Introduction to AI – Summary

Idan Orzach, Mike Greenbaum

07/07/2020

©iStock-PlargueDoctor

This beauty summarizes the course *Introduction to Artificial Intelligence (67842), Spring 2020*, as taught by Jeff Rosenschein's lectures and Maya Cohen & Yoni Sher's recitations.

Unless explicitly stated otherwise (hyperlinks/footnotes/direct credit), everything that's written – including images, is based on the course's slides or manually created.

If you have any corrections, additions, hesitations, enlightenments or free food, contact us:

idan.orzach@mail.huji.ac.il

mike.greenbaum@mail.huji.ac.il

Table of Contents

1 Uninformed Search

1.1 Definitions

A search problem has the following:

- Single agent (player who can think and do things)
- Static – the agent doesn't change the landscape
- **Deterministic** – the agent knows the results of his actions
- **Fully observable** – the agent always knows where it is
- Finite space state

Definition: We denote a ***search problem*** as a tuple:

$$< S, s_0, G, A, F, C >$$

$S = \{set\ of\ states\}$
$s_0 \in S$ *is the start state*
$G = \{set\ of\ goal\ states\} \subseteq S$
$A = \{set\ of\ actions\}$
$F: S \times A \to S$ *is a Transition Function,* *translating a state + an action → a new state*
$C: S \times A \to \mathbb{R}^+$ *is a Cost Function*

Example:

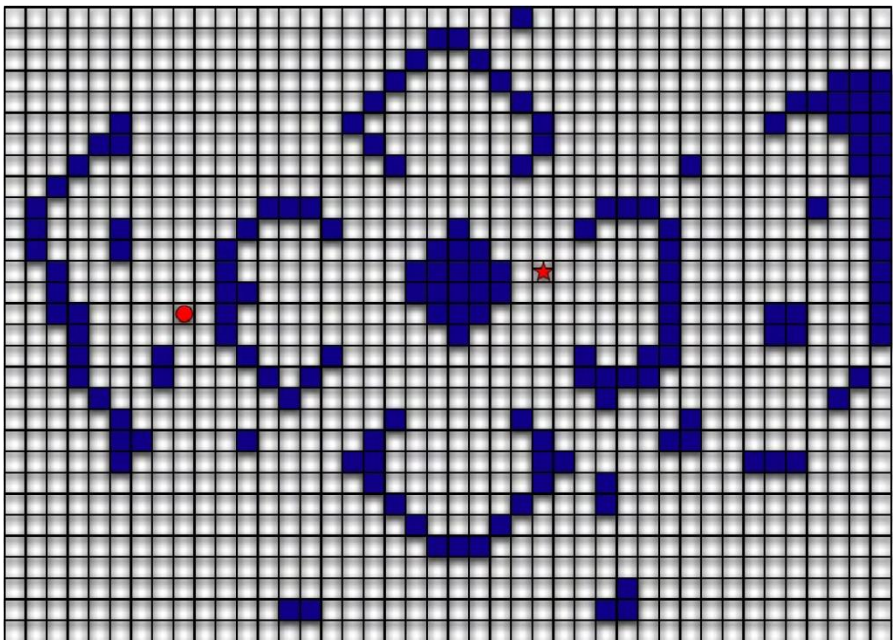


Figure 1.1

The Problem: bring ● to ★.
States: Every configuration of the board where circle is on a white tile.
Start state: The image above.
Goal states: Any state where circle and star are on the same tile.
Actions: move circle up / down / right / left 1 tile.
Transition function: Gets a state (the board and position of circle) and an action, and returns the matching state **after** making that action. Note that moving the circle won't always work.
Cost function: In this example each step, if the circle moves, cost 1. Else 0.

Sometimes we also want to know – what do we do in order to go from s_0 to a goal state $g \in G$?

Definition:

1. A ***solution*** is a tuple

$$< \{s_i\}_{i=0}^{n}, \{a_i\}_{i=0}^{n-1} >$$

where s_0 is the start state, and $\forall i > 1 \;\; s_i = F(s_{i-1}, a_{i-1})$.

2. The ***solution cost*** is $\sum_{i=0}^{n-1} C(s_i, a_i)$.

In the previous example, 2 solutions may look like this:

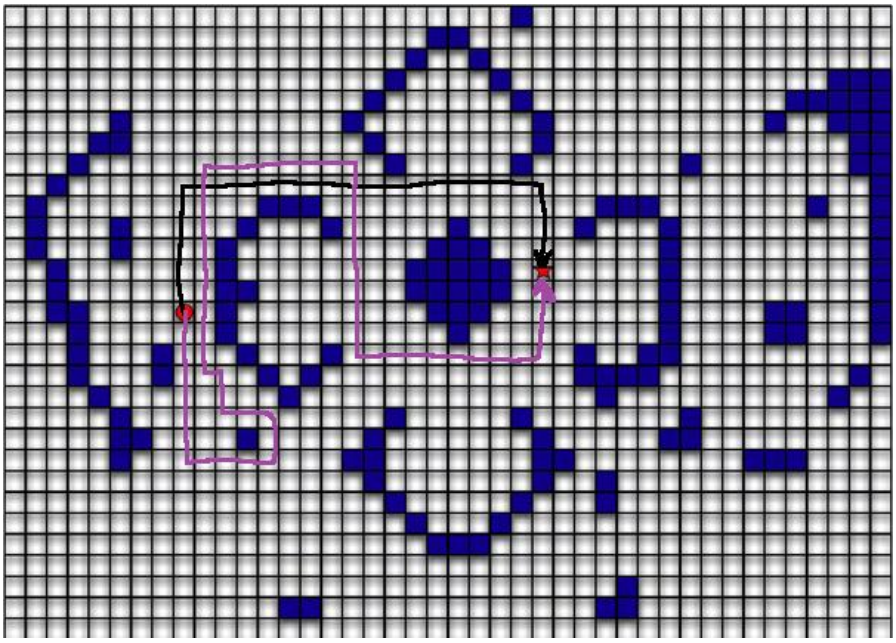


Figure 1.2

The black path's cost is 27, and the purple one's more.

1.2 Implementing Search – How do we put that in a computer?

Since the problem space can get (very easily) enormous, we will keep only the relevant parts of the problem in the memory – where the agent is right now, where the agent was, and sometimes where the agent can go. What we basically need is:

- **DAST** to hold states.
- **Nodes** to connect states and give context – the root is the **start state**.
  each node corresponds to a *single* state (but not the opposite!).
- **Connecting** nodes represents the transition between states using *actions*.
- **Expanding nodes** – generating a state's successors. since we don't the entire space in the memory, creating this option is recommended.

These combine into a data structure (V, E), where

$$V = \{nodes\ representing\ states\},\ E = \{< u, a, v > \mid u, v \in V,\ a \in A\ \ s.t.\ F(u, a) = v\}$$

The search can be represented as a *tree* or as a *graph*. Both options are nice, although trees are inefficient, and searching over graphs needs to be smarter.

General Algorithm – **Tree**

Let $(V = \{v_1\}, E = \emptyset)$, s.t. $L(v_1) = s_0$, $fringe = \{v_1\}$

While $fringe \neq \emptyset$:

$Current \leftarrow choose(fringe)$

$fringe = fringe \setminus \{Current\}$

If $L(Current) \in G$, then DONE

return path to $Current$ from $Root(V, E)$

else:

let $V_{new} = \{v \mid \exists s, a: L(v) = s , s = f(L(Current), a)\}$

let $E_{new} = \{ e = (Current, v): v \in V_{new}\}$

$fringe = fringe \cup V_{new}$

$V = V \cup V_{new}, E = E \cup E_{new}$

Can't solve

Figure 1.3 General algorithm for tree search

Since the same state can be found in **different** nodes, 2 issues can come up with the tree version:

1. A linear problem can turn into an exponential one, and that's big.
2. A finite space can turn into an infinite one.

Converting the tree search to a **graph search** can solve these issues, and the modified algorithm is as follows:

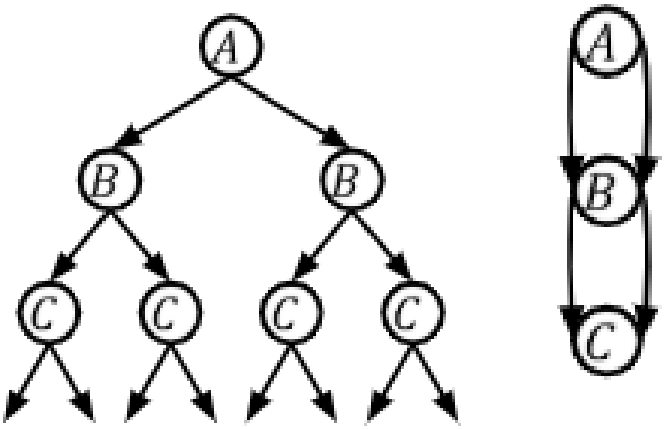


Figure 1.4 A 7-nodes tree that can be reduced to a 3-nodes graph

General Algorithm – **Graph**

Let $(V = \{v_1\}, E = \emptyset)$, s.t. $L(v_1) = s_0$, $fringe = \{v_1\}$, $closed = \emptyset$

While $fringe \neq \emptyset$:

$Current \leftarrow choose(fringe)$

$fringe = fringe \setminus \{Current\}$

If $L(Current) \in G$, then DONE

return path to $Current$ from $Root(V, E)$

else if $L(Current) \notin closed$:

let $V_{new} = \{v \mid \exists\, s, a\colon L(v) = s\,,\ s = f(L(Current\,),\ a)\,\}$

let $E_{new} = \{\, e = (Current,\ v\,)\colon v \in V_{new}\}$

$fringe = fringe \cup V_{new}$

$V = V \cup V_{new}, E = E \cup E_{new}$

$closed = closed \cup \{Current\}$

Can't solve

Figure 1.5 General algorithm for graph search

The solution isn't necessarily single. If there's more than one way to get from s_0 to $g \in G$, there can (and will) be a lot of solutions. The actual search can be done in different ways, each way yields different results.

- What if we want to get the straightest path to our goal states? curviest? shortest? longest? only right turns (saves fuel if driving on the right)?
- Do we want to use a tree, a graph, or maybe something else?
- Expanding direction – from start to goal, goal to start, *both*?

Search quality

To formally measure a search's quality, we use parameters:

- **Completeness**: Finds a solution if exists.
- **Soundness**: Signals if there's no solution.
- **Optimality**: Finds the lowest cost solution, if exists.
- **Time Complexity**: Asymptotic # of operations during the search.
- **Space Complexity**: Asymptotic amount of memory used at one time (=nodes).

And for the computational complexity:

- ***b***: Maximum branching factor of the search tree/graph.
- ***d***: Depth of the shallowest solution (with the fewest actions).
- **m**: Maximum depth of the problem space (may be ∞).

In the following search algorithms, what defines them is the **fringe**, and more specifically:

1. Ordering of the fringe.

2. Repeating states – check them or not? Note that the first occurrence of a state is the first node that comes out of the fringe.
3. Reset of the fringe.

In addition, the next algorithms use only the problem's structure – they are *uninformed*.

Breadth First Search (BFS): Searching over the whole layer before deepening the search.

Fringe:

1. Ordering: Queue – FIFO.
2. Repeating states: Doesn't matter.
3. Reset: No reset – single run.

Complete: Yes.

Sound: If m < ∞.

Optimal: If all actions cost the same.

Time complexity: BFS opens every node shallower than the goal, and nodes in the goal's layer until it gets to the goal.

$$T(n) = O\left(1 + b + \cdots + b^d + b \cdot \left(b^d - 1\right)\right) = O\left(b^{d+1}\right)$$

Space complexity: Since BFS keeps all nodes in memory, it is the same as the time complexity.

$$S(n) = O\left(b^{d+1}\right)$$

Depth First Search (DFS): Searching a specific path until getting stuck before moving on.

Fringe:

1. Ordering: Stack – LIFO.
2. Repeating states: Doesn't matter.
3. Reset: No reset – single run.

Complete: If m < ∞ and visiting only new nodes.

Sound: If m < ∞.

Optimal: No. The order of expansion is arbitrary, so a solution will be accordingly.

Time complexity: DFS can go through almost every possible path until it finds a solution path.

$$T(n) = O(b^m)$$

Space complexity: Like BFS - $O(b^m)$. Note that the fringe takes only $O(b \cdot d)$ memory, and a smart implementation can reduce the space complexity to $\boldsymbol{O(b \cdot d)}$.

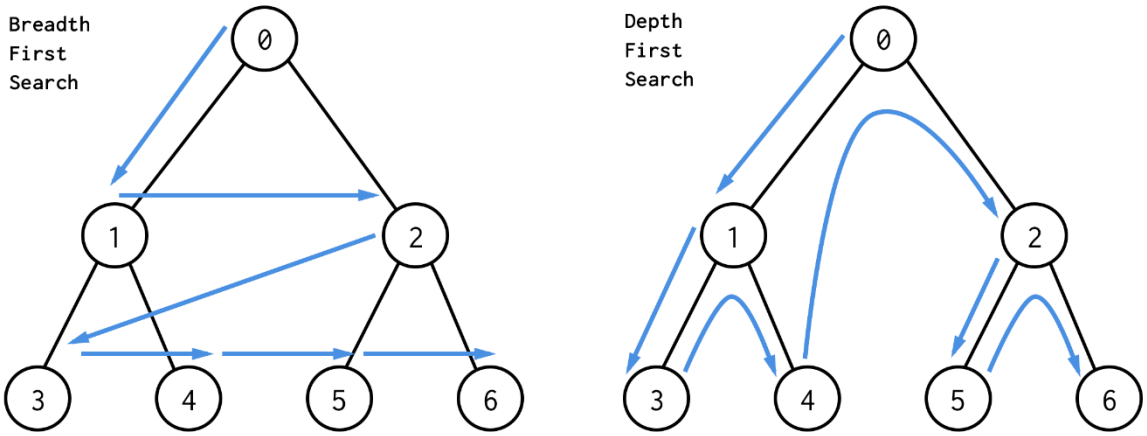


Figure 1.6 BFS and DFS illustration[1]

Uniform Cost Search (UCS): Searching over cheaper paths first. The equivalence of Dijkstra.

Fringe:

1. Ordering: Priority Queue – FIFO (by cost).
2. Repeating states: Doesn't matter.
3. Reset: No reset – single run.

Complete: Yes.

Sound: If m < ∞.

Optimal: Yes.

Time complexity: UCS opens every node shallower (by terms of **cost**) than the goal, and nodes in the goal's layer until it gets to the goal.

$$T(n) = O\left(1 + b + \cdots + b^d + b \cdot \left(b^d - 1\right)\right) = O\left(b^{d+1}\right) \text{ *(if actions have same cost)}$$

Space complexity: Since UCS keeps all nodes in memory, it is the same as the time complexity.

$$S(n) = O\left(b^{d+1}\right) \text{ *(same)}$$

Depth-limited Depth First Search (D-DFS): Searching a specific path until getting stuck before moving on, while **bounding the maximum search depth** – d_{max}.

Fringe:

1. Ordering: Stack – LIFO.
2. Repeating states: Doesn't matter.
3. Reset: No reset – single run.

Complete: No (if $d > d_{max}$).

Sound: Yes.

Optimal: No. The order of expansion is arbitrary, so a solution will be accordingly.

[1] https://www.freelancinggig.com/blog/tag/bfs-vs-dfs-algorithms/

Time complexity: D-DFS can go through almost every possible path with depth ≤ $d_{max}(n) = O(b^{d_{max}})$.

Space complexity: Like DFS - $O(b^{d_{max}})$, and like DFS – can be reduced to $\boldsymbol{O(b \cdot d_{max})}$.

Iterative Depth-limited Depth First Search (ID-DFS): D-DFS while **iteratively increasing** d_{max}.

Fringe:

1. Ordering: Stack – Expand recently opened nodes first, so last in first out.
2. Repeating states: Doesn't matter.
3. Reset: Upon failure – Increase d_{max}d start again.

Complete: Yes.

Sound: if m < ∞.

Optimal: If we increase d_{max} 1 per iteration.

Time complexity: ID-DFS can go through almost every possible path with depth ≤ d_{max}, repeating it until $d_{max} = d(n) = O(b + b\text{^}2 + \cdots + b\text{^}d + b^{d+1}) = O(b^{d+1})$.

Space complexity: Like D-DFS – $O(b^{d+1})$, and like DFS – can be reduced to $\boldsymbol{O(b \cdot d)}$.

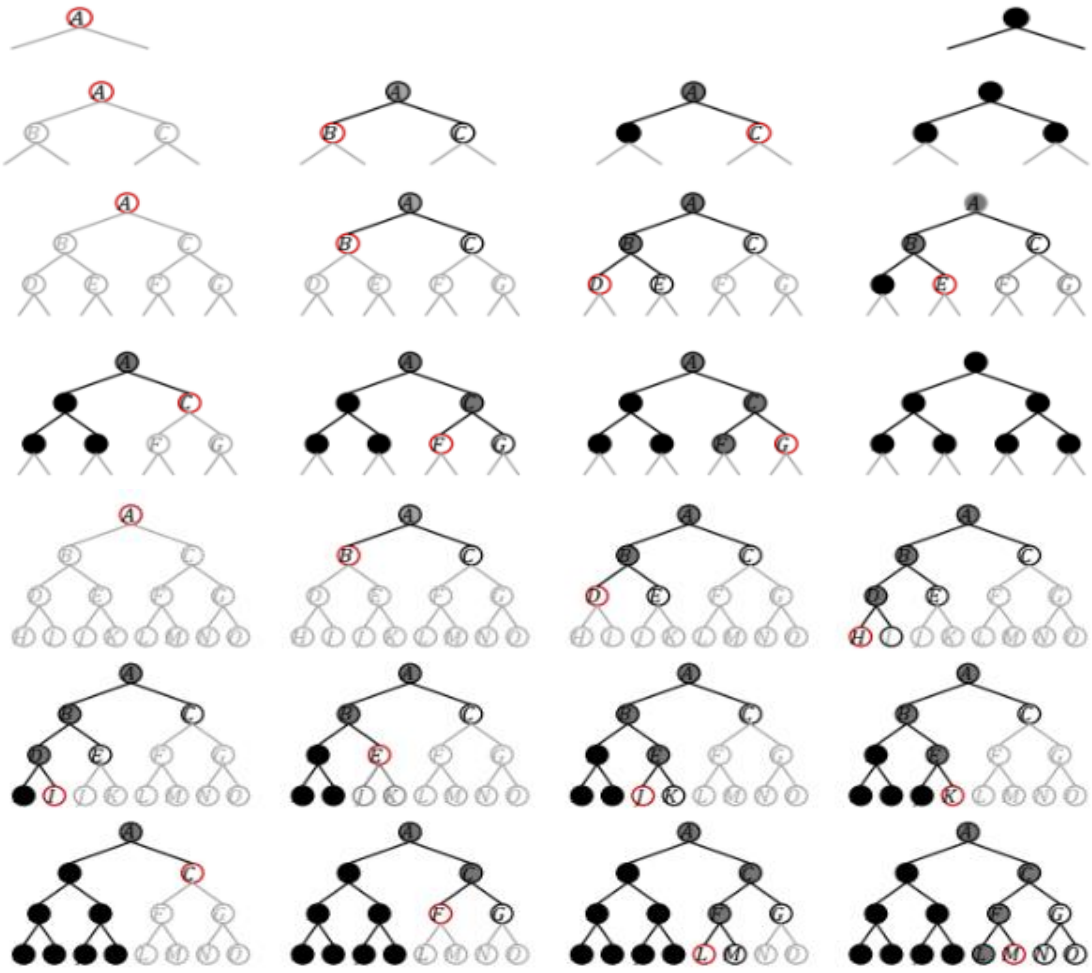


Figure 1.7 ID-DFS illustration

Search Strategies – Summary

	BFS	DFS	UCS	D-DFS	ID-DFS
Complete	Yes	Yes**	Yes	No	Yes
Sound	Yes**	Yes**	Yes**	Yes	Yes**
Optimal	Yes*	No	Yes	No	Yes*
Time	$O(b^{d+1})$	$O(b^m)$	$O(b^{d+1})$	$O(b^{d_{\max}})$	$O(b^{d+1})$
Space	$O(b^{d+1})$	$O(b^m)$	$O(b^{d+1})$	$O(b^{d_{\max}})$	$O(b^{d+1})$

Figure 1.8 Summary of uninformed search algorithms
** if cost is uniform ** if $m < \infty$*

2 Informed & Local Search

2.1 Informed Search

This section's agenda is that most searches have very large search spaces and we have more information about the problem than its structure (which is what the current search algorithms use), for example: 8-puzzle. The figures below illustrate the 8-puzzle's search space.

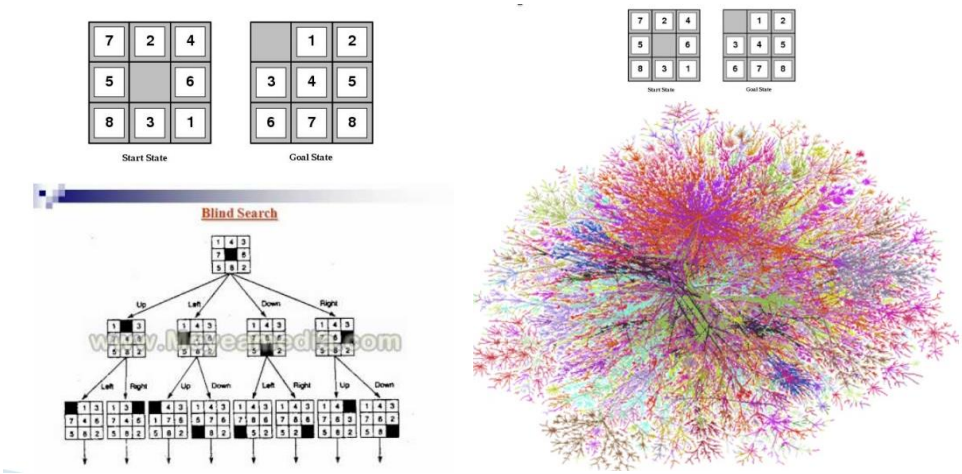


Figure 2.1 The 8-puzzle search space

However, we want to be able to solve much bigger puzzles in a reasonable time, so "reducing" the search space will be good.

Definition:

> A ***heuristic*** is a function that tries to approximate the cost of a state.
> Formally, let $< S, s_0, G, A, F, C >$ be a search problem, then any function $h: V \to \mathbb{R}_{\geq 0}$ is called a heuristic function if $\forall g \in G$ it holds that $h(g) = 0$.

We will try to use the heuristic function to obtain information about the problem and search less in the search space while still using the generic algorithms proposed in Figure 1.3 General algorithm for tree search and Figure 1.5 General algorithm for graph search.

Our first try is being Greedy, formally known as:

Best First Search (BFS): Searching on nodes with lowest heuristic value (trying to minimize the heuristic function and get to the goal)

Fringe:

1. Ordering: Priority queue with heuristic function – expand the node that has the least h value "in hope for being the closest to the goal"
2. Repeating states: Doesn't matter.
3. Reset: No reset – single run.

Complete: If m < ∞ and visiting only new nodes.

Sound: No.

Optimal: No.

Time complexity: Best First Search can go through almost every possible node until it finds a solution path, so $T(n) = O(b^m)$.

Space complexity: Keeps all nodes in memory – $O(b^m)$.

Notice that this has horrible Time complexity and Space complexity, however it can work amazingly well depending on the heuristic function itself. For example:

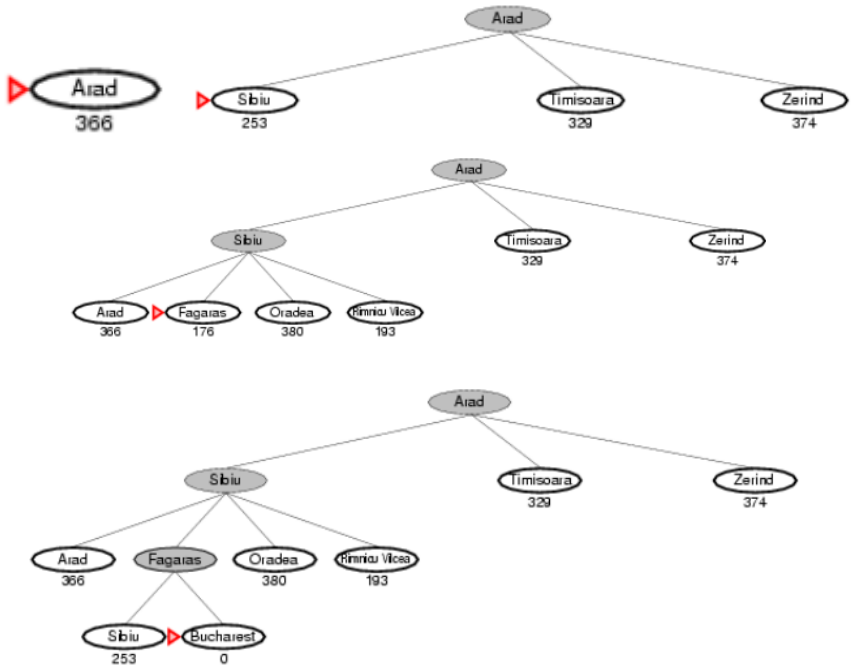


Figure 2.2 Greedy search algorithm, searching for the optimal path from Arad to Bucharest

Our second try is to involve both the heuristic and its current cost to the node by defining $f(n) = g(n) + h(n), \forall n \in V$ when h is the heuristic function and g is the cost of getting to node n.

A* (A-star): Searching on nodes with lowest f value (looking for the shortest path from node n to a goal while taking into account the distance **already** covered).

Fringe:

1. Ordering: Priority queue with f function – expand the node that has the least f value "in hope for looking only on promising paths to the goal".
2. Repeating states: Doesn't matter.
3. Reset: No reset – single run.

Complete: Yes, unless there are infinitely many nodes with $f(n) < C^*$ when C^* is the optimal cost.

Sound: Yes, if $m < \infty$.

Time complexity: A* is exponential in the solution length and in error to the heuristic.

Space complexity: Keeps all nodes in memory – $O(b^m)$.

Optimal: Yes! if the heuristic function is *consistent*, which leads us to the following
Definitions:

Definitions:

1. We will say that a heuristic function $h: V \to \mathbb{R}_{\geq 0}$ is ***admissible*** if $\forall v_0 \in V$ and for every path from v_0 to a goal, denote this path with $v_0, \dots, v_n$, $v_n \in G$ the next equality holds

$$h(v_0) \leq \sum_{i=1}^{n} C(v_{i-1}, v_i)$$

The idea is the heuristic doesn't overestimate the real cost.

2. We will say that a heuristic function $h: V \to \mathbb{R}_{\geq 0}$ is ***consistent*** if $\forall e = (v, v') \in E$ the next equality holds

$$h(v) \leq c(e) + h(v')$$

The idea is the heuristic doesn't overestimate the real cost and is similar to the triangular inequality.

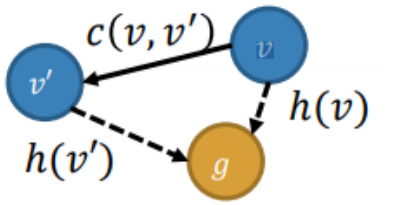


Figure 2.3 Illustration of a consistent heuristic function h

Definition: we will say that a heuristic function $h_1: V \to \mathbb{R}_{\geq 0}$ ***dominates*** a heuristic function $h_2: V \to \mathbb{R}_{\geq 0}$ if $\forall v \in V$ it holds that $h_2(v) \leq h_1(v)$.

Claim 1: if h is an admissible heuristic then A-star finds the **optimal** solution in a **TREE search**.

Proof: Denote $v \in V$ an unexpended node in the fringe such that v is on the optimal path to an optimal goal G_1.
Denote $G_2 \in G$ a suboptimal goal in the fringe (suboptimal means **NOT** optimal), therefore

$$f(v) = g(v) + h(v) \overset{h\ is\ asmissible \to\ has\ a\ value\ less\ than\ the\ real\ cost\ to\ G_1 \in G}{\leq} g(G_1)$$

$$\overset{G_1\ is\ optimal\ goal\ and\ G_2\ is\ suboptimal}{<} g(G_2) \overset{G_2 \in G \to h(G_2)=0}{\leq} g(G_2) + h(G_2) = f(G_2)$$

So, we got $f(v) < f(G_2)$ and therefore, expand v before G_2 for every v on the optimal path and therefore expand G_1 before expanding G_2. ■

Claim 2: if h is a consistent heuristic then f is non-decreasing along any path.

Proof: Denote $e = (v, v') \in E$ and notice that

$$f(v) = g(v) + h(v) \overset{h\ is\ consistent \to h(v) \leq c(e) + h(v')}{\leq} g(v) + c(e) + h(v') = g(v') + h(v') = f(v')$$

So, we got $f(v) \leq f(v')$ for every two nodes on a path, so f is non-decreasing along any path.
■

Conclusion: if h is a consistent heuristic then A-star finds the **optimal** solution in a **GRAPH search**.

A unique property of A*, it expands ALL nodes with $f(n) \leq C^*$, some $f(n) = C^*$ and no $f(n) > C^*$ when C^* is the optimal cost.

We can define a *weighted A-star* which uses $f(n) = g(n) + w \cdot h(n)$ for $w > 1$ and tries to use the heuristic more than the path, however it doesn't provide optimality ☹.

Overall, we have an algorithm that is informed and can find the optimal solution if the heuristic is consistent. However, it uses too much memory, so we will try and solve this problem in a couple of ways:

Simple Memory Bounded A-star (SMA*):

Just run A-star and if there is no memory left, throw the node with the highest cost from the fringe.

This idea tries to deal with the memory constraints given by an exponential Memory Space complexity.

ID-A*: Performing D-DFS while **iteratively increasing** f_{max} and expand only nodes with $f \leq f_{max}$.

Fringe:

1. Ordering: Stack – Expand recently opened nodes first, so last in first out.
2. Repeating states: Doesn't matter.
3. Reset: Upon failure – Increase f_{max} and start again.

Complete: Yes, unless there are infinitely many nodes with $f(n) < C^*$ when C^* is the optimal cost.

Sound: Yes, if $m < \infty$.

Time complexity: exponential in the solution length.

Space complexity: Note that the fringe takes only $O(b \cdot d)$ memory.

Optimal: No!

Recursive best first search: try being greedy but go back if there are other interesting nodes to expand recursively.

Complete: Yes, unless there are infinitely many nodes with $f(n) < C^*$ when C^* is the optimal cost.

Sound: Yes, if $m < \infty$

Time complexity: exponential in the solution length.

Space complexity: Note that the fringe takes only $O(b \cdot d)$ memory.

Optimal: Yes, on tree with an admissible heuristic.

Running example: doing greedy until the given state and then noticing that a neighbour and a lower f value:

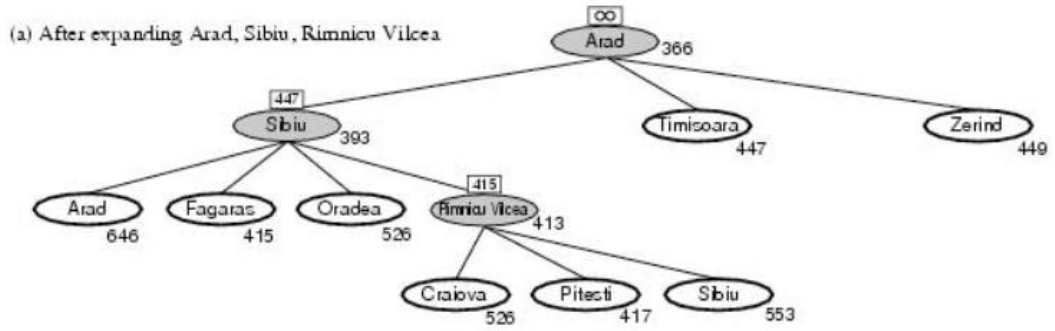


Figure 2.4 While searching in Rimnicu Vilcea's (413) neighbours, the RBFS algorithm notices that Faragas (415) has a lower h *than them (≥417)*

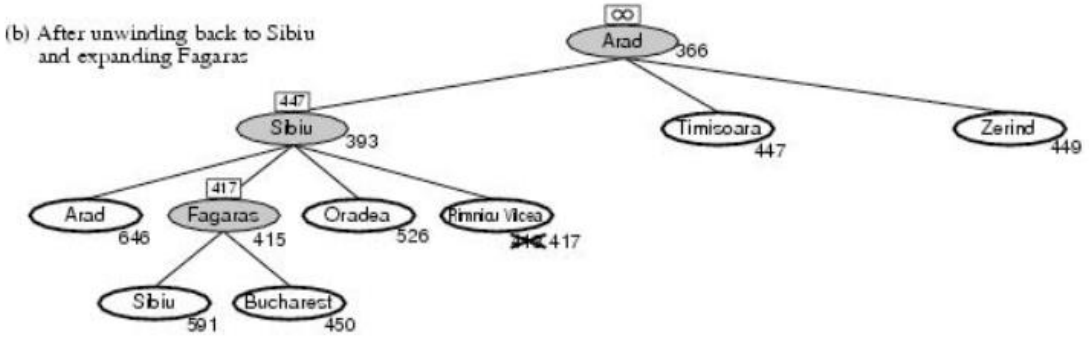


Figure 2.5 Updates Rimnicu Vilcea to 417 and goes to Faragas

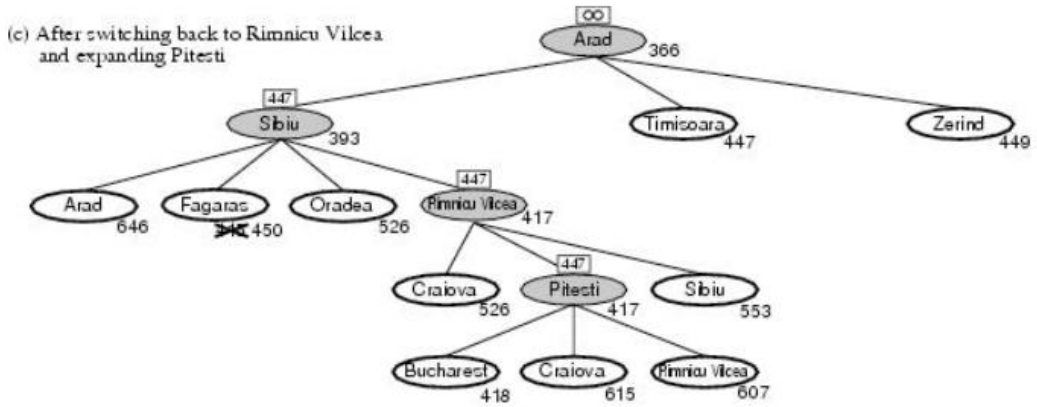


Figure 2.6 Same here. Updates Faragas to its neighbours' minimum h *value (450), then going once more to Rimnicu Vilcea*

Sometimes recursive best first search is more efficient than SMA*, but their problem is that even if there was more memory available, they won't be able to use it (because they bound themselves).

Now it looks like we are in an amazing world, we have optimal algorithms to search with information on a problem. However, until now, we assumed a magical heuristic that contains all the information of the problem. Our next problem is to figure out how to define such heuristics that will be admissible (and maybe consistent) so the optimality can be guaranteed.

1. Our first way to calculate heuristics is through relaxation/abstraction of the problem to the problem with less constraints.
2. Our second way is to take heuristics and weight them. Let $\{h\}_{i=1}^{n}$ be heuristics then we can create $h = \sum_{i=1}^{n} w_i \cdot h_i$ when $\sum_{i=1}^{n} w_i = 1$.
3. The third way is to take heuristics and compose them. Let $\{h\}_{i=1}^{n}$ be heuristics then we can create $h = F(h_1, \dots, h_n)$ when $F = max, sum, \dots$.

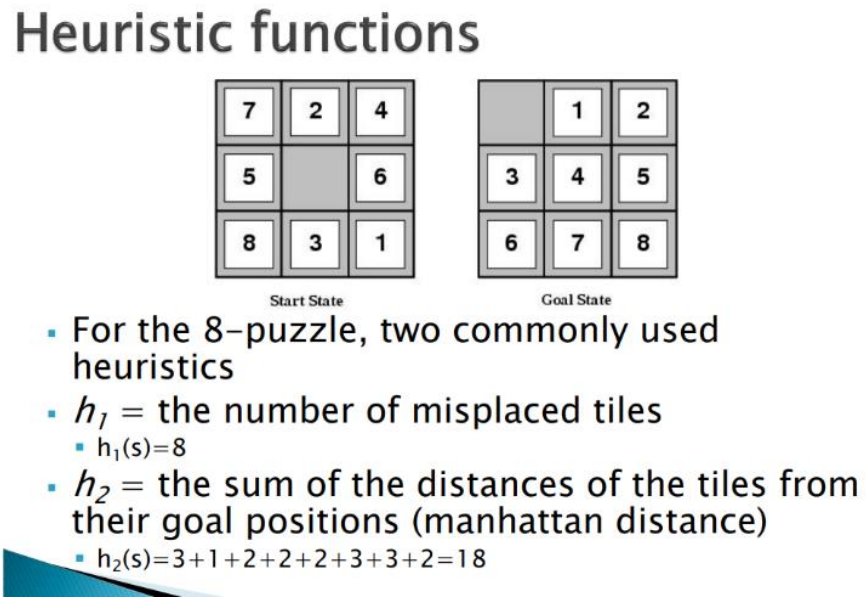


Figure 2.7 Some 8-puzzle example heuristics

2.2 Local Search

Up until this point, we were always interested in finding the best solution and the path to that solution, if we remove the second desire to know the path, we can propose some interesting algorithms that work very well, this is called local search.

Gradient Descent is the idea of going down the slope up to the point of reaching local minimum:

Let $\mathcal{S}$ denote current solution.

- If there is a neighbour $\mathcal{S}'$ of $\mathcal{S}$ with strictly lower cost, replace $\mathcal{S}$ with the neighbour with the minimal cost.
- Otherwise terminate.

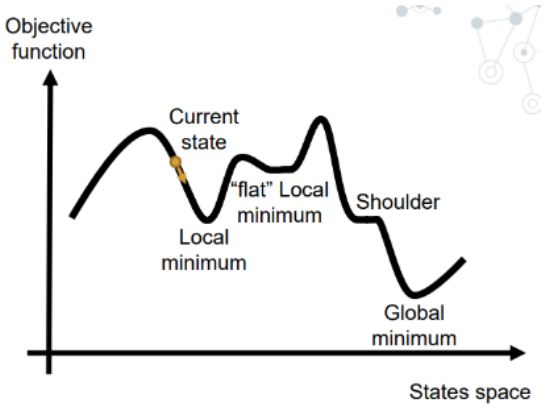


Figure 2.8 A simplified states space and gradient descent illustration

After seeing this definition, we will try once again the greedy algorithm:

Hill Climbing (Gradient *Ascent*): always go to the neighbour with the highest h value.

The biggest problem with Hill Climbing is that it can get stuck in a local maximum, as in the figure below. However, if we allow the agent to take risks (by going to worse states, or descending), then it might find a *global* maximum.

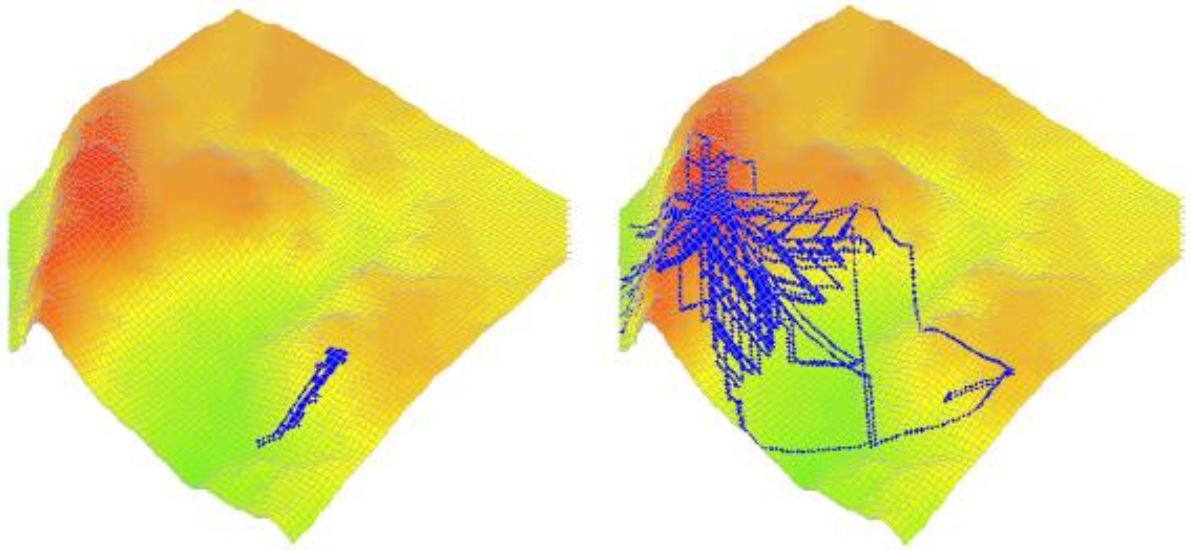


Figure 2.9 Hill climbing (left). However, allowing randomness can lead to better results (right)

So, we will try and modify the hill climbing a bit to be able to escape local maxima:

1. Stochastic Hill Climbing – choose a random neighbour with a higher value than the current state, random climbing, gives it an option to explores other hills.
2. First Choice Hill Climbing – run Stochastic Hill Climbing but generate successors when needed randomly, may help if states contain many successors.
3. Random Restart Hill Climbing – we will run Hill Climbing and each time it gets stuck, restart the search from a random point in space, whilst remembering the best answer found. By this we are giving the algorithms a couple of tries to find the global maxima.

Next idea comes from the random restart hill climbing – what if there are multiple climbers who can share their knowledge?

Local Beam Search:

1. Randomly initialize k states.
2. Each time we choose the best k states of **all** neighbours of the current states.
3. If one of the states is a goal, we found what we were searching for, else we repeat step 2.

The problem we might encounter with Local Beam Search is that the random states might all lead to the same local maxima.

Again, we can make the Local Beam Search be random by choosing successors by their function value (height), this is called **Stochastic Beam Search**.

We can try and take the hill climbing algorithm in a different way:

Simulated Annealing:

1. With a low probability go to a random successor.
2. Else go to the best successor of the current state.
3. And sometimes reset to avoid local maxima.

The idea is to let someone drunk to climb a hill; he may sometimes stray from the best path to find a better one.

There is a proof that is we decay the low probability slowly enough, the probability of finding the global maxima approaches 1.

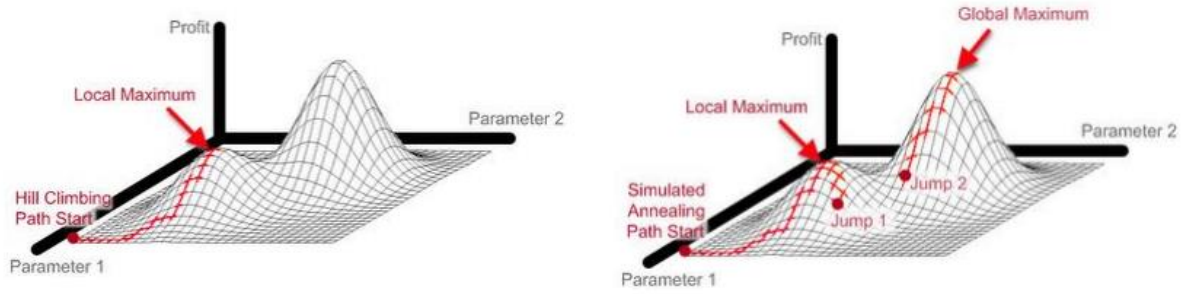


Figure 2.10 Illustrations of "naive" hill climbing (left) and simulated annealing (right)

After all our previous attempts, this time we will try and simulate mother nature's evolution cycle:

Genetic Algorithm:

First, we will use the term *individual* instead of state. Each individual is a vector $v \in \mathbb{R}^n$. We'll call the group of individuals a *population*. This algorithm contains a couple of algorithms to simulate the evolution in nature by having a lot of individuals and letting them evolve from experiences:

1. **Mutation:** this is an algorithm that takes the current population and changes them in hope to improve them, as in nature evolutions occur from mutations.
2. **Crossover:** this is an algorithm that gets a couple of individuals and tells how to create new states from its parents, as in nature children get their DNA by combining their parent's DNA.
3. **Fitness function:** a function that calculates how good a given individual preformed. We will use this function to simulate the law of survival of the fittest.
4. **Selection:** The probability to be chosen for reproducing is related to the fitness function that rates the individuals (higher values for better states).

Below is a scheme of how the genetic algorithm runs:

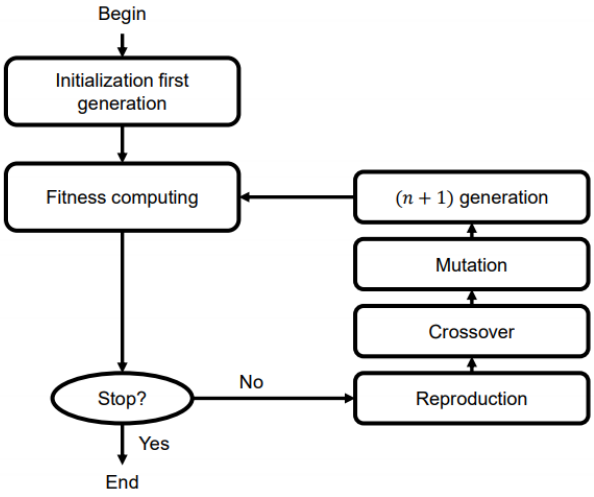


Figure 2.11 Genetic algorithm

Example: 8-queens. Each agent in the population is a vector of the row of each queen (the columns are always 1-8), and the fitness function is the number of non-attacking pairs on the board.

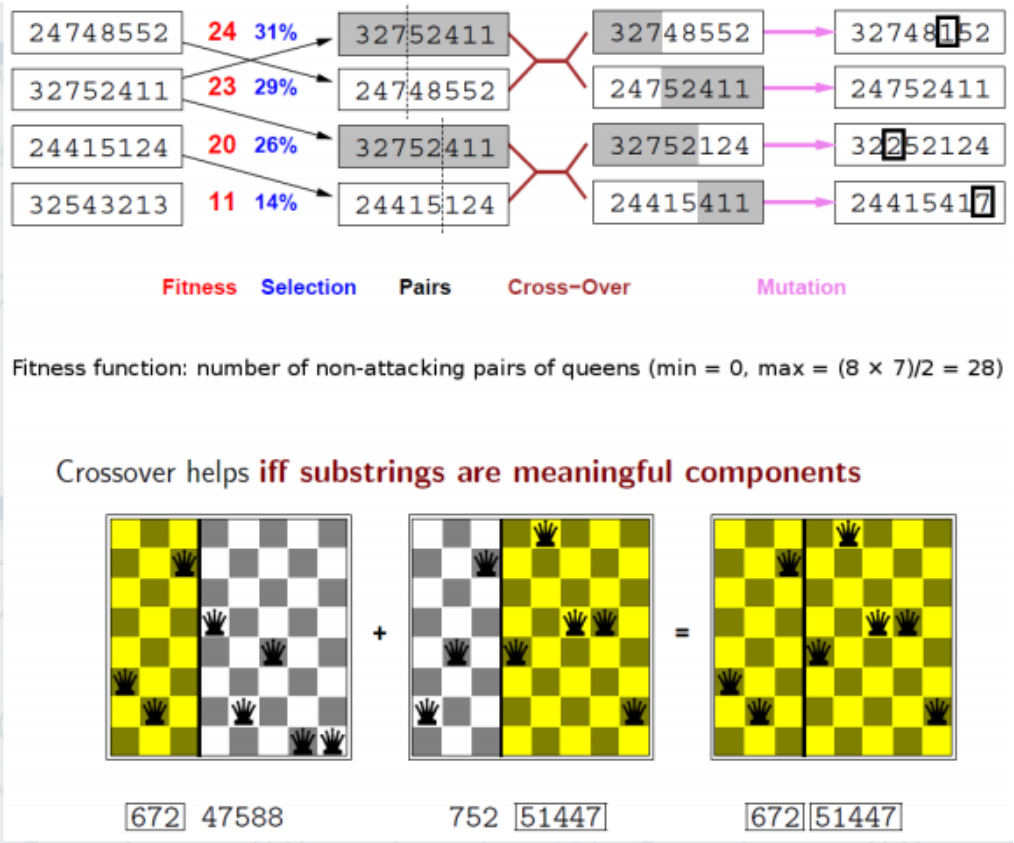


Figure 2.12 8-queens problem, solved with a genetic algorithm

3 Adversarial Search & Constraint Satisfaction Problems

3.1 Constraint Satisfaction Problems

Definition: A ***CSP*** is a tuple:

$$< \mathcal{X},\ \mathcal{D},\ \mathcal{C} >$$

$\mathcal{X} = \{x_1, \dots, x_n\} = \{set\ of\ variables\}$
$\mathcal{D} = \{D_1, \dots, D_n\} = \{set\ of\ domains\}$
$\mathcal{C} = \{C_1, \dots, C_n\} = \{set\ of\ constrains\},\ \ C_j \subseteq D_1 \times \dots \times D_n$
* $\mathcal{C}$ will be usually presented as a function or with hand-waving (simpler).

Example: Consider Australia map, where the goal is to colour its territories so that no 2 adjacent part have the same colour.

Figure 3.1 Australia map as a CSP problem

Here, $\mathcal{X} = \{WA, NT, Q, NSW, V, SA, T\},\ \mathcal{D} = \{\{red, green, blue\} \cdot 7\}$,
$\mathcal{C}$ = adjacent regions must have different colours.
e.g. $WA \neq NT$ or $(WA, NT) \in \{(r, g), (r, b), \dots, (b, g)\}$

Definitions:

1. A ***state*** is an **assignment** of values from some/all domains to their corresponding variables.
2. A ***consistent assignment*** is a state that doesn't violate any constraint.
3. A ***complete assignment*** is a state where each variable has a value.
4. A ***solution*** is a complete & consistent assignment.

In the previous example, one solution looks like this:

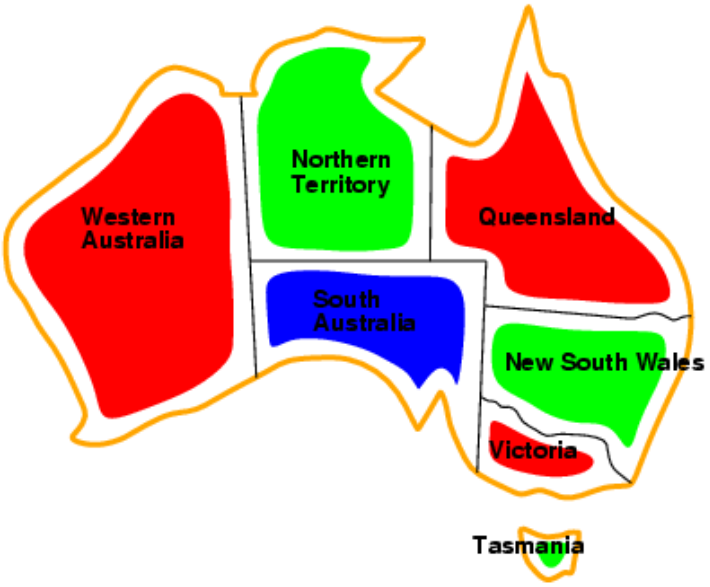


Figure 3.2

And formally, $WA = red, NT = green, Q = red, NSW = green, SA = blue, V = red, T = green.$

Definitions:

1. ***Unary Constraint***: A constrain containing only one variable.
2. ***Binary Constraint***: A constrain containing two variables.
3. ***Binary CSP***: A CSP with only unary or binary constrains.
4. Given a binary CSP we define a ***Constraint Graph*** as:

$$V = \{x_1, \dots, x_n\}, \ (x_i, x_j) = e_k \in E \Leftrightarrow \exists (d_1, \dots, d_n) \in C_k$$
$$\wedge \ \exists \ (d_1, \dots, d_i', \dots, d_n), (d_1, \dots, d_j', \dots, d_n) \notin C_k$$

Meaning, the vertices are the **variables** and the arcs are **constraints** which can be *violated*. If there are constraints which aren't unary/binary, a Constraint Hypergraph can be defined, and be *reduced* to a binary graph.

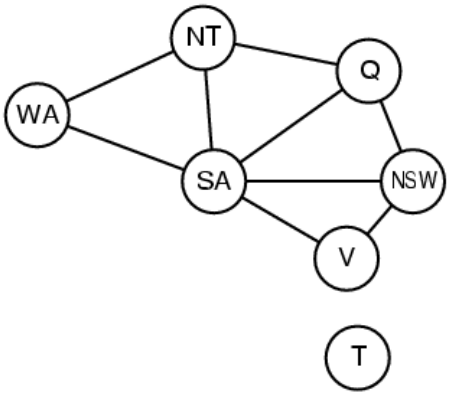


Figure 3.3 Australia as a CSP graph

3.2 Solving CSPs

There are 2 approaches to solving a CSP problem:

1. Starting from an **empty** assignment and assigning *legally* values to variables (backtracking).
2. Starting from a **complete** assignment and *iteratively* changing assignments to get a solution.

Backtracking

The previous search algorithms' runtimes on CSPs are exponential, since in CSPs the states are *commutative*, something that algorithms like A* don't handle. Therefore, we need to recall intro week 8 and use (DFS) **Backtracking**. In backtracking each variable (at its turn) is given a value until a dead end is reached. If the algorithm found a solution, yay. Else, undo the last steps and try something *different*. This will be the base for the heuristics to come.

Backtracking Search
return $RecursiveBacktracking(\{\}, csp)$

Recursive Backtracking

if $assignment$ is complete then return $assignment$
$var \leftarrow SelectUnassignedVariable(csp.variables, assignment, csp)$
for each $value$ in $OrderDomainValues(var, assignment, csp)$:
  if $value$ is consistent:
  add $\{var = value\}$ to $assignment$
  $result \leftarrow RecursiveBacktracking(assignment, csp)$
  if $result \neq failure$ then return $result$
  remove $\{var = value\}$ from $assignment$

Figure 3.4 Recursive backtracking algorithm

Note: The implementations of $SelectUnassignedVariable$ and $OrderDomainValues$ aren't single and affect the backtracking's runtime. Also, that algorithm can be upgraded even further (soon).

If we want a fast backtracking, we want to know:

1. **Which variable** should be assigned next?
2. In **what order** should its **values** be tried?
3. Can we **detect inevitable failure** early?
4. Can we **take advantage** of **problem structure**?

Parts 1-2 will be answered with *heuristics* and 3-4 with upgrades to the general algorithm.

1: Which variable should be assigned next?

Minimum remaining values: choose the variable with the fewest legal values.

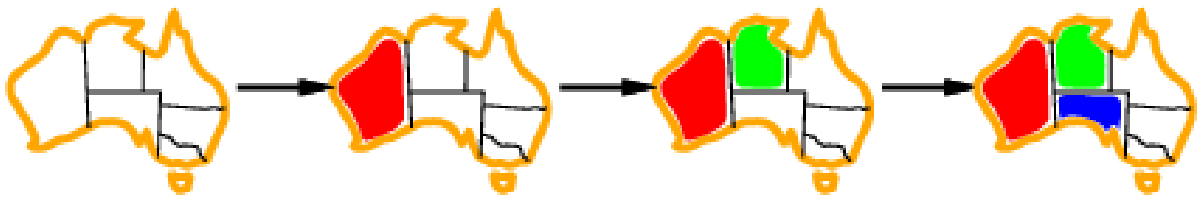


Figure 3.5

Most constrained variable: choose the variable with the most constraints – the vertice with the *highest* degree (on remaining variables).

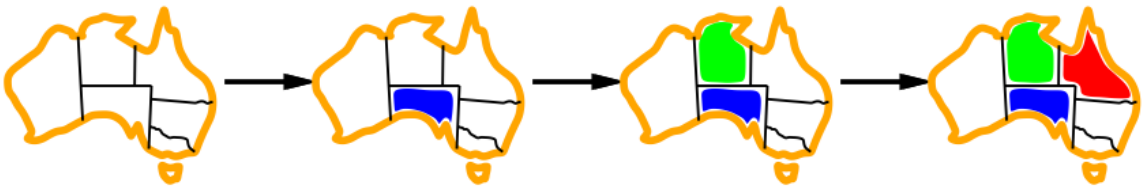


Figure 3.6

Note: The 2 heuristics are **not** commutative – therefore their order must be predefined.

2: In what order should its values be tried?

Least constraining value: given a variable, choose the value that creates the **least** constraints between the given variable and the others. This is equivalent to choosing the value that rules out the least values (of other variables).

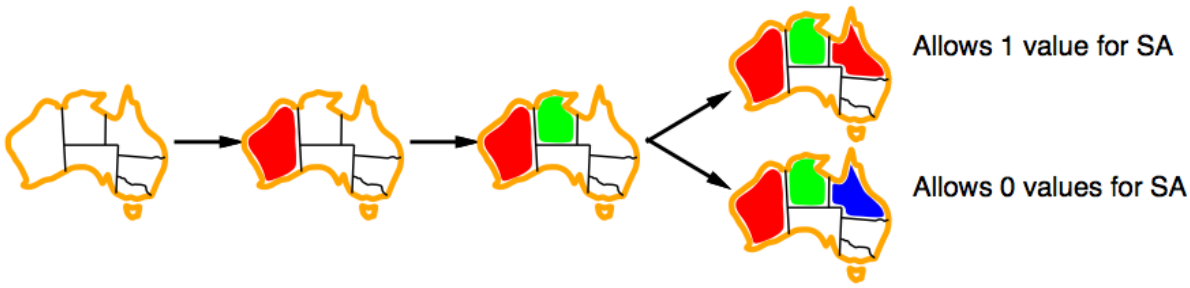


Figure 3.7

3: Can we detect inevitable failures early?

Yes – Forward checking: keep track of each variable's remaining, legal values. From that it's easy to see that if in a state *s* a variable has been left with 0 legal values, a failure is *inevitable*.

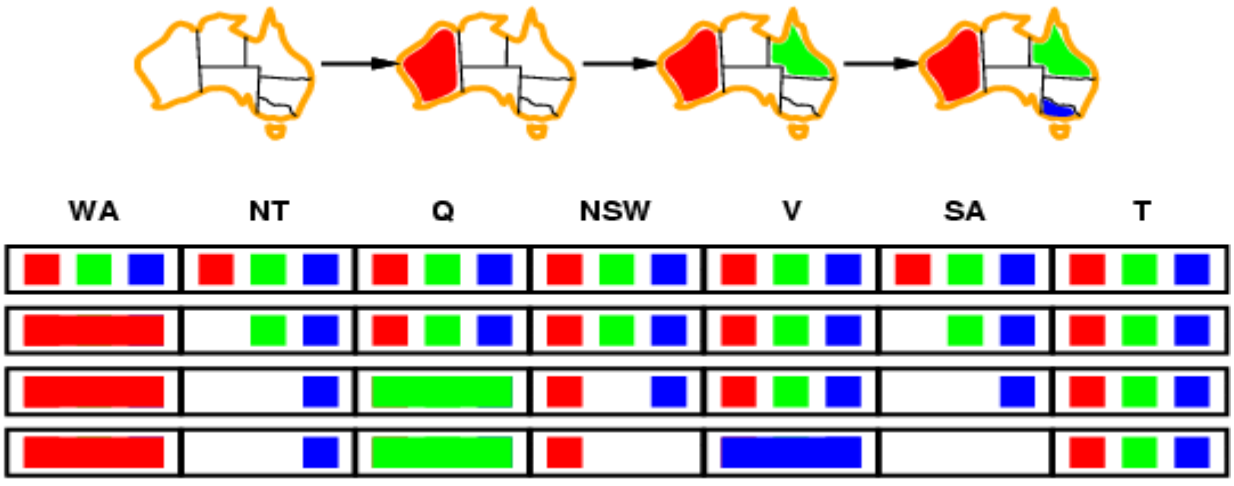


Figure 3.8

In the figure above, colouring WA in red implies that its neighbours, NT & SA, can't be red.

Yes – Constraint propagation: This takes forward checking one step, well, forward. When an illegal value is being removed from a variable x, check x's neighbours and delete inconsistent values. If any of x's neighbours' possible values change, apply this again on them and so on

(similar to flood-fill). Since there are multiple ways of implementing constraint propagation, we'll focus on one:

Arc consistency (AC-3): A simple form of the above. For every arc (x, y) delete values from x's domain until (x, y) is consistent, meaning $\forall a \in D_x \; \exists b \in D_y \; \{x = a, y = b\} \; is\ consistent$.

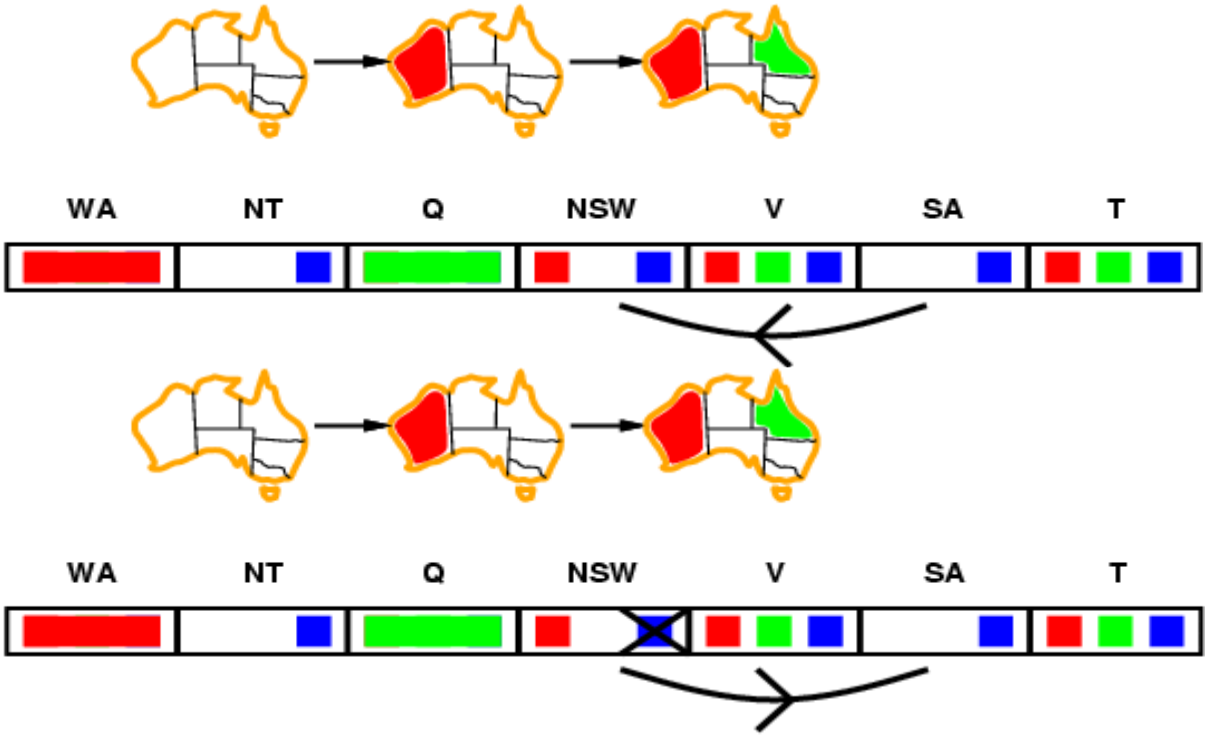


Figure 3.9

(SA, NSW) is consistent because (blue, red) is legal; After removing **blue** from NSW, (NSW, SA) is consistent too, because setting **blue** to NSW leaves SA with 0 legal values. NSW's domain changed, therefore check its neighbours…

4: Can we take advantage of the problem structure?

Yes, yes and yes! – in our example, since Tasmania (T) has no constraints, we can treat it as a *different*, independent problem. An example that shows the benefits of it is the world's map. Since Europe and South America are disconnected, we can save a lot of time by treating those 2 as different problems. In general:

If every variable has a domain of size d, and every **sub-problem** has c variables out of n, therefore treating them as 1 problem is $O(d^n)$ time, while treating them as **multiple** problems is $O\left(\frac{n}{c} d^c\right)$ time. Much faster.

A Few Words on Tree CSP

If the CSP graph is a tree, then we can reduce the algorithm's runtime to $\boldsymbol{O(d^2 n)}$. The proof is an algorithm:

1. Topologic-sort the tree's vertices to get $(v_1, \ldots, v_n)$ (if v comes after u then $(v, u) \notin E$).
2. Forall v_i, check for arc-consistency, between $(parent(v_i), v_i)$ **backwards** (j = n,…,2).
3. Forall v_i assign consistent values **forwards** (j = 1,…,n) – no backtracking required.

The algorithm's runtime is a result of topologic-sort's runtime + checking for arc-consistency for the variables. The algorithm's correctness is because after checking for arc-consistency, for

every value assigned to v_i there's a legal value for $child(v_i)$, and it can be proved by induction that assigning values **forwards** will result in a **solution**.

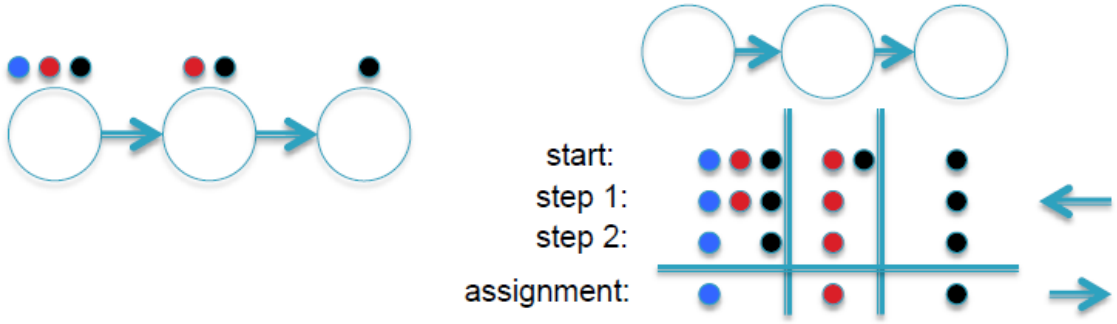


Figure 3.10 Solving a tree-CSP

Note: If the CSP graph is *almost* a tree, then it's possible to reduce it to a tree by assigning values to "key" vertices, then solving the rest of the graph as a tree. In Australia's example, that "key" vertice is SA. So, we can assign **red** to it, and then the rest of the graph can be solved using the ^ algorithm.

2. Solving CSP: Similarities to Informed Search

When trying to solve CSP in the second approach, it can look very much like an *informed search* problem. Unlike the first approach, we can apply informed search algorithms and expect good runtimes because the search tree is no longer factorial, but exponential. Since the start state is a (random) complete one, it can be assumed that it's illegal. This defines the search problem as *changing variable assignments until a solution is reached*. As always with not-perfect evaluation function, stochastic algorithms may have the upper hand here: (First Choice) Hill-Climbing, Simulated Annealing, Local Beam Search etc. Below is an example of 4-queens where the variables are chosen **randomly**, and the values are chosen using **min-conflict** heuristic.

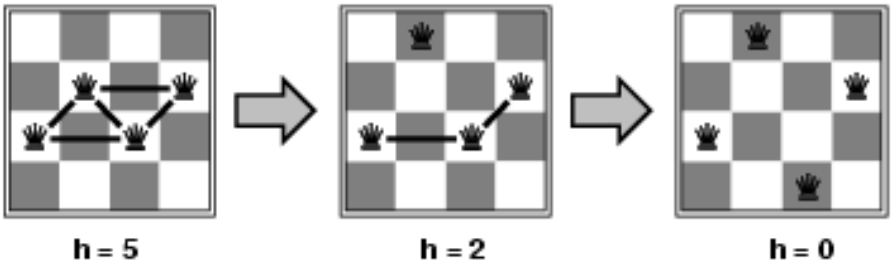


Figure 3.11 4-queens problem. The h *function is the number of pairs attacking each other*

A stochastic search algorithm using the **min-conflict** heuristic can solve the n-queens problem for large n's in almost constant time with high probability (e.g. $n = 10{,}000{,}000$). The ratio between number of constraints and number of variables has a part in it. That's because when $R = \frac{\#\ of\ constraints}{\#\ of\ variables}$ is low it's easy to assign values, and when R is high it's easy to rule out values.

$$R = \frac{\text{number of constraints}}{\text{number of variables}}$$

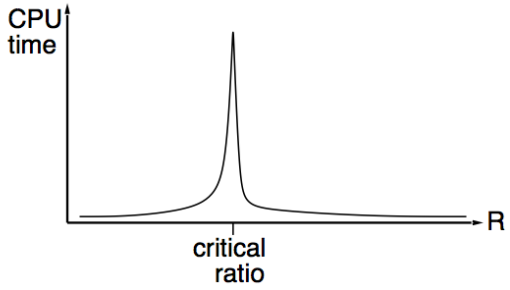


Figure 3.12

3.3 Adversarial Search

Until now the search problems were single agent problems. Consider the case of **adversarial search** – 2 (or more) agents are in the same world, each tries to maximize its profit on the back of the other.

The search tree becomes a **Game Tree**: Every node is a state, but now actions can be made by 2 or more agents, and affect the others' future actions. The solution is a **strategy**: The best course of actions for a **specific** agent (given the start state). We'll focus on 2-agents **Zero-Sum Games**: $u_1(s) = -u_2(s)$ (u for $u$tility). The game tree is layered with alternating max and min nodes.

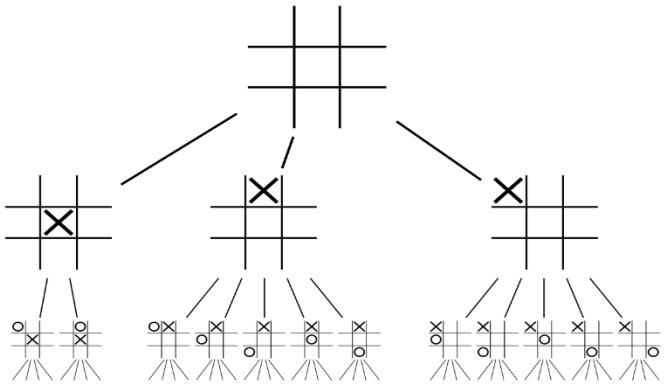


Figure 3.13 Tic Tac Toe game tree

In the example[2] to the right, the agents are the X player and the O player. One is the *maximizing* player – wants to maximize its utility, and the other is the *minimizing* player – wants to minimize the other's utility.

3.4 The Minimax Algorithm – Finding the Perfect Play

How do we find the best strategy? The best strategy is a strategy the player would user against a **perfect rival**. Why? If that strategy can win against a perfect rival, it can win against any rival (because the rival's loss is *exactly* our profit). And this is what the **Minimax Algorithm** does.

Minimax(v, depth, isMaxNode)
if $depth = 0$ or v is a terminal node then return $u(v)$ else if isMaxNode then return $\max_{s \in succs(v)}(Minimax(s, depth - 1, false))$ else (isMaxNode is false) then return $\min_{s \in succs(v)}(Minimax(s, depth - 1, true))$

Figure 3.14 The Minimax algorithm

[2] https://www.pngegg.com/en/png-thczy

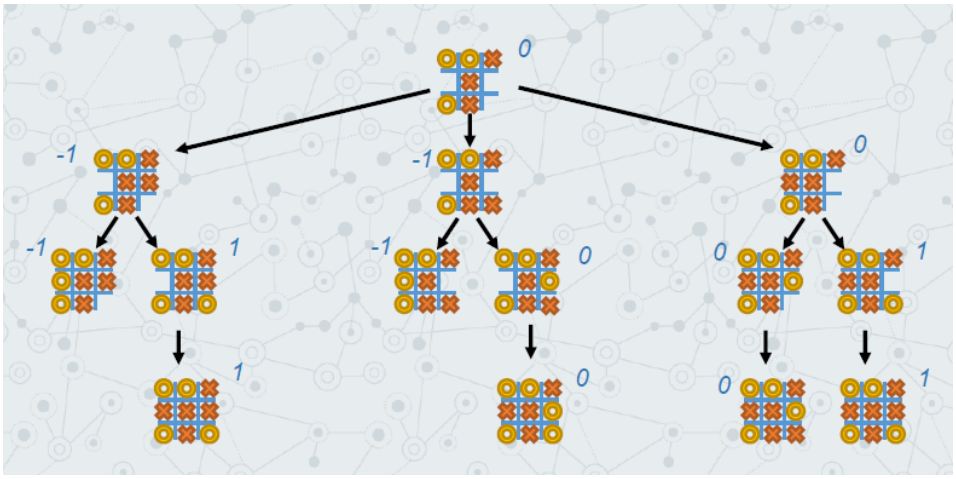


Figure 3.15 Tic Tac Toe, where the X player is the maximizing player, and the O player is minimizing player

After recursively deepening in the tree, Minimax assigns values for the leaves, then pumping them upwards, taking the maximum/minimum values for each non-terminal node (depending on the layer – the player that's about to play).

Minimax properties:

- Complete: in a finite game.
- Optimal: against a perfect opponent (because we assume the opponent will choose the best move for him).
- Time complexity: $O(b^m)$.
- Space complexity: $O(b^m)$ for BFS, $O(bm)$ for DFS.

The Need for Heuristics

In many games the game tree can very big very fast. For example, in Chess there are $\approx 10^{120}$ states (or nodes in the full game tree). In those cases we can practically open the tree only up to a certain depth bound – 2 moves ahead, 5 moves ahead… and from that another question comes up: How to evaluate these terminal nodes? For that we use *heuristics* once more. This time, since the goal state can change every turn and the search space isn't fully observable, the heuristics used for this problem are less restricted (admissibility, consistency).

Example: Connect-4 heuristic. The heuristic here is creating a map from each position on the board to a number representing *in how many ways it's possible to create 4's* that contain this square. Therefore, the corner squares are mapped to **3**, while the middle ones get **13**.

3	4	5	7	5	4	3
4	6	8	10	8	6	4
5	8	11	13	11	8	5
5	8	11	13	11	8	5
4	6	8	10	8	6	4
3	4	5	7	5	4	3

Figure 3.16 Connect-4 heuristic

Of course, there are more good heuristics to that game (e.g. its solution), and we can combine all those heuristics to a **weighted sum** of heuristics.

$\alpha - \beta$ Pruning

Even though the game tree is big, we can take advantage of the assumptions. One advantage is skipping nodes that we know we/the opponent won't choose. For that we'll use $\boldsymbol{\alpha - \beta}$ **Pruning**.
This is an upgraded version of the Minimax algorithm.

Formally: The algorithm keeps track (for **every** node) of 2 bounds:

1. **α** – the best value for the **maximizing** player.
2. **β** – the best value for the **minimizing** player.

Thus, if α ≥ β then we can conclude that one of the players *won't* choose the current node (depends on the node's layer) and we can cut the node's branch from the tree.

$\alpha - \beta$(v, depth, α = −∞, β = ∞, isMaxNode)

if $depth = 0$ or v is a terminal node then return $u(v)$

else if isMaxNode then

    for $s \in succs(v)$ do

        $\alpha \leftarrow \max_{s \in succs(v)}(\alpha,\ \alpha - \beta(s, depth - 1, \alpha, \beta, false))$

        if α ≥ β then **break** // β cut-off

    return α

else (isMaxNode is false) then

    for $s \in succs(v)$ do

        $\beta \leftarrow \min_{s \in succs(v)}(\beta,\ \alpha - \beta(s, depth - 1, \alpha, \beta, true))$

        if α ≥ β then **break** // α cut-off

    return β

Figure 3.17 $\alpha - \beta$ Pruning algorithm. For an animated example, check out this video by Sebastian Lague

<u>$\alpha - \beta$ Pruning properties:</u>

- Complete: in a finite game.
- Optimal: against a perfect opponent (because it cuts branches under the assumption the opponent is a perfect player).
- Time complexity: $O(b^m)$.
  With perfect ordering it can be reduced to $O(b \cdot 1 \cdot b \cdot 1 \ldots) = O\left(b^{\frac{m}{2}}\right)$.
- Space complexity: $O(b^m)$ for BFS, $O(bm)$ for DFS.

<u>Expectimax</u>

What if the opponent (the minimizing player) is a random player (choosing moves at random)? Then the opponent is clearly not an optimal player. But can we make some changes to the original Minimax algorithm, so it fits this problem (hint – $\mathbb{E}$).

If we know the probabilities for choosing each move, instead of taking the minimum value for minimizing nodes we can take the **Expected** value. If a minimizing (random) node v can choose between nodes $v_1, \ldots, v_n$, with probability of p_i and value of u_i, then $u(v) = \sum_{i=1}^{n} u_i p_i$.

Expectimax(v, depth, isMaxNode)
if $depth = 0$ or v is a terminal node then return $u(v)$ else if isMaxNode then return $\max_{s \in succs(v)}(Expectimax(s, depth-1, false))$ else (isMaxNode is false) then return $\sum_{s \in succs(v)}\big(Expectimax(s, depth-1, true) \cdot \mathbb{P}(s \mid v)\big)$

Figure 3.18 Expectimax algorithm. I like Meanimax more

4 Propositional Logic

4.1 Definitions

Some problems in the real life are logic ones, or can be modelled as logic problems, such as proving things and *planning* (we'll get to that later). We will define what propositional logic is and then see how to convert it to our beloved graph search.

To define propositional logic, we will use the term connectors which are $\{\wedge, \vee, \neg, \Rightarrow, \Leftrightarrow\}$.

Definitions:

1. The ***connectors*** are defined as follows:
   - $x \wedge y$ is (x and y)
   - $x \vee y$ is (x or y)
   - $x \Rightarrow y$ is (if x then y)
   - $x \Leftrightarrow y$ is (x iff y)
   - $\neg x$ is (not x)
2. A ***sentence*** in propositional logic is one of the following:
   - A variable, $p, q, \ldots$
   - True \ False
   - $S_1\ op\ S_2$ when $op \in \{\wedge, \vee, \Rightarrow, \Leftrightarrow\}$ and S_1, S_2 are sentences
   - $\neg S_1$ when S_1 is a sentence

We can take real world statements and convert them to propositional logic. For example:

If there are no clouds, it cannot rain → denote $clouds = p$ and $rain = q$. Then the statement is

$$\neg p \Rightarrow \neg q$$

Like in every logic, we need rules/axioms to proof claims. We will choose a group of axioms which contains enough to proof everything.

Definition: Let ϕ, ψ be sentences and we will write $\phi \vdash \psi$ and say ψ is ***inferred*** from ϕ if there exists a **proof** from ϕ to ψ using only the axioms we chose.

Note: If $\phi \vdash \psi$ is an **axiom** we can write $\phi \vdash \psi$ defined as "ψ is inferred from ϕ". Examples of axioms are: $p \vdash p$, $\neg\neg p \vdash p$ and $(p \wedge q) \vdash p$.

With these definitions we now can think of proofs in a more formal way. So, we see it is only *syntax* (things that follow from the axioms) we need to follow, but we know there are statements that are *semantically* correct (where things get true/false values), which leads to the next definition.

Definition: A ***model*** to a sentence is a **function** from its variables to $\{True, False\}$.

Using models, we now can calculate the **truth of the sentence in the model** in the following way: Create a truth table that contains all models, then by the definitions above conclude whether the sentence is true or false. This is called *model enumeration*.

We can calculate the value of any sentence by doing this process recursively.

P	Q	$\neg P$	$P \wedge Q$	$P \vee Q$	$P \Rightarrow Q$	$P \Leftrightarrow Q$
false	*false*	*true*	*false*	*false*	*true*	*true*
false	*true*	*true*	*false*	*true*	*true*	*false*
true	*false*	*false*	*false*	*true*	*false*	*false*
true	*true*	*false*	*true*	*true*	*true*	*true*

Figure 4.1 Model enumeration

Definitions:

1. If $S \Rightarrow p$ is true in every model, then we can write $S \vDash p$ and the claim is ***valid***.
2. S is ***satisfiable*** if there exists a model that S is true in that model.
3. S is ***unsatisfiable*** if there doesn't exists a model that S is true in that model.

More Definitions:[3]

1. We will say that a system of inference S (set of sentences) is ***sound*** if for every sentence p, it holds that

$$S \vdash p \Rightarrow S \vDash p$$

2. We will say that a system of inference S (set of sentences) is ***complete*** if for every sentence p, the following is true:

$$S \vDash p \Rightarrow S \vdash p$$

So now we can check if p can be inferred from S by finding a *proof* – $S \vdash p$, or checking that in *every* model the formula $S \Rightarrow p$ is true – $S \vDash p$.

4.2 AI'ing Propositional Logic

In order to encode this concept to a language that a computer can understand, we will denote our knowledge of the world as ***KB*** (knowledge base), which is the set of sentences S. Assuming this KB the computer now can tell us whether it ***entails*** α.

So, as always, the naïve approach to figure out if α is inferred by KB is to just go through all the models and check if KB is true then if α is true. This approach is called ***inference by enumeration***.

Time Complexity: $O(2^n)$ and proven to be in NP-complete.

[3] In logic course we prove that $S \vDash p \Leftrightarrow S \vdash p$ with our known set of inferences S (set of axioms).

Space Complexity: $O(n)$ with DFS.

Now we will try and improve our average run time, because it is in NP, we don't hope to improve the worst case (since it'll always be bad).

Modus Ponens: A rule that claims $\{\alpha_1, \dots, \alpha_n, \alpha_1 \wedge \dots \wedge \alpha_n \vdash \beta\} \vDash \beta$.

Modus Ponens leads us to 2 algorithms that use this rule to find a solution quickly.

Forward Chaining

Definition: A ***Horn clause*** is a clause of the form $(A \wedge B) \rightarrow C$, where A is a variable or a conjunction of variables ($\wedge$ between variables), and B, C are any variables.

The idea is to use add to the KB any conclusion results from it (which iteratively increases the KB) until α is concluded **or** nothing more can be concluded (that is, failure).

```
function PL-FC-ENTAILS?(KB, q) returns true or false
  inputs: KB, the knowledge base, a set of propositional Horn clauses
          q, the query, a proposition symbol
  local variables: count, a table, indexed by clause, initially the number of premises
                   inferred, a table, indexed by symbol, each entry initially false
                   agenda, a list of symbols, initially the symbols known in KB
  while agenda is not empty do
      p ← POP(agenda)
      unless inferred[p] do
          inferred[p] ← true
          for each Horn clause c in whose premise p appears do
              decrement count[c]
              if count[c] = 0 then do
                  if HEAD[c] = q then return true
                  PUSH(HEAD[c], agenda)
  return false
```

Figure 4.2 Forward chaining algorithm

Forward chaining is **complete** for Horn KBs (KBs that contain only Horn clauses).

In other words, the idea is to try and prove as much as we can in hope to reach the desired result. Here is an example: the number is how much unsatisfied edges there are to the sentence.

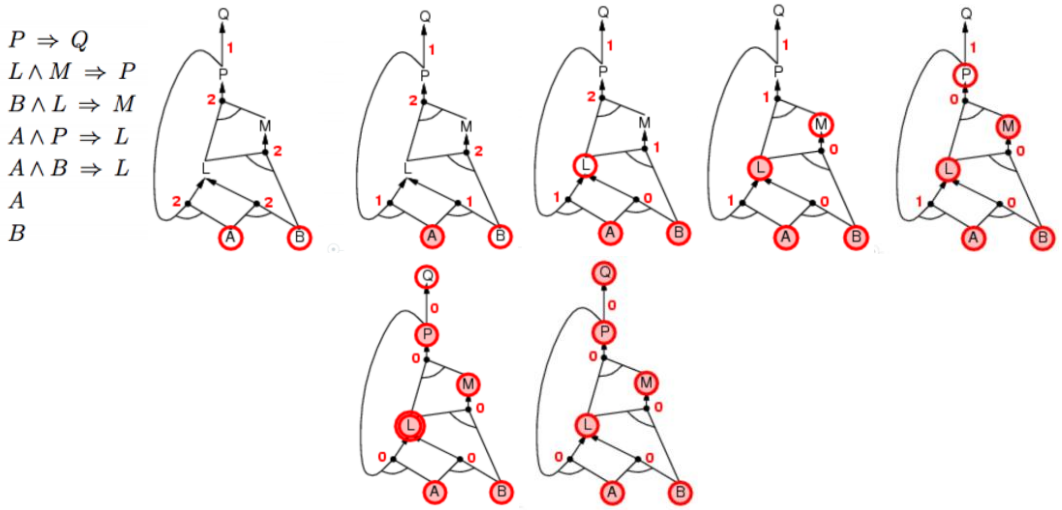


*Figure 4.3 Forward chaining example. The **red circles** are what's concluded by the KB, the red rings are what's has been concluded at the last iteration, and the numbers written next to the vertices are how many unsatisfied edges there are (which are connected to that vertice)*

Backward Chaining

The idea is like Forward Chaining but from the *end*. Backward chaining starts from what we want to prove and goes **backwards** until we find assumptions we can use (the *KB*), then backtracks and fills all the required to get the proof, and the conclusion.

Avoiding loops can be done by not expanding an expanded node and not trying to prove something more than once.

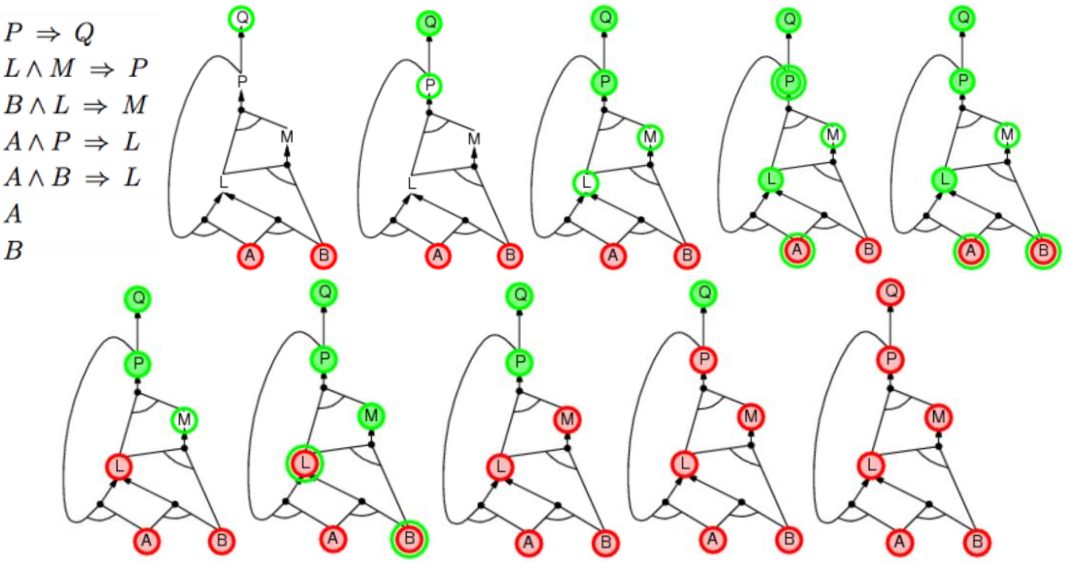


Figure 4.4 Backward chaining example. The top left figure is the KB*; the green ring is what's to be proved, the **green circles** are what's assumed to be correct, and the **red circles** are what's concluded from the* KB

So, forward chaining proves as much as he can until the goal is reached, meanwhile backward chaining searches the space (backwards) from the target.

Backward chaining is **complete** for Horn KBs, and can be *less* than linear in the size of the KB (since this is a *recursive* algorithm).

4.3 Conjunctive Normal Form (CNF) KBs

Definition: ***Conjunctive Normal Form*** a sentence in the form of conjunction of disjunctions of literals or their negations.

Example: $(A \vee B \vee \neg C) \wedge (\neg A \vee D), (A \vee \neg B) \wedge (B \vee \neg C \vee \neg D)$.

But why does CNF get a subsection of its own? As we'll see, CNFs are a nice and simple structure that simplify more complex sentences and many times are found in nature.

First thing – converting a propositional logic to CNF. The algorithm is simple and *logical*:

1. Eliminate $x \Leftrightarrow y$ by replacing it with $(x \Rightarrow y) \wedge (y \Rightarrow x)$.
2. Now eliminate $x \Rightarrow y$ by replacing it with $\neg x \vee y$.
3. Push $\neg$ into the parentheses via de-Morgen laws.
4. Apply distribution over $\vee$,$\wedge$ to get the CNF.

So now we can solve propositional logic problems *only* with CNF KBs and that's enough!

Note:

1. Due to Modes Ponens, if we know $l_i = \neg m_j$ and we have a $(l_1 \vee l_2 \ldots \vee l_n) \wedge (m_1 \vee \ldots \vee m_k)$ we can remove l_i, m_j from the sentence **without** changing its soundness. This is called ***resolution***. It will be **very** useful from now on.
2. $KB \models p$ iff $KB \Rightarrow p$ iff $KB \wedge \neg p$ is unsatisfiable.

These observations lead us to form the following algorithm:

```
function PL-RESOLUTION(KB, α) returns true or false
  inputs: KB, the knowledge base, a sentence in propositional logic
          α, the query, a sentence in propositional logic
  clauses ← the set of clauses in the CNF representation of KB ∧ ¬α
  new ← { }
  loop do
      for each C_i, C_j in clauses do
          resolvents ← PL-RESOLVE(C_i, C_j)
          if resolvents contains the empty clause then return true
          new ← new ∪ resolvents
      if new ⊆ clauses then return false
      clauses ← clauses ∪ new
```

Figure 4.5 Resolution

The idea is: Given sentences of the form $l_1 \wedge l_2 \wedge \ldots \wedge l_n \wedge \neg\alpha$, we convert it to graph (where the vertices are the clauses, and the edges mark direct conclusions) and combine any 2 clauses to remove some of the variables in the previous sentences. If we reach an empty clause then we reached a contradiction – thus $l_1 \wedge l_2 \wedge \ldots \wedge l_n \wedge \neg\alpha$ is **false** and $l_1 \wedge l_2 \wedge \ldots \wedge l_n$ is **true**,

therefore $\boldsymbol{l_1 \wedge l_2 \wedge \ldots \wedge l_n \vdash \alpha}$. If we did all the options possible and couldn't find a contradiction the algorithm (and us) concludes that $l_1 \wedge l_2 \wedge \ldots \wedge l_n \vdash \neg\alpha$.

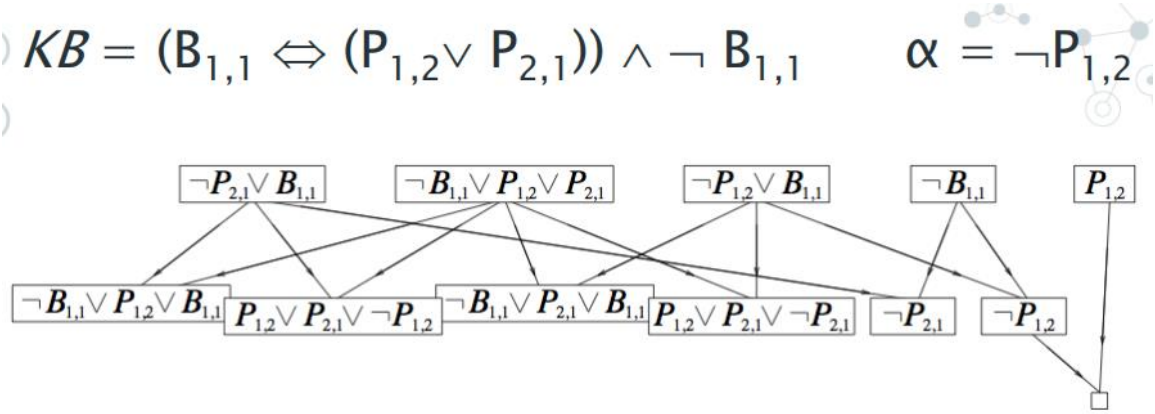


*Figure 4.6 Resolution example. The empty vertice (or clause) in the bottom right indicates that ¬α is **false**, thus α is **true** (and more formally, $l_1 \wedge l_2 \wedge \ldots \wedge l_n \vdash \alpha$)*

Note: The goal now is searching for the empty clause, and as with searching heuristics always **love** to jump into the mix.

We will see 2 different algorithms to search this space, one is based on complete backtracking search and one is local search.

DPLL

1. Choose a literal.
2. Assign a value to it.
3. Simplify the formula.
4. Recursively check if the simplified formula is satisfiable.
    a. If this is the case, the original formula is satisfiable.
    b. Otherwise, the same recursive check is done assuming the opposite truth value for the literal.

And given a clause α, DPLL tries to conclude the empty clause from $KB \wedge \neg\alpha$.

It's obvious that DPLL can work much better with heuristics than on its own. 2 helpful heuristics to the DPLL are:

1. **Pure symbolic heuristic:** If the variable appears with the same sign in all clauses, assign that sign to the variable.
2. **Unit clause heuristic:** If the clause is a variable, assign only its sign to the variable.

The bonus of DPLL over the naïve search is that DPLL can try (using more heuristics) to get an early termination and successfully does so most of the time, for example

- A clause is true if any literal is true.
- A sentence is false if any clause is false.

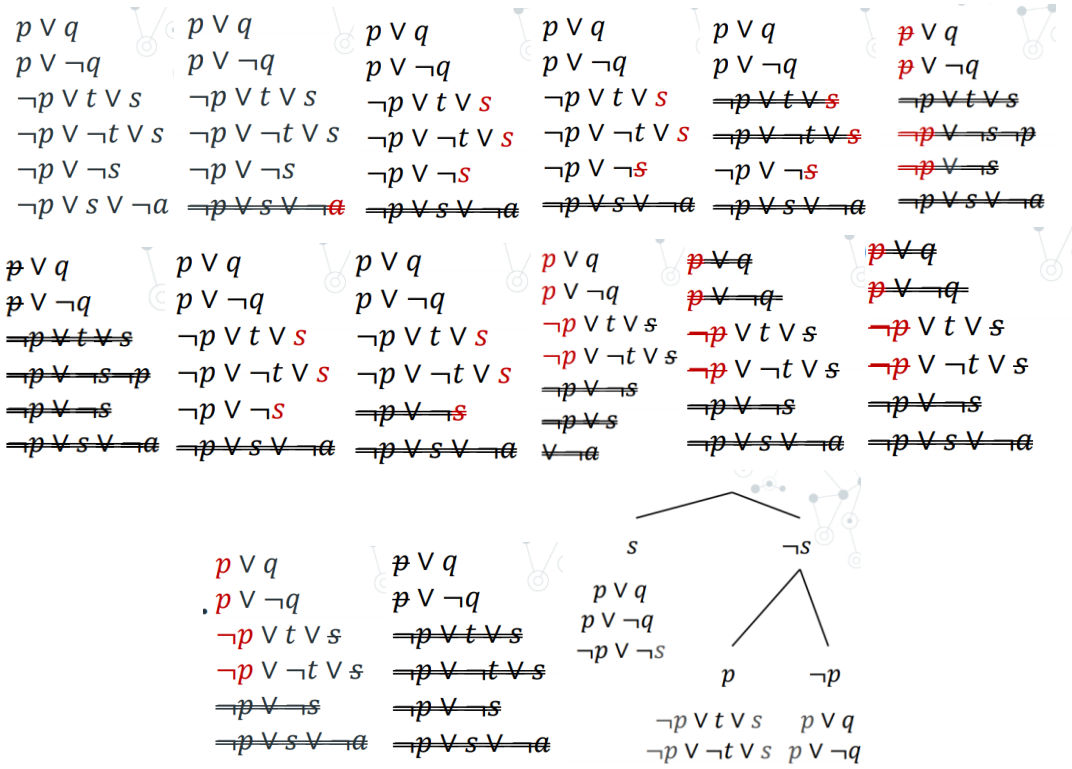


Figure 4.7 DPLL illustration. a is what we want to prove/disprove. Since each path lead to an early termination ($X \wedge \neg X$) the conclusion is $KB \vdash a$. The last figure is the tree that summarizes the DPLL's run (where gray literals are false)

And notice we explored very little (4) of all the possible assignments (16).

Since this is a **complete** backtracking algorithm, DPLL's time complexity is $O(2^n)$, however most of the times it checks much less (using heuristics for searching and terminating).

WalkSAT

1. Pick a random assignment.
2. Pick a random unsatisfied clause.
3. Select and flip a variable from that clause.
    a. With probability p, pick a random variable.
    b. With probability $(1 - p)$, pick greedily (a variable that minimizes the number of unsatisfied clauses).
    c. If we found a solution return it.
4. Repeat steps 2-3 to predefined maximum number of flips.
5. If no solution found, restart.

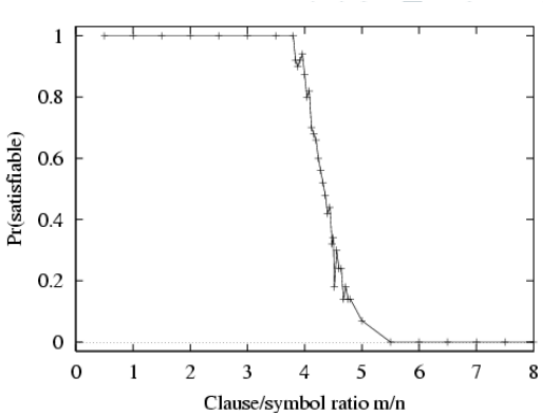


Figure 4.8 WalkSAT's probability of finding a valid assignment as a function of the clause/symbol ratio

Intuitively, the WalkSAT algorithm should perform well with the minimum conflict heuristic and randomness. In practice WalkSAT outperforms DPLL, and with using local search techniques its chances of finding an optimal solution is good.

Interestingly, when the clause/symbol ratio is near 4.3, both algorithms' runtimes increase significantly (similar to what we've seen in Solving CSPs).

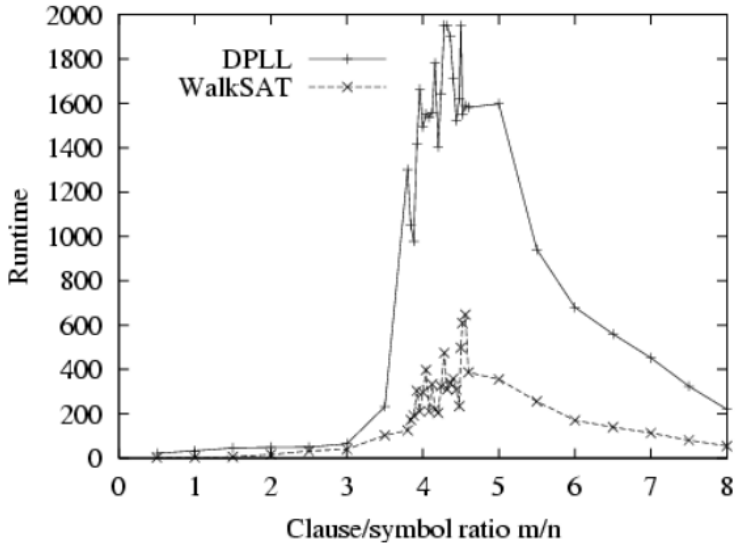


Figure 4.9

5 First Order Logic

5.1 Definitions

Like in the last section, we will define a *new* logic with its own rules and try to solve it by using the propositional logic. So, this section's structure will be similar to the previous section's.

Definition: An ***atomic sentence*** in first order logic is one of the following:

1. ϕ is a constant.
2. ϕ is a variable.
3. $\phi = R(t_1, \dots, t_n)$ when t_i is an atomic sentence.

Definition: A ***sentence*** in first order logic is one of the following:

1. $\phi = \forall x[\psi]$ when ψ is a sentence.
2. $\phi = \exists x[\psi]$ when ψ is a sentence.
3. $\phi = \psi \; op \; \tau$ when ψ, τ are sentences and op is a connector.
4. ϕ is an atomic sentence.
5. $\phi = (\psi = \tau)$ when ψ, τ are atomic sentences.
6. $\phi = \neg\psi$ when ψ is a sentence.

Examples: $\exists x\big[\forall z[Loves(x,z)]\big]$, $\forall x[Love(x) = Love(x)]$.

5.2 AI'ing First Order Logic

As defined before, CNF is a conjunction of sentences containing only disjunctions (∨) between them.

The next steps show how to convert a sentence in **first order logic** to CNF:

1. Eliminate $x \Leftrightarrow y$ by replacing it with $(x \Rightarrow y) \wedge (y \Rightarrow x)$.
2. Now eliminate $x \Rightarrow y$ by replacing it with $\neg x \vee y$.
3. Distribute negation:
   a. $\neg(\phi \wedge \psi)$ becomes $\neg\phi \vee \neg\psi$.
   b. $\neg(\phi \vee \psi)$ becomes $\neg\phi \wedge \neg\psi$.
   c. $\neg\forall x[\phi]$ becomes $\exists x[\neg\phi]$.
   d. $\neg\exists x[\phi]$ becomes $\forall x[\neg\phi]$.
4. Rename variables to have unique names.
5. Replace existentials: If the existential doesn't depend on anything add it add **constant**, else as a **function**:

$\exists x \forall y\big(P(x,y)\big)$ becomes $\forall y\big(P(A,y)\big)$

$\forall y \exists x \Big(\big(Q(x) \wedge P(x,y)\big) \vee \forall z \exists w\big(R(w)\big)\Big)$

becomes

$\forall y \Big(\big(Q\big(H(y)\big) \wedge P\big(H(y),y\big)\big) \vee \forall z \Big(R\big(I(y,z)\big)\Big)\Big)$

Figure 5.1

6. Remove universals. Every variable is universally quantified without needing to right it down.

7. Distribute over ∨,∧ to get CNF form.
8. Replace operators: Replace the conjunction $\phi_1 \wedge \ldots \wedge \phi_n$ with the set of clauses $\{\phi_1, \ldots, \phi_n\}$.
9. Rename variables such that each variable appears only in 1 clause.

Now we can once again only worry about CNF sentences!

The next problem we have to deal with is the following: $G(x), G(Alice)$, we want to be able to find a **substitution** to be able to understand the formulas the same way. This process is called **unification**.

For example, Knows(John, x) and Knows(y, z), we can find the next substitutions $\theta = \{y/John, x/z\}$ and $\theta = \{y/John, x/John, z/John\}$. The first one is better because it is more general, we would like to be able to find the most general substitution and call it **MGU** (most general unifier). The MGU is **unique** up to renaming variable names.

<u>The MGU algorithm</u>

1. we get x, y, θ as input
2. if x == y
    a. return without adding anything
3. if x and y are constants
    a. then failure because we can't replace x with y or y with x
4. if x is a variable or y is a variable
    a. add replacing x with y (or y with x, depends which is a variable), add only one of the replacements and update the mapping and the sentences according to this replacement
    b. return
5. call recursively on all elements in x, y

We are happy now because we can find MGU rather fast! :D

Note: We saw last week that $(P \vee Q) \wedge (\neg Q \vee R) \vdash (P) \wedge (R)$. Now we can do something similar with first order logic: If $(\Phi \wedge \Psi)$ and if Φ, Ψ are clauses and contain $\phi, \neg\phi$ respectively, then

$$(\Phi \wedge \Psi) \vdash (\Phi \setminus \{\phi\}) \wedge (\Psi \setminus \{\neg\phi\})$$

Now we want to decide whether the KB infers ϕ.

Forward Chaining

The idea is to try and prove as much as we can in hope to reach the desired result.

Example: The law says that it is a crime for an American to sell weapons to hostile nations. The country Nono, an enemy of America, has some missiles and all its missiles were sold to it by Colonel **West**, who is American. Prove that Colonel West is a *criminal*.

Forward chaining algorithm

Functoin $FC(KB, \alpha)$ returns a substitution or $False$
repeat until $new = \emptyset$
  $new = \{\ \}$
  For each sentence r in KB do
    $(p_1 \wedge p_2 \wedge \cdots \wedge p_n \Rightarrow q) \leftarrow standarize - apart(r)$
    For each θ such that $subset(\theta, p_1 \wedge p_2 \wedge \cdots \wedge p_n)$
    $= subset(\theta, p'_1 \wedge p'_2 \wedge \cdots \wedge p'_n)$ for some $p'_1, p'_2, \ldots, p'_n$ in KB
    $q' \leftarrow subset(\theta, q)$
    If q' is not renaming of a sentence already in KB or in new then
      Add q' to new
      $\phi = mgu(q', \alpha)$
      If ϕ is not fail then return ϕ
  Add new to KB
Return $false$

Figure 5.2 Forward chaining algorithm

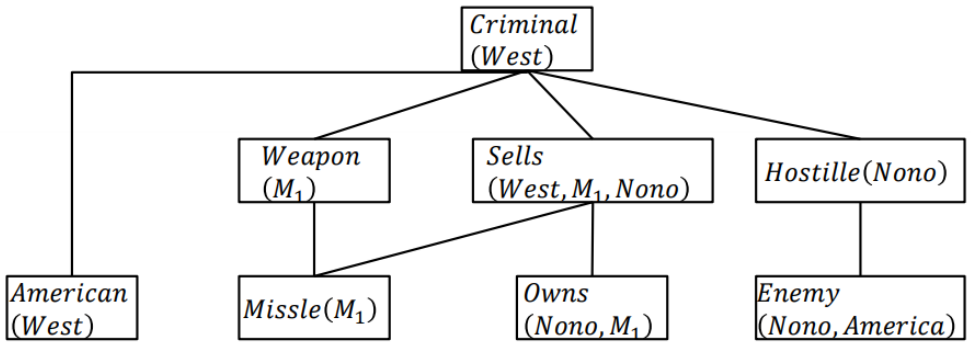


Figure 5.3 Forward chaining example (from the bottom up)

Forward chaining is complete and sound for first-order definite clauses (implication clauses with no negative literals) + no functions.

Backward Chaining

The idea is like Forward Chaining but from the end. The idea is to try start from what we want to proof and go backwards until we find assumptions we can use and then backtrack and fill all the required to get the proof.

Avoiding loops can be done by not expanding an expanded node and not trying to prove something more than once.

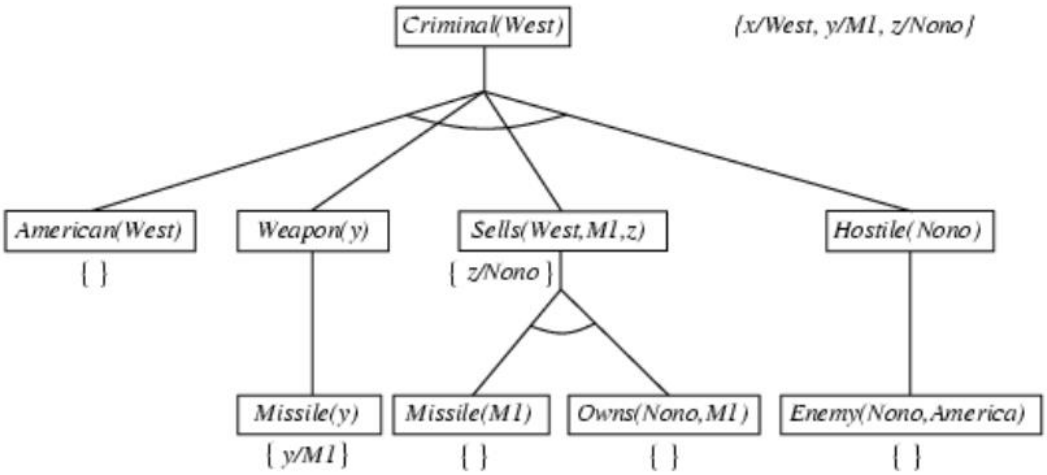


Figure 5.4 Backward chaining example (from top to bottom)

Resolution

Same idea as in propositional logic – create all clauses from $KB \cup \{\neg\alpha\}$ (equivalent to $KB \wedge \neg\alpha$) and try to reach the empty clause.

1	$\{P\}$		1	$\{P\}$		1	$\{P\}$	
2	$\{\neg P, Q\}$		2	$\{\neg P, Q\}$		2	$\{\neg P, Q\}$	
3	$\{\neg Q, R\}$		3	$\{\neg Q, R\}$		3	$\{\neg Q, R\}$	
4	$\{\neg R\}$		4	$\{\neg R\}$		4	$\{\neg R\}$	
			5	$\{Q\}$	1,2	5	$\{Q\}$	1,2
			6	$\{\neg P, R\}$	2,3	6	$\{\neg P, R\}$	2,3
			7	$\{\neg Q\}$	3,4	7	$\{\neg Q\}$	3,4
						8	$\{R\}$	3, 5
						9	$\{R\}$	1,6
						10	$\{\neg P\}$	4,6
						11	$\{\neg P\}$	2,7
						12	$\{\}$	5,8

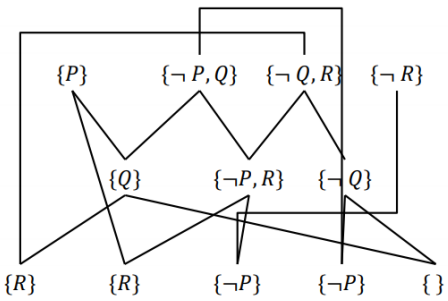


Figure 5.5 Resolution example. 1-4 is the KB, 5-12 are the conclusions (left); graph representation of the algorithm (right)

Resolution can be thought of as the bottom-up construction of a search tree, where the leaves are the clauses produced by KB and the negation of the goal:

- When a pair of clauses generates a new **resolvent clause**, add a new node to the tree with arcs directed from the resolvent to the two parent clauses.
- Resolution succeeds when a node containing the **false** (or empty) **clause** is produced, becoming the *root* node of the tree.
- A **strategy** is complete if its use guarantees that the empty clause (i.e. false) can be derived whenever it is entailed.

And again, we reach the same problem – how to efficiently traverse the created graph by the clauses? And the answer is **search** with ***heuristics*** (like always 😊).

Breadth-first Search Heuristic:

The levels (or layers) are:

1. Level 0 are the original clauses given and the negation of the goal.
2. The k level clauses are all the resolution produced from levels $k-1$ and below.

So, the idea of Breadth-first Search heuristic is to search the graph by the **level** of the resolution.

1	$\{P,Q\}$	KB	9	$\{R\}$	3, 5	17	$\{\}$	4, 9
2	$\{\neg P,R\}$		10	$\{Q\}$	4, 5	18	$\{R\}$	3, 10
3	$\{\neg Q,R\}$		11	$\{R\}$	3, 6	19	$\{\}$	8, 10
4	$\{\neg R\}$	Goal neg	12	$\{P\}$	4, 6	20	$\{\}$	4, 11
			13	$\{Q\}$	1, 7	21	$\{R\}$	2, 12
5	$\{Q,R\}$	1,2	14	$\{R\}$	6, 7	22	$\{\}$	7, 12
6	$\{P,R\}$	1, 3	15	$\{P\}$	1, 8	23	$\{R\}$	3, 13
7	$\{\neg P\}$	2, 4	16	$\{R\}$	5, 8	24	$\{\}$	8, 13
8	$\{\neg Q\}$	3, 4				25	$\{\}$	4, 14
						26	$\{R\}$	2, 15

Figure 5.6 BFS resolution heuristic

It is complete but very inefficient. For example, in the figure to the right, the algorithm produces **22** new clauses where only **2** are needed!

Deletion Strategies

Eliminate certain clauses before they are ever used. There are a couple of heuristics to help us delete non relevant information.

1. **Pure literal elimination heuristic:** Remove any clause containing a "pure literal" – a literal that has no complementary instance in the data base. In the example below, any clause S is deleted since there is no $\neg S$, thus we can assign $S \leftarrow true$ and satisfy these clauses, independently of the others.

$$\{\neg P, \neg Q, R\}, \{\neg P, S\}, \{\neg Q, S\}, \{P\}, \{Q\}, \{\neg R\}$$

$$\{\neg P, \neg Q, R\}, \{\neg P, S\}, \{\neg Q, S\}, \{P\}, \{Q\}, \{\neg R\}$$

Figure 5.7

2. **Tautology elimination heuristic:** Eliminate clauses that contain complementary literals – **tautologies**: $\{P(G(A)), \neg P(G(A))\}$ and $\{P(x), Q(y), \neg Q(y), R(z)\}$ are always true.
3. **Subsumption elimination heuristic:**
   a. A clause F ***subsumes*** a clause Y iff there is a substitution σ such that $F\backslash\sigma \subset Y$
   b. $\{P(x), Q(y)\}$ subsumes $\{P(A), Q(v), R(w)\}$, since the substitution $x\backslash A$, $y\backslash v$ makes the first a subset of the second. So, we can throw away the second, as it's concluded from the first clause.

Length Heuristic (Shortest-clause heuristic): generate clauses with the **fewest** literal first.

Unit Resolution heuristic: create only clauses which contain one sentence in them. **Not complete** but helps us search in a specific direction. However, unit resolution **is** complete for Horn KBs (Horn clauses are defined in FOL similarly to propositional logic).

Linear Resolution heuristic: At level k we can resolve clauses only from level 0 and level $k-1$ clauses.

Input Resolution heuristic: At level k we can resolve clauses only from level 0 and level $k-1$ clauses and must use at least **one** clause from level 0. **Not complete** but helps us search in a specific direction. It is **complete** for Horn KBs.

Note: $Unit = Input$, meaning there's a unit refutation iff there's an input refutation.

Set of Support Resolution heuristic

We start by saving a set Γ to be the negation of the goal and for every clause resolved we must use a clause from Γ and then add it to Γ. This is **complete**. The idea is assuming the goal is wrong and find *where* is it mistaken.

1	$\{P, Q\}$	Δ
2	$\{\neg P, R\}$	Δ
3	$\{\neg Q, R\}$	Δ
4	$\{\neg R\}$	Γ
5	$\{Q, R\}$	1,2
6	$\{R\}$	3, 5
7	$\{\}$	4, 7

Figure 5.8 Set of support resolution

Ordered Resolution heuristic

We sort the clauses into literals and allow resolution only on the first literal in the clause. **Not complete** but helps us search in a specific direction. It is **complete** if the Data is stored as Horn clauses.

1	$\{P, Q\}$	Δ
2	$\{\neg P, R\}$	Δ
3	$\{\neg Q, R\}$	Δ
4	$\{\neg R\}$	Γ
5	$\{\neg P\}$	2, 4
6	$\{\neg Q\}$	3, 4
7	$\{Q\}$	1, 5
8	$\{P\}$	1,6
9	$\{R\}$	3, 7
10	$\{\}$	6, 7

Figure 5.9 Ordered resolution

Note:

- Unit and input resolutions are refutation complete for Horn KBs.
- Linear, set of support and ordered resolutions are refutation complete. Ordered resolution is also complete for Horn KBs.
- All these resolution heuristics rely on the assumption that the input is a **CNF** KB.

6 Planning

6.1 Definitions & Modelling

Consider the following: Given a start state and a set of actions, we want to reach a goal state using these actions. However, at each state only **some** of the actions can be used. How would construct a **plan** – a sequence of actions that will lead you to the goal?

Definition: We denote a ***planning problem*** as a tuple:

$$< S, I, O, G >$$

Where:

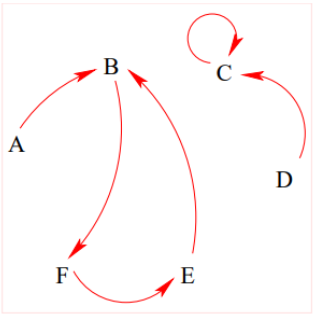
$S = \{set\ of\ state\ variables\}$
$I \in S$ *is the start state*
$G = \{set\ of\ goal\ states\} \subseteq S$
$O = \{set\ of\ actions\} \subseteq S \times S$

And we will say that π is a **plan** if $\pi = (a_1, \dots, a_n)$ is a sequence of legal actions from state I to $g \in G$.

There are many ways to model planning problems in a more concrete way. However, some are more efficient than others. So, our goal for now is to find some of these efficient ways.

The first is the naïve approach. Given $S = \{s_1, \dots, s_n\}$, we'll define a matrix M such that $M[i,j] = 1$ iff there exists an action to transition from state s_i to state s_j.

We can calculate reachability with n states by $R_n = \sum_{i=0}^{n} M^i$ and we can reach from i to j within n steps iff $R_n[i,j] = 1$. The only problem is that if S is very large then this representation will take a lot of time to calculate, as it requires storing and multiplying matrices.



	A	B	C	D	E	F
A	0	1	0	0	0	0
B	0	0	0	0	0	1
C	0	0	1	0	0	0
D	0	0	1	0	0	0
E	0	1	0	0	0	0
F	0	0	0	0	1	0

Figure 6.1 Transition matrix example

Why do we even need to have a better representation than the definition?

1. Concise model description.
2. Computation: Reveal useful information about problem's structure (could be useful for discovering a specific heuristic automatically).

Let's define languages to try and be computation efficient and concise.

Our second attempt is the ***SAS*** representation (over some state space):

1. V = group of variables with some domain for each variable.
2. I is the initial assignment to the variables in V.
3. G is a partial assignment to the variables we want to reach.
4. $O = A$ will be split into $pre(A), eff(A)$ for each action. We will be able to use action A only if the current assignment fulfils all the preconditions of $pre(A)$ and when using the effect will change the variables as the $eff(A)$ defines.

Formally, an action a is applicable in a state $s \in dom(V)$ iff $s[v] = pre(a)[v]$ whenever $pre(a)[v]$ is specified.
Applying an applicable action a changes the value of each variable v to $eff(a)[v]$ if $eff(a)[v]$ is specified.

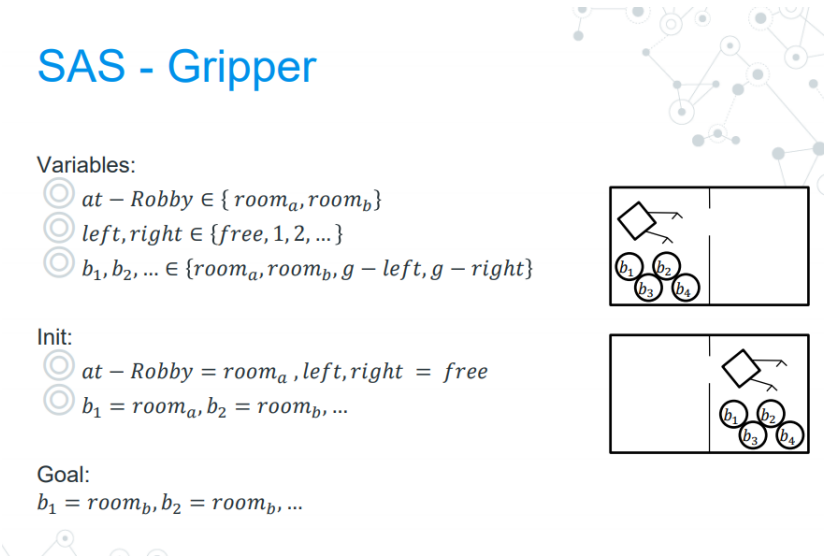


Figure 6.2 SAS representation example

This is a good attempt because we don't need a lot of variables, but the domains are not Boolean, and we remember how much we didn't like first order logic. Therefore, we will do something similar for propositional logic.

Our third attempt is the ***STRIPS*** representation (over Boolean state space):

1. V = group of variables with Boolean value for each variable.
2. I is the initial assignment to the variables in V.
3. G is a partial assignment to the variables we want to reach.
4. $O = A$ will be split into $pre(A), del(A), add(A)$ for each action. We will be able to use action A only if the current assignment fulfils all of the preconditions of $pre(A)$ and when using the action, we first delete all the variables as specified in $del(A)$ and then update all specified in $add(A)$, a little more formal $new - assigment = (old - assigment \setminus del(A)) \cup add(A)$.

Example: There's a rover on Mars, and it wants to do stuff on Mars, but the rover's creators are on Earth. So instead of manually controlling it, the programmers (or whoever put it on Mars) tell the rover to reach different goals, and the rover does that somewhat automatically using the STRIPS modelling.

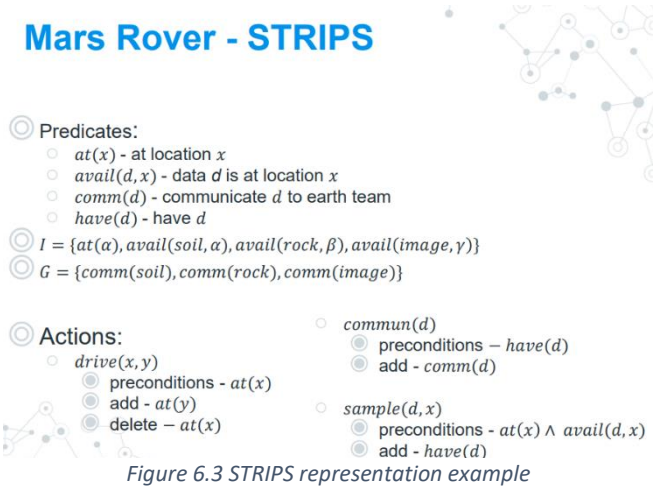


Figure 6.3 STRIPS representation example

6.2 Solving Planning

Cool, so now we can represent planning problems, but how do we solve them?

As always, we first will try the naïve approach, a planning problem can be searched so we can just use A^*, BFS, DFS and all the other algorithms to find a plan or the optimal plan, however there are many problems with this approach.

1. Firstly, for A^* we need to define a heuristic for each different planning problem, and we want one algorithm for all planning problems.
2. Local search or uninformed search? Or informed search?
3. How to prune (if we can)?

Ok, so searching from the initial state is problematic. What about doing the same search, but from the goal state and try to reach the initial state? The same problems arise as in searching from the initial state to the goal.

Therefore, now we will try to solve planning problems via new methods, that might be better than the classical searching.

Solving with SAT

Our first technique will be to use the Conjunctive Normal Form (CNF) KBs to solve the STRIPS planning language. That is, if we can convert the problem to a sat problem and extract the solution from the sat than we can solve the original problem.

Figure 6.4

Wait, but how do we encode a planning problem into a propositional equation? Many possible answers. Most (in use up to now) share:

- Time steps $0 \leq t \leq b$.
- Fact variables p_t: Is p TRUE or FALSE at time t.
- Action variables a_t: Is action a performed at time t. The size of the encoding grows linearly in b.

Ok, but what is the equation that the SAT needs to solve? Planning can be done by proving a theorem in situation calculus.

- Test the satisfiability of a logical sentence: initial state ∧ all possible action descriptions ∧ goal.
- Sentence contains propositions for every action occurrence.
- A model will assign true to the actions that are part of the correct plan and false to the others.
- An assignment that corresponds to an incorrect plan will not be a model because of inconsistency with the assertion that the goal is true.

- If the planning is *unsolvable*, the sentence will be **unsatisfiable**.

It will look something like the following code:

```
function SATPLAN(problem, T_max) return solution or failure
  inputs: problem, a planning problem
          T_max, an upper limit to the plan length
  for T= 0 to T_max do
      cnf, mapping ← TRANSLATE-TO_SAT(problem, T)
      assignment ← SAT-SOLVER(cnf)
    if assignment is not null then
       return EXTRACT-SOLUTION(assignment, mapping)
  return failure
```

Figure 6.5 SATPLAN

Finally, we have an algorithm to generate plans! But as always we want more tools. As much as we can to solve the problem.

Solving with POP

We will define another term in planning to help us define another algorithm:

Definition: ***Causal link*** – $a_p \xrightarrow{q} a_c$ records the use of a_p to produce the **precondition** q of a_c.

Definition: We denote a ***partial order plans*** as a tuple:

$$< A, O, L >$$

Where:

$A = \{set\ of\ actions\}$
$O =$ {set of ordering constraints}

$L =$ {set of causal links}

Definition: Causal links are used to detect when a newly introduced action interferes with past decisions. Such an action is called a ***threat***. Suppose that $a_p \xrightarrow{q} a_c$ is a causal link in L (of some plan {A, O, L}), and a_t is yet another action in A.

We say that at ***threatens*** ap ac if $O \cup \{a_p < a_t < a_c\}$ is **consistent**, and q is in the **delete list** of a_t.

The idea of these definitions is that sometimes we don't care about the order of the plan **unless** it ruins something else in our plan.

Now we can define the **POP** algorithm. This algorithm searches plan space:

1. Starts with the null plan.
2. Makes non-deterministic plan refinement choices until:
    - All preconditions of all actions in the plan have been supported by causal links.
    - All threatened causal links have been protected from possible interference.

Few observations on the POP:

1. A plan is consistent iff there are **no cycles** in the ordering constraints and **no conflicts** with the causal links.
2. A consistent plan with no open preconditions is a **solution**.
3. A partial order plan is executed by repeatedly choosing *any* of the possible next actions.

The POP algorithm over STRIPS problem will:

1. Start with the start state.
2. Choose a legal action and update the links until all open casual links are solved.
3. Return the plan.

Note: To ensure consistency among threats we need to update the casual links after *each* action.

The summary of the POP algorithm idea:

1. Operators on partial plans:
   - Add link from existing plan to open precondition.
   - Add a step to fulfil an open precondition.
   - Order one step w.r.t. another to remove possible conflicts.
2. Gradually move from incomplete/vague plans to complete/correct plans.
3. Backtrack if an open condition is unachievable or if a conflict is irresolvable.

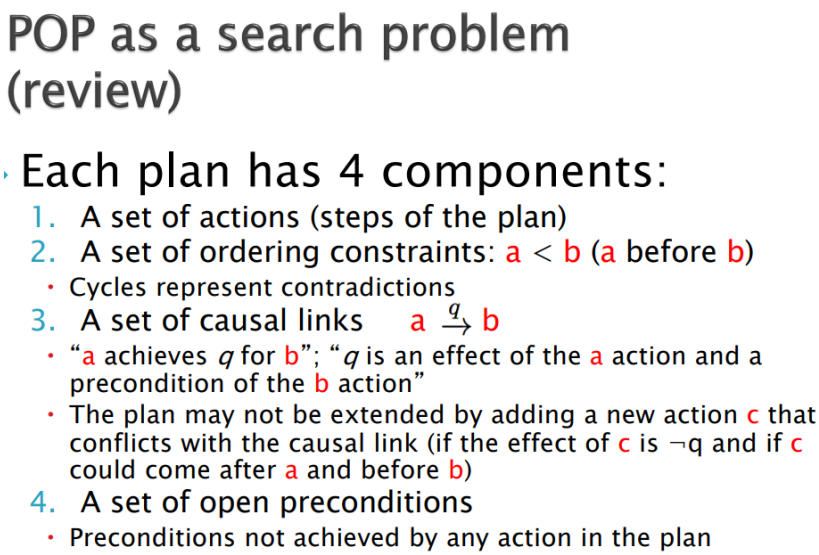


Figure 6.6 Solving with POP – overview

Graphplan

After seeing 2 different approaches to solve this question we will present one final one via **graph construction** for the STRIPS problem. ***Planning Graph*** is a special data structure that can be used to build heuristics, and they can be applied to any previous search technique.

- $P_0(s)$ is defined as the set of propositions in the state s.
- $A_i(s)$ consists of all actions that have all their preconditions' propositions in $P_{i-1}(s)$.

- $P_i(s)$ is the set of all propositions given by the add list of an action in $A_i(s)$.
- Using a **noOp** action to ensure that propositions in $P_{i-1}(s)$ persist to $P_i(s)$.

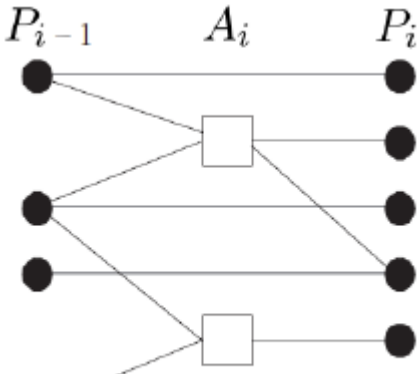


Figure 6.7 Planning graph

The construction continues until one of the following:

- The graph has levelled off, i.e. two subsequent proposition layers are **identical**.
- the goal is reachable, i.e. every goal proposition is present in a proposition layer.

One such graph is:

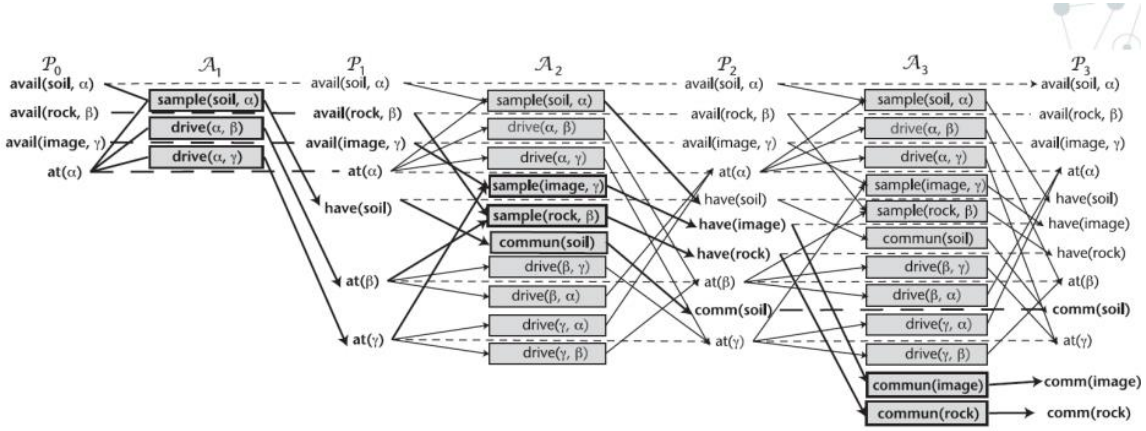


Figure 6.8 A (more realistic) planning graph

If we search this graph until all the variables we wanted appear in the same layer – we will get a relaxed plan **without** an option for one action to sabotage another. Therefore, we need to try and figure out if two actions sabotage each other.

Definition: 2 **actions** are ***mutex*** if one of the following occurs:

1. inconsistent effects: One action adds a proposition that the other action deletes.
2. interference: One action deletes the precondition of another.
3. competing needs: The actions have preconditions that are mutually exclusive at the proposition layer of the previous level.

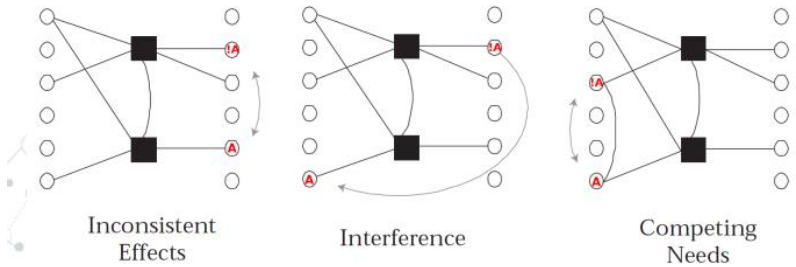


Figure 6.9 Mutex actions

Definition: 2 **propositions** are ***mutex*** if one of the following occurs:

1. One is the negation of the other (relevant for other languages) interference: One action deletes the precondition of another.
2. Inconsistent support: If all ways of achieving the propositions (that is, actions at the same level) are pairwise mutex.

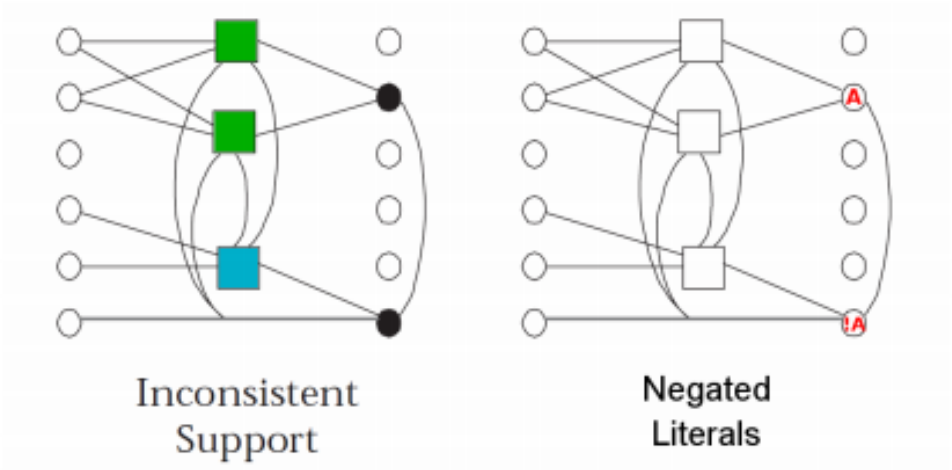


Figure 6.10 Mutex propositions

With those definitions and constructions, how do we extract a solution – a sequence of actions, avoiding mutex actions & propositions? **Graphplan**. Since most of the work is defining and constructing the planning graph, there's not much to add, and the algorithm is straight forward.

GRAPHPLAN alternates between:

1. **Graph expansion** phase
   - Stop expansion when:
     - All goal literals are present at the last proposition layer, and
     - None of them are in mutex with each other
2. **Solution extraction** phase
   - Consider each subgoal in turn
   - For each such literal at level i choose an action a at the same level, i (this is a backtrack point in case of mutex) that achieves the literal
   - Continue recursively for the set of literals at level $i - 1$ that include all preconditions needed for the actions chosen at level i

```
RPExtract(PG(s), G)
1:  Let n be the index of the last level of PG(s)
2:  for all p ∈ G ∩ P_n do /* Initialize Goals */
3:      P_n^RP ← P_n^RP ∪ p
4:  end for
5:  for i = n ... 1 do
6:      for all p ∈ P_i^RP do /* Find Supporting Actions */
7:          Find a ∈ A_{i-1} such that p ∈ eff+(a)
8:          A_{i-1}^RP ← A_{i-1}^RP ∪ a
9:      end for
10:     for all a ∈ A_{i-1}^RP, p ∈ pre(a) do/* Insert Preconditions */
11:         P_{i-1}^RP ← P_{i-1}^RP ∪ p
12:     end for
13: end for
14: return (P_0^RP, A_0^RP, P_1^RP, ..., A_{n-1}^RP, P_n^RP)
```

Figure 6.11 Graphplan in words (left) and pseudo-code (right)

Note: There are several options to how to expand the graph. Therefore, we can add *heuristics* to help us search on this graph:

1. **Max level heuristic:** Max level for any goal conjunct (admissible but inaccurate).
2. **Set level heuristic:** The level they all occur on without mutex links (better, also admissible).
3. **Level-sum:** The sum of the levels of every goal conjunct (inadmissible but works well in practice).
4. **Relaxed plan:** The length of the relaxed plan without noOps (noOp is an action that just carries the proposition to the next layer).

Sometimes, it's very hard to search all the space to find a solution if it is too large, therefore we might want to solve relaxed planning problems, store them as a pattern DB and use this as a *heuristic* for our search.

Formally,

Consider a SAS problem $\Pi = \langle V, A, I, G \rangle$.

Each variable subset $V' \subseteq V$ defines approximating abstraction

$$\Pi' = \langle V', A^{[V']}, I^{[V']}, G^{[V']} \rangle$$

that is obtained by projecting the initial state, the goal, and all the actions preconditions and effects onto V'

$$\forall a \in A, a^{[V']} = \langle pre(a)^{[V']}, eff(a)^{[V']} \rangle$$

and so

$$A^{[V']} = \{a^{[V']} | a \in A \wedge eff(a)^{[V']} \neq \emptyset\}$$

Given a SAS problem $\Pi = \langle V, A, I, G \rangle$

- Select subsets $V_1, \ldots, V_m$ of V such that for $1 \leq i \leq m$, the size of V_i is sufficiently small to perform exhaustive-search reachability analysis in $\Pi^{[V_i]}$
- Let $h^{[V_i]}(s)$ be the optimal cost of achieving the abstract goal $G^{[V_i]}$ from the abstract state $s^{[V_i]}$ in $\Pi^{[V_i]}$
- Clearly, each $h^{[V_i]}(\cdot)$ is an admissible estimate of the true cost $h^*(\cdot)$
- $\max_{1 \leq i \leq m} h^{[V_i]}(s)$ is always admissible
- $\sum_{1 \leq i \leq m} h^{[V_i]}(s)$ is sometimes admissible

Figure 6.12

Theorem: Let $\Pi = (V, A, I, G)$ be a SAS planning task and $V_1, \ldots, V_k \subseteq V$.
If there exists no operator $a \in A$ that has an effect on a variable $v_i \in V_i$ and on a variable $v_j \in V_j$ for some $i \neq j$, then $\sum_{1 \leq i \leq k} h^{[V_i]}(s)$ is an **admissible and consistent** heuristic.

Meaning, summing the optimal costs over a set of **cliques** is an admissible and consistent heuristic.

Now we can try and use this to find the best combination of patterns that will be addmisble:

Finding additive pattern sets

Given a pattern collection $\mathcal{P}$ (i.e., a set of subsets), we can use this information as follows:

1. Build the compatibility graph for $\mathcal{P}$
   - Vertices correspond to patterns $V \in \mathcal{P}$
   - There is an edge between two vertices iff no operator affects both incident patterns.

2. Compute compatibility graph of the graph. These correspond to maximal additive subsets of $\mathcal{P}$.
   - Computing large cliques is an NP-hard problem, and a graph can have exponentially many maximal cliques.
   - However, there are output-polynomial algorithms for finding all maximal cliques which have led to good results in practice.

Figure 6.13

And the result is the **canonical heuristic function**:

Canonical Heuristic Function

Let $\Pi = \langle V, A, I, G \rangle$ be a SAS Planning task and $\mathcal{P}$ be a planning task, and let $\mathcal{P}$ be a pattern collection for Π. The canonical heuristic $h^{\mathcal{P}}$ for pattern collection $\mathcal{P}$ is defined as

$$h^{\mathcal{P}}(s) = \max_{\mathcal{D} \in \text{cliques}(\mathcal{P})} \sum_{V \in \mathcal{D}} h^{[V]}(s)$$

where cliques($\mathcal{P}$) is the set of all maximal cliques in the compatibility graph for $\mathcal{P}$.

For all choices of $\mathcal{P}$, heuristic $h^{\mathcal{P}}$ is admissible and consistent.

Figure 6.14

Example: Consider the following task:

- $V = \{v_1, v_2, v_3\}$.
- $\mathcal{P} = \{P_1, \dots, P_4\}$ where $P_1 = \{v_1, v_2\}$, $P_2 = \{v_1\}$, $P_3 = \{v_2\}$, $P_4 = \{v_3\}$.
- Each operator affects a single variable, except for some operators that affect v_1 and v_3.

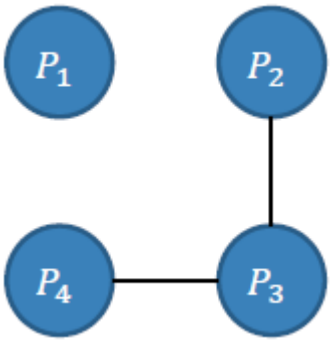


Figure 6.15 Canonical heuristic function – maximal cliques example

So, the maximal cliques in the compatibility graph for $\mathcal{P}$ are $\{P_1\}, \{P_2, P_3\}, \{P_3, P_4\}$, and the *canonical heurstic* of this problem is:

So the canonical heuristic function $h^{\mathcal{P}}$ is:

$$\max \begin{Bmatrix} h^{P_1}, \\ h^{P_2} + h^{P_3}, \\ h^{P_3} + h^{P_4} \end{Bmatrix} = \max \begin{Bmatrix} h^{\{v_1, v_2\}}, \\ h^{\{v_1\}} + h^{\{v_2\}}, \\ h^{\{v_2\}} + h^{\{v_3\}} \end{Bmatrix}$$

Figure 6.16 Canonical heuristic function – example

7 Markov Decision Processes (Markov Chains)

Markov Systems are ways of modelling **probabilistic** environments. They both assume the *Markov property*: What happens at state s depends **only** on what's in the state s. In addition, there are rewards *along the way*, like heuristics. However, since immediate profit often worth more than future profit, we add a *discount factor* – that exponentially reduces the profit with progress of time.

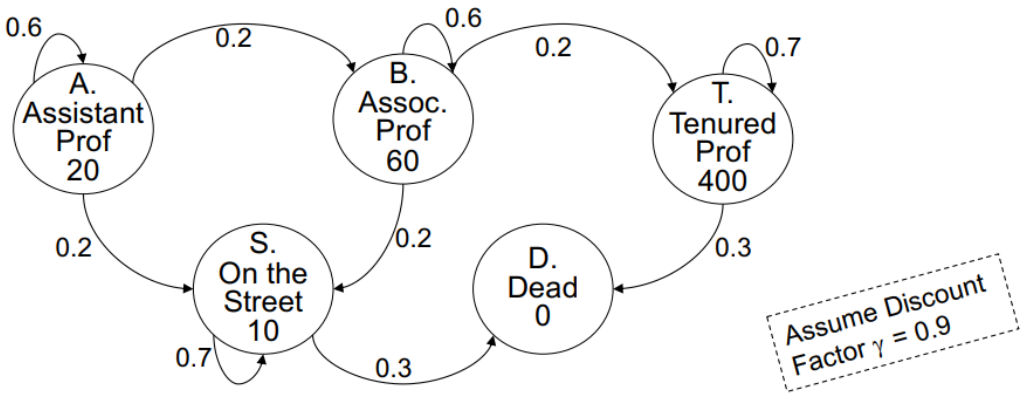


Figure 7.1 A Markov system. Note that there aren't actions, but there are transitions.

7.1 Markov Decision Processes

Until now the (search) space was (almost) always deterministic. However, there are many cases where the model matching the situation is **non-deterministic** – they can be "pure" random, or they can be sporadic enough, so it'll be more accurate to describe the outcomes using their *distributions*. For that purpose, Markov was kind enough and lent us some of his power in the form of MDPs.

Model assumptions & definitions

- Actions' outcomes are **non-deterministic**, and we know their *distribution*.
- (this week) the model is **fully observable**.

Goal: Maximize the revenue.

Definitions:

1. **MDP** is a tuple where:

$$< S, A, \{P_{sa}\}, \gamma, R >$$

$S = \{set\ of\ states\}$
$A = \{set\ of\ actions\}$
$P_{sa}(s') = \mathbb{P}(\, s \to s' \mid did\ a\ in\ s\,)$
$\gamma \in [0,1] = discount\ factor\ \big(more\ interesting\ when\ \gamma \in (0,1)\big)$
$R: S \to \mathbb{R} - reward\ function$

2. A **walk** on MDP is a series of states $s_0, \dots, s_n \in S$ with **legal** actions $a_0, \dots, a_{n-1} \in A$, meaning:

$$\forall i < n \quad P_{s_i a_i}(s_{i+1}) > 0$$

Therefore, the probability of a walk happening is always *positive*.

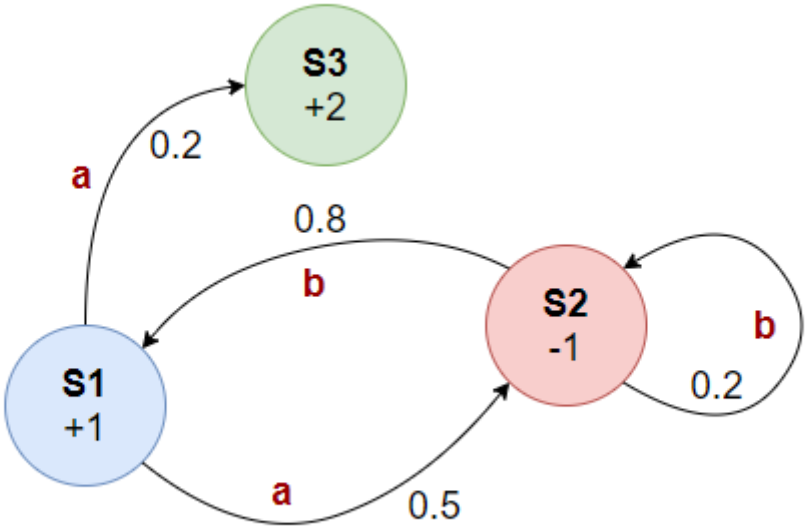


*Figure 7.2 MDP example. The states are $\{S1, S2, S3\}$ with their matching rewards and the actions are $\{a, b\}$ with their matching probabilities and outcomes. The same action can lead to **multiple** states*

Note:

1. For every $s, s' \in S, a \in A$, the probability $P_{sa}(s')$ is **independent** of the previous actions/states:
   $\mathbb{P}(s_n = y_n \mid s_{n-1} = y_{n-1} \wedge \ldots \wedge s_0 = y_0 \wedge did\ a) = \mathbb{P}(s_n = y_n \mid s_{n-1} = y_{n-1} \wedge did\ a)$
2. The discount factor γ is how fast the rewards *drop* with progress of the process (can be measured by times, actions). The drop is *exponential*; therefore, they will always **converge**.

7.2 How to AI this?

The goal is to maximize the reward. Since $\gamma < 1$, all rewards will converge (exponentially) to 0, so we **don't need** to travel on the graph for eternity… therefore, we can define a *payoff* function.

Definition: A ***payoff function*** is a function $V: W = \{set\ of\ possible\ walks\} \to \mathbb{R}$ indicating the **payoff** of a walk. V is defined by:

$$s = s_0, s_1, \ldots, s_n,\ \ V(s) = R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \cdots + \gamma^n R(s_n) = \sum_{k=0}^{n} \gamma^k R(s_k)$$

Since the actions' outcomes are probabilistic, deterministic search over the space won't work (the search we used to do until now). What can we do? Provide the agent a *policy* – instead of finding the best course of states, the goal will be obtained by finding the best course of **actions**.

Definitions:

1. **Policy** is a function $\pi: S \to A$ telling the agent what to do in each state: $\pi(s) = a$.
2. **Policy Payoff Function** is a function $V^\pi: S \to \mathbb{R}$ indicating the *payoff* of policy π. Since the space is non-deterministic, we define V^π as the **expected payoff** of the policy:

$$V^{\pi}(s) = \mathbb{E}\left(\sum_{n=0}^{\infty} \gamma^n R(s_n) \mid s_0 = s\right), \quad s_{n+1} = F\big(s_n, \pi(s_n)\big)$$

Now maximizing the payoff means maximizing $V^{\pi}(s_0)$ (where s_0 is the start state), thus the goal is to find $\arg\max_{\pi} V^{\pi}(s_0)$. First, let's simplify the formula above by finding a *recursive* definition:

$$V^{\pi}(s) = \mathbb{E}\left(\sum_{n=0}^{\infty} \gamma^n R(s_n) \mid s_0 = s\right) = \mathbb{E}\left(R(s_0) + \gamma\left(\sum_{n=1}^{\infty} \gamma^{n-1} R(s_n)\right) \mid s_0 = s\right)$$

$$= R(s) + \gamma \cdot \mathbb{E}\left(\sum_{n=1}^{\infty} \gamma^{n-1} R(s_n) \mid s_0 = s\right) = R(s) + \gamma \cdot \sum_{s' \in S} P_{s,\pi(s)}(s')\, \mathbb{E}\left(\sum_{n=1}^{\infty} \gamma^{n-1} R(s_n) \mid \begin{matrix} s_0 = s \\ s_1 = s' \end{matrix}\right)$$

The last transition is by the law of total probability. Since the sum in $\mathbb{E}$ is now independent of s_0:

$$V^{\pi}(s) = R(s) + \gamma \cdot \sum_{s' \in S} P_{s,\pi(s)}(s') \cdot V^{\pi}(s')$$

and this matches the principle that the actions from state s' are *independent* of what happened in s.

Assuming S is finite, we can represent V^{π} as a **vector** of expected payoffs from each state:

$$\begin{bmatrix} V^{\pi}(q_1) \\ \vdots \\ V^{\pi}(q_n) \end{bmatrix} = \begin{bmatrix} R(q_1) \\ \vdots \\ R(q_n) \end{bmatrix} + \gamma \cdot \begin{bmatrix} p_{q_1 \to q_1} & \cdots & p_{q_1 \to q_n} \\ \vdots & \ddots & \vdots \\ p_{q_n \to q_1} & \cdots & p_{q_n \to q_n} \end{bmatrix} \cdot \begin{bmatrix} V^{\pi}(q_1) \\ \vdots \\ V^{\pi}(q_n) \end{bmatrix}$$

$$\Rightarrow \vec{V} = \vec{r} + \gamma \cdot P \cdot \vec{V} \;\Leftrightarrow\; \vec{V}(I - \gamma \cdot P) = \vec{r} \;\Leftrightarrow\; \vec{V} = (I - \gamma \cdot P)^{-1}\vec{r}$$

Therefore, finding V^{π} requires **no lookahead** (+ inverting a matrix), and the total runtime is $\boldsymbol{O\big(|S|^3\big)}$.

Note that after finding $\arg\max_{\pi} V^{\pi}(s_0)$, we are not bounded to that policy – it's plausible that after 1 move there's a better policy. Overall the result is a new policy which is a combination of parts of different policies, where each state we choose a new, **best** (expected) policy.

Definition: $V^*: S \to \mathbb{R}$ is the **expected reward** if the agent follows the *best* policy from state s.

$$V^*(s) = \max_{\pi} V^{\pi}(s)$$

Since $V^{\pi}(s) = R(s) + \gamma \cdot \sum_{s' \in S} P_{s,\pi(s)}(s') \cdot V^{\pi}(s')$, we can (intuitively) get:

$$V^*(s) = R(s) + \max_{\pi}\left\{\gamma \cdot \sum_{s' \in S} P_{s,\pi(s)}(s') \cdot V^*(s')\right\} \Rightarrow \pi^*(s) = \arg\max_{a}\left\{\gamma \cdot \sum_{s' \in S} P_{s,a}(s') \cdot V^*(s')\right\}$$

where π^* is the **best policy** matching V^*. Note: $V^* = V^{\pi^*}$ (thanks, Bellman).

Thus, given V^* **or** π^* we can find the *other* one. The problem is we don't have **neither** V^* nor π^*. Therefore, what we'll do is guess one, and iteratively improve and converge to an optimal solution (converges because the search space is convex – thus each local maximum is a global one).

There are 2 approaches. The first one is – start from $V^0 \equiv 0$, and iteratively maximize it. Then use V to draw the matching policy, like the iterative improvement. Both use *dynamic programming*.

Value Iteration (V^*)
$t \leftarrow 0,\ \forall s \in S:\ V^0(s) \leftarrow 0$ while $\exists s \in S\ \ \lvert V^t(s) - V^{t-1}(s)\rvert \geq \varepsilon$: $t++,\ \forall s \in S:\ V^t(s) \leftarrow R(s) + \gamma \cdot \max_a\{\sum_{s' \in S} P_{s,a}(s') \cdot V^{t-1}(s')\}$ return $\pi(V) \leftarrow \arg\max_a\{\sum_{s' \in S} P_{s,a}(s') \cdot V^{t-1}(s')\}$

Figure 7.3 Value Iteration algorithm

Runtime: Single iteration - $O(|S|^2|A|)$; Number of iterations - $O\left(\frac{\max(R)}{\varepsilon} \cdot \frac{1}{1-\gamma} \cdot \frac{1}{\gamma}\right)$.

Second approach: Start from some policy π, and iteratively update the π array by its V^π as a **vector**. This time there's a **definite** stopping condition – when π hasn't changed in some iteration it can be concluded that π won't change anymore, thus the algorithm is completed.

Policy Iteration (π^*)
$t \leftarrow 0,\ \forall s \in S:\ \pi^0(s) \leftarrow arbitrary\ action$ while $\pi^t \neq \pi^{t-1}$: $t++,\ \overrightarrow{V^{t-1}} \leftarrow \left(I - \gamma \cdot P^{\pi^{t-1}}\right)^{-1} \cdot \vec{r}$ $\forall s \in S:\ \pi^t(s) \leftarrow \arg\max_a\{\sum_{s' \in S} P_{s,a}(s')\left(R(s,a) + \gamma \cdot V^{t-1}(s')\right)\}$ return $\pi(V) \leftarrow \arg\max_a\{\sum_{s' \in S} P_{s,a}(s') \cdot V^{t-1}(s')\}$

Figure 7.4 Policy Iteration algorithm

Runtime: Single iteration - $O(|S|^2|A|) + O(|S|^3)$ Note: Before updating, π^t **is** π^{t-1}?

In the next section we'll go over reinforcement learning, POMDPs, and even more algorithms that can come in handy in MDPs.

8 Reinforcement Learning

Many times, trying to model the world as an MDP will be very inefficient, taking too much space & time. Furthermore, sometimes we don't even have enough information to model the world as an MDP (we need the *state space*, set of *actions*, the actions' outcomes *distributions* and their *rewards*).

Figure 8.1 Illustration

Therefore, we should find a more general way of modelling situations in the world, where some of the information may be hidden, or even *unknown*. Sometimes the agent will know its surroundings, sometimes it'll know the rewards, but only later in the game (for example what to study, and where?). What's known and unknown can be played with to create different models.

8.1 Partially Observable Markov Decision Process

Consider the following: Given a random variable X with **unknown** distribution, how to estimate $\mathbb{E}(X)$? The natural approach to this problem can be sampling X over and over and taking the average of the results.

Definition: A_k is the ***approximated expected value*** which defined by z_t (the t^{th} **sample**):

$$A_k = \frac{1}{k}\sum_{t=1}^{k} z_t$$

Now we have another sample, z_{k+1}. How do we sum up the new sample with the previous ones?

$$A_{k+1} = \frac{1}{k+1}\sum_{t=1}^{k+1} z_t = \frac{1}{k+1}\sum_{t=1}^{k} z_t + \frac{1}{k+1} z_{k+1} = \frac{k}{k+1} A_k + \frac{1}{k+1} z_{k+1}$$

Denote $\alpha_k = \frac{1}{k}$ as the **learning rate**:

$$= \left(1 - \frac{1}{k+1}\right) A_k + \frac{1}{k+1} z_{k+1} = (1 - \alpha_{k+1}) A_k + \alpha_{k+1} \cdot z_{k+1} = A_k + \alpha_{k+1}(z_{k+1} - A_k)$$

$$\Rightarrow A_{k+1} = A_k + \alpha_{k+1}(z_{k+1} - A_k)$$

The learning rate tells us how much weight we put on latest results.

- Large α_{k+1} – then $z_{k+1} - A_k$ has more weight relative to A_k, and A_{k+1} can change easily.
- Small α_{k+1} – then $z_{k+1} - A_k$ has less weight relative to A_k, and A_{k+1} is closer to A_k.

For the learning rate to be meaningful (that is, to **learn**) we want to have $\lim_{k\to\infty} \alpha_k = 0$, meaning – the agent assumes that it gets better with progress of time, so at the beginning the object is to explore the world, then the object gradually changes to optimizing the current method.

Definition: Given policy π, an ***episode*** is a sequence of actions, with their immediate implications and rewards (bounded in some clear way).

Episodes help us estimate policies' values in a more accurate way – instead of looking 2-3 moves ahead, divide the world to chronological episodes that start & end in critical points. We'll use episodes (somewhat implicitly) and the previous definitions to present 5 POMDP-search algorithms: Guessing, Monte Carlo, Temporal Difference Learning, SARSA and Q-Learning.[4]

Definitions: Since the world isn't fully observable, we denote a few things to store our current knowledge/observations, *extending* the MDP definition:

1. $Z = \{set\ of\ observations\}$.
2. $O: S \times A \times Z \to \mathbb{R}$ is the ***conditional observation probability function***, meaning $\mathbb{P}(z|s,a)$.
3. $R: S \to \mathbb{R}$ is the ***aggregate reward function***, including γ (can be also $S \times A \times S \to \mathbb{R}$).
4. $T: S \times A \times S \to \mathbb{N}$ is the ***conditional transition counting function***. It counts how many times action a got us from state s to a **specific** s'.
5. $N: S \times A \to \mathbb{N}$ is the ***transition counting function***. It counts how many times action a was applied on state s, **regardless** of the outcome. We can say that $\sum_a N(s,a)$ counts how many times we've visited s.

The principle: At state s, accept reward r, apply action a and go to the outcome state s'. Finally update the model's estimation using the current knowledge and do it again from s'. Formally:

$$R(s) \leftarrow R(s) + r,\ \ T(s,a,s') \leftarrow T(s,a,s') + 1,\ \ N(s,a) \leftarrow N(s,a) + 1$$

$$\hat{P}_{s,a}(s') = \frac{T(s,a,s')}{N(s,a)},\ \ \hat{R}(s) = \frac{R(s)}{\sum_a N(s,a)}$$

This forms *belief states* – states with the distributions observed. With these belief states a fully observable MDP can be formed, and an optimal policy π^* can now be found – since the POMDP is Markovian over the belief state space. However, this is a continuous state space MDP, and not a discrete one. This makes the task of finding an optimal policy harder and more complex.
Let's see some algorithms.

8.2 Monte Carlo

This algorithm is a *lighter* version of the policy iteration from before, and a smaller, more specific version of Monte Carlo Tree Search. Given a policy π, MC **evaluates** the policy by simulating many, *many* episodes that follow the policy. Recall $V^\pi(s) = R(s) + \gamma \cdot \sum_{s' \in S} P_{s,\pi(s)}(s') \cdot V^\pi(s')$. But $P_{s,\pi(s)}(s')$ is unknown, so we can use the approximation of V^π. Replace $V^\pi \leftarrow A_k$, denote as V_k^π and we get an **approximation** for the expected reward by following π, based on k episodes:

$$V_{k+1}^\pi(s) = V_k^\pi(s) + \alpha_{k+1}\big(R(s) - V_k^\pi(s)\big)$$

Note: $R(s)$ is the approximation of the reward from s, with γ included.

So, Monte Carlo gets a policy π and a discount factor γ, and evaluates π under the restriction of partially observable space.

[4] Guessing won't be covered, as you can guess what the algorithm would look like.

Monte Carlo (π, γ)

$V \leftarrow \vec{0},\ \alpha \leftarrow learning\ rate$

for each episode do:

$s \leftarrow random\ state,\ \tau \leftarrow the\ walk,\ R \leftarrow \vec{0}$

while s is not terminal state do:

$\tau \leftarrow \tau \cup \{s\},\ a \leftarrow \pi(s),\ r \leftarrow reward$

for all $\tilde{s} \in \tau$ do:

$R(\tilde{s}) \leftarrow R(\tilde{s}) + \gamma^k r$ (k is the distance $s \rightarrow \tilde{s}$)

$s \leftarrow \pi(s)$

for all $s \in \tau$ do:

$V(s) \leftarrow V(s) + \alpha\big(R(s) - V(s)\big)$

decay α

return V

Figure 8.2 Monte Carlo algorithm

8.3 Temporal Difference Learning

If you noticed, Monte Carlo waits until the end of the episode to update V^π. Temporal Difference Learning updates V^π *online* – after each move. This is called in short – TD(0), since it looks 1 step ahead. Overall, we get the equation:

$$V_{k+1}^\pi(s) = V_k^\pi(s) + \alpha_{k+1}\left(\underbrace{r + \gamma \cdot V_k^\pi\big(\pi(s)\big)}_{reward\ from\ s + \pi(s)} - V_k^\pi(s)\right)$$

This can be generalized to TD(n) – n steps ahead:

$$V_{k+1}^\pi(s) = V_k^\pi(s) + \alpha_{k+1}\left(r + \sum_{i=1}^{n+1} \gamma^i \cdot V_k^\pi\left(\pi^i(s)\right) - V_k^\pi(s)\right)$$

Temporal Difference Learning (π, γ)

$V \leftarrow \vec{0}, \ \alpha \leftarrow learning\ rate$

for each episode do:

$s \leftarrow random\ state$

while s is not terminal state do:

$a \leftarrow \pi(s), \ r \leftarrow reward$

$V(s) \leftarrow V(s) + \alpha\left(r + \gamma \cdot V\big(\pi(s)\big) - V(s)\right)$ (k is the distance $s \to \tilde{s}$)

$s \leftarrow \pi(s)$

decay α

return V

Figure 8.3 Temporal Difference Learning algorithm

Here's an example which demonstrates how Monte Carlo and TDL are preformed.

- States - $\{0, 1, \ldots, N, END\}$
- Actions (for state s) - $\{0, 1, \ldots, \min\{s, N-s\}\}$, we can apply action a in state s if $a \leq s$ and $s + a \leq N$
- The probability for winning - p
- Policy $\forall s: \ \pi(s) = 1$
- from states 0 and N we move to the terminal state END
- $R(N) = 1$ and for every other state $R(s) = 0$
- $\gamma = 1$, $\alpha_k = \frac{1}{k}$

Figure 8.4 Gambler's Ruin. The agent starts with several coins. Each round, the agent bets k coins. With probability p it earns one coin and with probability $1 - p$ it loses one coin. The game ends when the agent has 0 or N coins

Episode	
4	$V(4) = 0 + 1 \cdot (1 - 0) = 1$
$3 \to 4$	$V(3) = 0 + \frac{1}{2} \cdot (1 - 0) = \frac{1}{2}$ $V(4) = 1 + \frac{1}{2} \cdot (1 - 1) = 1$
$2 \to 3 \to 4$	$V(2) = 0 + \frac{1}{3} \cdot (1 - 0) = \frac{1}{3}$ $V(3) = \frac{1}{2} + \frac{1}{3} \cdot \left(1 - \frac{1}{2}\right) = \frac{2}{3}$ $V(4) = 1 + \frac{1}{3} \cdot (1 - 1) = 1$
$3 \to 2 \to 1 \to 0$	$V(3) = \frac{2}{3} + \frac{1}{4} \cdot \left(0 - \frac{2}{3}\right) = \frac{1}{2}$ $V(2) = \frac{1}{3} + \frac{1}{4} \cdot \left(0 - \frac{1}{3}\right) = \frac{1}{4}$ $V(1) = 0 + \frac{1}{4} \cdot (0 - 0) = 0$ $V(0) = 0 + \frac{1}{4} \cdot (0 - 0) = 0$

Episode	
4	$V(4) = 0 + 1 \cdot (1 + 0 - 0) = 1$
$3 \to 4$	$V(3) = 0 + \frac{1}{2} \cdot (0 + 1 - 0) = \frac{1}{2}$ $V(4) = 1 + \frac{1}{2} \cdot (1 + 0 - 1) = 1$
$2 \to 3 \to 4$	$V(2) = 0 + \frac{1}{3} \cdot \left(0 + \frac{1}{2} - 0\right) = \frac{1}{6}$ $V(3) = \frac{1}{2} + \frac{1}{3} \cdot \left(0 + 1 - \frac{1}{2}\right) = \frac{2}{3}$ $V(4) = 1 + \frac{1}{3} \cdot (1 + 0 - 1) = 1$
$3 \to 2 \to 1 \to 0$	$V(3) = \frac{2}{3} + \frac{1}{4} \cdot \left(0 + \frac{1}{6} - \frac{2}{3}\right) = \frac{13}{24}$ $V(2) = \frac{1}{6} + \frac{1}{4} \cdot \left(0 + 0 - \frac{1}{6}\right) = \frac{1}{8}$ $V(1) = 0 + \frac{1}{4} \cdot (0 + 0 - 0) = 0$ $V(0) = 0 + \frac{1}{4} \cdot (0 + 0 - 0) = 0$

Figure 8.5 Monte Carlo (left), Temporal Difference Learning (right)

Maybe you noticed that these 2 algorithms are basically functions $\pi \mapsto V^{\pi}$. But we can't plug MC/TDL into the Figure 7.4 Policy Iteration algorithm, since that algorithm uses $P_{s,a}$ which is *unknown* here. In order to overcome this issue, we'll introduce an *upgraded* policy iteration.

8.4 Exploration vs. Exploitation

How should we split time between exploration and exploitation?

- Exploration – doing non optimal actions to find out how the world works.
- Exploitation – use everything we learned on the world to get maximum revenue.

On the one hand, without exploring the agent will lock on a non-optimal policy, but on the other hand without exploiting the agent will miss an optimal policy. What we can do is preferring exploration at the beginning, and gradually change the preference to exploitation. By "preferring", the meaning is randomness – at the beginning high randomness, and gradually decrease it so the agent will choose its optimal policy more often.

ε-greedy Exploration: Each iteration the agent explores with probability ε and exploits with probability $1-\varepsilon$. As in Simulated Annealing, ε decreases over time, in order to converge.

Q-function: Given $V^{\pi}(s) = R(s) + \gamma \cdot \sum_{s' \in S} P_{s,\pi(s)}(s') \cdot V^{\pi}(s')$ we can extract what's called the **Q-function** – what's the expected reward if we follow policy π, given that we start with doing action a in state s?

Definition: The ***Q-function*** of a policy π is a function $Q^{\pi}: S \times A \to \mathbb{R}$ defined (recursively) by:

$$Q^{\pi}(s,a) = R(s,a) + \gamma \cdot \sum_{s' \in S} P_{s,a}(s') \cdot Q^{\pi}\big(s', \pi(s')\big)$$

Note:

1. The Q-function overcomes the exploration vs. exploitation issue by allowing a "wandering interval" – the first action can be *any* action, but the next ones are by the policy.
2. The second replacement of V^{π} with Q^{π} is just **renaming** the function in the sum, since $Q^{\pi}\big(s', \pi(s')\big)$ is equivalent to following π from s' – that is $V^{\pi}(s')$.
3. If $P_{s,a}$ is unknown, a similar Q-function can be defined with the *approximation* $\hat{P}_{s,a}$.

Recall that given V^* we can derive π^* and vice versa. We can also do a similar thing with the Q-function. Given π^* finding Q^* is trivial, and given Q^*, finding the best policy π^* can be done by:

$$\forall s \in S: \quad \pi^*(s) = \arg\max_{a}\{Q^*(s,a)\}$$

And now we can value iteration this thing.

8.5 SARSA

A generalized value iteration process. The Q-function-SARSA relation is very analogous to the MC-TDL relation – each time act, observe and update the Q-function from 1 time earlier.

$$Q(s_t,a_t) = Q(s_t,a_t) + \alpha\big(r_t + \gamma \cdot Q(s_{t+1},a_{t+1}) - Q(s_t,a_t)\big)$$

SARSA (γ, ε)

$Q \leftarrow 0_{|S|\times|A|}$, $\alpha \leftarrow learning\ rate$

for each episode do:

$s \leftarrow init\ state$, $a \leftarrow init\ action$

while s is not terminal state do:

$s' \leftarrow F(s,a)$, $r \leftarrow reward$

$$a' \leftarrow \begin{cases} random\ action & with\ prob.\ \varepsilon \\ \pi(s') & with\ prob\ 1-\varepsilon \end{cases}$$

$$Q(s,a) \leftarrow Q(s,a) + \alpha\big(r + \gamma \cdot Q(s',a') - Q(s,a)\big)$$

$$\pi(s) \leftarrow \arg\max_{\tilde{a}} Q(s,\tilde{a})$$

$s \leftarrow s'$, $a \leftarrow a'$

decay α

return Q

Figure 8.6 SARSA algorithm

8.6 Q-learning

What if at time $t+1$ there's a bad action? The naive equation above will conclude that the state $\boldsymbol{s_t}$ is bad, but that action *isn't* mandatory. If I hold money in my hand and I can burn it, instead of concluding that holding money is bad, I can just not burn money. Therefore, we'll use a *better* equation – **Q-learning**:

$$Q(s_t,a_t) = Q(s_t,a_t) + \alpha\left(r_t + \gamma \cdot \max_{a_{t+1}}\{Q(s_{t+1},a_{t+1})\} - Q(s_t,a_t)\right)$$

Q-learning (γ, ε)

$Q \leftarrow 0_{|S|\times|A|}$, $\alpha \leftarrow learning\ rate$

for each episode do:

$s \leftarrow init\ state$

while s is not terminal state do:

$$a \leftarrow \begin{cases} random\ action & with\ prob.\ \varepsilon \\ \arg\max_{a} Q(s,a) & with\ prob\ 1-\varepsilon \end{cases}$$

$s' \leftarrow F(s,a)$, $r \leftarrow reward$

$$Q(s,a) \leftarrow Q(s,a) + \alpha\left(r + \gamma \cdot \max_{a'} Q(s',a') - Q(s,a)\right)$$

$s \leftarrow s'$

decay α

return Q

Figure 8.7 Q-learning algorithm

9 Supervised Learning

9.1 Curve Fitting

We want to learn a function f from examples – pairs $(x, f(x))$. Usually the function is very complicated, so we instead try to find a **hypothesis** h such that $h \approx f$, given a set of examples. We say that h is *consistent* if it agrees with f on all examples. Many learning problems come down to curve fitting – approximating a function, assuming it exists. The naive solution would be "bending" h to agree with f on all examples. The issue with that approach is what's called *overfitting*: *"the production of an analysis that corresponds too closely or exactly to a particular set of data, and may therefore fail to fit additional data or predict future observations reliably."*. [5]

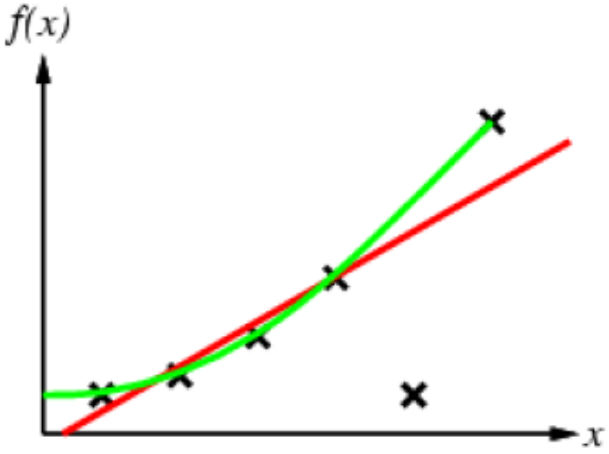


Figure 9.1 Simple approximating

Meaning, if there will be more examples, a better approach would be approximating f as simple a possible.

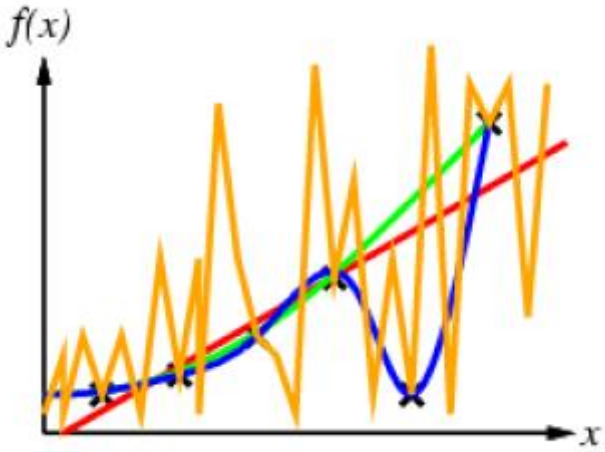


Figure 9.2 Some better hypotheses, and too good ones

For example, in the figure above we would prefer the blue hypothesis over the orange one, and the green one over the blue one. Why? Maybe the examples' values are stochastic, or some examples in the training set are inaccurate (*noises*). Therefore if an example doesn't align "nicely" with the other examples we should consider ignoring it. However, we don't want something too simple like the red hypthesis – this is called *underfitting*.

We will *learn* now a more specific way to approximate f, which is decision trees.

[5] https://en.wikipedia.org/wiki/Overfitting

9.2 Decision Trees

The goal is to classify an event based on its attributes, meaning we want to approximate a function f that gets a vector of attributes and returns its classification. The motivating and the main example here will be this: We want to predict the winner in the upcoming finals match between Hapoel Jerusalem and Maccabi Tel-Aviv and get rich. This example will make this section clearer, by demonstrating the main techniques on it.

Definition: A ***decision tree*** is a tree where:

1. Each interior node is labelled with a feature.
2. Each arc going out of interior nodes is labelled with a value for the node's feature.
3. Each leaf is labelled with a category.

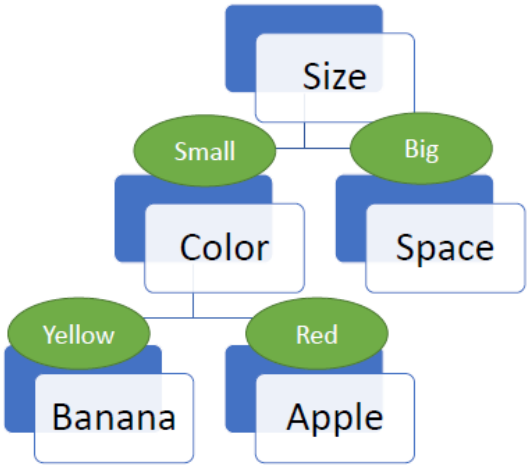


Figure 9.3 A decision tree example

Back to the main example. We denote Lior Eliyahu and Yotam Halperin as 2 players for Hapoel Jerusalem. The game's attributes are:

1. Home/away?
2. Start time?
3. Does Lior Eliyahu (E) start?
4. Does Yotam Halperin (H) play Point Guard/Shooting Guard?
5. Is the opponent's center tall/short (where tall ≥ 2.08)?
6. Does Yotam Halperin guard the opponent's center or one of the forwards (small/power)?
7. **Winner?** ← This is what we want to know.

To learn and approximate f, we need examples. The example set here is Hapoel's previous games, with their results. Below them is the game (vector) we want to predict.

Where	When	Eliyahu Starts	Helperin Offence	Halperin Defence	Opponent Center	Outcome
Home	7pm	Yes	Point	Forward	Tall	Won
Home	7pm	Yes	Shooting	Center	Short	Won
Away	7pm	Yes	Shooting	Forward	Tall	Won
Home	5pm	No	Shooting	Center	Tall	Lost
Away	9pm	Yes	Shooting	Forward	Short	Lost
Away	7pm	No	Point	Forward	Tall	Won
Home	7pm	No	Shooting	Center	Tall	Lost
Home	7pm	Yes	Point	Center	Tall	Won
Away	7pm	Yes	Point	Center	Short	Won
Home	9pm	No	Shooting	Center	Short	Lost
Away	7pm	No	Shooting	Forward	Short	Lost
Away	5pm	No	Point	Forward	Tall	Won
Home	7pm	No	Point	Center	Tall	Lost
Home	9pm	No	Shooting	Forward	Short	Lost
Away	9pm	Yes	Point	Forward	Short	Lost
Home	7pm	Yes	Point	Center	Short	Won
Home	7pm	Yes	Point	Forward	Short	Won
Home	5pm	No	Shooting	Center	Short	Lost
Home	7pm	Yes	Point	Forward	Tall	Won
Away	5pm	No	Point	Center	Tall	Lost
Where	**When**	**Eliyahu Starts**	**Helperin Offence**	**Halperin Defence**	**Opponent Center**	**Outcome**
Away	9pm	No	Point	Forward	Short	???

Figure 9.4 The previous results, and the game that we want to predict

Note: This game's setups **don't match** any previous game. One way to look at this is to split by attributes. Once we have split the games by attributes, we have a classifier which gets a game description and goes down the tree according to its attributes, until reaching a terminal node – win/lose. The one thing that's left is learning such a good tree.

9.3 Learning Decision Trees

A good decision tree will be one that's as consistent as possible, while keeping it simple (as in the last section). Brute force won't help, as there are 2^{2^n} distinct decision trees with n boolean attributes. In a more expressive hypothesis space, the chances to express the target function increases (as there are more consistent hypotheses in relation to all hypotheses), but they may get *worse* predictions.

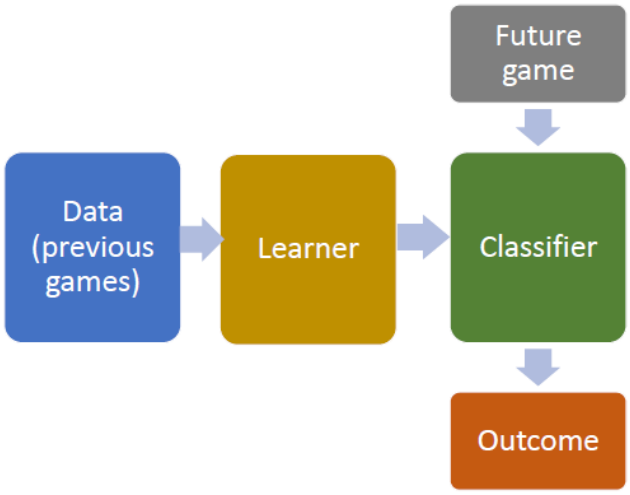


Figure 9.5 The learning and predicting structure

A simple algorithm for generating a decision tree is a recursive one – ID3:

1. If all the events in this branch have the same classification, create a leaf with that prediction.
2. Otherwise: Pick an attribute and create a node;
   For each value of that attribute, use ID3 to build a subtree for the events that match the value.

However, different attributes hold different information: The overall win/lose ratio is 0.5 and so is the away win/lose ratio; the 9pm win/lose ratio is 0, but on the other hand, the point win/lose ratio is 8/11. How will we choose a good attribute to split by? A good attribute is one that increases the tree's accuracy, while keeping its simplicity (try splitting the world's population by age vs. name).

9.4 Information Theory

Definition: Given a discrete random variable X with distribution $p: \{x_0, \dots, x_{n-1}\} \to [0,1]$, the ***entropy*** of X is defined by:

$$H(X) = -\sum_{x \in dom(p)} p(x) \log_b \left(p(x)\right)$$

Note:

1. If $p(x) = 0$ for some x then we'll look at $p \setminus \{(x, 0)\}$.
2. $H \geq 0$.
3. When $b = 2$ the entropy is measured in bits. In this section, $b = 2$.

What this definition does is quantifying the amount of uncertainty X holds, which is identical to *how much information we need in order to know X's outcome*.

For example, let $X \sim Ber(p)$. Then $H(X) = -p \log p - (1-p)\log(1-p)$.
If $p = \frac{1}{2}$, then $H(X) = -\frac{1}{2}\log\frac{1}{2} - \frac{1}{2}\log\frac{1}{2} = -\log\frac{1}{2} = 1$, meaning with 1 bit we can know $Ber\left(\frac{1}{2}\right)$'s outcome.
If $p = 1$, then $H(X) = -\log 1 - 0 = 0$, meaning we know X's outcome – since it holds that $X = 1$ with probability 1.

If $p = \frac{1}{4}$, then $H(X) = -\frac{1}{4}\log\frac{1}{4} - \frac{3}{4}\log\frac{3}{4} \approx 0.81$, meaning X isn't totally unexpected, but we still need 0.81 bits to know its outcome.

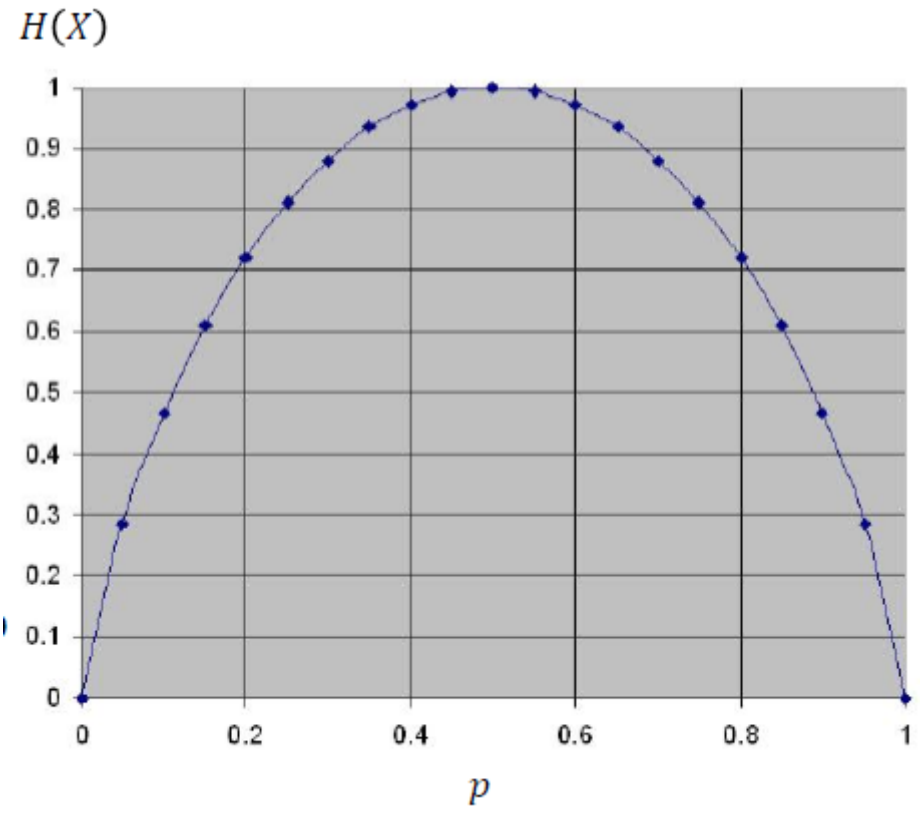


Figure 9.6 The entropy of $Ber(p)$

Definition: Let X, Y be 2 discrete random variables with joint distribution $p(x, y)$. The **joint entropy** of X, Y is:

$$H(X,Y) = -\sum_{x,y} p(x,y)\log\big(p(x,y)\big)$$

Definition: Let X, Y be 2 discrete random variables. The **conditional entropy** of X given Y is:

$$H(X|Y) = \sum_{y} p(y)H(X|Y=y) = -\sum_{y} p(y)\sum_{x} p(x|y)\log\big(p(x|y)\big) = -\sum_{x,y} p(x,y)\log\big(p(x|y)\big)$$

The intuition for those 2 definitions is:

1. How much information de we need in order to know X **and** Y?
2. How much information do we need in order to know X **given** Y (meaning given that we know Y's outcome)?

By how H is defined, we get a *chain rule*: $H(X,Y) = H(Y) + H(X|Y)$. Also, not always $H(X|Y) = H(Y|X)$, as much as one variable may reveal more information about the second than vice versa.

Example: Let X, Y be random variables as in the following table:

Y/X	1	2
1	¼	¼
2	0	½

Figure 9.7

The marginal distribution of X is $\left(\frac{1}{4},\frac{3}{4}\right)$, of Y is $\left(\frac{1}{2},\frac{1}{2}\right)$. Therefore $H(X) \approx 0.81$ and $H(Y) = 1$.
Also, $H(X|Y) = \sum_{i=1}^{2} p(Y=i)H(X|Y=i) = \frac{1}{2}H\left(\frac{1}{2},\frac{1}{2}\right) + \frac{1}{2}H(0,1) = \frac{1}{2}$.
Similarly, $H(Y|X) = \sum_{i=1}^{2} p(X=i)H(Y|X=i) = \frac{1}{4}H(1,0) + \frac{3}{4}H\left(\frac{1}{3},\frac{2}{3}\right) \approx 0.69 \left(\neq \frac{1}{2}\right)$.
And finally, $H(X,Y) = -\left(\frac{1}{4}\log\frac{1}{4} + \frac{1}{4}\log\frac{1}{4} + 0 + \frac{1}{2}\log\frac{1}{2}\right) = \frac{1}{2}\cdot 2 + \frac{1}{2}\cdot 1 = \frac{3}{2}$ (all in bits).

Definition: Let X, Y be 2 discrete random variables with joint distribution $p(x,y)$. The ***mutual information*** of X, Y is:

$$I(X;Y) = \sum_{x,y} p(x,y)\log\frac{p(x,y)}{p(x)p(y)} = I(Y;X)$$

Claim: $I(X;Y) = H(X) - H(X|Y)$.
Proof:

$$\begin{aligned} I(X;Y) &= \sum_{x,y} p(x,y)\log\frac{p(x,y)}{p(x)p(y)} = \sum_{x,y} p(x,y)\log\frac{p(x|y)}{p(x)} \\ &= -\sum_{x}\sum_{y} p(x,y)\log\big(p(x)\big) + \sum_{x,y} p(x,y)\log\big(p(x|y)\big) \\ &= -\sum_{x} p(x)\log\big(p(x)\big) - \left(-\sum_{x,y} p(x,y)\log\big(p(x|y)\big)\right) = H(X) - H(X|Y) \end{aligned}$$

Note that this claim says – $I(X;Y)$ is the reduction on the entropy of X achieved by learning Y.

Denote:

1. $Attr$ – set of attributes (features).
2. Ex – set of training examples.
3. $val(x,a)$ the value of attribute a in example x.
4. $Goal \in Attr$ the attribute we want to predict.
5. $Ex_{a,v} = \{x \in Ex | val(x,a) = v\}$ for $a \in Attr, v \in val(a)$.

Definition: The ***information gain*** for an attribute a is defined by:

$$IG_{Ex}(Goal,a) = H_{Ex}(Goal) - H_{Ex}(Goal|a) = H_{Ex}(Goal) - \sum_{v \in val(a)} \frac{|Ex_{a,v}|}{|Ex|} H_{Ex_{a,v}}(Goal)$$

Where $H_s(a)$ is the entropy of attribute a in the subset s.

Now that we are equipped with these tools, we can go back to the main example. Recall that we want to find the best attribute to split the data by (and create the decision tree). The overall win/lose ratio is 0.5, therefore $H(Goal) = H_{Ex}(Goal) = H\left(\frac{10}{20},\frac{10}{20}\right) = 1$. Let's calculate $IG_{Ex}(Goal;When)$.

4. 4 records with $When = 5pm$ with win ratio of 0.25: $H\left(\frac{1}{4},\frac{3}{4}\right) \approx 0.81$.
5. 12 records with $When = 7pm$ with win ratio of 3: $H\left(\frac{3}{4},\frac{1}{4}\right) \approx 0.81$.
6. 4 records with $When = 9pm$ with win ratio of 0: $H(0,1) = 1$.

Therefore, $H(Goal|When) \approx H_{Ex}(Goal|When) = \frac{4}{20}\cdot 0.81 + \frac{12}{20}\cdot 0.81 + \frac{4}{20}\cdot 1 = 0.65$; So the information gain of splitting by $When$ is $H(Goal) - H(Goal|When) \approx 1 - 0.65 = 0.35$.

In this example, $IG(When)$ is also the maximum out of all attributes, therefore we should choose $When$ as the top-level split. After calculating this in every non-terminal node the resulting tree is:

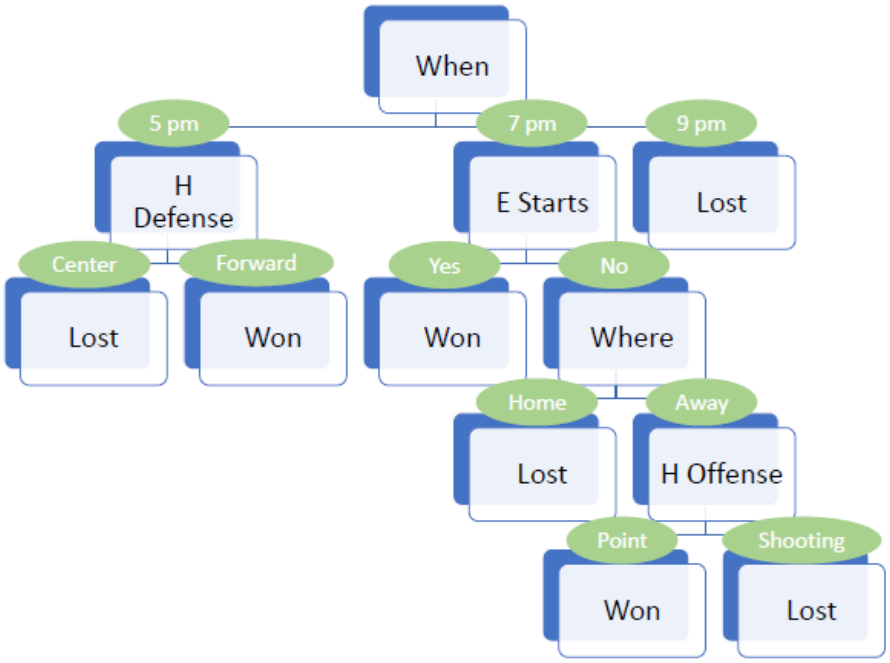


Figure 9.8 The resulting tree

And its prediction is that Hapoel will lose.

Problem: Information gain prefers attributes with many large domains (e.g. [0,10000]), and in other words – overfitting. Why? Because an attribute with k possible values can hold $\log k$ bits of information, so for 1024-ary attribute a and 2-ary attribute b it probably holds that $IG(a) > IG(b)$. Intuitively, splitting to many values is like "many splits in one layer" – we get a lot of "classifying power". But these splits aren't necessarily good, and don't guarantee a small tree.

One way to avoid this is to use **gain ratio**, which is normalizing IG.

Definition: Denote $IV_{Ex}(a) = -\sum_{v \in val(a)} \frac{|Ex_{a,v}|}{|Ex|} \log \frac{|Ex_{a,v}|}{|Ex|}$ as the ***intrinsic value***. The ***gain ratio*** is:

$$IGR_{Ex}(Goal, a) = \frac{IG_{Ex}(Goal, a)}{IV_{Ex}(a)}$$

Intuitively: $IV_{Ex}(a)$ is the amount of information generated by splitting the data on a, $IG_{Ex}(Goal, a)$ is the amount of information that's **relevant** to $Goal$, so $IGR_{Ex}(Goal, a)$ is the proportion of $IV_{Ex}(a)$ that's relevant to $Goal$.

This solves the problem of preferring attributes with large domains. However, this tree learner will still try to generate a perfect tree for the examples, but as we've seen earlier it can cause problems with future examples. And as we've seen, the approach to this will be generalizing the tree.

9.5 Evaluating Decision Trees

To evaluate a decision tree, we can do that with what we have – the previous data. It makes sense to evaluate the tree based on its predictions for unseen examples. So, we can divide the example data to 2 – *training data* and *hold out data*. Instead of training on everything, train on the first part and test on the second.

Let's take this one step further – train **several** trees, each time on a different subset of the examples. Then test each tree on the examples it hasn't seen. Choose the most successful tree and then train it on its test data. This is called **Cross Validation**.

There are many options for how to divide the data and training the trees:

1. **Simple partition**: 90-10, 70-30, 20-80 (training-testing, in %).
2. **K-fold cross-validation**: Partition the data to k (disjoint) subsets of equal size and run the learner k times (each time leaving out one subset).
3. **Leave-one-out**: Same except k=1. The accuracy of each learner is the average of the accuracy of the k learned trees.

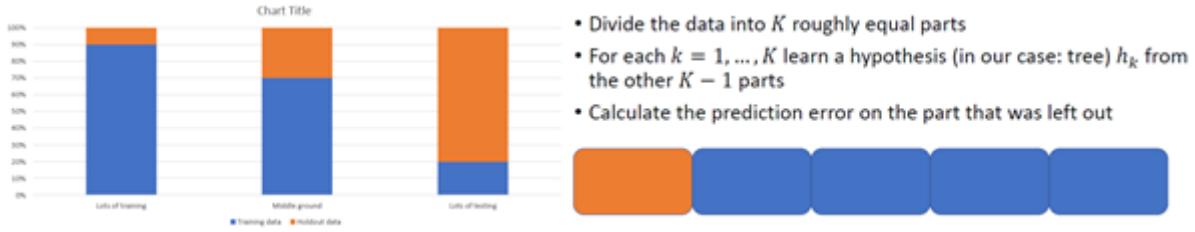


Figure 9.9: simple partition (left), k-fold cross-validation (right)

With those techniques we have another way of generalizing/simplifying our decision tree – *pruning*. In the main example, if the resulting tree has 7000 nodes then we should probably make it smaller. To do that we can do the following algorithm:

1. Generate the decision tree using ID3.
2. Test performance on the hold-out set.
3. For each leaf: **If** the performance is better without its parent, **then** prune it.
4. Repeat the latter until there's nothing left to prune (under the condition above).

Note:

1. Pruning a node which is a parent of a leaf means removing its leaves and replacing the node with the label that matches most of the examples that it reaches.
2. Since the tree is finite – this algorithm will converge.
3. The method of generating a tree using ID3 and pruning it is an example of *local search*.

And to wrap this thing up, we can use several trees – **Random Forest**.

For $b = 1, \dots, B$:

5. Draw N samples from the training data.
6. Select m attributes at random (iid).
7. Learn the tree T_b.

Return $T_1, \dots, T_B$.

And given a new instance x, predict its outcome using majority vote over $T_1(x), \dots, T_B(x)$.

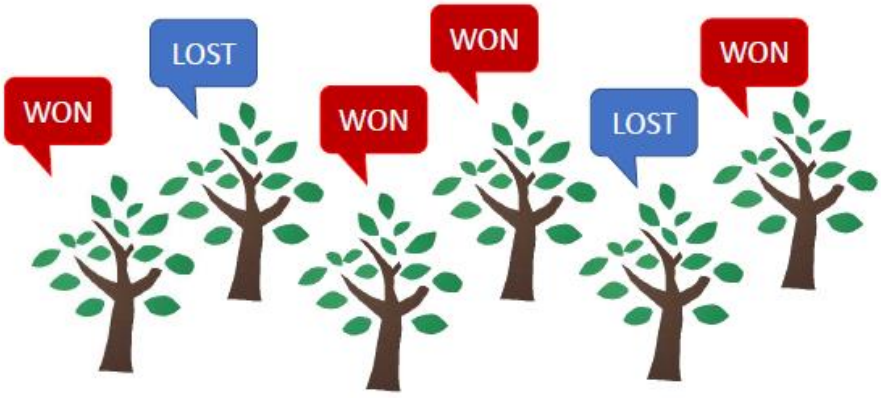


Figure 9.10 Random forest

To summarize, here is a comparison of trees vs. random forests:

- Prediction of trees tend to have a high variance
- Easy to tune parameters
- Rather fast
- Trees yield insight into decision rules

- RF has smaller prediction variance and therefore usually a better general performance
- Easy to tune parameters
- Rather slow
- "Black Box": Rather hard to get insights into decision rules

Figure 9.11 Trees vs. random forests

10 Game Theory

Definition: A ***game*** is a tuple $< n, \{S_i\}_{i=1}^n, \{u_i\}_{i=1}^n >$ where

- n – the **players** in the game $(1, \dots, n)$.
- S_i – set of possible **strategies** for player i.
- u_i – the **utility function** of player i, which is a function $S_1 \times \dots \times S_n \to \mathbb{R}$.

Note:

1. We assume that the players are **rational**, meaning each player wants to maximize its utility.
2. We assume each player knows every other player's possible strategies and their utilities.

Games can be often represented as a *game tree*, and this form is called **extensive form**.

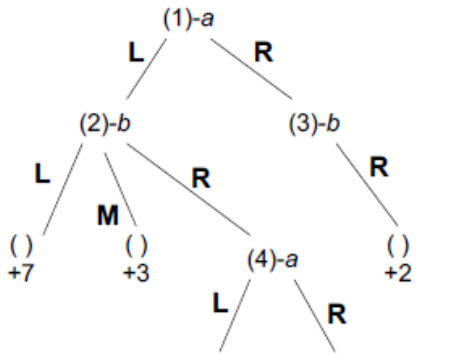
Definition: A ***pure strategy*** for a player is a mapping from all the possible states that player could see to the move the player would make.

In the example to the right, there are 4 pure strategies for A:

1. Strategy I: (1→ L, 4→ L)
2. Strategy II: (1→ L, 4→ R)
3. Strategy III: (1→ R, 4→ L)
4. Strategy IV: (1→ R, 4→ R)

and 3 pure strategies for B:

1. Strategy I: (2→ L, 3→ R)
2. Strategy II: (2→ M, 3→ R)
3. Strategy III: (2→ R, 3→ R)



	B-I	B-II	B-III
A-I	7	3	-1
A-II	7	3	4
A-III	2	2	2
A-IV	2	2	2

Figure 10.1 Extensive Form (up) and Normal Form (down) examples

Games can be also represented as a *matrix*, and this form is called **normal form**, as the matrix to the right. Note that in this example, the 2 representations are equivalent. In addition, this example is a **zero-sum game** (recall week 3). A fundamental result in this field is the *Minimax Theorem*, proved by Von-Neumann (1928): In a 2-player, zero-sum game of perfect information, **Maximin** = **Minimax**. And there always exists an **optimal** pure strategy for each player. This however doesn't hold for non-zero-sum games, and doesn't hold for games with hidden information either.

Another example: The Little Mermaid. $n = 2$ where the players can be denoted also as $\{Mermaid, Prince\}$. Their strategies are $S_M = \{A, W\}, S_P = \{M, S, N\}$. So, the normal form of this game is:

	M	S	N
A	80 101	75 -150	-5 -115
W	60 100	100 -170	10 -120

$S_M = \{Ask\ him\ out, Wait\ for\ him\ to\ ask\ first\}$

$S_P = \{marry\ the\ Mermaid, marry\ the\ Sea\ witch, Nothing\}$

Figure 10.2 Normal Form example

For example, $u_M(A, M) = 101$ and $u_P(W, S) = 100$.

10.1 Nash Equilibriums and Pareto Optimality

The utilities functions (mostly) are **not** independent. If the mermaid chooses A then the prince can only get -5, 75, or 80. Sometimes these games have some sort of local maximum – a strategies combination where there's no change of 1 strategy that results in all players (weakly) improving their utility.

Definition: $S^* = (s_1^*, \dots, s_n^*)$ is a ***pure Nash equilibrium*** if

$$\forall i, s_i' \quad u_i(s_i', s_{-i}^*) \leq u_i(s_i^*, s_{-i}^*)$$

Where s_{-i} is some combination of strategies for all players except i, s_{-i}^* is a short for $(s_1^*, \dots, s_{i-1}^*, s_{i+1}^*, \dots, s_n^*)$, and s_i' is any pure strategy of player i.

In the previous example, (A, N) is not a pure NE because – assuming mermaid chooses A, prince can change his choice to S and improve his utility from -5 to 75. (A, M) *is* a pure NE – assuming mermaid chooses A, the prince's best strategy is to choose M and vice versa.

However, a pure NE doesn't always exist: In the example to the right, for each tuple of strategies (i, j) there's a player who can improve its utility by changing its strategy choice.

The motivation behind NE's is to predict (up to certain confidence) how the players in a game will act.

If we allow **randomness** things get different and more interesting. Instead of choosing a strategy, each player can choose a distribution over its set of strategies.

	1	2
1	-1 1	1 -1
2	1 -1	-1 1

Figure 10.3 A game without a pure NE

Definitions:

1. A ***mixed strategy*** is denoted by:

$$\Delta(s_i) = (p_1, \dots, p_m),\ \ p_j \in [0,1],\ \ \sum_{j=1}^{m} p_j = 1$$

2. $\sigma^* = (\sigma_1^*, \dots, \sigma_n^*)$ is a ***mixed Nash equilibrium*** if
$$\forall i, \sigma_i' \quad u_i(\sigma_i', \sigma_{-i}^*) \le u_i(\sigma_i^*, \sigma_{-i}^*)$$
Where σ_i is a mixed strategy of player i.

Another very important result in this area has been proved by John Nash (1950) – If n is finite and $\forall i \le n\ \ s_i$ is finite, then there **exists** at least one **NE** (possibly involving mixed strategies).

So in the previous example (figure 7.3), $([0.3,0.7],[0.5,0.5])$ is **not** a mixed NE because
$u_2([0.3,0.7],[0.5,0.5])$
$= 0.15 \cdot 1 + 0.15 \cdot -1 + 0.35 \cdot 1$
$+ 0.35 \cdot -1 = 0 < 0.4$
$= u_2([0.3,0.7],[1,0])$

But Nash promises that there **is** a NE here. Assume those 2 strategies are $\sigma_1 = [x, 1-x]$ and $\sigma_2 = [y, 1-y]$. Then $u_1(\sigma_1, \sigma_2) \ge u_1(1, \sigma_2)$ and $u_1(\sigma_1, \sigma_2) \ge u_1(2, \sigma_2)$ (where the first player is the rows player, and 1,2 are pure strategies). It also holds that for **some** strategies σ_1, σ_2 that
$u_1(\sigma_1, \sigma_2) > u_1(1, \sigma_2) \Leftrightarrow u_1(\sigma_1, \sigma_2) < u_1(2, \sigma_2)$

Figure 10.4 Mixed strategies example

Thus $u_1(\sigma_1, \sigma_2) = u_1(1, \sigma_2) = u_1(2, \sigma_2)$ and symmetrically $u_2(\sigma_1, \sigma_2) = u_2(\sigma_1, 1) = u_1(\sigma_1, 2)$ for **optimal** σ_1, σ_2, meaning the optimal strategy for player i is to make each strategy of player j as good as the other strategies of j. So, in this example, a mixed NE is $([0.5,0.5],[0.5,0.5])$. Here's another, more detailed example that demonstrates how to find a mixed NE:

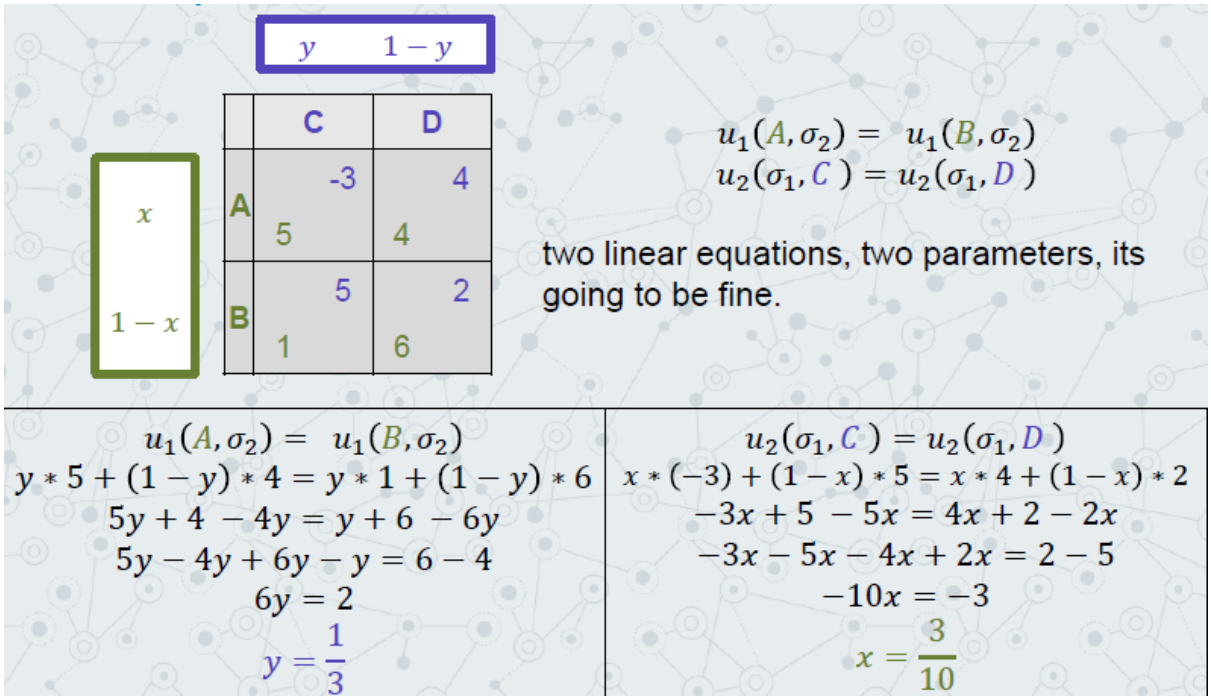


Figure 10.5 Extracting a mixed NE

Note: Nash's theorem guarantees there *exists* a Nash Equilibrium. There may be more than one, and there may be pure and mixed NE's (since every pure NE is also a mixed NE).

Another way to find a NE is to use the fact that the domains are continuous: In the last figure, to find y (which defines player **2**'s strategy) we can sketch the graphs of $u_1(A, \sigma_2)$ and $u_1(B, \sigma_2)$:

Remember that σ_2 is optimal for player 2 if player 1 won't prefer A over B and vice versa. As in the graph to the right, it happens when $\boldsymbol{y = \frac{1}{3}}$, and it can be seen why it matches the previous calculations.

The intuition for looking for *that* y is – if player 1 plays optimal then both A **and** B must be the best responses to **my** strategy – otherwise (if $u_1(A, \sigma_2) > u_1(B, \sigma_2)$ for example) player 1 could adjust his strategy and **increase** its utility, and that contradicts the optimality of its strategy.

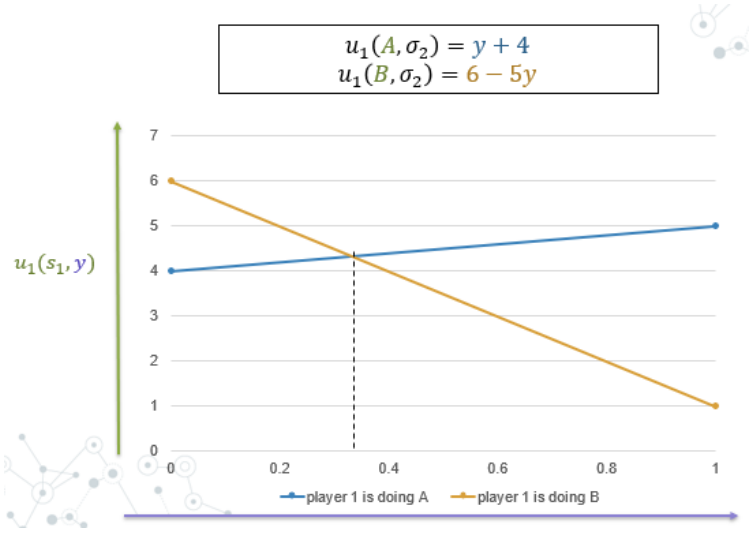


Figure 10.6 Another way to extract a mixed NE

Definition: An outcome is said to be ***Pareto optimal*** (or ***Pareto efficient***) if there is no other outcome that makes one agent better off without making another agent worse off.

Note: By outcome we mean the actions that each player **chose** (in total).

For example, in the figure to the right – CC and DD are NE's; but DD is **not** pareto optimal, since rechoosing CC does not decrease any player's utility.

Note: As not every NE is pareto optimality, not every pareto optimality is a NE! In the green figure to the right, DD is pareto optimality but **not NE**.

	C	D
C	4, 4	1, 2
D	2, 1	3, 3

	C	D
C	0, 0	0, 0
D	0, 1	1, 0

Figure 10.7 Pareto Optimality examples

Definition: The ***social welfare*** of an outcome ω is the sum of the utilities that the agents get from ω:

$$\sum_{i=1}^{n} u_i(\omega)$$

10.2 Dominated Strategies

Definition: s_i is a ***dominated strategy*** if: $\exists s_i' \quad \forall s_{-i} \quad u_i(s_i, s_{-i}) \leq u_i(s_i', s_{-i})$.

Note that s_i' in the definition is the **same** s_i' for every s_{-i}.

If one of a player's strategies is **dominated**, then assuming it's a "classical" game (every player has perfect information and rationality), we can deduce that player **won't** play that strategy. This can make things easier when looking for NE's of a specific game. For example, in Figure 10.2 Normal Form example – N is dominated by S, thus N is removed; The same goes for W (dominated by A), then S (by M) and in the end the only outcome is (A, M). This method is called *iterated elimination*.

2 more fundamental results related to iterated elimination are:

1. In the n-player pure strategy game $G = \{S_1, \dots, S_n; u_1, \dots, u_n\}$, if iterated elimination of strictly dominated strategies eliminates all but the strategies $(s_1^*, \dots, s_n^*)$ then these strategies are the **unique NE** of the game.
2. Any NE will survive iterated elimination of *strictly* dominated strategies (where strictly dominated strategies are the same as the last definition but with $<$ instead of $\leq$).

10.3 Cake Cutting

The goal is to cut a cake between several agents in a way that'll maximize their utilities/the social welfare (or just maximize something non-trivial). Let's formalize this:

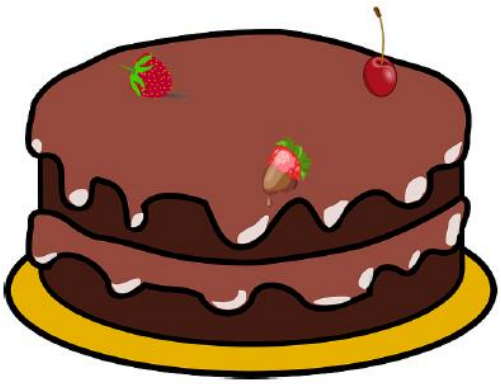
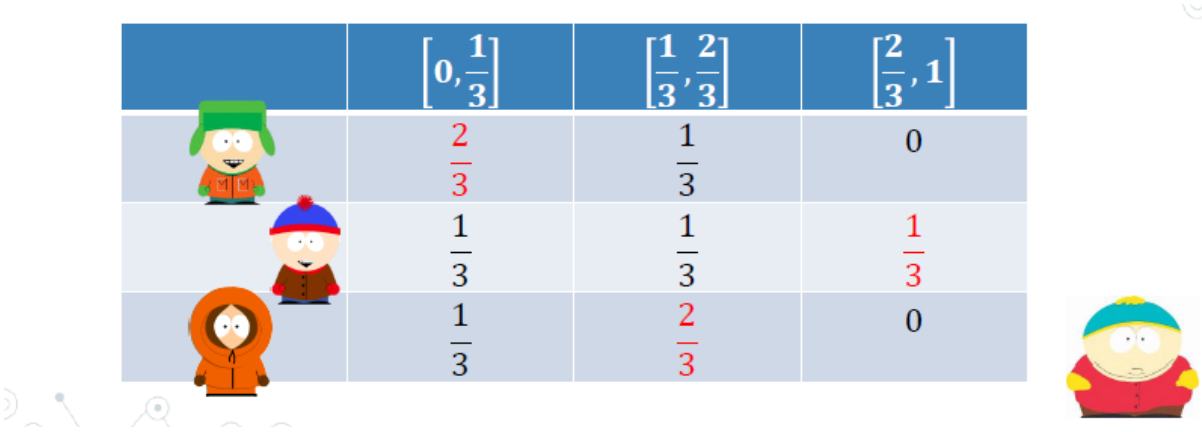


Figure 10.8 A cake. Notice the classic yet bold double-layer structure, inhabited by the unavoidable toppings, bathing in the victory sauce.

- The cake is represented by [0,1].
- Each player's preferences are represented by a function $V_i : [0,1] \to \mathbb{R}$ that satisfies:
  - $\forall A \subseteq [0,1] \;\; V_i(A) \geq 0$.
  - $\forall x \in [0,1] \;\; V_i([x,x]) = V_i(\emptyset) = 0$.
  - $V_i([0,1]) = 1$.
  - $\forall A, B \subseteq [0,1], \; A \cap B = \emptyset :$ $V_i(A \cup B) = V_i(A) + V_i(B)$.
- We assume there's a black box poly-time algorithm that can answer the following queries:
  a. $V_i(a,b)$ given $a < b$.
  b. Given a, x finds b such that $V_i(a,b) = x$.
- A division $\mathcal{S} = \{S_1, \dots, S_n\}$ is a partition of the cake and an allocation to n agents and satisfies:
  a. S_i is a **finite** set of *intervals*.
  b. $\bigcup_{i=1}^{n} S_i = [0,1]$.
  c. $\mathcal{S}$ is a set of pairwise disjoint sets.

We want to add an attribute for a division $\mathcal{S}$ – fairness. We use 2 different concepts of fairness:

1. A division is ***proportional*** if $\forall i \leq n \;\; V_i(S_i) \geq \frac{1}{n}$.
2. A division is ***envy-free*** if $\forall i, j \leq n \;\; V_i(S_i) \geq V_i(S_j)$.



	$\left[0,\frac{1}{3}\right]$	$\left[\frac{1}{3},\frac{2}{3}\right]$	$\left[\frac{2}{3},1\right]$
	$\frac{2}{3}$	$\frac{1}{3}$	0
	$\frac{1}{3}$	$\frac{1}{3}$	$\frac{1}{3}$
	$\frac{1}{3}$	$\frac{2}{3}$	0

Figure 10.9 A division example. Each row is V_i, and the division is coloured in red

Lemma: If a division $\{S_1, \dots, S_n\}$ is **envy-free**, it's also **proportional**.
Proof: for every agent i, $\sum_{j=1}^{n} V_i(S_j) = 1$, thus there's j for which $V_i(S_j) \geq \frac{1}{n}$, thus $V_i(S_i) \geq V_i(S_j) \geq \frac{1}{n}$ – thus $\{S_1, \dots, S_n\}$ is proportional.

Lemma: For $\boldsymbol{n = 2}$, a **proportional** division is also **envy-free**, and therefore the 2 are equivalent.
Proof: $V_1(S_1) \geq \frac{1}{2}$ thus $V_1(S_2) \leq \frac{1}{2} \leq V_1(S_1)$ and the same goes for $V_2(S_2)$.

If $n = 2$ then there is a *protocol* which guarantees an envy-free division.

Cut-and-Choose Protocol:

1. Agent 1 divides the cake to 2 parts A, B where $A \sqcup B = [0,1]$ and $V_1(A) = V_1(B) = \frac{1}{2}$.
2. Agent 2 chooses the part that maximizes 2's utility ($\geq \frac{1}{2}$), leaving the other part to agent 1.

This protocol guarantees an **envy-free** division because $V_1(S_1) = \frac{1}{2}$ and $V_2(S_2) \geq \frac{1}{2}$. However, it's not the optimal protocol. Finding the optimal protocol (assuming no hidden information) requires a little bit of algebra.

10.4 Cooperative Games

Cooperative games model scenarios where:

- agents benefit from cooperating.
- binding agreements are possible.

Their utilities can be either transferrable or non-transferrable (between agents). We'll concentrate on games with **transferrable** utilities.

Phases of coalitional action:

1. Agents form coalitions (teams).
2. Each coalition chooses its action.
3. Transferable utility (TU) games: The choice of coalitional actions (by all coalitions) determines the payoff of each coalition.
    a. The members of the coalition then need to divide this joint payoff.

4. Non-transferable utility (NTU) games: The choice of coalitional actions (by all coalitions) determines each player's payoffs.

In general TU games, the payoff obtained by a coalition depends on the actions chosen by other coalitions. These games are also known as ***partition function games*** (PFG). If the payoff of each coalition only depends on the action of that coalition, we call it a ***characteristic function game*** (CFG). In such games, each coalition can be identified with the profit it obtains by choosing its best action.

Figure 10.10 Every TU is a NTU

Even though agents work together they are still **selfish**, therefore the partition into coalitions and payoff distribution should be such that no player (or group of players) has an incentive to *deviate*. Let us now formalize all of this.

Definitions:

1. A ***transferable utility game*** is a pair (N, v) where:
   - $N = \{1, \dots, n\}$ is the set of players.
   - $v : 2^N \to \Re$ is the utility function.

   Usually, it is assumed that v is normalized – $v(\emptyset) = 0$; non-negative – $v \geq 0$; monotonic – $C \subseteq D \Rightarrow v(C) \leq v(D)$.

2. A ***coalition*** is any subset $A \subseteq N$. N itself is called the ***grand coalition.***

Example: 2 Simpsons (boy, girl, nerd) want ice cream. The setup is as follows:

w	500	750	1000
p	7	9	11

Figure 10.11 Transferable game example

And v is defined by – $v(\emptyset) = v(\{B\}) = v(\{L\}) = v(\{M\}) = 0$; $v(\{L, M\}) = 500$; $v(\{B, L\}) = v(\{B, M\}) = 750$; $v(\{B, L, M\}) = 1000$.

Definition: An ***outcome*** of a TU game $G = (N, v)$ is a pair (CS, x), where

- $CS = (C_1, \dots, C_k)$ is a k-partition of N, which represents a **coalition structure**.

- $x = (x_1, \dots, x_n)$ is a **payoff vector** that satisfies – $x_i \geq 0$; $x(C) = \sum_{i \in C} x_i = v(C)$ for $C \in CS$.

Example: $v(\{1,2,3\}) = 9, v(\{4,5\}) = 4$. Then $\big((\{1,2,3\},\{4,5\}),(3,3,3,3,1)\big)$ is an outcome; $\big((\{1,2,3\},\{4,5\}),(2,3,2,3,3)\big)$ is **not** an outcome – transfers between coalitions are not allowed.

Definition: An outcome (CS, x) is called an ***imputation*** if it satisfies individual rationality:

$$\forall i \in N \quad x_i \geq v(\{i\})$$

Definition: A (TU) game $G = (N, v)$ is ***superadditive*** if

$$\forall C, D \subseteq N \quad C \cap D = \emptyset \Rightarrow \quad v(C \cup D) \geq v(C) + v(D)$$

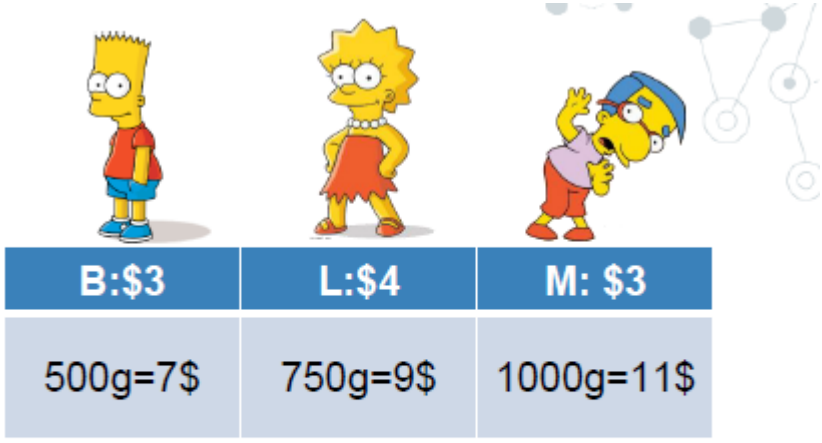
Example: if $v(C) = |C|^2$ then v is in a superadditive game –

$$v(C \cup D) = (|C| + |D|)^2 \geq |C|^2 + |D|^2 = v(C) + v(D)$$

In superadditive games, two coalitions can always merge **without** losing money; hence, we can assume that players form the *grand coalition* N.

Let's return to the Simpsons example. However, since Bart was pickpocket, the setup is now this:

This is a *superadditive* game, therefore we can deduce that they'll form the grand coalition and buy a medium-sized ice cream.



How should the players share the ice cream (**TU**)?

$$v(\emptyset) = v(\{B\}) = v(\{L\}) = v(\{M\}) = V(\{B,M\}) = 0,$$
$$v(\{B,L\}) = v(\{L,M\}) = 500,$$
$$v(\{B,L,M\}) = 750.$$

Figure 10.12

If the payoff vector is (350,200,200), Lisa and Milhouse can get more ice cream by buying a 500g tub on their own and splitting it equally. Therefore, the outcome (350,200,200) is **not** stable (when $CS = \{N\}$)!

Definition: The ***core*** of a (TU) game is the set of all stable outcomes, i.e., outcomes from which no coalition wants to deviate from:

$$core(G) = \{(CS, x) | \sum_{i \in C} x_i \geq v(C) \ \ for\ all\ C \subseteq N\}$$

In other words, each coalition earns at least as much it can make on its own (note that G isn't necessarily superadditive).

In the last example:

- (350,200,200) is not in the core because $x(\{L,M\}) = 400 < 500 = v(\{L,M\})$.
- (250,250,250) is in the core.
- (0,750,750) is also in the core!

The core is a very attractive solution concept. However, some games have empty cores:

$$G = (N, v), N = \{1,2,3\}, v(C) = 1_{|C|>1}$$

Consider an outcome x: It holds that $x_i > 0$ for some i. Then $x(N \setminus \{i\}) < 1 = x(N)$, yet $v(N \setminus \{i\}) = 1$ – thus $core(G) = \emptyset$.

10.5 Incomplete Information (in zero-sum games)

In some games there's information that's *hidden* – Poker, Bridge, etc. Some of the results mentioned in this section don't hold in this kind of games, since they rely on the perfect-information attribute.

Example: Imagine a version of mini poker in which Red cards are bad for A and Black cards are good.

Player A is dealt a card. It is red or black with 50% probability.

A may resign if the card is red → A loses 20c.
Else A "holds".

B may then resign → A wins 10c.
B may "see":

If card is red → A loses 40c.
If card is black → A wins 30c.

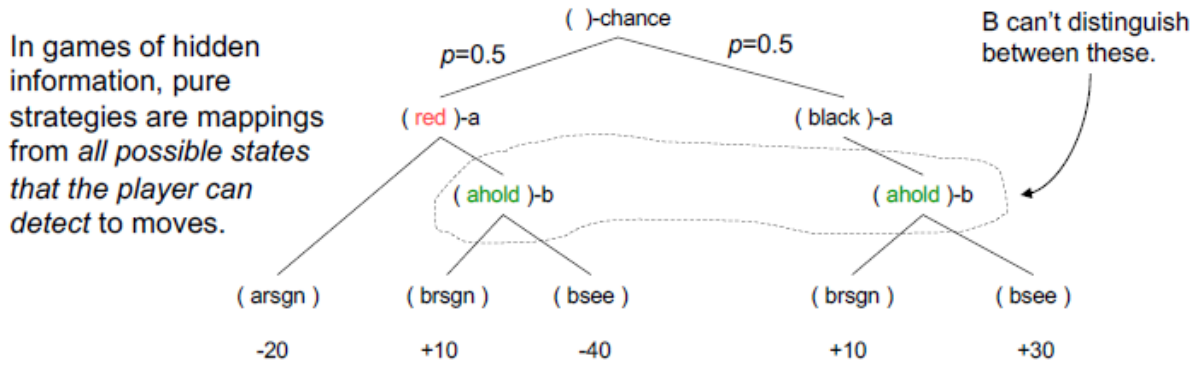


Figure 10.13 A game with hidden information (circled)

To convert this to a normal form, the i, j entry of the matrix will be the **expected** utility results from A playing i and B playing j (remember that this is a *zero-sum* game).

	B-resigner	B-seer
A-resigner	-5	+5
A-holder	+10	-5

*Figure 10.14 Normal form (entries are now **expected** values)*

Note:

- It turns out for the game of mini poker, the game theoretic value for A is 1 cent. A can expect to win 1 cent per game on average if A does the right thing.
- Furthermore, A can even tell B in advance its strategy. That information won't help B.

11 Auctions and Voting

11.1 Auctions

In auctions there are usually an **auctioneer** which desires to *maximize* the price, and **bidders** which desire to *minimize* the price. Usually, the parameters that define auctions are:

- Goods' value: Private, common, correlated.
- Winner determination: By first price, by second price.
- Bids: Open cry, sealed bid.
- Bidding: Ascending, descending.

We'll review kinds of auctions.

English Auctions:

- First price, open cry, ascending.
- Dominant strategy is for agent to successively bid a small amount more than the current highest bid until it reaches their valuation, then withdraw.
- Susceptible to:
    - Winner's curse – the winner's bid may be higher than the winner's evaluation of the item.
    - Shills – "fake" bidders who raise the item's price.

Dutch Auctions:

- First price, open cry, descending – the first bidder to bids win the item.

First-Price Sealed-Bid Auctions:

- First price, sealed bid – bidders submit a single, sealed bid with their offer (meaning they don't know what the others offer), and the item is allocated to the highest bidder.
- The best strategy is to bid less than true valuation.

Vickrey Auctions:

- **Second price**, sealed bid – like the previous kind but the winner pays the second highest bid.
- Bidding to your true valuation is dominant strategy in Vickrey auctions:
    - Bidding high opens a gap where others may bid, and you'll have to pay too much. This is the only situation where bidding high made you win and bidding true valuation didn't.
    - Bidding low opens a gap where others may win but doesn't affect your payment.

11.2 Voting

Motivation: Agents have to reach a consensus regarding a preferred alternative in a shared environment.

Definition: A ***voting protocol*** is a function from all the voters' (linearly ordered) preferences to an outcome represented by either an ordered list or a winning candidate.

There is no "best" function that chooses the "right" answer (objectively), but different functions can satisfy different criteria that look important. However, some combinations of criteria are **impossible** to achieve.

Arrow's Impossibility Theorem: Define:

- ***Universality*:** Should create a deterministic, complete social preference order from every possible set of individual preferences orders.
- ***Citizen sovereignty*:** Every possible order should be achievable by some set of individual preference orders.
- ***Non-dictatorship*:** The social welfare function should be sensitive to more than the wishes of a single voter.
- ***Monotonicity*:** Changing favourable to candidate x does not hurt x.
- ***Independence of irrelevant alternatives*:** If we restrict attention to a subset of options and apply the social welfare function only to those, then the result should be compatible with the outcome for the whole set of options.

Then **no** system meets all these criteria if there are two or more voters, and three or more choices.

Gibbard–Satterthwaite Theorem (regarding systems that choose a single winner):
For three or more candidates, one of the following three things **must hold** for every voting rule:

- The rule is *dictatorial*.
- There is some candidate who *cannot* win, under the rule, in any circumstances.
- The rule is *manipulable*.

Manipulations are when agents vote strategically – for example in order to lower one candidate's chances of going through the first round, placing their real preferred choice against a weaker candidate.

We'll now go over some voting protocols and see how each one fails in its own unique way, where each voter has a *list* of preferences, linearly ordered.

Sequential Pairwise Voting

2 candidates are matched-up; then the winner is matched-up against a third candidate and so on, $n-1$ times (where n is the number of candidates). Therefore, there are $\frac{n!}{n-1}$ agendas, and theoretically anyone can be a winner – as in the example to the right.

This voting protocol is susceptible to *manipulations* – if the **third** voter switches between b and c and agenda ii is played,

1 voter	1 voter	1 voter
a	c	b
b	a	d
d	b	c
c	d	a

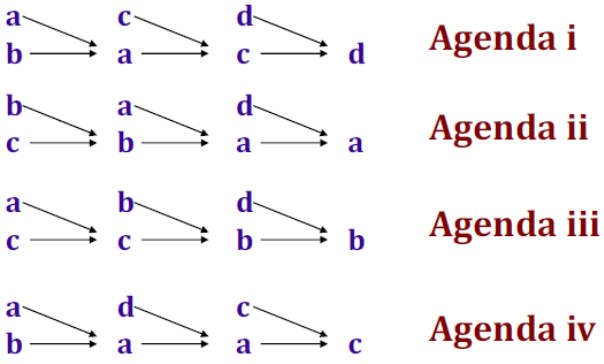


Figure 11.1 Sequential Pairwise Voting. Anyone can be the winner!

then c will win against b – leading to **d** winning the elections.

Definition: If a voting protocol satisfies: "If every voter prefers an alternative x to an alternative y, the voting rule should not produce y as a winner" then we say that it satisfies the ***Pareto criterion***.

Sequential Pairwise Voting does **not** satisfies the Pareto criterion: in agenda i **d** wins – even though the majority prefers *b* over d.

Plurality Voting

Each agent votes for 1 alternative; the candidate with the most votes wins.

Plurality voting *violates* the Pareto criterion as well – here *c* wins, even though the majority (5-4) rate c as last.

Moreover: In pairwise decisions, b, which came in last in the plurality vote, would beat both c and a; and c, which won the plurality vote, would have lost all pairwise contests.

Definition: A candidate *a* is ***Condorcet winner*** if *a* beats every other alternative in pairwise election. It follows that if such *a* exists, it's *unique*.

Definition: In a voting protocol, if a Condorcet winner is always elected, then the protocol is ***Condorcet consistent***.

3 voters	2 voters	4 voters
a	b	c
b	a	b
c	c	a

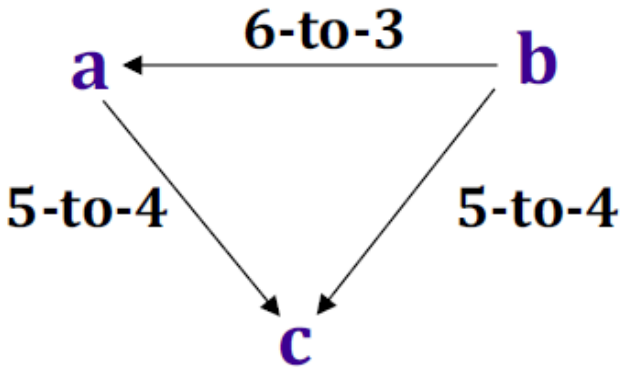


Figure 11.2 Plurality Voting. The winner may not have the support of the majority.

Plurality voting is **not** Condorcet consistent. Furthermore, it might choose a candidate which *loses* to every other candidate (see the last example). However, sequential pairwise voting *is* Condorcet consistent.

Another Condorcet consistent voting rule is Copeland: The winner is the candidate with the most wins in pairwise elections.

Plurality with Runoff

Same as plurality voting but another round is pulled off where the only candidates are the first & second place from the previous round.

6 voters	5 voters	4 voters	2 voters	(last 2 voters, strategic)
a	c	b	b	a
b	a	c	a	b
c	b	a	c	c

Figure 11.3 Plurality with runoff. Susceptible to manipulations

In the example above if the last 2 voters don't "lie" – a wins. However, if the 2 voters vote strategically – a and c are the top 2 – and **c** wins.

Definition: The ***monotonicity criterion*** is: If x is a winner under a voting rule, and one or more voters change their preferences in a way favourable to x (without the changing the order in which they prefer any other alternatives), then x should still be the winner.

Plurality voting *is* monotonic, but **not** with runoff.

Borda Count

Each voter submits his n-list of preferences. The k'th alternative receives $n - k$ points. The candidate with the most points wins.

With Borda count, a: 6, b: 7, c: 2. So b wins, but *a* is the Condorcet winner. Moreover, a has an absolute majority of first place votes.
The existence of c allows the 2 voters to weight b over a more heavily than the 3 voters choose to weight a over b, enabling b to win the Borda count. So, Borda's method aggregates not only the first choices, but *all* choices, so changes in preferences are less subtle here.

3 voters	2 voters	
a	b	2 points
b	c	1 point
c	a	0 points

Figure 11.4 Borda count. Doesn't care who's first

Definition: A ***scoring rule*** is a vector $\alpha = (\alpha_1, \dots, \alpha_n)$ where $\alpha_i \geq \alpha_{i+1}$, and represents the voter's preferences. The candidate that matches the i'th place (in vector α) gets α_i.

Examples: Plurality: $(1,0,\dots,0)$; Veto: $(1,\dots,1,0)$; Borda: $(n-1, n-2, \dots, 0)$.

Single Transferable Vote

This voting rule's approach is to eliminate unwanted candidates. In each round, each voter awards one point to candidate he ranks **highest** out of *surviving* candidates – candidate with least points is eliminated.

Here's an example with shows that every voting rule of the above can choose a different winner:

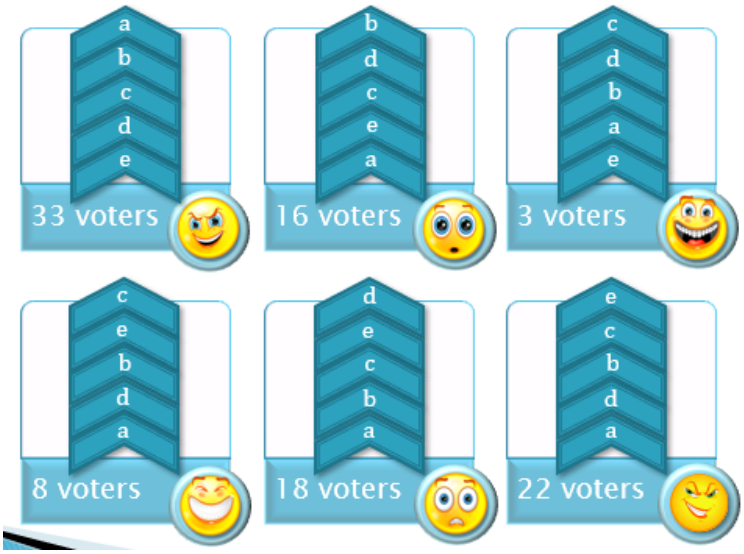


Figure 11.5 An example where each voting rule chooses a different winner

11.3 Voting Power

Consider a 4-members body, A B C D, and they vote for something (yes/no). If there's a tie, A gets to break the tie. How much voting power does A have?

Definition: Assume all orderings are possible, see when each member is pivotal (i.e., the losing coalition becomes the winning coalition when that member joins it, left to right). Then the ***Shapley-Shupik index*** of a voter A is the number of orderings where A is the pivot.

In the last example, there are $4! = 24$ orderings, and A is the pivot in 12 of them, therefore A's Shapley-Shubik index is $\frac{1}{2}$.

ABCD	ADBC	BCAD	CABD	CDAB	DBAC
ABDC	ADCB	BCDA	CADB	CDBA	DBCA
ACBD	BACD	BDAC	CBAD	DABC	DCAB
ACDB	BADC	BDCA	CBDA	DACB	DCBA

Figure 11.6 The pivot in each ordering is underlined

Sometimes there are bodies where different members' votes are weighted: We denote the game as $[q; w_1, \dots, w_n]$ – $\boldsymbol{q}$ is required to pass a law, each member i gets w_i weight for his vote. This is a generalization of the last example – it can be written as [3; 2,1,1,1].

Example: [5; 2,2,1,1]. The orderings are: 2211, 2121, 2112, 1221, 1212, 1122. 2 is pivot in $\frac{5}{6}$ of them, and there are 2 2's. Therefore, the *power indices* (Shapley-Shubik indices) are $\left[\frac{5}{12}, \frac{5}{12}, \frac{1}{12}, \frac{1}{12}\right]$.

Another example with **committees**. 3-member committee (A's) in 9-member body (6 other B's). To be approved, law must have support of 5 members, **including** at least 2 A's. There are

$\binom{9}{3} = 84$ orderings. B is pivot in $\binom{4}{2}\binom{4}{1} + \binom{4}{3}\binom{4}{0} = 24 + 4 = 28$ orderings, so A is pivot in 56 orderings. In total, the power index of each A is $\frac{1}{3} \cdot \frac{56}{84} = \frac{2}{9}$, and B's is $\frac{1}{6} \cdot \frac{28}{84} = \frac{1}{18}$.

The same method holds if A's and B's groups share the same power (passing a law need most A's **and** most B's), or if A's must agree unanimously. The approach to find the power indices in each setup is by combinatorics and sometimes a little bit of determination as well.

12 Deep learning

12.1 Neural Networks[6]

In real life, $|S|, |A|$ of the are too big & complex to use the Q-learning algorithm directly. However, we can *approximate* the Q-function. An effective and popular approach is **(Artificial) Neural Networks**. Neural networks try to imitate the brain – they consist of neurons and connections between them. Each neuron holds a value, and if that value is big enough the neuron will fire it to the next neurons that are connected to it. A simple neural network is a fully connected layers graph, divided to 3 parts:

1. Input layer: Stores the input data.
2. Hidden layers: Manipulate the input data in order to approximate the Q-function.
3. Output layer: Stores the output data – what should be done next.

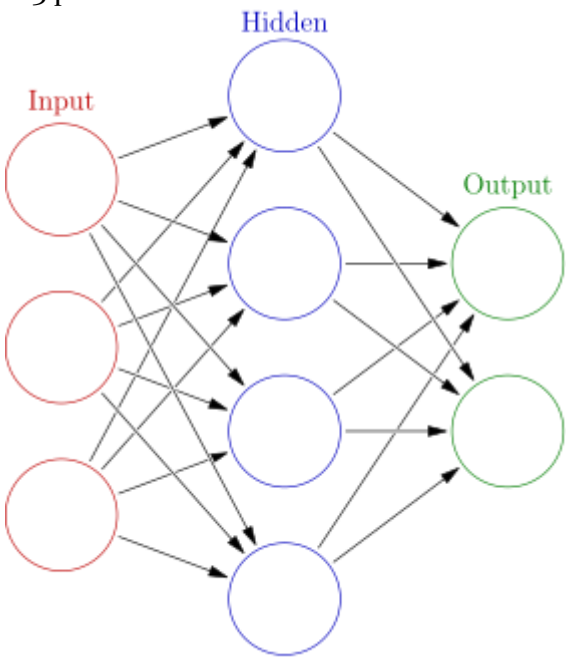


Figure 12.1: A simplified neural network's architecture[7]

More specifically, each neuron holds a number x, weight w, and bias b. Then it computes $wx + b$, applies an **activation function** on the result, and passes its output to its neighbours in the next layer. The activation function is a function that decides if the neuron should be activated or not.

Since the graph is fully connected, the input for a layer $i + 1$ can be written as vector multiplication:

$$x_{i+1} = activation(Wx_i + b)$$

Where:

$$x_i = \begin{bmatrix} x_i^1 \\ \vdots \\ x_i^n \end{bmatrix}, W = \begin{bmatrix} w_1^1 & \dots & w_n^1 \\ \vdots & \ddots & \vdots \\ w_1^m & \dots & w_n^m \end{bmatrix}, b = \begin{bmatrix} b^1 \\ \vdots \\ b^m \end{bmatrix},$$

$$x_{i+1} = \begin{bmatrix} x_{i+1}^1 \\ \vdots \\ x_{i+1}^m \end{bmatrix}$$

Note that m, n can be different, meaning that layers can have different number of neurons.

After calculating the output, usually we apply a normalization function (e.g. SoftMax) to convert the outputs to probabilities. Each output neuron basically represents an action a and tells how much does the NN thinks it's good to do a, with confidence as a number in [0,1].

Training the Neural Network After the whole calculation (or feedforward), given a **loss function** we can check how close are we to the correct answer. Then, if we want to minimize the loss, the solution is to tune the weights and the biases. To do that, a simple solution is to use a local search method called Gradient Descend. This method is a function which tells us where in the evaluation function's space we should go and how much. The result is back

[6] It is very recommended to watch 3Blue1Brown's series: *But What is a Neural Network?*
[7] https://en.wikipedia.org/wiki/Artificial_neural_network

propagated from the output layer all the way to the input layer, then doing the process again with another input.

12.2 Monte Carlo Tree Search

Recall Minimax and Monte Carlo from weeks 3 and 8, and the rules of *Go*.[8] In Go's case, the game tree is stupidly large, so Minimax is useless. However, if looking at the problem as finding the Q-function, another solution rises: *Approximating* the Q-function (or the best strategy) by simulating random episodes, and that's exactly what **Monte Carlo Tree Search** does.[9] The Monte Caro Tree Search consists of 4 steps:

1. Select: Explore vs. Exploit random choice based on ε parameter.
    a. If explore – select a random action.
    b. If exploit – select the most promising leaf.
2. Expand: In order to grow the tree, we wish to expand some child nodes. Sample one child randomly.
3. Simulate (also called rollout): Start a random game, step by step until a critical point – e.g. win/lose. Then evaluate the path based on the terminal node (might run all the way down to a leaf).
4. Backpropagation: Update the winning rates of all known and involved choices.

12.3 AlphaGo

Doing this many, *many* times can produce an estimation of the minimax tree and it turns out to be very good, compared to humans. How good? In 2016, DeepMind's *AlphaGo* challenged Lee Sedol (a very good Go player) to a 5-games matchup and won 4-1.[10]

A later, even better version, named AlphaZero was released by DeepMind in 2017.

[8] https://en.wikipedia.org/wiki/Rules_of_Go

[9] For a more detailed explanation with images (which doesn't fit here well) see MCTS on Wikipedia.

[10] https://www.youtube.com/watch?v=WXuK6gekU1Y. The Netflix film, AlphaGo, is also recommended.