T	$P(I\|G, E) = 160/200$
T	F	$P(I\|G, \bar{E}) = 240/280$
F	T	$P(I\|\bar{G}, E) = 10/70$
F	F	$P(I\|\bar{G}, \bar{E}) = 10/30$

Similarly, we could construct the CPTs at nodes D and J as well, and be able to derive any probabilities that we may require.

10.3.3 Space consumption

Did our new model of the problem actually help with reducing the size of our table?

- Without adopting the aforementioned model, a network of n nodes would require a table with 2^n entries.
- With the use of Bayesian networks, the number of entries is upper bounded by $n \times 2^p$, where p represents the maximum number of parents that a node in the network can have.

In the previous example, we were able to reduce the number of entries from 32 to 10, merely by adopting the Bayesian network model. Admittedly, if there is a node with $n-1$ parents, then we would not be able to have a significant reduction of entries, but such networks are uncommon.

10.3.4 More on Bayesian networks

Bayesian networks have the following **axiom**:

> Each node in a Bayesian network is independent of its *nondescendants*, given its parents.

For instance, with respect to Figure 10.1,

$$P(I|G,E,D) = P(I|G,E)$$
$$P(D|E,G) = P(D|E)$$

However, note that this axiom does not apply to the descendants of nodes. For example,

$$P(I|J,G,E) = \frac{P(I,J,G,E)}{P(J,G,E)} \text{ is not necessarily equal to } P(I|G,E).$$

This highlights the idea that causation implies correlation, but correlation does not necessarily imply causation.

10.3.5 Computing probabilities

Suppose we want to compute $P(G,I,J,E,D)$. We can apply the chain rule which we have discussed in a previous section:

$$P(G,I,J,E,D) = P(G|I,J,E,D)P(I|J,E,D)P(J|E,D)P(E|D)P(D)$$

However, note that the probabilities on the right hand side of the equation cannot be obtained from the CPTs of the Bayesian network, and thus this computation would not be possible.

Instead, we may want to apply the chain rule in a different way, i.e. after rearranging the elements. Then,

$$\begin{aligned} P(G,I,J,E,D) &= P(J,I,G,D,E) \\ &= P(J|I,G,D,E)P(I|G,D,E)P(G|D,E)P(D|E)P(E) \\ &= P(J|I)P(I|G,E)P(G)P(D|E)P(E) && \text{from the axiom of BN} \end{aligned}$$

Now, the probabilities on the right hand side of the equation can be simply extracted from the CPTs of the Bayesian network, and thus the computation would be trivial.

Q: How do we figure out the arrangement/ordering of the variables?

When ordering the variables $X_{i+1}, X_{i+2}, \ldots, X_n$ in $P(X_i, X_{i+1}, X_{i+2}, \ldots, X_n)$, we would want to ensure that:

1. none of these variables is a descendant of the variable X_i, and
    - this is because if X_{i+j} for some j is a descendant of X_i, then $P(X_i|\ldots, X_{i+j})$ would be hard to compute (because X_i and X_{i+j} are not conditionally independent of each other), and we wouldn't want to have to compute such a probability.
2. it would also be nice to select a variable X_i such that $X_{i+1}, \ldots, X_n$ contains as many parents as possible.

Since the network is a directed acyclic graph, it is possible for us to recursively choose the nodes without children and place them at the start of the ordering; this could be done easily using **topological sort**. Hence, for our previous example, we could pick the nodes in the order $J \to I \to G \to D \to E$, and this arrangement would enable us to compute the probability $P(G, I, J, E, D)$ easily via the chain rule.

In summary, the algorithm for computing probabilities is:

1. Rearrangement of variables.
2. Apply chain rule.
3. Lookup the conditional probabilities from the CPTs.

CS3243: Introduction to Artificial Intelligence **Fall 2020**

Lecture 11: November 4

Lecturer: Prof. Kuldeep S. Meel *Scribe: Ang Zheng Yong*

11.1 Bayesian Networks

Recall from last week, we discussed a problem which gave us the following Bayesian network:

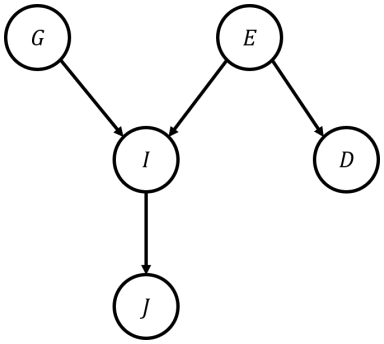

Figure 11.1: Bayesian network model for example

We also assumed that we have the following table of data:

G	E	I	freq.	Pr
T	T	T	160	$P(G, E, I) = 160/600$
T	T	F	60	$P(G, E, \bar{I}) = 60/600$
T	F	T	240	$P(G, \bar{E}, I) = 240/600$
T	F	F	40	$P(G, \bar{E}, \bar{I}) = 40/600$
F	T	T	10	$P(\bar{G}, E, I) = 10/600$
F	T	F	60	$P(\bar{G}, E, \bar{I}) = 60/600$
F	F	T	10	$P(\bar{G}, \bar{E}, I) = 10/600$
F	F	F	20	$P(\bar{G}, \bar{E}, \bar{I}) = 20/600$

Today, we will be discussing how are we going to fill up the entries of the conditional probability tables (CPTs) for each node from our data.

11.1.1 Filling up CPTs

Let us focus on a subset of nodes from Figure 11.1:

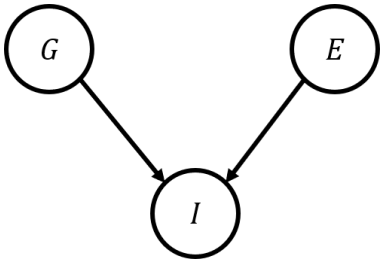

Figure 11.2: Subset of nodes from Figure 11.1

By processing the table from the previous page in a *row-by-row manner*, we will be able to fill up the CPTs for the nodes G, E, and I. The tables for each of the nodes are given below:

$\#G$	#total	$P(G)$
$160 + 60 + 240 + 40 = 500$	$\sum \text{freq} = 600$	$\frac{\#G}{\#\text{total}} = \frac{500}{600}$

$\#E$	#total	$P(E)$
$160 + 60 + 10 + 60 = 290$	$\sum \text{freq} = 600$	$\frac{\#E}{\#\text{total}} = \frac{290}{600}$

G	E	$\#I$	#total	$P(I\|G, E)$
T	T	160	$160 + 60 = 220$	$\frac{\#I}{\#\text{total}} = \frac{160}{220}$
T	F	240	$240 + 40 = 280$	$\frac{\#I}{\#\text{total}} = \frac{240}{280}$
F	T	10	$10 + 60 = 70$	$\frac{\#I}{\#\text{total}} = \frac{10}{70}$
F	F	10	$10 + 20 = 30$	$\frac{\#I}{\#\text{total}} = \frac{10}{30}$

Sometimes, we may be given data in factored form. In other words, we may receive the data separately for E and B. In this case, for each data point, we can go to each entry in our CPTs, update the total count, and calculate the probabilities accordingly.

On the other hand, if we encounter missing data, we may choose to use our background knowledge or prior understanding of the underlying distributions to assign approximate values to the entries.

11.1.2 Deciding on a model

Suppose that we believe that the mood of the driver, D, is not affected by the cost of ERP, E. Instead, we believe that the mood of the driver depends on the students' grades, G. In this case, our Bayesian network model will look as such:

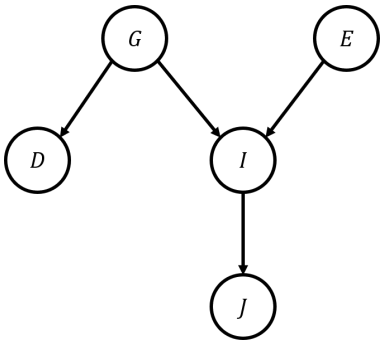

Figure 11.3: Bayesian network model for our new perception of the world

How do we determine which model is able to provide a more accurate description of the world? We first denote Figure 11.1 as M_1 and Figure 11.3 as M_2, and attempt to fill up the CPT entries for M_2. Then, we will collect a number of observations/evidence, and then compare between the models to determine which model provides a better description of the evidence collected.

For instance, we might have collected the following data:

$$e_1 : G = T,\ E = T,\ D = F,\ I = T,\ J = F$$
$$e_2 : G = T,\ E = F,\ D = T,\ I = T,\ J = F$$

To compare the between the models based on our evidence, we will be comparing between $P(M_1|\{e_1, e_2\})$ and $P(M_2|\{e_1, e_2\})$, where by Bayes' rule,

$$P(M_1|\{e_1, e_2\}) = \frac{P(\{e_1, e_2\}|M_1)P(M_1)}{P(\{e_1, e_2\})} \text{ and } P(M_2|\{e_1, e_2\}) = \frac{P(\{e_1, e_2\}|M_2)P(M_2)}{P(\{e_1, e_2\})}$$

Effectively, we are using *prior distributions* (i.e. $P(M_i)$) to update *posterior distributions* (i.e. $P(M_i|\text{evidence})$). We will now proceed to analyze how we could obtain each of the components on the R.H.S. of the equations.

- As discussed in the previous lecture, we can ignore the computation of the denominator, $P(\{e_1, e_2\})$, since both $P(M_1|\{e_1, e_2\})$ and $P(M_2|\{e_1, e_2\})$ share the same denominator.
- We may also assume that the data collected are independent of each other. Thus, we have

  $$\forall i,\ P(\{e_1, e_2\}|M_i) = P(e_1|M_i)P(e_2|M_i)$$

  Since $\forall i, j,\ P(e_j|M_i)$ can be computed easily using topological sort as taught in last week's lecture, where

  $$P(e_j|M_i) = P(G, E, D, I, J|M)$$

  thus the computation for $P(\{e_1, e_2\}|M_i)$ is trivial.
- The final component to consider is $P(M_1)$ and $P(M_2)$. How do we calculate these values?

11.1.3 Dealing with $P(M)$

$P(M_1)$ and $P(M_2)$ reflect the probabilities of each of the models occurring in real life. In fact, it is often difficult to obtain these probabilities, and there are several possible ways to do so:

1. We may choose to start with prior distributions for $P(M_1)$ and $P(M_2)$, based on our own past experiences.
2. We may also base the values of $P(M_1)$ and $P(M_2)$ on historical information from the usage of either of these models.

However, note that it is only necessary for us to figure out the ratio $P(M_1)/P(M_2)$, since we are only trying to compare the magnitude of $P(M_1|\{e_1, e_2\})$ with $P(M_2|\{e_1, e_2\})$.

11.1.4 Modelling

How do we come up with models for comparison? The idea is to start with some arbitrary model M, and move on to a new model M' by making some minor edits to our original model M. For instance, we may have an initial model M_a, given by

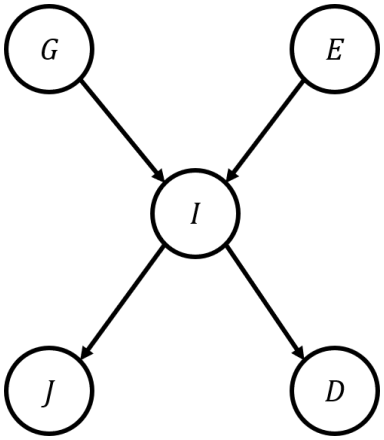

Figure 11.4: Model M_a

We can then make some minor edits to M_a, and arrive at a new model M_b.

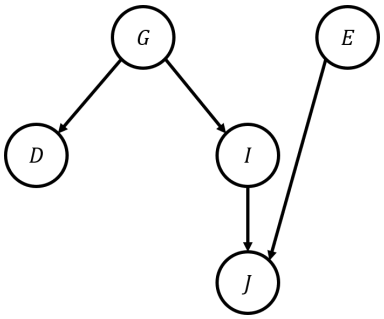

Figure 11.5: Model M_b

After coming up with a new model, we can compute the probability ratio of the two models, and keep the model which gives the higher probability ratio (i.e. the model that more accurately describes the evidence). In order to obtain a model that best describes our evidence, we can repeat this process of formulating a new model and comparing the probability ratio of the 2 models, until we reach some model which always gives a higher probabiliy ratio than other models.

Notice that this concept is exactly similar to local search. In local search, we also begin with an arbitrary state, and gradually transition to another state depending on the value assigned to each state.

Thus, the algorithm for obtaining an optimal model based on some evidence is as follows:

1. Initialize some random model.
2. At every step, look for a neighbour $N(M)$ of the current model.
3. Compute $\frac{P(\text{evidence}|N(M))}{P(\text{evidence}|M)}$, where $N(M)$ is a neighbour of model M.
4. Transition to a new model (or remain on the current model) based on the probability ratio that is computed.

Notice that we are computing $\frac{P(\text{evidence}|N(M))}{P(\text{evidence}|M)}$ instead of $\frac{P(\text{evidence}|N(M))P(N(M))}{P(\text{evidence}|M)P(M)}$. This is because we do not know that value of $\frac{P(N(M))}{P(M)}$, and thus we can assume that $P(N(M)) = P(M)$. Alternatively, we can assign a value based on our prior knowledge of the probability ratio of $\frac{P(N(M))}{P(M)}$.

11.2 Tutorial Discussion

Q: In the following Bayesian network, are F and G independent, given knowledge of the value of C?

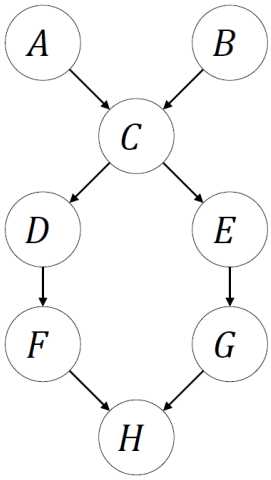

Figure 11.6: Bayesian network from Tutorial 10

In order to show independence, we would want to show that for some arbitrary values f, g, c_i,

$$P(F = f|G = g, C = c_i) = P(F = f|C = c_i)$$

$$\begin{aligned}
P(F = f|G = g, C = c_i) &= \sum_{d_i \in Dom(D)} P(F = f \cap D = d_i|G = g, C = c_i) && \text{axiom of probability}[1] \\
&= \sum_{d_i \in Dom(D)} P(D = d_i|G = g, C = c_i)P(F = f|D = d_i, G = g, C = c_i) && \text{defn of cond. prob.} \\
&= \sum_{d_i \in Dom(D)} P(D = d_i|C = c_i)P(F = f|D = d_i) && \text{axiom of BN} \\
&= \sum_{d_i \in Dom(D)} P(F = f \cap D = d_i|C = c_i) && \text{defn of cond. prob.} \\
&= P(F = f|C = c_i) && \text{axiom of probability}
\end{aligned}$$

□

[1] $P(A = a) = P\left(\bigcup_{b \in Dom(B)} (A = a, B = b)\right) = \sum_{b \in Dom(B)} P(A = a, B = b)$

11.3 Knowledge Representation

Recall from week 2, we have discussed about a mopbot agent, which is able to move in the following state space:

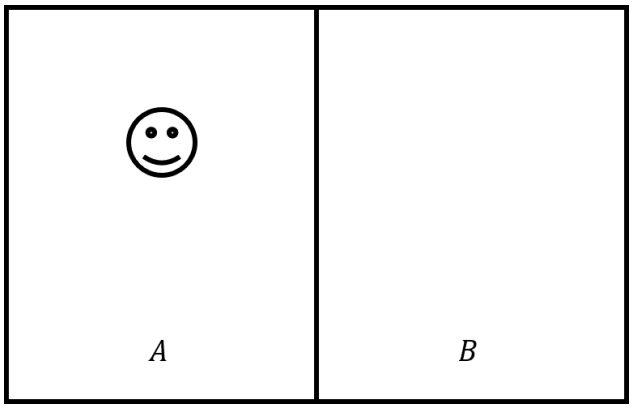

Figure 11.7: State space for mopbot, where the paggro face denotes the current location of the mopbot :)

Our mopbot has:

- Sensors, which enables it to detect whether there is dirt in the room.
- Actuators, which enables it to clean the room or move to another room.

Suppose that the mopbot is given the goal: "go to room B and leave after the room is clean". It might develop the following simple plan p_1 to achieve the goal:

1. Move to B.
2. Clean B.
3. Leave B when no dirt is detected.

Alternatively, the mopbot may also adopt a more complex plan p_2 to achieve the goal:

1. Move to B.
2. If dirt is detected, clean B. Otherwise, there is no need to clean.
3. Leave B when no dirt is detected.

However, how does the mopbot know that it has achieved its goal? The mopbot's plan consists of "no dirt $\implies$ leave room", and we know that "no dirt $\implies$ room is clean", but how does would the mopbot be able to deduce that when it has left the room, the room will be clean? In other words, how does the mopbot know that is has achieved the goal of "room is clean $\implies$ leave room"?

11.3.1 Propositional logic

Knowledge is a concept that is very hard to define, but one definition of knowledge that has been generally agreed upon is:

An agent A knows that statement S is true if and only if:

1. S is true.
2. A believes S is true.
3. A is justified in believing that S is true.

In Bayesian networks, we have to first come up with some model, before we decide whether something is true or false. Since the model is abstract, we have to rely on our beliefs to determine whether something is true or false, and thus we are not going to differentiate between points 1 and 2.

When given the statements "all Greeks are human" and "all humans are mortal", we can deduce that "all Greeks are mortal". However, when given "not all Greeks are human" and "all humans are mortal", we are not able to deduce that "not all Greeks are mortal". Why is this so?

We will now introduce the idea of *propositions*, which are variables that take on values *True* or *False*. According to George Boole (1854), we can decide whether or not we can make such deductions by replacing these statements with propositions. For instance, we can let

- g represent Grades,
- h represent Humans,
- m represent Mortals.

Thus, the first statement is equivalent to:

$$g \to h \wedge h \to m \implies g \to m$$

After introducing propositions, we need a way of combining them to construct meaningful sentences. This led to the introduction of operators, including:

- unary operators: $\neg$
- binary operations: $\wedge$ and $\vee$

Finally, we would want to be able to express our knowledge base. We thus express:

- PROP: the set of all propositions, and
- FORM: the set of all formulas that can be expressed by propositional logic.

Using these expressions, we will be able to classify which statements are allowed in propositional logic, and which are not allowed. For instance, $(p \vee q)$ is allowed under PROP and FORM, whereas $()p \wedge q$ is not allowed.

11.3.2 Recursive definition of FORM

FORM can be recursively defined as such:

$$\begin{aligned} \text{FORM}_0 &= \text{PROP} \\ \text{FORM}_{i+1} &= \text{FORM}_i \cup \{(\alpha \circ \beta) | \alpha, \beta \in \text{FORM}_i\} \cup \{(\neg\alpha) | \alpha \in \text{FORM}_i\} \\ \text{FORM} &= \bigcup_{i=0}^{\infty} \text{FORM}_i \end{aligned}$$

where $\alpha \circ \beta \equiv \{\alpha \vee \beta\} \cup \{\alpha \wedge \beta\}$.

CS3243: Introduction to Artificial Intelligence **Fall 2020**

Lecture 12: November 11

Lecturer: Prof. Kuldeep S. Meel *Scribe: Ang Zheng Yong*

12.1 Propositional Logic

In the last lecture, we have discussed the idea of a knowledge base, which can be seen as a database of everything that the agent knows. In this lecture, we will be representing the knowledge base as *KB*.

We would want every statement $\alpha \in KB$ to also be in *FORM*. In other words, we do not want to deal with sentences which are not in *FORM*, such as $()pq$.

12.1.1 Truth assignments

A **truth assignment** τ is defined as $\tau : PROP \to \{0, 1\}$. For each statement, there are $2^{|PROP|}$ possible truth assignments, where $|PROP|$ represents the number of distinct propositions in the statement.

We say τ models ϕ, or $\tau \models \phi$, if:

- ϕ holds at τ, or
- ϕ is true at τ, or
- τ satisfies ϕ.

Hence, if we are given the statement $p \vee q$, we can construct the following truth table:

p	q	$p \vee q$
0	0	0
0	1	1
1	0	1
1	1	1

and we see that $\tau \models \phi$ when τ has either of the following assignments: $(p \to 0, q \to 1)$, $(p \to 1, q \to 0)$, or $(p \to 1, q \to 1)$. We notice that $\tau \models \phi$ if and only if $\phi(\tau) = 1$.

Another example given in lecture is the statement $\phi = ((p \vee q) \wedge (\neg r))$. For this statement, if we were given the following two truth assignments:

$$\tau_1 : \begin{cases} p \to 1 \\ q \to 1 \\ r \to 1 \end{cases} \qquad \text{and} \qquad \tau_2 : \begin{cases} p \to 1 \\ q \to 0 \\ r \to 0 \end{cases}$$

we notice that $\phi(\tau_1) = 0$ and $\phi(\tau_2) = 1$, so we conclude that $\tau_1 \models \phi$ and $\tau_2 \not\models \phi$.

12.1.2 Satisfiability

In this section, we began with the following definitions:

1. ϕ is *SAT* (satisfiable) if $\exists\tau$ such that $\tau \models \phi$.
2. ϕ is *VALID* if $\forall\tau,\ \tau \models \phi$.
3. ϕ is *UNSAT* (unsatisfiable) if $\forall\tau,\ \tau \not\models \phi$.
    - In other words, there does not exist τ such that $\tau \models \phi$.

Notice that ϕ is *VALID* implies that ϕ is a tautology, and thus a statement that is *VALID* will also be *SAT* (i.e. *VALID* $\implies$ *SAT*). An example of a statement that is *VALID* is $(p \vee \neg p)$, and an example of a statement that is *SAT* but not *VALID* is $(p \vee q)$. On the other hand, examples of statements that are *UNSAT* include $(p \wedge \neg p)$ and $(p \vee q) \wedge (\neg p \vee q) \wedge (p \vee \neg q) \wedge (\neg p \vee \neg q)$.

We also define the following:

- $\phi \models \psi$ if and only if $\phi \rightarrow \psi$ is *VALID*.
- $\phi \equiv \psi$ if and only if $\phi \leftrightarrow \psi$ is *VALID*.
- $\phi \leftrightarrow \psi$ is *VALID* if $\forall\tau,\ \phi(\tau) = \psi(\tau)$.
    - This is equivalent to saying that ϕ and ψ are semantically equivalent.

12.1.3 Conjunctive Normal Form (CNF)

Suppose we have $C_1, C_2, C_3, C_4 \in FORM$. Then, we can see that

$$C_1 \wedge C_2 \wedge C_3 \wedge C_4 \Leftrightarrow (C_1 \wedge (C_2 \wedge (C_3 \wedge C_4))) \Leftrightarrow (((C_1 \wedge C_2) \wedge C_3) \wedge C_4)$$

In fact, the order of the bracketing would not matter, if all the operators in between the clauses C_i are $\wedge$ (note that if the operators consist of $\vee$ and $\wedge$, then it may no longer be the case). We say that such a statement has a **Conjunctive Normal Form** (CNF).

More formally, we say that a statement is a CNF if that statement has the form

$$C_1 \wedge C_2 \wedge \ldots \wedge C_n$$

and every **clause** C_i is of the form

$$C_i = (l_1 \vee l_2 \vee \ldots \vee l_k)$$

and each **literal** l_j is of the form

$$l_j \in \{p \mid p \in PROP\} \cup \{\neg p \mid p \in PROP\}$$

An example of a CNF consisting of the literals p, q, r, s is $(p \vee q \vee r) \wedge (q \vee \neg s \vee r) \wedge (p \vee \neg q \vee s)$.

Theorem 12.1. *Every formula ϕ can be converted into a CNF.*

This theorem states that we can assume that $KB \in CNF$. This is essential for our subsequent discussion on resolution.

For example, if we were given the statement $\phi = ((p \wedge q) \vee (r \wedge s))$, we can convert it to its CNF:

$$\begin{aligned}
\phi &= ((p \wedge q) \vee (r \wedge s)) \\
&= (p \vee (r \wedge s)) \wedge (q \vee (r \wedge s)) && \text{distributive law} \\
&= ((p \vee r) \wedge (p \vee s)) \wedge ((q \vee r) \wedge (q \vee s)) && \text{distributive law} \\
&\equiv (p \vee r) \wedge (p \vee s) \wedge (q \vee r) \wedge (q \vee s) && \because \text{CNF}
\end{aligned}$$

12.1.4 Resolution

We are interested to know if $KB \rightarrow \alpha$ is *VALID*, and we say $KB \models \alpha$ (i.e. KB entails α) if and only if $KB \rightarrow \alpha$ is *VALID*. Observe that ϕ is *VALID* if and only if $\neg\phi$ is *UNSAT*. Thus, we can determine whether ϕ is *VALID* by checking whether $\neg\phi$ is *UNSAT*. How could we check if ϕ is *SAT* or *UNSAT*?

One way to do so is by checking the value $\phi(\tau)$ for all τ. However, recall that there are many possible truth assignments τ, on the scale of $2^{|PROP|}$.

Another way to check if ϕ is *SAT* or *UNSAT* is by checking whether there exists a contradiction in the statement. In other words, if we find a clause C_i in the CNF such that C_i and $\neg C_i$ coexist, then we can immediately conclude that the CNF is *UNSAT*.

- For example, if we are given a statement $\phi = (p_1) \wedge (\neg p_1) \wedge (p_2 \vee p_3 \vee p_4) \wedge \ldots$ where $PROP = \{p_1, p_2, \ldots, p_{100}\}$, we can immediately conclude that ϕ is *UNSAT* because $p_1 \wedge \neg p_1$ is *UNSAT*.

By generalizing the above idea, suppose now that we are given $\phi = (\alpha \vee p) \wedge (\neg p \vee \beta)$. We can construct the truth table for this statement

α	β	p	ϕ	$\alpha \vee \beta$
0	0	0	0	0
0	0	1	0	0
0	1	0	0	1
0	1	1	1	1
1	0	0	1	1
1	0	1	0	1
1	1	0	1	1
1	1	1	1	1

and from the truth table, we can derive that $\phi \rightarrow (\alpha \vee \beta)$. This step is also known as a **resolution**. Another important observation is that if $C_1 \wedge C_2 \rightarrow C_3$ is *VALID*, then $(C_1 \wedge C_2) \leftrightarrow (C_1 \wedge C_2 \wedge C_3)$. By riding on this observation and the idea of resolutions, we can solve a CNF by repeatedly applying resolution, and if we encounter $(C_i \wedge \neg C_i)$ at any point in time, we can then conclude that the formula is *UNSAT*.

For example, for the question posed in tutorial, we can solve it using the resolution approach in the following manner:

Given $(p \vee q \vee r) \wedge (\neg p \vee q \vee s) \wedge (\neg s \vee r) \wedge (\neg q \vee r) \wedge (\neg q \vee \neg r) \wedge (q \vee \neg r)$,
Assuming $C_1, C_2, \ldots, C_6$ are assigned to each of the above clauses,
C_7(resolution over $C_1 \wedge C_2$) : $(q \vee r \vee s)$
C_8(resolution over $C_7 \wedge C_3$) : $(q \vee r)$
C_9(resolution over $C_8 \wedge C_4$) : r
C_{10}(resolution over $C_5 \wedge C_6$) : $\neg r$
C_{11}(resolution over $C_9 \wedge C_{10}$) : *UNSAT*

Recall from last week that we have discussed the problem posed by Greek philosophers, and modelled it using propositions:

$$\text{All Greeks are human} : g \rightarrow h \equiv (\neg g \vee h)$$
$$\text{All humans are mortal} : h \rightarrow m \equiv (\neg h \vee m)$$

How could we determine if $(g \rightarrow m)$ is *SAT*? We could also use the resolution approach to determine whether $(\neg g \vee h) \wedge (\neg h \vee m)$ entails $g \rightarrow m \equiv (\neg g \vee m)$.

Previously, we have mentioned that $KB \models \alpha$ if and only if $KB \rightarrow \alpha$ is *VALID*. So,

$$(KB \rightarrow \alpha) \text{ is } \textit{VALID} \leftrightarrow (\neg KB \vee \alpha) \text{ is } \textit{VALID} \leftrightarrow (KB \wedge \neg\alpha) \text{ is } \textit{UNSAT}$$

and thus we can determine if $(g \rightarrow m)$ is *SAT* by checking if $(\neg g \vee h) \wedge (\neg h \vee m) \wedge g \wedge (\neg m)$ is *UNSAT*. Using the resolution approach,

Let $C_1 = (\neg g \vee h), C_2 = (\neg h \vee m), C_3 = g, C_4 = \neg m$, then
C_5(resolution over $C_1 \wedge C_3$) : h
C_6(resolution over $C_2 \wedge C_4$) : $\neg h$
C_7(resolution over $C_5 \wedge C_6$) : *UNSAT*

Thus, we can conclude that $(g \rightarrow m)$ is *VALID*.

12.1.5 Resolution refutation

We define that resolution refutation of ϕ (given in CNF) to be a list of clauses $C_1, C_2, \ldots, C_t$ such that either $C_i \in \phi$ or C_i is derived from C_a, C_b such that $a, b < i$, and $C_i = \textit{UNSAT}$.

So, for our example from the tutorial, the resolution refutation would be $C_1, C_2, C_3, C_4, C_5, C_6, C_7, C_8, C_9, C_{10}, C_{11}$.

Theorem 12.2. *ϕ is UNSAT if and only if there exists a resolution refutation of ϕ.*

Thus, if we can find a resolution refutation for a statement ϕ, we would have proven that ϕ is *UNSAT*.

Remark: However, note that we still need to find a way of applying resolution between clauses, and this step is hard and may take an exponential amount of time. In fact, the brute force approach may have a better performance than the resolution approach in some circumstances. Given ϕ, checking if ϕ is *SAT* is *NP*-complete.