CS3243 Introduction to Artificial Intelligence

Choong Wey Yeh

AY2019/20 Semester 2

1 Uninformed Search

In this section, we will be discussing some of the basic uninformed search algorithms where these algorithms do not make use of certain heuristics to optimize their search strategy. Most of the time, we will observe that these basic uninformed search algorithms perform complete searches; that is they iterate over all the possible elements in the solution space before terminating.

For this section, we will be working with:

- **Fully-Observable Environment:** Each item in the solution space is known to the algorithm; i.e there are no hidden states
- **Deterministic Setting:** No uncertainty/probabilistic notions involved in state change
- **Discrete Environment:** States consist of discrete variables instead of continuous uncountable ones

Some of these examples include winning a sequence of moves in a single-player game, assembly of machine parts and boolean satisfiability(SAT) problem. Essentially, we can formulate each of these examples as a search problem where we iterate over the possible solutions. For example, in the SAT problem, we can perform a complete uninformed search over each combination of variables such that the boolean formula is satisfied. Of course, we will eventually see that this is not an efficient(nor practical) way to attain a solution for various problems.

1.1 Problem Formulation

Formally, we can define the basic elements of a problem definition using **states**, which are the current configurations for the problem(e.g the variable values for a SAT problem), and **actions** which causes a transition of one state to another via changing configurations(setting a boolean variable from true to false). When considering the configurations for state and actions, most of the factors are unnecessary and may be **abstracted out**. For the route planning problem in particular, we are concerned with how route the driver takes to get to a location, but *not exactly how he drives there*, such as which lane or which car he uses. In this case, such information may be left out of the problem definition.

Additionally, we also have an **initial state**, which are the parameters that the algorithm starts with to search for the solution. This initial state can be default setting of certain parameters, or random initialization. But, we also need the following definitions:

1.1.1 Goal State

The *goal state* is defined as the state that determines the termination of the algorithm as it has reached the objective of the solving the problem. When the search algorithm reaches a state that is deemed as the goal state, it will stop and return it's current solution. The goal state can be defined in various ways:

- **Explicit Set** of goal states, which can be the configurations that the algorithm permutates through. Often for problems with large solution spaces, this is not very practical as the set could be very large.
- **Implicit Function** $f(x)$ that evaluates the current configuration of the algorithm and outputs if the configuration is indeed a goal state or not, where x is the current configuration used as the input.

Ideally, we want the evaluation of whether the current configuration is the goal state or not to be *efficient*, as this evaluation function is called every single time the algorithm changes states, and could be called a large number of times(e.g if boolean formula in SAT is very long). Hence, an inefficient evaluation function could be computationally expensive and slow down the search process immensely.

1.1.2 Path Cost

When we are discussing about a search problem, very often each transition from one state to another incurs a certain cost or penalty, such in the route planning problem where there is a cost incurred moving from a location to another. Then, a sequence of state transitions. which are viewed as actions, is a *search path* taken by the algorithm to find the optimal solution by viewing the possible actions that can be taken by the algorithm now as a decision tree like structure. We denote the cost of an action, such as a transition from one state to another as the following function:

$$c(s, a, s')$$

where an action of a causes the algorithm to transition from state s to s'. Here, if a_i is the i^{th} action taken by the algorithm, then the sequence of actions $a_1, a_2, ..., a_n$ is the search path of the algorithm. If this sequence of actions results in the final state being the goal state, then it is a solution to the problem. Note that the solution **may not be unique**.

1.1.3 Performance

Not required for the problem formulation, but it is also eventually necessary to make sure that our algorithm works, and is efficient. In particular, we need to consider:

- **Termination:** We want for our solution to eventually find a solution. If the algorithm does not terminate at all, then we may never get an answer as the algorithm continues running
- **Optimality:** We want a feasible solution, but above that we want to find the optimal feasible solution. That is, for a solution y, the solution is optimal if for all other feasible solutions y', $c(y) \leq c(y')$. That is, the cost of our solution is minimal
- **Efficiency:** defined as the search cost as well, is the time and space complexity of our algorithm with respect to the size of the solution space

1.2 Search Strategies

1.2.1 Search Representation

We shall represent the states and actions in our search problem as nodes and edges in a tree or graph, such as in **tree search** and **graph search**. Each node represents a state from the state space, and the edge connecting node s and s' represents a transition from state s to s'.

Additionally, we also introduce the definition of a **frontier**. After we have reached a node x, the other nodes connected to x are divided into those that are visited, and those that are not. The nodes that are visible from node x, but are not yet visited and can be expanded for exploration. These nodes form the frontier. Our search algorithm makes use of the frontier as a pool of nodes that are available to be visited next down it's search path. In tree search, we are allowed to repeat the nodes that we have visited(e.g backtracking), but in graph search we no longer return to a node once we have visited it.

1.2.2 Search Performance

Previously, we mentioned that the following criteria will be used to judge the performance of our algorithm:

- **Completeness:** Is the algorithm guaranteed to find a solution(if the solution exists)?
- **Time Complexity:** How long does the algorithm take to find a solution in the worst case?
- **Space Complexity:** How much space is required for the algorithm to perform the search?
- **Optimality:** Does the algorithm find the highest quality solution in the presence of multiple feasible solutions?

We shall explicitly evaluate the upcoming algorithms using the following parameters:

- **Branching Factor** b**:** The maximum number of children, or successors of a node in a search tree of graph
- **Depth of Shallowest Goal Node** d**:** Height of the tree before the shallowest goal node is reached
- **Maximum Depth** m**:** Maximum depth of the search tree

Instead of the standard time complexity evaluation using V and E as the number of nodes and edges, due to the large size of search problems.

1.2.3 Breadth First Search

The basic idea of BFS is that at any point of time in the search, the shallowest nodes are explored first. In particular, if the frontier consists of nodes from depth $d, d+1, ...$ then all nodes at depth d are visited first before $d+1$. BFS can easily be implemented by using a queuing function that accepts the queued neighbours of visited nodes, and outputs them in the order that it is received where newly generated states are polled last.

- **Completeness:** BFS is a very systematic strategy, where it observes all nodes of depth 1, then depth 2 and so on. Hence, at depth d, BFS would have finished visiting all nodes of depth $d-1$ and lesser. By the time BFS reaches depth $m+1$, it would have visited all nodes of depth m and before, which is all the nodes in the tree. Hence if a solution exists, it is sure to have found one.

- **Time Complexity:** Consider the case where each node in the search tree has the maximum number of children b. Then, the first level of the search tree would generate b children, and each of these children continues to generate another b and so on. Then, the maximum number of nodes that the algorithm has to visit would be:

  $$1 + b + b^2 + ... + b^d = O(b^d)$$

  where at the i^{th} level, the search tree has b^i children. Since the algorithm terminates when it has found the first goal node, if the shallowest goal node is at level d then BFS would have eventually found it and does not continue past depth d.

  While the time complexity is exponential in the input, BFS can be useful if heuristic says that the search tree is wide but shallow, or if the goal node is within a certain depth from the root.

- **Space Complexity:** Similar to the time complexity, assuming the worst case scenario of a complete tree with branching factor b, then there are $O(b^d)$ children, and hence $O(b^d)$ amount of memory required to store all these children.

- **Optimality:** If we consider a special case of search where the search step cost is 1, then the optimal solution for this search problem would be a feasible solution with the least number of nodes in the search path, which would be the shallowest goal node which has the lowest possible depth. Since BFS traverses the tree level by level, it would find the shallowest goal node first among all other goal nodes, which is the optimal solution.

  However, most of the time the step cost is not unit, and in that case BFS does not usually find the optimal solution. This is because it finds the shallowest goal node, but it is not necessarily the *least cost* path solution. We shall try to rectify this issue in the next search algorithm.

1.2.4 Uniformed Cost Search

UCS modifies the BFs strategy by always expanding the lowest cost node in the frontier, measured by a path cost function $g(n)$ as the cost of the path from the root to the current node n. It is trivial to see that BFS is UCS with path cost function $g(n) = depth(n)$. In implementation, we modify from BFS slightly again by changing the queue used a priority queue, or a heap data structure that emphasises cheapest paths.

- **Completeness:** UCS is complete, under the caveat that the minimum step cost is $\geq \epsilon$, where ϵ is a non-negative number. If there is no bound on the step cost, then we could have cases of negative costs or exponentially decreasing costs. In this case, the algorithm could be stuck without termination.

- **Optimality:** Under the same conditions as completeness, the solution is optimal provided step costs are lower bounded by a non-negative number. This is because the first solution

found by the algorithm is guaranteed to be the cheapest cost solution. If there were a cheapest cost solution, the algorithm would have found it first.

Note the observation that because step cost $\geq \epsilon$, the step cost must never decrease. In other words, $g(successor(n)) > g(n)$. However, if we lift the lower bound on step cost, then the search problem is reduced to that of BFS, because a long and expensive path may become the optimal by running into a node with extremely high negative cost. As a result, we need to search all possible paths to obtain the optimal.

- **Time Complexity:** Remember that in the queue, we keep at most b^i nodes in the queue if we are at depth i. Since the step cost $\geq \epsilon$, when the algorithm goes down a search path from depth i to $i+1$, the overall current cost increases by at least ϵ. Consequently, if the optimal path cost is C^*, then there are at most $\frac{C^*}{\epsilon} + 1$ nodes in the search path since the cost increases in sequence of $0, \epsilon, 2\epsilon, ..., \frac{C^*}{\epsilon}$. This also means the maximum depth the algorithm will go is at most $\frac{C^*}{\epsilon} + 1$, and hence the maximum number of nodes that it will visit is $b^{\frac{C^*}{\epsilon}+1}$
- **Space Complexity:** Similarly, since the maximum depth is $\frac{C^*}{\epsilon}+1$, the maximum number of nodes that will be stored in the heap will be at most $b^{\frac{C^*}{\epsilon}+1}$.

1.2.5 Depth-First Search

DFS expands one of the nodes the deepest level of the tree during the search instead of covering all nodes in a level. Only when the search hits the most bottom of the search path then does it go back to a shallower level and expand the nodes at those levels. This is good for search problems where it is beneficial to explore deeper rather than breadth. Implementation wise, DFS uses a stack instead of a queue to pick the next node.

- **Completeness:** Similar to BFS, DFS is complete as it eventually expands all nodes in a frontier and explore all possible paths, given that the depth of the search tree is *finite*. For trees with infinite tree depths, then the algorithm may never terminate.
- **Optimality:** Like BFS, DFS is not optimal for the same reason that it does not use any heuristic on path cost at all.
- **Time Complexity:** Assuming finite search tree height of m, let us consider the worst case that the optimal solution is a search path of length m and that it is the rightmost path. Then, notice that because of the way DFS explores paths, DFS will exhaust all other possible paths before obtaining the optimal by depth-first traversal. If all nodes have the maximum successor branching of b, then that means DFS will go through all combinations of b^m search paths in the worst case.
- **Space Complexity:** DFS has a better space complexity than BFS of $O(bm)$. Each time the algorithm expands a node, it pushes all the successors of that node into the stack, which is at most b. Since the maximum depth, and the length of a search path is m, and DFS fully explores and exhaust a search path before it expands another node, the maximum number of items pushed on the stack is bm.

  In fact we can continue to improve this space complexity further. Notice when we push the successors onto the stack, we pick one successor to continue exploring down. In this process, other successors are not used until all the possible paths following the picked

successor is exhausted. In that case, we merely have to push the *current* successor onto the stack to be tracked without the other successors. The algorithm can then backtrack and then pick another successor to expand along later in the search. Then, the space complexity is equivalent to the number of successors picked in a path, which is also equivalent to the maximum path length that is $O(m)$.

1.2.6 Depth-Limited Search & Iterative Deepening Search

We have seen that for trees with infinitely long tree heights, then the algorithm could potentially run forever. If we knew that we only had limited time to run our search, or that the goal node is within a certain height, then we could limit the maximum height that DFS runs to. That is, we can force our algorithm to terminate once there are no longer any more search paths with length $l \leq l*$ where $l*$ is the maximum height specified.

However, note that if we do so, then our DLS algorithm is **not complete**. Because it only considers search paths of length l^* and lesser, any search paths with longer lengths is ignored. If the goal node is of length $\geq l*$, then our algorithm would never return a solution.

To address this, we introduce an additional condition: To perform DLS *until a solution is found*. This guarantees that the algorithm is complete and will return a solution to us. Additionally, this algorithm is similar to BFS in the sense that the search deepens level by level, like how BFS expands nodes in a depth first.

Algorithm 1 Iterative Deepening Search

```
1: for depth = 0 → ∞ do
2:     result ← DEPTH-LIMITED-SEARCH(problem, depth)
3:     if result ≠ cutoff then
4:         return result
5:     end if
6: end for
```

This allows us to maintain the space complexity benefit of DFS over BFS, but also allows us to address the problem of large state space and unknown tree depth(which is often most of the time).

However, now we have another downside: we essentially repeat searches over subtrees of the search tree. We first perform DLS with depth d, then $d+1$ and so on. Notice that the search space on the search tree limited with depth d, is a subspace of the search space on a tree with depth $d+k$. Hence, we are repeating some of the search from the previous iteration of IDS. In particular, the number of nodes expanded by DLS on depth d is:

$$1 + b + b^2 + ... + b^{d-1} + b^d$$

and the number of nodes expanded by IDS is:

$$(d+1)1 + (d)b + (d-1)b^2 + ... + 2b^{d-1} + 1b^d$$

Notice that of all the subtrees of different depths of the search tree, IDS explores subtrees of shallowest heights the most, while the bottom-most parts of the trees is explored lesser. This is because of the fact that in every iteration of IDS, it performs DLS restarting from the root again. Hence the initial parts of the search tree are traversed more often.

In particular, the further down the tree the search goes, the number of successors increases, perhaps exponentially. Since these parts of the tree are explored much lesser than the top, which has much lesser nodes, in perspective the search is repeated on a very small subset of the nodes of the entire search space considered, and hence the overhead in practice is actually within accepted bounds.

Remarks

- If b is not finite, then all the uninformed algorithms we have discussed are not complete, since they will never be able to exhaust fully nodes of any depth
- If b is finite, then BFS and IDS is complete, since IDS iteratively increases the depth similar to BFS. Additionally, if the step cost is lower bounded by ϵ, then UCS is also complete
- If m is finite, then DFS is complete, otherwise it may continually run forever down an infinite search path
- if the step cost is unit(i.e $= 1$), then UCS is reduced to BFS, and BFS and IDS are optimal because the step cost is equivalent to the number of nodes in search path

2 Informed Search

Previously, we have seen search strategies that makes use of the problem definition, such as the states and actions, to perform a general search action. This usually results in some form of queuing function that determines which node to expand next deterministically. However, more often than not we do not have enough information to full model the problem to determine the full cost. For example,

- Depth of search tree or size of solution space could be possibly infinite
- Repeated states could be present
- Certain constraints could be hard to model, or cause problems to be difficult to solve

If we use uninformed search strategies for these problems, optimality and completeness may not be guaranteed, in addition to the exponential time search efficiency. In this case, we can introduce additional information or heuristics that can be exploited in order to improve our search strategy. To do this, we introduce an evaluation function $f(n) : X \to R$ which returns a number indicating the desirability of the node to expand next. This could be in terms of shortest path cost for example. The evaluation function that we define here serves to approximate the cost estimate from node to node in problems where full information is not present for us to perform uninformed search.

2.1 Best-First Search

In best-first search, we make use of the evaluation function by expanding the node that has the most desirable evaluation function score, by ordering the nodes by their evaluation function values. If the evaluation function is able to perfectly reflect the information stated in the problem definition, that is if the evaluation function is a true one, then expanding the node by the evaluation function score will be the optimal path and indeed be the best node to expand at the current point in time. However, this is often not the case and the evaluation function will reflect the node that *appears* to be the best node to expand in that point in time.

Algorithm 2 Best-First Search

1: $result \leftarrow$ next node ordered by $f(n)$
2: **if** $result ==$ *goal state* **then**
3: return *search path*
4: **else**
5: goto 1
6: **end if**

2.1.1 Greedy Best-First Search

A good Best-First Search strategy is to *minimize estimated cost to reach the goal*. That is, we define a **heuristic function** $h(n)$ that estimates the cost from a node to the goal node. Often this function provides not an exact formulation of the actual cost from a node to the goal node, but a good enough estimate of this cost. Then, we set:

- $f(n) = h(n)$ and use the heuristic function to expand the next node
- $h(t) = 0$ where t is the goal node, as the distance from the goal node to itself is clearly 0

2.1.2 Performance Measure

- **Completeness:** Similar to the uninformed search functions, Best-First Search also makes use of a queuing function, albeit using an evaluation function. This means that every node will eventually be added into the queue, and expanded until the solution is found if b is finite.

- **Optimality:** However, it is easy to see that Best-First Search **is not optimal**. Notice that Best-First Search aims to obtain low cost solutions by using the estimated cost from the node in the frontier to the goal node. However, *it does not make use of past information of the cost of the path travelled so far.* Best-First Search focuses on greedily reducing the current estimated remaining cost to the goal, and completely ignores the overall path cost to reach the goal node.

  In order to focus the search on lowest cost paths toward the goal, our evaluation function must incorporate the *cost of the path* from a state to the goal state, instead of ignoring the path of the cost so far to the current node.

- **Time Complexity:** In the worst case where the path containing the least cost weights has length of that of the height of the search tree, then the search will incur runtime exponential in the order of the branching factor $O(b^m)$

- **Space Complexity:** Similar to the time complexity, in the worst case we may maintain b^m elements in the queue for a search tree with branching factor of b and a search tree depth of m.

Apart from in-optimality issues due to ignoring of overall path cost, Best-First Search also has various other issues:

- **Susceptible to False Starts:** Because the search strategy only minimizes the current remaining estimated cost, it could be possible for the algorithm to pick the shortest node that leads to a dead end, resulting in unnecessary nodes being expanded.

- **Repeated States:** Furthermore, if we allow repeated states such as in Graph Search, then it could be possible for the algorithm to oscillate between 2 visited states if the estimated cost between the two is the cheapest among those in the frontier.

- **Infinite Search Path Length:** Similar to Depth-First Search, if the search tree path has infinite length, then it is possible for the algorithm to keep searching down that path and never terminate

2.2 A* Search

We have seen that minimizes the estimated cost to the goal using the heuristic function $h(n)$ in a bout to reduce search cost, but it is not optimal. On the other hand, Uniformed-Cost Search(UCS) reduces the cost of the path so far to the goal, but it is inefficient due to it's exponential time complexity. But we can combine both strategies to obtain a search strategy with the advantages of both. We do this by summing up the the evaluation functions of both search algorithms where:

- $g(n)$ is the cost function from UCS that measures the cost from the start node to the current node n

- $h(n)$ is the cost estimate heuristic from Best-First Search that gives an estimate of the remaining cost from the node n to the goal node
- $f(n) = g(n) + h(n)$ is the combined heuristic function that estimates the cost of the cheapest path from the start node to the goal node by expanding node n from the current node.

Hence if we are aiming to minimize the overall cost of the search path, then minimizing $f(n)$ is a reasonable strategy. In fact, we will soon prove that given the following restriction on $h(n)$, then the search strategy is complete and optimal:

Theorem 1. *If $h(n)$ is an **admissible heuristic**, then A* Search using Tree Search is optimal*

Definition 1. *A heuristic function $h(n)$ is an **admissible heuristic**, if $\forall n, h(n) \leq h^*(n)$ where $h^*(n)$ is the optimal true cost to reach the goal state from node n.*

That is, the algorithm *never overestimates the true cost of the path from node n to the goal.*

Furthermore, we shall assume the **monotonic** property for the path costs for A* search that, all path costs are non-decreasing. That is for the path containing the node n to the goal node $t*$, $f(n) \leq f(t*)$. In particular we say that the heuristic function is **consistent**.

Definition 2. *A heuristic function $h(n)$ is **consistent**, if $\forall n$, and it's successors n', $h(n) \leq c(n, n') + h(n')$.*

In some cases, we also call this heuristic function metric because it obeys the triangle inequality. A result from this definition is that path costs are non-decreasing, or monotonic. This can be seen via the following equation:

$$\begin{aligned} f(n') &= g(n') + h(n') \\ &= g(n) + c(n, n') + h(n') \\ &\geq g(n) + h(n) = f(n) \end{aligned}$$

In particular, the following lemma holds:

Lemma 1. *If a heuristic is consistent, then it is also admissible*

A simple proof of this is that, we can take any node n, then we see that:

$$\begin{aligned} h(n) &\leq c(n, n') + h(n') \\ &\leq c(n, n') + c(n', n'') + h(n'') \\ &... \\ &\leq c(n, n') + c(n', n'') + ... + c(n^{t-1}, t) + h(t) \\ &\leq c(n, n') + c(n', n'') + ... + c(n^{t-1}, t) (\text{remember } h(t) = 0) \\ &\leq c(n, t) \end{aligned}$$

If the path costs for a problem is non-monotonic, then we can use the following function to enforce monotonicity using the following *pathmax* function:

$$f(n') = max(f(n), g(n') + h(n'))$$

By the monotonicity property of the path costs, that means that A* will expand all nodes n with evaluation function costs $f(n) < f*$ where $f*$ is the cost of the optimal solution path. This is similar to the BFS behaviour where A* expands nodes layer by layer, where each layer is a contour represented by it's evaluation function cost.

2.2.1 Proof of Optimality of A* Search

Let the optimal goal state in the search tree be $t*$, and a suboptimal goal state be t. Then, $f(t) = g(t) \leq f(t*) = g(t*)$. Let us then assume then A* algorithm has picked t as the solution and terminated. We shall see that this is not possible.

Let n be an unexpanded node on the path to the optimal goal state $t*$. Such a node exists because if it does not, that means that the algorithm that already expanded all nodes on the path to $t*$, and hence found the optimal goal state. Additionally, let it be such that

$$f(t) \leq f(n)$$

such that t is expanded before n, which can be rewritten as:

$$\begin{aligned} f(t) \leq f(n) &= g(t) + h(t) \leq g(n) + h(n) \\ &= g(t) \leq g(n) + h(n) \end{aligned}$$

since $h(t) = 0$ as t is a goal state. Then, because path costs are non-decreasing, that means that:

$$\begin{aligned} g(t) &\leq g(n) + h(n) \\ &= f(n) \\ &\leq f(t*) \\ &= g(t*) + h(t*) = g(t*) (\text{since } h(t*) = 0) \end{aligned}$$

Which means that the path cost of reaching the suboptimal goal state t is lesser than that of the optimal goal state $t*$, which contradicts our initial assumption that $f(t) = g(t) \leq f(t*) = g(t*)$. Hence, we have proven that A* search is guaranteed to find the optimal goal state. Intuitively, we can see that this is true because $f(t) \leq f(t*) = g(t*) = g(n) + h*(n) \leq f(n) = g(n) + h(n)$ for any node n that is on the optimal path from the start node to the goal node due to non-decreasing costs, and by the admissible heuristic that $h(n) \leq h*(n)$. Moreover, it is also easy to see that if we have a heuristic that *overestimates* the path cost to the goal node, then the algorithm might evaluate $f(n) > f(t)$ and eventually find the suboptimal goal first.

Additionally, a stronger claim is that when a node n is selected for expanding, that means that *the path leading up to n is the shortest path found*. To see why this is so, we shall assume that the algorithm expands node n via a suboptimal path with last node q_2 instead of a optimal path with last node q_1. We know that since n was expanded before q_1, that means that $f(q_2) \leq f(n) \leq f(q_1)$. But this contradicts the optimality of the optimal path with cost $f(q_1) < f(q_2)$, and the non-decreasing property since q_1 is on the path to n and hence $f(q_1) < f(n)$.

In particular, it is also proven that A* is **optimally efficient**. That is, any other search algorithm that uses an admissible heuristic that finds the optimal goal state *does not expand less nodes* than A*. This also means any algorithm that does not expand nodes n with $f(n) \leq f(t*)$ run the risk of non-completeness and not finding the optimal goal state.

2.2.2 Proof of Completeness of A* Search

Recall that we mentioned that path costs are non-decreasing along the path. We can visualize the search landscape of the search tree in contours, where each contour is in order of increasing path costs. We can see that A* Search eventually expands all nodes in a contour before moving

onto the next. In particularly, A* search moves in increasing order of contours using the evaluation function. This means that A* search will expand up the contours, before it eventually reaches the optimal shallowest goal node.

However, this is only true under 2 conditions:

- If there are finite number of nodes with evaluation function costs $f(n) < f*$
- If the branching factor b in the search tree is finite

Similar to problems with uninformed search, we can see that if the above 2 conditions do not hold, it is possible for the algorithm to be stuck within a contour and never terminate.

2.2.3 Time and Space Complexity of A* Search

While A* Search is optimally efficient, it still incurs time complexity exponential in the base b, to the power of the error incurred in the estimation of the heuristic function resulting in $O(b^{h^*(n)-h(n)})$, unless the error grows no faster than the logarithm of the actual path cost(proof omitted):

$$|h^*(n) - h(n)| \leq O(\log h^*(n))$$

In particular, we wish to use dominant heuristic functions over a set of possible $h(n)$

Definition 3. *For 2 heuristic functions $h_1(n), h_2(n)$ that are admissible, if $\forall n, h_1(n) \geq h_2(n)$, then $h_1(n)$* ***dominates*** *$h_2(n)$*

This also means that $h_1(n)$ incurs lower search costs, because since both functions are admissible, values for $h_1(n)$ are closer to the true cost function $h^*(n)$ than $h_2(n)$ is.

This means that A* is greatly affected by the error incurred by the heuristic function estimation. However, computational time is not the biggest drawback. A* incurs a space complexity of $O(b^m)$ by maintaining nodes generated in the frontier in memory. More often than not, A* Search will run out of space before it finishes running.

2.3 Deriving Admissible Heuristics

Often we have to invent our own heuristic functions in order to tackle a search problem. One of the ways that we can do this is by **relaxing** the constraints to the problem. This means removing some of the restrictions placed on the problem itself, leading to special cases of the problem that can be solved using specific derivable heuristics. It is often that *optimal solutions to a relaxed problem is often a good heuristic to solving the original*. By relaxing a problem, we can develop a set of admissible heuristic functions $\{h_1, h_2, ..., h_n\}$ that provides optimal or near optimal solutions to specific configurations to the search problem, which can be used to evaluate different scenarios.

With a set of specific heuristic functions, it follows that each of these heuristics that developed to solve a specific scenario, and not a general best heuristic for the best solution itself. If we pick one heuristic over another, it could be that the heuristic does optimally in one scenario, but very bad in another. Hence instead of picking, we can make use of the dominant property to pick heuristic functions at the current state:

$$h(n) = \max(h_1, h_2, ..., h_n)$$

Which means that always have the best option of a heuristic at any state in the search. Using the dominant property, as well as the fact that the heuristics are admissible, guarantees the best choice of heuristics among all we have by picking the heuristic closest to the upper bound of the true cost.

Other methods of derivation of heuristics involve statistical learning and feature evaluation as well, which allows the algorithm to derive importance of various features of heuristics in different situations by itself.

3 Local Search

In some problems, the state description contains all the information about the problem that we need to make informed searches, and the path in which we take during our search may not be so useful anymore. We only require that the goal state be reached. In this case, local search, or **iterative improvement** algorithms may provide a practical approach that does not require searching over search paths. More often than not, it is often a good idea to *start with a complete configuration of the problem, and make gradual improvements to the solution.* We can easily see that such algorithms search a smaller portion of the search space while still being able to provide reasonable solutions to the problem. Local search algorithms do not look far into a search path, and usually only considers neighbours within close proximity.

3.1 Hill-Climbing Search

The hill-climbing algorithm is one that continually moves the current position in the direction of an increasing objective value. The algorithm always aims to make an improvement through a movement to it's surrounding neighbours if possible. Variants of this algorithm in other problems such as convex optimization may also be known as gradient descent.

Algorithm 3 Hill-Climbing Search

```
1: current ← problem.initial-state
2: loop
3:     neighbour ← highest-valued successor of current
4:     if neighbour.Value ≤ current.Value then
5:         return current.State
6:     else
7:         current ← neighbour
8:     end if
9: end loop
```

Note that unlike the previous algorithms that we have seen, the Hill-Climbing search algorithm does not maintain any form of search tree or graph, but only the current state, it's neighbour states and their evaluations. This cuts down the need for large memory requirements immediately.

However, the Hill-Climbing search paradigm does suffer from certain drawbacks:

- **Local Maxima:** The algorithm may run into a point where the current state is a peak among all other neighbours, and returns this state. That is, the algorithm is guaranteed to return a *maxima*, but it could be a local maxima and be maximal with respect to the neighbours, but not the entire state space and be a *global maxima*. The algorithm does not know this and only recognises the local maxima as the highest value, since it does not look past the neighbours of the current state. In many cases, the local maxima returned by the algorithm may be far from satisfactory compared to the global maxima.
- **Plateau:** This is an area of the state space where the value does not change from current to neighbouring states. In this case, the algorithm does not know for sure which move to take, and either uses heuristics to decide or makes a random walk.

The drawbacks that Hill-Climbing search has points to the fact that once the algorithm is unable to make any further improvement, it returns the current best solution without checking other

peaks in the state space, and most of the time the current highest returned is a local maxima and not the optimal solution to the problem. That is, Hill-Climbing fully uses exploitation, but not exploration in it's search. We can combat this in various ways depending on the problem, but the most commonly known is **Random-Restart Hill-Climbing** by restarting the algorithm initialized to different starting states in a bout to obtain a higher maxima.

4 Adversarial Search

Unlike previous search problems that were defined before, we now look at problems where *more than one player* is involved. For example, chess can be modelled as a search problem playing against another player and trying to make the best moves at each point in time. These problems are also called *games*, and model real world problems much more accurately than previous problems.

Unfortunately, the presence of additional agents also introduce multiple factors that increase the intractability of solving the problem:

- **Uncertainty** of the next environment states in the game, since the states are no longer influenced just by the agent's actions, but also other agent's actions as well.
- **Larger search space** due to the increase in the total number of combinations of moves resulting from additional agents participating in the game. The agent needs to account for the possible moves that the opponent might take in chess, for example.

4.1 Two-Person Games

Let us being by considering games that involves 2 agents or players. In particular, we wish to look at **zero-sum games**, which are games where the gain or loss by a participant is exactly balanced and reflected in the other player's utility. For example, in a gambling problem involving a banker and the player, if the banker has a utility of x, that means that he/she has won x dollars and consequently the player has lost x dollars and has a utility of $-x$. Additionally, we reformulate the following:

- **Initial state** includes the starting board position, and who goes first
- **Set of operators** determines the legal moves that the current play can make
- **Terminal test** defines whether or not the game is over depending on win conditions, or if no more legal moves can be made
- **Utility/Payoff Function** returns a numerical value on the outcome of the game. In the game of chess, a win for the MAX player would give +1, a win for MIN player would give -1 and a draw would be 0

We also see that for zero-sum games, we define a MAX and MIN player, that aims to maximise and minimize the utility function respectively. We can see that the 2 agents now have objectives that contradicts each other, and makes the game harder for the opponent. More importantly, any gain or loss for the MAX player is a loss or gain for the MIN player respectively, and we can see that the total utility *sums to 0*, which reflects the definition of a zero sum game.

4.1.1 Minimax Algorithm

Let us consider the search tree for formulating a 2 player zero-sum game. Note that the plays alternate between the MAX and the MIN player, and hence each depth of the search tree alternates between player MAX and MIN. The minimax algorithm uses this search tree to formulate the optimal moves for the MAX player that starts the game first, by deciding the best optimal starting move.

1. The algorithm first generates the entire search tree for the game, all the way to the terminate states that are represented by the leaves in the search tree. Each depth of the tree alternates between MAX and MIN's moves, with the root of the tree representing the starting move by MAX

2. From the terminal states at the leaves, we apply the utility function to each leaf to determine the the utility value. This represents a +1 for a win for MAX, and -1 for a win by MIN for a game of chess for example.

3. We recursively move one level up the tree everytime the utility values of the current depth is determined for all nodes, and use the utility value for the current depth to decide the utility value for the nodes one level up. In particular, let S_i be the set of children residing in depth i of node v_{i+1}, and f the utility function used in the game. If level $i+1$ denotes a play by MAX, then

   $$f(v_{i+1}) = \max_{u \in S_i} f(u) \text{ if player} = \text{MAX}$$

   That is, at level $i+1$, MAX will pick the move out of all possible children that will lead it to the terminal state that yields the highest possible utility. Similarly for MIN, if the current level corresponds to a play by MIN, then

   $$f(v_{i+1}) = \min_{u \in S_i} f(u) \text{ if player} = \text{MIN}$$

   instead to minimize the overall utility.

   In particular, we see that everytime the levels alternate, the utility switches from maximizing to minimizing based on the MAX and MIN player. This is because the 2 objectives of the players are clashing, and everytime MAX makes a move that maximizes the utility, in the next half-move by MIN player, he/she will counter MAX's progress by choosing a path that minimizes the overall utility. As such, MAX and MIN never really often attain the maximal/minimal utility, but rather, the *maximum minimal utility* for MAX, and vice versa for MIN. Hence the name, Minimax algorithm.

In particular, at each level of the search tree, the move taken by the player to maximize/minimize the utility is known as a **strategy**.

Definition 4. *A strategy s_1 for player 1 is called **winning** if for any strategy s_2 by player 2, the game ends with player 1 as the winner.*
*A strategy s_1 for player 1 is called **non-losing** if for any strategy s_2 by player 2, the game ends in either a tie or a win for player 1.*

Another important observation is that at each level, the player that makes the play at that depth plays the best that they can to possibly maximize/minimize the utility, and we cannot change the search path choice to improve the utility. The minimax algorithm is also known to output a **sub-perfect Nash equilibrium**. While the choice over the entire tree may not be optimal, but at each play, the player makes the best choice they possibly can.

- **Completeness:** If the maximum depth of the tree m is finite, and the number of legal moves(which translates to the number of branches b) is finite, then the algorithm is able to fully generate the tree and is complete.
- **Optimality:** Similar to completeness, since m and b are finite, the entire search tree with it's respective utility costs is visible to the algorithm. That means that the algorithm has perfect information to the utility cost of each move, and is able to make the best move that maximizes/minimizes it's utility. Hence, it would be able to output the optimal solution.
- **Time Complexity:** For a search tree of branching factor b and depth m, in the worst case it needs to generate a tree with b^m nodes, and hence takes $O(b^m)$ time to do so
- **Space Complexity:** Because of the recursive nature of generating the nodes, Minimax algorithm maintains nodes in a search path by diving into the terminal states before going back up, using at most $O(bm)$ space.

4.2 Alpha-Beta Pruning

We have seen that for the minimax algorithm to output the optimal solution, it needs to generate the entire search tree which hinges on it being finite. However, even for finite trees, it can be seen that sometimes search trees can be very large, and it may take a long time for the algorithm to perform search.

Fortunately, it is possible for Minimax to be optimized to search and output the optimal solution despite not looking at every single node in the search tree, using the intuition that *we do not explore nodes that are already bad*. Consider a node n along the search tree. If the player has a better choice at node m which is a parent of, or higher up the search tree from n that improves his/her utility, then node n will never be reached. Hence, it is not necessary to explore the paths further down from n.

Since Minimax is depth first, we only need to consider nodes along a single path in the tree. We denote $\alpha(n)$ as the highest observed value found on any path from n initialized at $-\infty$, and $\beta(n)$ as the lowest observed value initialized at ∞. Intuitively, this can be seen as α is the maximal possible utility for MAX, and β the minimal for MIN, and any further values of β that do not improve the value of the current α will be pruned. In particular,

- given a node n at a level corresponding to a MIN play, we stop searching below n if there is a MAX ancestor i of n such that $\alpha(i) > \beta(n)$
- given a node n at a level corresponding to a MAX play, we stop searching below n if there is a MIN ancestor i of n such that $\beta(i) < \alpha(n)$

Note that alpha-beta pruning prunes aways branches of the search tree does will never get reached, and hence these branches do not influence the final decision. But how much computational effort does alpha-beta pruning actually save? This depends on the ordering of the moves that

are legal, which determines the search path ordering. With perfect ordering, alpha-beta pruning can give a time complexity on Minimax of $O(b^{\frac{m}{2}})$. This means that the algorithm can actually search twice as deep with the same time constraints. However, even without a good ordering, on average alpha-beta pruning gives a time complexity of $O(b^{\frac{3}{4}})$ in a random ordering.

4.3 Evaluation Functions

When search gets large, we also have the option of limiting search, just as in DLS. However, for cases like Minimax algorithm, we run into several major problems, the first being that since depth is limited, we may not reach the terminal states for certain search paths. Because the utility of intermediate noes rely on the terminal states to determine the actual utility, that means that we are unable to determine intermediate nodes of search paths that do not reach the terminal state.

We rectify this by introducing an **evaluation function**, which is an approximation function that evaluate how good an intermediate state is. We require that the evaluation function satisfy the following conditions:

- The evaluation function must agree with the utility function on terminal states
- The evaluation function cannot be too expensive, in particular it cannot be more expensive than the utility function as an approximation.
- The evaluation function must be accurate in reflecting the player's odds of winning

Additionally, we also need to ensure that we do not run into the **horizon problem** when we perform the search limited. If we cut off search at a state that is volatile, and is threatened by possible wild swings or moves that may turn the game around, then it could be possible that the winning odds reflected at the state before and after the cutoff reflect very different information. Hence, we also include the condition that the search be performed til a **quiescent** position before being cut off, which is one that is robust to large changes.

4.4 Expectimax Algorithm

In some games that involve stochastic layers, we also need to incorporate randomness into the game itself such as Poker. In addition to our standard minimax algorithm search tree, we also need to insert 'chance' nodes that represent as random choice between state changes. For each of the children of chance node i, x_j, we represent the probability that the search goes down to node x_j is denoted $p(x_j)$. At the terminal nodes, we continue to use our utility function to obtain the utility value. However when we propagate upwards to the chance nodes, then we are unsure of exactly which path we will take. Instead, we take the *expected* utility value upon reaching that node, which is the combination of the utility values among all children weighted by the probabilities of hitting that node.

$$f(x_i) = E[f(x_i)] = \sum_{j \in J} p(x_j) f(x_j)$$

By introducing chance nodes and randomness into our search algorithm, we have increases the time complexity from $O(b^m)$ to that of $O(b^m n^m)$ because of the fact that we could have up to n chance nodes at each of the m levels.

5 Constraint Satisfaction Problems

In some problems, in addition to the basic requirements of obtaining the optimal solution or reaching the goal state, we also need to satisfy additional requirements or properties for the solution to be feasible. These properties are called **constraints**, and the problem can be modelled as what is called a **Constraint Satisfaction Problem(CSP)**. In CSPs, the states are represented in the form of variables, and constraints that the goal state must obey. For example:

$$\begin{aligned} &\min c^T x \\ &s.t\ Ax = b \\ &x \geq 0 \end{aligned}$$

is commonly seen as the standard form of the Linear Programming problem.

Each variable x_i has a domain D_i that it takes it's values from, such as $D_i = \mathbb{R}$ for real numbers, or $D_i = \{0, 1\}$ for binary values.This domain can be discrete or continuous, but CSP under continuous domains falls under the previously mentioned Linear Programming problem, and for this section we focus on discrete values. Additionally, constraints can be the following:

- **Unary Constraints** involving one variable: $x_i = a$
- **Binary Constraints** involving 2 variables: $x_i + x_j \leq a$
- **Higher Order Constraints** involving multiple variables

Note that in this case, we are using integer operations on constraints. This is defined by our **Constraint Language**, which represents *how* our constraints should be formulated.

Converting our CSPs back to the form of search state spaces, we can denote our initial state as an empty assignment; that is no values are assigned to any variables, and the transition functions being the assigning of values to variables. Lastly, our goal state is an assignment for the variables such that the solution is feasible. In particular, we say that the assignment is *complete* if the assignment does not violate any constraints in the CSP, and $\forall i, y_i \in D_i$.

A standard search paradigm for working with CSPs is as follows:

Algorithm 4 Standard CSP Search

```
1: assignment ← []
2: while assignment is not complete do
3:     assignment ← assignment ∪ y_i where x_i = y_i, y_i ∈ D_i
4:
5:     if assignment is complete then return assignment
6:     end if
7: end while
```

Note that there are a few quirks of simply following the above routine:

- **Repeated States** but different order of assignments. This means that in term of a search tree, then we would have many unnecessary nodes that represent the same state(e.g assigned $x_i = a$ then $y_i = b$ in one path and the reverse order in another path. Notice order is not required. We can merely represent each level of the search tree as the assignments

for a *singlevariable*. Then, for d dimensional variable and m variables, the search tree is at most $O(d^m)$. This is also known as **Backtracking Search**, where we assign a variable per level.

- **Running into already invalid assignment** but still continuing to search. In many cases, we can already determine if progress from the current state will lead to a valid assignment. If the current state can no longer reach a valid assignment, we can simply stop the search down that sub tree and effectively prune search in that subtree.

5.1 Backtracking Search

Algorithm 5 Backtracking Search

```
1: assignment ← []
2: while assignment is not complete do
3:
4:     if assignment is complete then return assignment
5:     end if
6:     var ← Select-Unassigned-Variable(csp)
7:     for each value in Order-Domain-Values(var, assignment, csp) do
8:         if y_i is consistent with assignment then
9:             add {x_i = y_i} to assignment
10:            inferences ← Inference(csp, var, value)
11:            if inferences ≠ failure then
12:                add inferences to assignment
13:                results ← Backtracking Search(assignment, csp)
14:                if result ≠ failure then return result
15:                end if
16:            end if
17:        end if
18:    end for
19: end while
```

Notice that

- **Order of variable assignment is irrelevant**, similar to how we do not bother about the search path and concentrate on finding a valid assignment
- **At every level, consider a single variable**. This considerably reduces our search space for assignments
- **Use of inferences**, which tells us what we can say about the remaining values given the current assigned values. That is, the currently assigned values determine which of the remaining values are valid or not.

In particular, we can improve efficiency and save runtime by optimizing the subroutines: **Select-Unassigned Variables, Order-Domain-Values and Inferences**. Basically, *how do we choose what to assign each variable at each point in time.* The following as heuristics we can use:

- **Minimum Remaining Values(MRV):** Pick the variable with the *fewest* legal values left. This allows us to detect invalid assignments quicker before searching too deep and wasting search time.

- **Degree(Most Constraining Variable) Heuristic:** Pick the variable that has the most relations with other variables. That is, it *influences* other variables the most. Similar to MRV, selecting this variable first allows us to quick eliminate assignments to other variables made invalid because of this variable quickly.

- **Least Constraining Value** for value selection to be assigned to the variable. We want values assigned to variables to be permissive, so that we are more likely to find a valid assignment and not waste time searching only to reach an invalid solution.

5.2 Inferences

5.2.1 Forward Checking

In forward checking, the main idea is to keep track of remaining illegal values for unassigned variables. At any point in the time in the search, if any of the variables has no remaining legal values left, then it ay progress cannot output a valid assignment since one variable cannot be satisfied. Hence, we terminate search down the current subtree and return back to a state where a valid assignment is still possible.

5.2.2 Constraint Propagation

Using forward checking, we are able to terminate search down a subtree that is guaranteed to not have a valid assignment, and hence prunes portions of the search tree to reduce search space. But we can improve this further. Note that forward checking does not detect early signs of invalid assignments. This is where **constraint propagation** comes in. Using constraint propagation, we are able to observe if values are illegal by reducing and narrowing down the search space by removing assignments that have illegal values. In particular, constraint propagation enforces constraints **locally** for certain variables in order to continue search.

5.2.3 Arc-Consistency

Definition 5. *X_i is **arc-consistent** with X_j iff for every value $x_i \in D_i$, there exists some value $y_j \in D_j$ such that the binary constraint on the arc (X_i, X_j) is satisfied*

That is, an arc (X_i, X_j) is arc-consistent if there is a valid assignment for X_j for every value that we assign to X_i. As we perform search down the tree, it is likely that most of the variables in the state that we are in or will be in, will not be arc-consistent with every other variable. If that is the case, then most likely whatever value we pick for the variable X_i, the leading assignment will violate some constraint since it is not arc-consistent. However, we can **enforce** arc-consistency by removing illegal values from the domain D_i such that the resulting domain $D_i^{'}$ is arc-consistent with every other variable. Then, we have effectively reduced the domain D_i for X_i to one that contains legal values that do not lead to illegal assignments. This is done through:

- **Enforcing unary constraints** on the variable X_i. That is, we prune values from D_i that do not enforce unary constraints $X_i \geq a$ for example.

- **Propagating arc-consistency on binary constraints**. By enforcing arc-consistency on the arc (X_i, X_j), then the arc-consistency on the neighbouring arc (X_k, X_i) can be enforced on D_k on the reduced domain $D_i^{'}$. This could not have been possible without enforcing arc-consistency on (X_i, X_j) since we could have enforced arc-consistency on (X_k, X_i) with respect to illegal values in D_i. This represents transitive arc-consistency relations.

Hence, arc-consistency acts as a form of local repeated constraint propagation by enforcing it after every assignment. This way, we prune as much as of the domain space as possible by removing values that will possibly lead to illegal assignments.

Algorithm 6 Arc-Consistency Algorithm AC-3

```
1: procedure AC-3(csp) returns false if inconsistency found and true otherwise
2:     queue ← all arcs in csp
3:     while queue is not empty do (X_i, X_j) ← Remove-First(queue)
4:         if Revise(csp, X_i, X_j) then
5:             if size of D_i = 0 then return false
6:             end if
7:             for each X_k ∈ X_i.neighbours - {X_j} do
8:                 add (X_k, X_i) to queue
9:             end for
10:        end if
11:    end while
12: end procedure
13:
14: procedure REVISE(csp) returns true if the domain D_i of X_i is revised
15:    revised ← false
16:    for each x ∈ D_i do
17:        if no value y_j ∈ D_j allows (x_i, y_j) to satisfy the constraint between X_i and X_j then
18:            delete x from D_i
19:            revised ← true
20:        end if
21:    end for
22: end procedure
```

The AC-3 algorithm is one that allows us to ensure arc-consistency via constraint propagation. It first begins with arcs from the CSP problem, and prunes values from each domain D_i for the variable X_i if there are any values such that X_i is not arc consistent with any X_j. Once we do this, we re-enforce transitive arc-consistency by re-evaluating the values in the domains of the other neighbours of X_i, X_k such that once (X_i, X_j) is arc-consistent after the Revise procedure, each of the other neighbours of X_i, X_k needs to be arc-consistent with the revised domain of X_i, D_i'.

Notice that for a CSP problem, there are at most n^2 directed arcs in a complete graph where very node has a directed edge to other nodes. Additionally, each X_i has at most d values in their respective domains D_i, and hence on revising of neighbouring arcs, an arc (X_i, X_j) can be re-queued for the Revise operation at most d times(exactly d if everytime we only delete 1 element from the d elements in D_i). Each revise operation compares each cartesian product of the domains D_i, D_j and hence incurs at most d^2 comparisons. Putting all of these together, we have that the AC-3 algorithm incurs a total time complexity of $O(n^2d^3)$.

Hence, we can use the AC-3 algorithm as a routine on our CSP to ensure that all values in each domain is arc-consistent since it enforces transitive constraint propagation. In particular, we can re-execute this routine after every assignment to maintain the arc-consistency, and this entire process is known as **Maintaining Arc-Consistency** procedure. This ensures that after every assignment, we revise our domains to contains values that only lead to legal assignments.

Notice that the AC-3 algorithm is used to maintain transitivity of arc-consistency on *binary constraints*. In the case of ternary constraints where more variables are involved, we need to decompose the ternary constraints into multiple binary constraints if we wish to continue using the AC-3 algorithm. However, it is not always that ternary constraints can be decomposed into binary constraints, especially when constraints start to get complicated. Fortunately, we can extend the AC-3 algorithm from enforcing 2-consistency between pair of arcs (X_i, X_j) to a general algorithm that establishes k-consistency on ternary constraints.

5.3 Local Search on CSP

Previously, we have discussed how to perform standard search over the state space for CSPs, by starting with an empty assignment, and assigning a value to each variable at each level of the search tree. We then improved this to ensure we only assign legal values by performing forward-checking, in particular checking arc-consistency between variables to reduce search space to only valid assignments instead of searching down paths that lead to invalid assignments.

However, we can also consider local search techniques for solving CSPs, by first starting with an initial assignment(which can obviously be allowed to violate constraints, or else we are done), and perform iterative improvement by applying modification operators to move the current configuration towards a valid assignment. Often, these modification operators aims to reduce the number of inconsistencies in the constraints, and hence called **heuristic repair** methods. One commonly used heuristic that measures such inconsistencies in constraints is called the $min - conflicts$, which measures the $number of constraints that are violated$. Hence, by seeking a solution that minimizes this heuristic, we are aiming to find a solution that satisfies the maximal number of constraints possible.

5.3.1 Min-Conflicts

Algorithm 7 Min-Conflicts

1: $current \leftarrow$ assignment for csp
2: **for** $i = 1$ to max_step **do**
3: **if** $current$ is solution for csp **then return** $current$
4: **end if**
5: $var \leftarrow$ randomly chosen variable from csp.Variables
6: $value \leftarrow$ value v forvar that minimizes $conflicts(var, v, current, csp)$
7: $current.var \leftarrow value$
8: **end for**

Notice that we execute the Hill-Climbing search with min-conflicts as the heuristic function that we aim to minimize. Similar to Hill-Climbing search, we may face the issue of being stuck in a local minima that is not a global minima if the algorithm no longer finds any moves that can improve the current position, since it merely aims to reduce the heuristic function that leads to a better position now, but not in the long run and never explores nor backtracks. But surprisingly, in many cases, we are able to find the optimal solution with Hill-Climbing search with high probability.

Additionally, we can add element of exploration and exploitation by labelling constraints that are important, or have not been 'repaired' for a long time. We do this by assigning weights to constraints, which value can fluctuate over the period of the search. At each state of the search space, we aim to find an action that leads to a configuration that reduces the $total weight$ of all

constraint violated. Depending on how we define the value adjustments, this allows us to get out of local minima by marking constraints that are stagnant for long time, or label constraints that absolutely need to be satisfied among all constraints.

An important feature of local search is that it allows flexibility in time, by providing us with the best known solution at the time of termination, even if the search is not complete. This is particularly useful for extremely large search spaces. Additionally, it allows for existing solutions to be fixed due to change in constraints by using the current previously valid assignment, and running local search on it with new constraints.

5.4 Structured CSP

In general, solving CSPs is considered a computationally intractable problem. For CSPs with discrete valued variables, the search space is in the factor of $O(d^n)$ for all n variables. To reduce this computation time, we can break CSPs into independent CSPs. For reduced CSPs with c variables each, the computation time is reduced to $O(d^n) = O(d^c \frac{n}{c})$.

However, if we can find a substructure in the CSP, then we can exploit it to solve the CSP in polynomial time.

Theorem 2. *If the CSP constraint graph is a tree, then an assignment to the CSP can be found in polynomial time in factor of $O(nd^2)$ time*

To solve a tree-structured CSP, we first need to impose a hierarchical structure to the constraint graph. We do this by choosing an ordering of the graph that imposes **topological sorting** on the graph itself. Once that is performed, we can ensure arc-consistency of each arc (X_i, X_j) for each in the tree, from the leaf node X_j up to the parents. Since a tree with n nodes contains at most $n-1$ arcs, that means that the total time incurred for ensuring arc-consistency on all arcs is $O((n-1) \times d \times d) = O(nd^2)$.

Lemma 2. *If all arcs $(X_i, X_j), \forall$ i, j s.t $X_j \in$ children of X_i are consistent, then the following is true:*

- *There is legal assignment for CSP if $D_i \neq \emptyset$ for each parent X_i*
- *There is no legal assignment for CSP if $\exists D_i : D_i = \emptyset$*

The second part of the claim is easy to see: If one of the domains D_i is reduced to an empty set using arc-consistency, that means that for every variable in the domain D_i, it violates a constraint between X_i and at least one child X_j. This means that there is no legal assignment for X_i, and hence for the CSP itself.

If every domain is non-empty, the algorithm ensures that every arc between a parent and child node (X_i, X_j) is consistent. This means that for every value in the domain D_i for X_i, there is legal assignment for each child X_j. Notice that although the algorithm does not ensure that the reverse is true: the arc (X_j, X_i) is arc-consistent, we argue that the resulting reduced domains is valid. This is because while we ensure arc-consistency from the bottom of the tree up to the root, we assign values to variables by propagating down the tree. This means that we pick values from parent nodes *before* the child nodes, we only have to ensure that the values we pick for the parent nodes X_i induces valid values for the child node X_j that is selected after the parent node, which is exactly what arc-consistency on (X_i, X_j) ensures.

This also means that there is no need for backtracking when we assign values, since arc-consistency ensures that whatever values that we select for the parent nodes ensures that there is at least one valid value to be assigned to the child. Additionally, since the graph is a tree, each node only has at most one parent. Hence, the only values that can affect the selection of values for the current variable is the value of the parent node selected before. But because we ensured that each arc from parent to child node is arc-consistent, we know that we always have a valid value(is the domain is non-empty) for the current node no matter what the parent node is assigned.

Lemma 3. *Ensuring arc-consistency in a constraint graph that is a tree does not remove options for a valid CSP assignment*

Notice that by ensuring arc-consistency, we only remove values from a domain D_i if $\exists X_j : X_i = d \rightarrow D_j = \emptyset$. That is, the value d causes a child X_j to have only assignments that violate constraints. For a valid assignment for a CSP, every value d_i for each node X_i is valid. Assume that a value d_j^* for the variable X_j is removed by the arc-consistency algorithm. That means that $\exists X_i$ such that all $d_i \in D_i$ are invalid assignments if d_j is chosen. But that also means that the value $d_i^* \in D_i$ from the valid assignment is also removed by the arc-consistency algorithm, implying that if d_j^* is chosen, d_i^* will be an invalid assignment for X_i that violates some constraint. But this contradicts the fact that d_i^*, d_j^* is in the valid assignment since having both violates at least 1 constraint in the CSP. Hence, options for a valid assignment for the CSP will never be removed because for each arc (X_i, X_j), the value for the valid assignment d_i^* will at least have the valid value in the assignment d_j^* chosen for X_j, since both in the assignment implies they do not violate any constraints.

This brings us to our next lemma:

Lemma 4. *If a valid assignment is possible for the CSP, it will always be found regardless of the topological ordering induced on the constraint graph*

Observe that the valid assignment for the CSP is independent of the topological sorting of the constraint graph nodes. If we switch the topological ordering to form a new tree, the valid assignment values for CSP is still valid because picking each of the values $_i$ for variable X_i in this valid assignment implies that for each of the new children X_j of the reformed tree, we can minimally pick the valid values d_j from the valid assignment since these combination of values do not yield violating constraints. By the lemma before, this means that this valid assignment is not removed from by the arc-consistency algorithm, and hence can still be found if the topological ordering is different. Although, it is possible that a different valid assignment may be found instead.

6 Reinforcement Learning

Previously, we have discussed games in search trees using utility functions to determine which moves are the most optimal, and then propagating them back up using the Minimax and alpha-beta pruning algorithm. However, this requires that we be able to define the labelling for these outcomes of games. In some games, this may not be possible due to the sheer number of possible winning outcomes, or just that we do not perfectly know what is the best or most optimal outcome. In this case, it is difficult for the algorithm to search optimally without feedback from us. Hence, *the agent needs to be able to function without perfect feedback*. Instead of telling the algorithm explicitly which path is most optimal, we instead incentivize algorithms to take better paths by providing **rewards or reinforcements** to it, which is a signal to the algorithm that a particular action is favored over others.

We also impose the property of **Markov Decision Process** in reinforcement learning, where *the next state in the game is determined by the current state and actions, and not the entire sequence.*

The **policy** π that the agent choose determine the agent's behaviour and what actions it take at each state. The policies can be:

- **Deterministic:** $a_t = \pi(s_t)$. The agent chooses the best valued action at the current state
- **Stochastic:** $\pi(a|s_t) = \Pr[a_t = a|s_t]$. The policy is a distribution of actions that can be taken by the agent at the state in time.

In order to determine what actions to take, the agent need to evaluate if the action, and possibly the sequence of actions taken after it will yield optimal rewards. Consider the reward function $r_t(a_t, s_t)$ at the time t that infers the rewards received by the algorithm when taking the action a_t at state s_t. Then, we see the value function as:

$$V^\pi(s_t) = r_t(a_t, s_t) + \gamma r_{t+1}(a_{t+1}, s_{t+1}) + ... = \sum_{l=0}^{\infty} \gamma^l r_{t+l}(a_{t+1}, s_{t+1})$$

That represents the rewards that will be received by the algorithm by taking a sequence of action from the current state on subsequent resulting states, where $0 \leq \gamma \leq 1$ is known as the discount value. This discount value can be thought of as the *importance* of the rewards received upon reaching the state s_t. A high value of γ(e.g $= 1$) would mean that the agent is long-sighted: the rewards received in the next state is almost as, or equally important as the the rewards received in future states. However, notice if the value of γ is low, the value of γ^l decays much quicker. This means the agent is more concerned with maximizing immediate rewards rather than future rewards.

We wish to choose a value maximizing policy based on the above value function. Notice that we are missing information to do so, which are rewards of unobserved states, which are usually non-terminal states. Hence, we have to use rewards inferred from current and transitioned states to learn the expected utility that is associated with each non-terminal state:

$$U^\pi(s_t) = \mathbb{E}[V^\pi(s_t)] = \mathbb{E}[\sum_{l=0}^{\infty} \gamma^l r_{t+l}(a_{t+1}, s_{t+1})]$$

Note that the value function *has nothing to do with the agent's internal state of the environment.* It is fully objective, and aims to represent the rewards that the agent receives if it takes an action that transitions it to another state.

6.1 Markov Decision Processes

To define a Markov Decision Process(MDP), we first start with a fully-observable environment, such as the Tic-Tac-Toe game. In such cases, the agent is able to observe the entire environment during each of it's states, and also how it's actions taken affect the environment. In this case, as previously discussed, the agent would be able to construct an optimal strategy that reaches the goal state.

However, now let us look at the possibility that the *environment interferes with the agent's actions*. That is, every action the agent performs happens with a probability of $p \leq 1$, and with a total probability of $1 - p$ the environment could cause the agent to not be able to carry out it's intended action. In this case, the **transition model** is no longer deterministic that specifies the action taken at state $s, T(s, a, s')$, but rather in terms of probability:

$$P(s'|a, s)$$

which denotes the probability that the agent indeed reaches the state s' from s by taking the action a, should the environment not interfere successfully. Now our transition model becomes stochastic. Additionally, we assume that the transition model is **Markovian**: the odds of the agent transitioning to a state s' from the state s by the action a is affected *only by where it's current state s is*. The states that the agent were in in the past, $s_1, ..., s_{n-1}$ do not affect the probabilities of transition from it's current state to the next state. More formally:

$$P(s'|a, s_n, s_{n-1}, ..., s_1) = P(s'|a, s_n)$$

Lastly, the utility function for the agent is now dependent on the **reward function** $R(s)$ at state s. Unlike previous strategies, the reward function is now cumulative: it depends on the sequence of actions that the agent has taken. For example, the total utility that the agent receives using a policy π could just be the sum of the rewards at each state it has visited. A sequential decision problem for a fully observable, stochastic environment with a Markovian transition model and additive rewards is called a **Markov Decision Process(MDP)**.

6.1.1 Optimal Policy

Since we no longer have a deterministic transition model, a fixed action sequence that is used for a deterministic version of the search may or may not yield the best results. Hence instead of employing the strategy that we discussed in zero-sum games, we now make use of a **policy** π, which is a function that tells the agent the action that it *should* take at state s. Hence, the policy function at a state $s, \pi(s)$, explicitly determines the agent function.

Notice that since the environment is now stochastic in nature, running the agent with a fixed policy may return multiple different environment histories. Hence, the quality of a policy is no longer determined by the utility of a single environment history, but rather the **expected** utility from a series of environment histories returned by running the same policy on the environment several times. The optimal policy, is one that yields the **highest expected utility** out of all policies.

Additionally, it is also important to note that the reward function $R(s)$ influences the policy greatly, and how conservative it is. If rewards for reaching the goal state is low and penalty elsewhere is high, then the agent will tend to be more conservative.

6.1.2 Learning Optimal Policies

We have seen that the optimal policy is one that gives the highest expected returns for the agent at any state in the environment. However, we can only pick an optimal policy for the agent if we actually know the true probability distribution for the MDP, $\Pr(s'|a,s)$. Usually, in many real world problems, such a distribution is unknown to us, and hence we required the agent to *learn* this optimal strategy.

Remember that in Reinforcement Learning, we aim to maximize the utility that the agent gets through it's value function:

$$V^{\pi}(s_t) = r_t(a_t, s_t) + \gamma r_{t+1}(a_{t+1}, s_{t+1}) + ... = \sum_{l=0}^{\infty} \gamma^l r_{t+l}(a_{t+1}, s_{t+1})$$

where γ is the discount factor that determines how much the agent values long term rewards. If $\gamma = 1$, then

$$V^{\pi}(s_t) = r_t(a_t, s_t) + r_{t+1}(a_{t+1}, s_{t+1}) + ... = \sum_{l=0}^{\infty} r_{t+l}(a_{t+1}, s_{t+1})$$

are known as additive rewards. In this case, rewards in the future are as valuable as the immediate rewards. However, due to the possibly stochastic nature of environments, we may not always know what rewards that we will get by taking an action a. Instead, we wish to compute the expected utility:

$$U^{\pi}(s_t) = \mathbb{E}[V^{\pi}(s_t)] = \mathbb{E}[\sum_{l=0}^{\infty} \gamma^l r_{t+l}(a_{t+1}, s_{t+1})]$$

where s_t is the state that the agent is in at time t, following the actions that policy π recommends. Observe that if the policy is deterministic, then clearly $U^{\pi}(s_t) = V^{\pi}(s_t)$. Then, computing the optimal policy π^* is given as follows:

$$\pi^* = \arg\max_{\pi} U^{\pi}(s_0)$$

Where the optimal policy aims to provide the the best action starting at the current state s_0 that maximises **long term** expected rewards, as opposed to $r(a,s)$ that indicates short term rewards. Note also that as the discount factor γ goes to 0, $U(s)$ goes closer to $r(a,s)$. Additionally, because of the Markov property, when we start at a state, say s_0, and move onto the state s_1, the agent no longer needs to consider the previous state s_0. Hence, the utility function $U(s)$ allows the agent to select the best action using the principle of best expected utility at state s:

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

6.2 Bellman Equation

We have seen that the utility of a state is the expected sum of discounted rewards from that state onwards. From this, we can break the equation down further using the following reasoning: *the utility of a state is the immediate reward obtained from that state, followed by the expected discounted reward from the next state.*

$$
\begin{aligned}
U^{\pi}(s_t) &= \mathbb{E}[\sum_{l=0}^{\infty} \gamma^l r_{t+l}(a_{t+1}, s_{t+1})] \\
&= \mathbb{E}[r_t(a_t, s_t) + \gamma r_{t+1}(a_{t+1}, s_{t+1}) + \gamma^2 r_{t+2}(a_{t+2}, s_{t+2}) + ...] \\
&= \mathbb{E}[r_t(a_t, s_t) + \gamma(r_{t+1}(a_{t+1}, s_{t+1}) + \gamma r_{t+2}(a_{t+2}, s_{t+2}) + ...)] \\
&= \mathbb{E}[r_t(a_t, s_t) + \gamma \sum_{l=1}^{\infty} \gamma^l r_{t+l}(a_{t+1}, s_{t+1})] \\
&= \mathbb{E}[r_t(a_t, s_t)] + \gamma U^{\pi}(s_{t+1})
\end{aligned}
$$

The optimal policy would then be one that maximizes:

$$\pi^*(s) = \arg\max_{\pi} U^{\pi}(s) = \arg\max_{a \in A(s)} \{r(a, s) + \gamma U^{\pi}(T(a, s))\}$$

with a optimal utility of:

$$U^{\pi^*}(s_t) = r_t^*(a_t, s_t) + \gamma U^{\pi^*}(s_{t+1})$$

As in with MDP planning, we could formulate and compute the optimal policy π^* *if we knew the reward function* $r(a, s)$ *and the transition model* $T(a, s)$. However we don't, and therefore we cannot explicitly compute the optimal policy if we do not know what the rewards are if we take an action and what the next state is. Instead, we try to learn a policy that *converges* to the optimal one.

6.3 Q-Learning

Previously we have seen that computing and even learning the utility functions may not always be possible, as we require knowledge of the reward and the transition model. Instead, we learn an action-utility representation known as **Q-Learning**, denoting $Q(s, a)$ as the function that returns the value of performing action a at state s. More formally:

$$Q^{\pi}(s, a) = r(s, a) + \gamma U^{\pi}(s' = T(s, a))$$

is the utility the agent obtains by performing action a at state s(which may or may not be the one recommended by policy π), and then performing the actions recommended by policy π from state s' onwards. What we wish to learn is the optimal Q function. That is, the Q function that maximizes the possible rewards if an action a is taken. Using the Bellman equation, we have that:

$$
\begin{aligned}
Q^*(s, a) &= r(s, a) + \gamma U^*(s' = T(s, a)) \\
&= r(s, a) + \gamma \sum_{s'} \Pr(s'|s, a) \max_{a'} Q^*(s', a')
\end{aligned}
$$

$Q^*(s, a)$ represents the maximum utility that the agent can receive from state s', given that the agent has taken the action leading to s'. Observe further that the Q function and the utility function are related as follows:

$$U^*(s) = \max_{a} Q^*(s, a)$$

Because the maximum utility that the agent can receive at the next state $s', U^*(s)$, will be the maximum utility given an action a' at s' is taken. Hence, at every state we can replace the utility function $U^*(s)$ with the Q function.

Notice that the Q-Learning function does not require knowledge of the transition model $\Pr[s'|s,a]$: it gains this knowledge from learning or from observation by actually transitioning to the state by taking the action. Hence, it is a **model-free** method. Therefore, we have expressed the learning of our utility function in terms of a function that can observe these values without having to know the transition and reward functions.

6.4 Value-Iteration Algorithm

The Value-Iteration Algorithm aims to update the Q function values iteratively by observing the rewards and transitioned states as the agent performs actions.

Algorithm 8 Value-Iteration Algorithm

1: $\hat{Q}(s,a) \leftarrow 0, \forall a$
2: $s \leftarrow s_0$
3: **for** $t = 0, 1, ...$ **do**
4: $\quad a \leftarrow$ SELECT-ACTION(s)
5: $\quad r \leftarrow r(s,a)$
6: $\quad s' \leftarrow T(s,a)$
7: $\quad \hat{Q}(s,a) \leftarrow r + \gamma \max_{a'} \hat{Q}(s',a')$
8: **end for**

Let us assume that the rewards $r(s,a)$ are non-negative. Observe that in the beginning, all values of $\hat{Q}(s,a) = 0$, and $\hat{Q}(s,a) \leq Q^*(s,a)$. As we continue to observe the rewards and updates the values of $\hat{Q}(s,a)$, the value of $\hat{Q}(s,a)$ gets updated to the actual rewards received when action a is taken at state s. Because we observe and update the exact values of rewards, the value of $\hat{Q}(s,a)$ *never overestimates* $Q^*(s,a)$, and converges to $Q^*(s,a)$ over time as the actual values of the reward are observed by the agent.

6.4.1 Policy Update

Following the general algorithm, we have that the A values are constantly updated by new values as the agent observes rewards. This seems like a good idea, but notice that it completely overwrites the values of the previous $Q(s,a)$ with the new values. Complete ignoring the previous values in the computation of $Q(s,a)$ can result in large fluctuations of the Q values and lead to an underestimate of the actual utility.

Instead, we aim to stabilize the update of Q values by using a form of *learning rate*. As we observe the values of $Q(s,a)$ more and more times, we aim to incorporate the previous values seen and decrease the contribution of the new value to $Q(s,a)$:

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

where

$$\alpha_t = \frac{1}{1 + N[s,a]}$$

represents a fraction over the number of times we have visited the state. As we continue to observe more values, we wish to incorporate contributions of each value observed, and hence decrease contribution of new values to stabilize the updates.

6.4.2 SELECT-ACTION

In the Value-Iteration Algorithm, the action a_t is selected based on a routine SELECT-ACTION. However, how do we specify which action to select at time t and state s_t? We can perform the selection deterministically, such as specifying a criteria(e.g action with best expected rewards) using greedy selection, or incorporate randomization. One way to do this is the following:

$$\Pr[a|s] = \frac{e^{\epsilon \hat{Q}(s,a)}}{\sum_{a'} e^{\epsilon \hat{Q}(s,a)}}$$

where ϵ is a tunable parameter, represents the probability of being picked. Observe that we pick actions with higher expected utility value of $Q(s, a)$ with a higher probability. The sensitivity to Q values are influenced to the value of ϵ. Low values of ϵ(e.g 0) indicate that probabilities do not different too much with increasing values of $Q(s, a)$, with $\epsilon = 0$ being complete randomness. On the other hand, higher values of $Q(s, a)$ indicate biasedness towards high expected utilities, and we want the agent to be greedier.

6.5 Regret Minimization

Previously, we have mentioned that how we select which action to pick at each stage is critical for for update policy to converge to the optimal policy. What we wish to do in selecting the right action, is to pick the action that does not fall too far from the optimal action. That is, if we perform a series of actions over time T, we want that our expected rewards of picking actions a to not fall too far from the optimal utility, U^*. More formally, let

$$U^*(s) = Q^*(s, a^*) = \max_{a \in A(s)} Q^*(s, a)$$

again be the optimal utility achievable at state s by executing the optimal policy π^*. Note also again that we do not know what π^* is, else we could compute it. Then, let π be some other policy that may not be the optimal. The policy assigns the probability p_i^t to the action a_i^t at time t. Then, the expected utility of the agent at time t following the policy π is given by:

$$u_t^\pi = \mathbb{E}[r(s_t, a_t)] = \sum_i p_i^t u_i^t$$

And the expected utility of a policy π is given by:

$$U^\pi(s) = \sum^\infty u_t^\pi = \mathbb{E}[Q^\pi(s, a)]$$

which can also be represented by the expectation over the rewards of it's action-value representation in the form of the Q function we saw previously. Remember we aim to learn a policy π that maximize our cumulative value - then this policy should be one that is not too far from the optimal value U^*. More formally, we aim to reduce:

$$\frac{1}{T}(U_T^* - U_T^\pi)$$

which is the difference between the expected value that we get from a policy π, and the optimal policy. This is known as **regret**. We see that the probability distribution that we place on the SELECT-ACTION routine, influences the expected utility that the agent gets from the updated Q function. Bad action selection could lead to high underestimation of the actual optimal value that we could have received.

6.5.1 Greedy Algorithm

How we can pick an action deterministically, is to *greedily select an action that maximizes the total reward up to time t*. That is, we maintain a set of actions S_t that maximizes the total rewards from previous actions. Using a deterministic tie-breaking scheme, we pick an action from this set of actions.

On paper, this would mean that we would pick an action at time T is likely to get the highest expected rewards. However, we shall soon see that for any *deterministic* action-selection routine, we can always find a set of inputs such that it gets very bad rewards.

Theorem 3. *For any deterministic algorithm A, there is a sequence of actions for which* $U_T^* \geq \lceil \frac{(n-1)\times T}{n} \rceil$*, but* $U_T^A = 0$

Proof: If the algorithm is deterministic, and we know how the algorithm works, given a certain input, we can determine what action a_{t-1} the algorithm will pick at the next step. In that case, given an online learning setting where we can provide the input at each step, we can always construct an input where the action that the algorithm will pick will yield 0 rewards, and all other actions will yield 1.

In general, we can construct an input for a sequence of actions up to time T, such that the algorithm always gets 0 rewards at each time step but all other actions get 1, and hence after time T, $U_T^A = 0$. Given n actions, that means that at each round there are $n-1$ 1s associated to the other non-selected actions. Over time T, there are a total of $(n-1) \times T$ 1s, and hence an average of $\frac{(n-1)\times T}{n}$ 1s. But that means that *at least* 1 action would yield a utility of $U_T \geq \frac{(n-1)\times T}{n}$ 1s.

6.5.2 Randomized Greedy Algorithm

Stepping up from the greedy algorithm, we combat the adversarial aspect by introducing randomness. That is, the only change that we make is instead of a deterministic tie-breaking scheme in selection of actions from S_t, we instead pick actions from this set randomly with uniform probability.

Algorithm 9 Randomized Greedy Selection

for $t = 0, 1, ...$ **do**
  $U_t^{\max} \leftarrow \max_i U_t^i$
  $S_t \leftarrow \{i \in N : U_t^{i-1} = U_t^{\max}\}$
  for $i \in N$ **do**
    if $i \in S_t$ **then**
      $p_t^i \leftarrow \frac{1}{|S_t|}$
    else
      $p_t^i \leftarrow 0$
    end if
  end for
end for

That is, at each round we pick each action that is in the set S_t with uniform probability, and all other actions with probability 0. In fact, with some observations, we will see that the Randomized Greedy algorithm can lose at most $\log n + 1$ times between the loss of the best action.

First, observe that as the time t passes, actions are removed from the set S_t as these actions incur losses. It is also true that actions can also be added to the S_t, however, this actually lowers the probability of picking a losing action and does not provide a good bound on the number of losses Randomized Greedy will make.
Consider the number of times l that the best action loses. That is, the best action does not yield rewards at times $t_1, ..., t_l$. Then we wish to ask the question: in between losses t_j and t_{j+1}, how many times will the Randomized Greedy selection lose? Let the number of times that the algorithm loses be k_j. Then, we can observe that if there is a single time period where there are k_j actions that lost, then there is $\frac{k_j}{|S_t|}$ probability that the agent picks a losing action:

$$\mathbb{E}[U^*_{t_j} - U^{RG}_{t_j}] = \frac{k_j}{|S_t|}$$

Furthermore, observe that over a period of time between t_j and t_{j+1}, if actions are slowly removed from S_t(in particular, removed 1 at each round), then:

$$\begin{aligned}
\mathbb{E}[U^*_{t_j} - U^{RG}_{t_j}] &= \frac{1}{|S_{t_j}|} + \frac{1}{|S_{t_j}| - 1} + ... + \frac{1}{|S_{t_j}| - k_j + 1} \text{ (actions removed 1 at a time)} \\
&\geq \frac{l_1}{|S_{t_j}|} + \frac{l_2}{|S_{t_j}| - l_1} + ... + \frac{l_k}{|S_{t_j}| - l_{k-1}} \text{ (actions removed } l_i \text{ at a time)} \\
&= \underbrace{\frac{1}{|S_{t_j}|} + ...}_{l_1 \text{ times}} + \frac{1}{|S_{t_j}| - l_1} + ... \\
&\geq \underbrace{\frac{1}{|S_{t_j}|} + ...}_{k_j \text{ times}} = \frac{k_j}{|S_t|} \text{ (actions removed all at once)}
\end{aligned}$$

We can see that removing actions once at a time without having actions added into the set , yields the greatest loss. In fact, it acts as an upper bound for the number of times that the Randomized Greedy algorithm will lose, which is further upper bounded by:

$$\begin{aligned}
\frac{1}{|S_{t_j}|} + \frac{1}{|S_{t_j}| - 1} + ... + \frac{1}{|S_{t_j}| - k_j + 1} &\leq \log |S_t| + 1 \\
&= \log n + 1
\end{aligned}$$

Hence, the Randomized Greedy algorithm loses at most times in logarithmic factor to the number of actions available in between losses of the best action. Then, in total, the number of losses the Randomized Greedy algorithm will make is:

$$\log n + (\log n + 1) \times m$$

where m is the number of losses the best action makes up to time t.

6.5.3 Multiplicative Weights Update

Now, we have an algorithm that does not lose out too much from the optimal utility, with a logarithmic factor. However, it turns out that we can actually do better than this by adjusting the probabilities with which we select actions. With Randomized Greedy selection, the algorithm does not use knowledge of how good a move is. But, if we adjust probabilities to pick better moves with higher utilities, we can actually achieve close to the utility of the best action.
The weights used in the algorithm w^i_t represents how good the moves are with respect to the total utility that the action has earned up to time t. If the action yields positive rewards at

Algorithm 10 Multiplicative Weights Update

for $n = 1, .., n$ **do**
    $w_1^i \leftarrow 1$
    $p_1^i \leftarrow \frac{1}{n}$
end for
for $t = 0, 1, ...$ **do**
    for $i = 1, .., n$ **do**
        $w_t^i \leftarrow w_{t-1}^i \times e^{\epsilon u_t^i}$
    end for
    $W_t \leftarrow \sum_i w_t^i$
    for $i = 1, ..., n$ **do**
        $p_t^i \leftarrow \frac{w_t^i}{W_t}$
    end for
end for

time t, then v_t^i is positive and hence w_t^i increases in the update step. Then, each move with picked with respect to probability that scales with it's weight at time t.

More importantly, we shall see that this algorithm yields rewards that are *at least as good as* the best action. Firstly, we have that the weights with respect to time $t+1$ is:

$$\begin{aligned} W_{t+1} &= \sum_i w_{t+1}^i \\ &\geq w_{t+1}^* \\ &= w_t^* \times e^{\epsilon u_{t+1}^*} \\ &= w_1^* \times e^{\epsilon u_1^*} \times ... \times e^{\epsilon u_{t+1}^*} \\ &= e^{\epsilon(u_1^* + ... + u_{t+1}^*)} \\ &= e^{\epsilon U_{t+1}^*} \end{aligned}$$

On the other hand, we also have that:

$$\begin{aligned} W_{t+1} &= \sum_i w_{t+1}^i \\ &= \sum_i w_t^i \times e^{\epsilon u_{t+1}^i} \\ &\leq \sum_i w_t^i \times (1 + \epsilon u_{t+1}^i + (\epsilon u_{t+1}^i)^2) \text{ using the fact that for } x \in [-1, 1], 1e^x \leq 1 + x + x^2 \\ &\leq \sum_i w_t^i \times (1 + \epsilon u_{t+1}^i + \epsilon^2) \text{ since } 0 \leq u_{t+1}^i \leq 1 \rightarrow (\epsilon u_{t+1}^i)^2) \leq \epsilon^2 \\ &= \epsilon \sum_i w_t^i u_{t+1}^i + (1 + \epsilon^2) \sum_i w_t^i \\ &= \epsilon \times W_t \sum_i p_t^i u_{t+1}^i + (1 + \epsilon^2) \sum_i w_t^i \text{ using the fact that } \frac{w_t^i}{W_t} = p_t^i \\ &= \epsilon W_t(u_{t+1}^{MWU}) + (1 + \epsilon^2) W_t \text{ because } u_{t+1}^{MWU} = \sum_i p_t^i u_{t+1}^i \text{ is the time } t+1 \text{ expected reward of MWU} \\ &= W_t(1 + \epsilon^2 + \epsilon u_{t+1}^{MWU}) \\ &\leq W_t(e^{\epsilon^2 + \epsilon u_{t+1}^{MWU}}) \text{ using the fact that } 1 + x \leq e^x \end{aligned}$$

Applying the equation above recursively, we then have:

$$\begin{aligned}
W_{t+1} &\leq W_t(e^{\epsilon^2+\epsilon u_{t+1}^{MWU}}) \\
&= W_{t-1} \times (e^{\epsilon^2+\epsilon u_t^{MWU}}) \times (e^{\epsilon^2+\epsilon u_{t+1}^{MWU}}) \\
&... \\
&= W_1 \times (e^{\epsilon^2+\epsilon u_1^{MWU}}) \times (e^{\epsilon^2+\epsilon u_2^{MWU}}) \times ... \times (e^{\epsilon^2+\epsilon u_{t+1}^{MWU}}) \\
&= n \times e^{t\epsilon^2+\epsilon(\sum_{j=1}^{t+1} u_j^{MWU})} \\
&= n \times e^{t\epsilon^2+\epsilon U_{t+1}^{MWU}}
\end{aligned}$$

Taking the first part and second part,

$$\begin{aligned}
e^{\epsilon U_{t+1}^*} &\leq W_{t+1} \leq n + e^{t\epsilon^2+\epsilon U_{t+1}^{MWU}} \\
\epsilon U_{t+1}^* &\leq \log n + t\epsilon^2 + \epsilon U_{t+1}^{MWU} \\
U_{t+1}^* &\leq \frac{\log n + t\epsilon^2}{\epsilon} + U_{t+1}^{MWU}
\end{aligned}$$

which shows that the expected rewards of the MWU algorithm is at least as good as the expected reward of the best action.

7 Logical Agents

Previously in search, agents had knowledge of the environment, transition models and the possible utilities at several states. However, such knowledge is limited. For example, the agent playing chess using purely these information, may not be able to identify that it cannot reach the checkmate position to win from it's current position. Meaning, it cannot *infer* knowledge based on what it already knows. Instead, what we try to do is formulate smarter agents: Agents that are able to infer information based on an existing knowledge. What the agent currently knows is called a **knowledge base**. The knowledge base first starts off empty, but continue to grow as the agent makes new inferences, and the inferences made by the agent are *sentences* in a *formal language* in the knowledge base.

7.1 Knowledge-Based Agents

The difference between logical agents and the previous agents is that the logical agent has a **knowledge base** that tracks what the agent knows so far. The knowledge base consists of **sentences or axioms** that is expressed in a **knowledge representation language**. This knowledge base may contains background knowledge using prior knowledge or assumptions. A logical agent, very generally, can be formulated as such:

Algorithm 11 Knowledge-Based Agent

Persistent: KB - a knowledge base, t - time counter

TELL(KB, MAKE-PERCEPT-SENTENCE($percept, t$))
$action \leftarrow$ ASK(KB, MAKE-ACTION-QUERY(t))
TELL(KB, MAKE-ACTION-SENTENCE($action, t$))
$t \leftarrow t + 1$
return *action*

The knowledge-based agent is governed by 2 actions: TELL and ASK. They justify the actions that the logical agent must be able to perform. Firstly, the agent must be able to take in new percepts. This is seen by performing the TELL operation when it receives a new percept through sensors, and performs an action using actuators. Both operations update the knowledge base based on the perceived new environment, or the new action performed. For example, in the current state the agent may perceive that it cannot make a certain legal move based on the rules that it knows, and hence updates it's knowledge base on this information.

Then, using the existing knowledge and newfound ones(if any), the agent performs an action using the ASK operation, which makes a legal and perhaps even optimal move based on what it already knows. In relation to Reinforcement Learning, RL actually plays a big role in the ASK operation, where it selects the optimal ones in the presence of multiple valid actions.

Inference may be present in both TELL and ASK operations, where the agent derives new sentences from existing ones and the current percept. Additionally, we required that the knowledge base be updated with information that is correct. That is, new axioms must indeed follow from what is implied by the current knowledge. This is also called **entailment**. For now, we are interested in the TELL operation: how do we infer new information based on existing information and new percepts?

7.2 Logic

The fundamental concepts of logical representation and reasoning that allow us to formulate sentences are as follows:

- The **syntax** of the knowledge representation language tells us how we can express logic in the form of sentences, as well as if a sentence is valid. For example, the arithmetic language tells us that $x + y = m$ is valid, but $xy+ >$ is not.
- The **semantics** of the language tells us the meaning of the sentences formulated in that language, as well as the truth values behind these sentences. $x + y = 4$ would hold true in the arithmetic language where $x = 2, y = 2$, but false when $x = 1, y = 1$.
- a model m is said to **model** the logical sentence α, if α is true under m. m can usually be thought of as the 'world' or environment that the agent lives in. Following the previously example, $x + y = 4$ is satisfied by the environment where there are x red balls and y blue balls, and there are always a total of 4 balls. Since $x + y = 4$ is always in such a world, we say that it is a model of α.

7.2.1 Entailment

We say that a logical sentence, or a collection of logical sentences **entails** another, if the entailed sentence *follows logically* from the sentence. More formally, we say that if α entails β, then:

$$\alpha \models \beta$$

we say that β follows logically from α, or that α implies β. Following the property of models, let $M(\alpha)$ be the set of all models under which α is true. Then the following formal definition ensues for entailment:

Definition 6. *$\alpha \models \beta$ if and only if, in every model m in which α is true, β is also true.*

That is, if $m \in M(\alpha)$, then $m \in M(\beta)$, and hence $M(\alpha) \subset M(\beta)$. Hence, we can see that if $\alpha \models \beta$, then α is a *stricter* condition than β.

Then, using the definition of entailment, we wish that our inference algorithm be **sound**and **complete**. This ensures that our inference algorithm does indeed derive the optimal and correct solutions to our logic problem. Now, let the notation

$$KB \vdash_A \alpha$$

denote that the logical sentence α is derived from the knowledge base KB using inference algorithm A. Then, the inference algorithm A is **sound** if:

$$KB \vdash_A \alpha \rightarrow KB \models \alpha$$

That is, if a logical statement α is inferred by the algorithm A from the knowledge base KB, then that statement α indeed follows from the logical sentences in the knowledge base. Our inference algorithm only infers what is indeed correct, and not nonsense. We also say that such an algorithm is **truth-preserving**.

The inference algorithm A is **complete**, if it satisfies:

$$KB \models \alpha \rightarrow KB \vdash_A \alpha$$

If a logical statement α is true and follows from the logical sentences of the knowledge base, then it would be found by the inference algorithm A. Then, the agent would not having missing information using the inference algorithm A.

7.3 Propositional Logic

7.3.1 Truth Table Enumeration

One simple way to check if a sentence α is entailed by the knowledge base that we have, is to enumerate through the models in $M(KB))$ and check that α is indeed true for each model.

The algorithm is sound is because it simply uses the direct definition of entailment, and checks that the condition holds. It is also complete because given a finite number of symbols, then there are only a finite number of propositional logic statements. However, this still incurs a time complexity of $O(2^n)$ for n symbols. Using a depth first search technique, this incurs a space complexity of $O(n)$.

7.3.2 Propositional Theorem Proving

Previously, we have shown how we can determine entailment through basic model checking: by enumerating all possible table values and checking that the definition of entailment always holds. However, this can be very expensive, and not leveraging our knowledge of propositional logic. Instead, what we shall do is that we can show entailment(or lack of thereof) through **theorem proving** - by applying the rules of inference directly to sentences.

First, we introduce a series of concepts related to propositional logic:

- **Logical Equivalence:** two sentences α and β are logically equivalent, if they are true in the same set of models. For example $P \vee Q$ and $Q \vee P$ are logically equivalent. Then, we denote logical equivalence as:

Algorithm 12 TRUTH-TABLE ENUMERATION

```
1: procedure TT-ENTAILS?(KB, α) returns true or false
2:     inputs: KB - the knowledge base, α - sentence in propositional logic
3:     symbols ← a list of propositional symbols in KB, α
4: return TT-CHECK-ALL(KB, α, symbol, {})
5: end procedure
6:
7: procedure TT-CHECK-ALL(KB, α, symbols, model) returns true or false
8:     if EMPTY?symbols then
9:         if PL-TRUE?(KB, model) then return PL-TRUE?(α, model)
10:        else return true
11:        end if
12:    else
13:        P ← FIRST(symbols)
14:        rest ← REST(symbols)
15:        return TT-CHECK-ALL(KB, α, rest, model ∪ {P = true}) and
16: TT-CHECK-ALL(KB, α, rest, model ∪ {P = false})
17:    end if
18: end procedure
```

$$\alpha \equiv \beta$$

Alternatively, that means that if either of α or β is true, then the other must be true. Hence, one actually entails the other. We can then look at equivalence as:

$$\alpha \equiv \beta \leftrightarrow \alpha \models \beta \wedge \beta \models \alpha$$

- **Validity:** We say that a sentence is valid if it is true in *all* models. For instance, $P \vee \neg P$ is always valid, since it always holds. Valid sentences are also known as tautologies: they are necessarily true.
- **Satisfiability:** A sentence is satisfiable if it is *true* in *some* model. Additionally, a sentence is **unsatisfiable** if it is never true: it is $false$ in all models.

Using the above concepts, we can derive the following deduction theorem:

Theorem 4. *For any sentences α and β, $\alpha \models \beta \leftrightarrow (\alpha \implies \beta)$ is valid*

And this is exactly the same formal definition that we defined earlier on entailment: a sentence α entails another β, if β follows from α in all models. That is, $\alpha \implies \beta$ always holds for every model, which means that it is indeed valid.

Furthermore, notice that validity and satisfiability are linked: if a sentence α is valid, then it always holds in all models no matter what. That would imply that it's negation $\neg\alpha$ can never hold in any model. That is, $\neg\alpha$ is unsatisfiable. In fact that means that to show entailment, which originally is the following using Theorem 4:

$$KB \models \alpha \leftrightarrow (KB \implies \alpha) \text{ is valid}$$

We can also show that the *negation* of $(KB \implies \alpha)$ is unsatisfiable. That is, proving the above is logically equivalent to:

$$KB \models \alpha \leftrightarrow (KB \wedge \neg\alpha) \text{ is unsatisfiable}$$

7.3.3 Inference Rules

We now discuss some techniques known as inference rules that allow us to infer rules from existing ones, and eventually arrive at the a conclusion.

Modus Ponens: $\frac{\alpha \implies \beta, \alpha}{\beta}$

Says that if we have the rule $\alpha \implies \beta$, and we know the value of α, then we can infer what β is.

And-Elimination: $\frac{\alpha \wedge \beta, \alpha}{\alpha} \equiv \frac{\alpha \wedge \beta, \alpha}{\beta}$

And-Elimination tells us if we have a conjunction of clauses, if the conjunction holds true then we know each of the individual conjuncts are true as well.

Biconditional Elimination: $\frac{\alpha \iff \beta}{(\alpha \implies \beta) \wedge (\beta \implies \alpha)} \equiv \frac{\beta \iff \alpha}{(\alpha \implies \beta) \wedge (\beta \implies \alpha)}$

This allows us to convert if and only if statements into implication statements. Now, the next rule is important and will allow us to yield a sound and complete inference algorithm:

Resolution: $\frac{(x_1 \vee ... \vee x_{m-1} \vee x_m) \wedge (y_1 \vee ... \vee y_{m-1} \vee \neg x_m)}{x_1 \vee ... \vee x_{m-1} \vee y_1 \vee ... \vee y_{m-1}}$

The resolution says that, if a clause contains a literal, and another contains it's negation, then both can be removed from it's respective clauses, and the disjunction of the clauses with literal and it's negation removed is known as the **resolvent**. This is true because, consider the conjunction of clauses $(P \vee x) \wedge (Q \vee \neg x)$. For this sentence to hold, both clauses in the conjunction must be hold. But notice that only x and it's negation $\neg x$ can hold at once, and not both. If x holds, then $\neg x$ does not and hence Q must hold, and vice versa. Hence $P \vee Q$ holds.

Alternatively, we can also think of it this: Given an existing clause with prior knowledge in the form of the literal x. Now, we have an additional inferred clause that says $\neg x$ holds, which means x is no longer true. Hence, we resolve the clauses by removing each of the literals x and $\neg x$. Hence, the resolution rule is *sound*.

7.3.4 Resolution Algorithm

Remember that using propositional theorem solving, to show that KB indeed entails α, we want to show that $KB \implies \alpha$ is valid, or consequently that $KB \wedge \neg\alpha$. The resolution proves this by contradiction as follows:

1. $KB \wedge \neg\alpha$ is converted into conjunctive normal form(CNF). This is achieved using a combination of inference rules.

2. The resolution rule is applied to resulting clauses, which is the proof by contradiction process. Each pair of clauses that contains complementary literals is resolved to produce a new clause that is added back into the set. Then, one of 2 things will happen:

    - No new clauses can be added in the set. That means that there exists a pair of generating clauses such that they do not contain complementary literals, and that they can potentially hold in some models, and hence $KB \implies \alpha$ is satisfiable. Therefore, KB does not entail α.

- 2 clauses resolve to produce the empty clause. Remember that literals are removed when they cannot both be satisfied, that is they are contradicting/complementary literals. Then, 2 clauses that resolve to become the empty clause, means that the generating clauses in $KB \implies \alpha$ can be reduced to clauses that cannot both hold at the same time. Hence, $KB \implies \alpha$ is unsatistifable, and KB entails α.

Algorithm 13 PL-RESOLUTION

inputs: KB - knowledge base. α - propositional logic sentence

$clauses \leftarrow$ set of clauses in CNF representation of $KB \wedge \neg\alpha$
$new \leftarrow \{\}$
repeat
  for each pair of clauses C_i, C_j in $clauses$ **do**
    $resolvents \leftarrow$ PL-RESOLVE(C_i, C_j)
    if $resolvents = \emptyset$ **then return** $true$
    end if
    $new \leftarrow new \cup resolvents$
  end for
  if $new \subset clauses$ **then return** $false$
  end if
  $clauses \leftarrow clauses \cup new$
until all clauses resolved

We have seen that using the resolution rule yields inferences that are sound, because resolved rules are indeed entailed by KB using propositional logic rules. Now we shall argue that it is complete:

Completeness of Resolution

First, let us introduce the concept of a **resolution closure**, $RC(S)$ of a set of clauses S, which is the set of all clauses derivable by repeated application of the resolution rule to the clauses in S, as well as their derivatives. We can see that for a finite number of symbols involved in the clause $P_1, ..., P_k$ that appear in S, we can only construct a finite number of resolved clauses. Additionally, we repeat the resolution process for each pair of clauses everytime we introduce new clauses into the set.Since we introduce only a finite number of resolved clauses into the set, then consequently the number of iterations of PL-RESOLUTION is also finite.

To complete the proof for completeness, we state the **ground resolution theorem:**

If a set of clauses is unsatisfiable, then the resolution closure of those clauses contains the empty clause.

We can prove this by it's contrapositive. Suppose that the resolution closure $RC(S)$ does not contain the empty clause. Then, remember that resolution is a form of reduction using propositional logic. That is, the resolved clauses are entailed by the generating clauses that were involved in the resolution to it. Since the generating clauses entailed a non-empty clause, we can find assignments for this resolved clause such that it holds. Hence, this means that $KB \wedge \neg\alpha$ is satisfiable.

7.4 Model Inference

Because resolution is a sound and complete method for proving propositional logic, it is very important int he logic world. However, it is very expensive and runs in time exponential to the number of variables and clauses. Instead, we have more practical methods that runs more efficiently than the resolution algorithm.

7.5 Horn and definite clauses

For us to use more efficient algorithms, more often than not they are more specific and required certain conditions. Some of these conditions, is that the logic statements must be in **Horn clause** or **definite clause**. This allows us to use algorithms like Forward and Backward Chaining.

Definite clauses are clauses that contain a disjunction of literals, of which *at most one is positive*. For example, $(\neg x \vee \neg y \vee z)$ is a definite clause, while $(x \vee \neg y \vee z)$ is not. Slightly more general form of this is the **Horn clause**, which are clauses where *at most one* literal is positive.Clauses that do not have any positive literals are called **goal clauses**, and they also fall under Horn clauses.

We wish to express our knowledge base KB, in the form of a conjunction of Horn clauses. Doing so is particularly interesting because:

- Every definite clause can be expressed in it's implication form, whose premise is a conjunction of positive literals(derived from disjunction of negative literals), and the conclusion is a single positive literal. In general, we have that:

$$\begin{aligned}(\neg x_1 \vee ... \vee \neg x_i \vee y) &= (\vee_i \neg x_i \vee y) \\ &= (\neg(\wedge_i x_i) \vee y) \\ &= \wedge_i x_i \implies y\end{aligned}$$

- For goal clauses, we have that the implication form has all the literals(in positive form) as the premise, and $false$ as the conclusion:

$$\begin{aligned}(\neg x_1 \vee ... \vee \neg x_i) &= (\vee_i \neg x_i \vee y) \\ &= (\neg(\wedge_i x_i)) \\ &= \wedge_i x_i \implies false\end{aligned}$$

- Sentences in Horn clauses can be converted to implication forms, which makes it easier to understand. More importantly, it turns out deciding entailment with Horn clauses can be done in time *linear* in the size of the knowledge base using Forward and Backward Chaining.

7.5.1 Forward Chaining

Forward Chaining is a **data-driven** approach to model checking that sequentially infers new knowledge, given our knowledge base KB in the form of Horn clauses, as well as base facts that we already know(important, else we cannot infer anything at all from the start).

We can view the forward chaining procedure in an implication graph, or AND-OR graph as the algorithm sequentially adds new variables that it infers to it's knowledge. We start with base

facts: sentences containing knowledge of things we know. This, together with our rules in Horn clause form, allow us to **entail** new knowledge by definition of implication. More formally, the algorithm uses Modus Ponens:

$$\frac{a_1 \wedge ... \wedge a_k; \wedge_i a_i \implies b}{b}$$

That is, if we know facts a_1 to a_k, and we know that a conjunction of these facts imply b, then we know that b holds, and we can add it to our knowledge base. Generally, for each rule/clause c that is in Horn clause form: $\wedge_i x_i \implies y$. Then, let $count[c]$ be the number of literals in the clause's premise that have not yet been inferred, initialized to k. That is, we still need to infer that the remaining $count[c]$ literals in premise of c holds, before we can infer it's conclusion y by Modus Ponens.

Each time we infer a new variable x_i, if $x_i \in$ premise of c, then we decrement the $count[c]$ by 1 since we have inferred knowledge of a new variable that contributes to the conclusion of c. When $count[c]$ is decremented to 0, that means that we have inferred all knowledge in the premise of c, and hence by Modus Ponens, can conclude that c is true, adding the conclusion y of c to our knowledge base.

Algorithm 14 PL-FC-ENTAILS?

```
1: inputs: KB - knowledge base, q, the symbol queried
2: count ← a table, where count[c] is number of symbols in c's premise
3: inferred ← a table, where inferred[s] is initially false for all symbols
4: agend ← a queue of symbols, initially symbols known to be true in KB
5: while agend is not empty do
6:     p ← POP(agenda)
7:     if p = q then return true
8:     end if
9:     if inferred[p]pfalse then
10:        inferred[p] ← true
11:        for each clause c in KB where p is in c.PREMISE do
12:            count[c] ← count[c] − 1
13:            if count[c] = 0 then add c.CONCLUSION to agenda
14:            end if
15:        end for
16:    end if
17: end while
18: return false
```

It is clear that Forward Chaining is indeed **sound**, because it infers new knowledge using Modus Ponens, which is known to be sound and follows the definition of entails. Forward Chaining is also **complete:** every entailed atomic sentence will be derived by the algorithm. When the algorithm terminates, the final state of the $inferred$ table will contain $true$ if it was inferred by FC at any point in the search, and $false$ otherwise. Furthermore, *every definite clause in the original KB is true in this model.*

Assume this is false. Then, that means that there exists a clause $a_1 \wedge ... \wedge a_k \implies b$ such that $a_1 \wedge ... \wedge a_k$ holds, but b is $false$. But that means that the FC algorithm has yet to infer the clause, and hence has not yet terminated. Therefore, we can see that the FC algorithm will infer all the knowledge that it can possibly infer before terminating. That is, if the knowledge base KB entails b given certain ground truths, then Forward Chaining would have inferred it.

7.5.2 Backward Chaining

Backward Chaining is a form of **goal-directed reasoning**. The algorithm starts with the query q and works backwards to infer if q is true. If the query q is known then no work is needed. Otherwise, the algorithm finds implications in the knowledge base whose conclusion is q. If all premises of one of those implications can be proved true by Backward Chaining, then q is true. Else, we propagate backwards to find implications where our current variable is a conclusion, until we can prove all premises of the implication by Modus Ponens.

Unlike Forward Chaining that infers new knowledge using existing data, Backwards Chaining is less expensive because it only uses relevant implications and facts, compared to using all data in the knowledge base.

7.6 Propositional Model Checking

7.6.1 DPLL Algorithm

The DPLL(Davis, Putnam Logemann, Loveland) algorithm, is an improved backtracking search algorithm that uses multiple heuristics to decide how to assignments for sentences in conjunctive normal forms. These heuristics include:

- **Early Termination:** The algorithm can detect whether the entire sentence is true or false without having to complete the assignments for all variables, similar to constraint propagation. Notice since the sentence is in CNF, as long as any clause is false, then the entire sentence is false.

  Simiarly, a clause is true is any of it's literal is true. This reduces the amount of checking that we have to do over all clauses, if we know that a symbol within it is already true. Early termination allows us to prune large subspaces of search.

- **Pure Symbol Heuristic:** A pure symbol is one that always appears either as it's literal, or it's negation in all clauses. For example, in the clause $(x \vee \neg y) \wedge (z \vee y) \wedge (x \vee z)$, x and z are pure symbols. When we have pure symbols in the sentence, we start assigning values to these symbols first. This is because since they either only appear as it's literal or the negation, then making it true will cause all clauses that has this variable to be true, since negations of the symbol do not appear. Hence, such assignments will only make clauses true, and never false, thus increasing the number of clauses that are true. This is similar to the least constraining value heuristic in CSPs.

- **Unit Clause:** If a clause contains only a single literal, then that literal must be evaluated to true, else the sentence cannot hold true. This includes clauses with multiple literals assigned except one that still evaluates to false. Then, the last literal must evaluate to true. For example, in the clause $(\neg B \vee C)$, if $B = true$, then we have no choice but to set $C = true$ for the clause to be true. This mirrors the most constrained variable heuristic that we have seen in CSPs.

Additionally the DPLL algorithm has various other ways to optimization that allows it to be scaleable to larger graphs:

- **Component Analysis:** As DPLL assign truth values to variables, the set of clauses may become separated into disjoint subsets called components, that shared no unassigned variables. Efficiently detecting these subsets may allow for increased speed by working on each component separately.

Algorithm 15 DPLL

```
1: procedure DPLL-SATISFIABLE?(s) returns true or false
2:     inputs: s, a sentence in propositional logic
3:
4:     clauses ← set of clauses in the CNF representation of s
5:     symbols ← a list of propositional symbols in s
6:     return DPLL(clauses, symbols, {})
7: end procedure
8:
9: procedure DPLL(clauses, symbols, model) returns true or false
10:    if every clause in clauses is true in model then return true
11:    end if
12:    if some clause in clauses is false in model then return false
13:    end if
14:    P, value ← FIND-PURE-SYMBOL(symbols, clauses, model)
15:    if P is non-null then return DPLL(clauses, symbols − P, model ∪ {P = value})
16:    end if
17:    P, value ← FIND-UNIT-CLAUSE(clauses, model)
18:    if P is non-null then return DPLL(clauses, symbols − P, model ∪ {P = value})
19:    end if
20:    P ← FIRST(symbols); rest ← REST(symbols)
21:    return DPLL(clauses, symbols − P, model ∪ {P = true}) ∨ DPLL(clauses, symbols −
       P, model ∪ {P = false})
22: end procedure
```

- **Variable and value ordering:** The DPLL algorithm above uses the values *true* before *false* for tie-breaking. As in the CSP, we can design better tie-breaking heuristics such as the degree heuristic.
- **Intelligent Backtracking:** SAT solvers can make use of intelligent backtracking by making use of some form of conflict clause learning to record conflicts so that they won't be repeated later in search.
- **Random restarts:** may be required when the search seems to not be making progress sometimes, allowing for reduced search time doing making little progress.

7.6.2 Walk-SAT

Because the problem of solving the satisfiability of boolean CNF-SAT formulas is intractable similar to solving general CSPs, we often look towards local search methods which are able to produce approximately good solutions in reasonable time . Just like how we have MIN-CONFLICTS in CSPs, we also have the Walk-SAT algorithm for CNF formulas.

With local search methods, we require that the algorithm knows that it is moving in the direction of improvement: There is a need for an evaluation function to estimate how good a certain configuration is. Similar to the number of conflicts in the MIN-CONFLICTS algorithm, we approximate our goal of finding an assignment of satisfying every clause by counting the number of unsatisfied clauses.

The Walk-SAT algorithm greedily chooses to either perform exploitation by picking an

assignment for a variable that minimizes the number of unsatisfied clauses in the new state, or perform a "random walk" by picking a random symbol for assignment in the exploration stage with a probability p

Algorithm 16 WALK-SAT

inputs:
clauses, a set of clauses in propositional logic
p, the probability of choosing to do a 'random walk' move
max_flips, the number of flips allowed before giving up

$model \leftarrow$ a random assignment of $true/false$ to the symbols in the clause
for $i = 1$ to max_flips **do**
    if *model* satisfies *clauses* **then return** *model*
    end if
    clause $\leftarrow$ a randomly selected clause from *clauses* that is false in *model*
    with probability p flip the value of a randomly selected symbol in *clause*
    else flip whichever symbol in *clause* maximizes the number of satisfied clauses
end for
return $failure$

When WALK-SAT returns a model, that means that it has indeed found a satisfying solution. However, if it returns failure instead, this could be for 2 reasons: Either the sentence itself is unsatisfiable, or that the algorithm did not manage to find it due to it's probabilistic nature. Increasing the number of flips with max_flips will increase the chances of WALK-SAT finding a satisfying solution if there exists one, and even more so if $max\_flips = \infty$, then it will return return a model(if it exists) because eventually it will hit the solution. Unfortunately, this is so in theory, but in practice that means that the algorithm may never terminate.

Because of this, WALK-SAT is **incomplete**, since there is no guarantee that with every iteration the number of satisfied clauses is non-decreasing. More importantly, it cannot be used to detect unsatisfiability, since upon termination with failure we cannot tell if indeed the sentence is unsatisfiable, since it does not necessarily search the entire space.

8 First Order Logic

Using propositional logic, we are able to declaratively tell the logical agents what we want them to do. However, even for simple and small problems, we will see that the number of inferences and rules blow up exponentially: There can be a large number of propositional logic sentences. Instead the knowledge base of our logical agents in propositional logic, can more often than not be expressed more concisely using **first-order logic**. Hence, first order logic is said to be more *expressive*.

8.1 First Order Inference

The **syntax** of FOL consists of the following:

- **Constants:** 2, John
- **Predicates:** $brother(John, Doe), isSmart(x)$
- **Functions:** $myFunc(x), \sqrt{x}$

- **Variables:** x, y, z
- **Connectives:** $\neg, \vee, \wedge, \implies$
- **Equality:** $=$

A **term** refers tp a constant, variable or function, and an **atomic sentence** a single sentence containing a predicate, without the use of connectives: $sibling(John, Richard)$. **Complex sentences** are then constructed from atomic sentences using connectives: $(aleqb) \vee (a > b)$ or $sibling(John, Richard) \implies sibling(Richard, John)$. Unlike in propositional logic where each expression has a fixed meaning to it, in FOL each predicate and function is defined by what meaning we assign it. For example, in the example before, the predicate $sibling(x, y)$ means that persons x and y are siblings.

In propositional logic, remember that if we had a composition of rules, for instance a conjunction rules, we would express them as:

$$\begin{aligned} & King(John) \wedge Greedy(John) \implies Evil(John) \\ & \wedge King(Richard) \wedge Greedy(Richard) \implies Evil(Richard) \\ & \wedge King(father(John) \wedge Greedy(father(John)) \implies Evil(father(John)) \\ & ... \end{aligned}$$

Which says that if a king is greedy, then he is evil. But this single english sentence is yet expressed in a possibly infinitely many propositional logic sentences, if we were to list out all possible kings! Fortunately, FOL allows us to concisely pack them using $quantifiers$, in particular for this one using **universal instantiation**:

$$\forall x, King(x) \wedge Greedy(x) \implies Evil(x)$$

Similar if we wish to say that there is **some** king who is greedy and evil, instead of saying:

$$\begin{aligned} & King(John) \wedge Greedy(John) \implies Evil(John) \\ & \vee King(Richard) \wedge Greedy(Richard) \implies Evil(Richard) \\ & \vee King(father(John) \wedge Greedy(father(John)) \implies Evil(father(John)) \\ & ... \end{aligned}$$

we can use **existential instantiation** to rephrase it as:

$$\exists x, King(x) \wedge Greedy(x) \implies Evil(x)$$

However, we have a simpler reduction technique known as **Skolemization** by adding a new constant known as the **Skolem constant** as follows:

$$\exists x : P(x) \implies P(x_0)$$

where x_0 is a Skolem constant that we introduce into our KB that wasn't originally there. This basically means there is *some* variable x_0(which we may not know what is now), such that $P(x_0)$ holds.

Functions and predicates are treated as constant variables in the propositional logic sentence(e.g $King(John)$).

This technique of **propositionalization** can be made completely general by applying the above methods. In fact, any first order logic knowledge base and query can be propositionalized in

a way such that entailment is preserved. However, while the result of propositionalization of FOL knowledge bases are complete, the procedure of conversion is not. In particular, when the knowledge base contains function symbols(with infinite domains), then the set of possible ground term substitutions are infinite. For example, if we have the the original FOL:

$$x > y \iff x - next(y) \vee \exists z, x = next(z) \wedge z > y$$

can be converted into:

$$x > y \implies x = next(y) \vee x = next(next(y)) \vee ...$$

results in an infinitely nested term during propositionalization. Propositional algorithms will run for infinitely long periods of time with KB having infinitely many terms. Fortunately, Herbrand's theorem says the following:

Theorem 5. *If α is entailed by a First Order Logic knowledge base KB, then is entailed by a **finite subset** of the propositionalized KB*

Since any such subset has a maximum depth of nesting among it's ground terms, we can find the subset by first generating all the instantiations with constant symbols up to a specific depth k that is non-decreasing, and then perform a check for entailment as before using algorithms like resolution and forward/backward chaining. This is similar to the IDS search algorithm that we have done before.

Observe that if the algorithm returns true, then we know that a finite subset of depth k of the original KB does indeed entail α. However, if the algorithm returns false, we cannot immediately decide that KB does not entail α, since the current check for entailment is only up to depth k. In fact, we cannot know for sure at all that KB does not entail α if the algorithm returns false. This is known as **semi-decidability**.

Definition 7. *The question of entailment for first order logic is **semi-decidable**, if there exists algorithms that say yes to every entailed sentence, but no algorithm exists that say no to every non-entailed sentence*

More importantly, further observe that any predicate or function involving k variables each with d values, results with a d^k number of instantiations. This exponential blow up of sentences is unnecessary because more often than not we do not need most of them.

8.2 Unification

Unification is a key component of all first order logic inference algorithms, and it is a process that aims to make 2 logical expressions in FOL language look identical. The UNIFY algorithm takes in 2 sentences p, q, and returns a unifier θ for them if one exists:

$$\text{UNIFY}(p, q) = \theta \text{ where } \text{SUBST}(\theta, p) = \text{SUBST}(\theta, q)$$

where the unifier θ is a set of variable assignments involved in the 2 sentences p and q to make them look identical. For example, if we wish to ask the question $AskVars(Knows(John, x))$: Whom does John know? The answer to this can be found by finding all sentences in the knowledge base that unify with $Knows(John, x)$. If we know that John knows Jane: $Knows(John, Jane)$, then we can make the following unification:

$$\text{UNIFY}(Knows(John, x), Knows(John, Jane) = \theta = \{x \leftarrow Jane\}$$

Where the fact follows that if John knows everyone and John knows Jane, then we can infer that John knows Jane(trivially) by unifying the 2 sentences. More generally, if we have that $Knows(y, Jane)$, that is, everyone knows Jane, then the following unification holds:

$$\text{UNIFY}(Knows(John, x), Knows(y, Jane) = \theta = \{x \leftarrow John, y \leftarrow Jane\}$$

which has the following meaning entailed by the substitution: if John knows everyone and everyone knows Jane, then definitely John knows Jane. However, the following does not work:

$$\text{UNIFY}(Knows(John, x), Knows(x, Jane) = \theta = fail$$

because the 2 sentence share the same variable x, which results in name clashes and complications arising. This problem can be avoided by **standardizing apart** one of the 2 sentences being unified by renaming the variables to form:

$$\text{UNIFY}(Knows(John, x), Knows(x', Jane) = \theta = \{x \leftarrow John, x' \leftarrow Jane\}$$

8.2.1 Most General Unifier

Previously, we have mentioned that the UNIFY procedure should return a single substitution that makes the 2 sentences look identical. However, we may have situations where there are multiple possible substitutions that fulfil the unification. In $\text{UNIFY}(Knows(John, x), Knows(y, z)$, we can use $\theta = \{x \leftarrow z, y \leftarrow John\}$ as a substitution. But we can also use $\theta = \{x \leftarrow John, y \leftarrow John, z \leftarrow John\}$ since the 2 sentences are still identical. We say that the first unifier is *more general* than the second, since it places less restrictions than the second and yet captures the information that the second unifier implies. In particular, it turns out that for every unifiable pair of expressions, there is a single **most general unifier(MGU)**, that entails the information of every other possible unifier, and it is unique up to renaming and substitution of variables. Hence, we wish to find the MGUs for each unification process, and the algorithm for doing so can be found below:

Algorithm 17 UNIFICATION

procedure UNIFY(x, y, θ) **returns** substitution θ to make x, y identical
inputs:
x, y - variable, constant, list, or compound expression to be unified,
θ - the substitution to be return(initialized to empty)

if θ = fail **then return** fail
else if $x = y$ **then return** θ
else if VARIABLE?(x) **then return** UNIFY-VAR(x, y, θ)
else if VARIABLE?(y) **then return** UNIFY-VAR(y, x, θ)
else if COMPOUND?(x) $\wedge$ COMPOUND?(y) **then**
return UNIFY(x.ARGS, y.ARGS, UNIFY(x.OP, y.OP, θ))
else if LIST?(x) $\wedge$ LIST?(y) **then**
return UNIFY(x.REST, y.REST, UNIFY(x.FIRST, y.FIRST, θ))
else return fail
end if
end procedure

procedure UNIFY-VAR(var, x, θ) **returns** substitution θ
if $var \leftarrow val \in \theta$ **then return** UNIFY(val, x, θ)
else if $var \leftarrow val \in \theta$ **then return** UNIFY(val, x, θ)
else if OCCUR-CHECK?(var, x) **then return** fail
else return $\theta \cup var \leftarrow x$
end if
end procedure

The unification algorithm for finding MGU for 2 sentences is composed of 2 components. The first component is a general UNIFY procedure, which takes in 2 sentences P, Q, as well as an empty unifier θ at the start. Along the way, if the unifier contains $failure$, or if a valid unifier has been found($P = Q$), then the unifier θ is returned.

If both sentences P, Q are functions, then we cannot directly compare them. Instead, we break them down into 3 components: the arguments of P and Q in lists, as well as the function names of P and Q. Because functions are treated as unique variable names, we cannot unify P and Q if they are of different function names(e.g $Knows(x, y)$ and $Greedy(u, v)$). Hence, the algorithm will return false in this case.

If the function names agree, then the algorithm compares each respective argument from both sentences, using the UNIFY-VAR if there exists at least one variable argument. If the variable var already has a value val assigned to it from earlier substitutions, then we unify the sentence x with the val by getting the 2 to equate. Conversely, if x is already assigned a value, then we unify x with the variable var. Similar to before, if x contains var, then the algorithm automatically returns $failure$. Otherwise, we set $var \leftarrow x$. We repeat this process until we find a valid unifier, or failure.

8.3 Generalized Modus Ponens

In the original modus ponens in propositional logic given as $\frac{\alpha_1 ... \alpha_k; (\alpha_1 \wedge ... \wedge \alpha_k) \implies \beta}{\beta}$ says that if the facts α_1 to α_k are known, as the conjunction of these facts imply β, then we know that β is true. In FOL, we can perform modus ponens as well using **lifting**. Lifted or generalized modus

ponens allows us to perform modus ponens in the FOL context.

More specifically, given an implication $p_1 \wedge ... \wedge p_k \implies q$, and atomic sentences p_i, p_i', if there is a substitution $\text{SUBST}(\theta, p_i) = \text{SUBST}(\theta, p_i')$ for all i, then we can conclude that q holds.

$$\frac{p_1', ..., p_k'; (p_1 \wedge ... \wedge p_k) \implies q}{q}$$

This is intuitively saying that given whatever we know and can infer from $KB = \{p_1', ...\}$ we can substitute these facts into the implication statement by unifying each respective implicants p_i, p_i'.

We can show that generalized modus ponens is indeed a sound inference rule. Observe that for any sentence p whose variables are universally quantified, for any valid substitution θ, $p \models \text{SUBST}(\theta, p)$ by property of universal instantiation. Thus, for $p_1', ..., p_k'$, we can infer

$$\text{SUBST}(\theta, p_1') \wedge ... \wedge \text{SUBST}(\theta, p_k')$$

and from the implication premise $p_1 \wedge ... \wedge p_k$, again by property of universal instantiation entails $\text{SUBST}(\theta, p_1) \wedge ... \wedge \text{SUBST}(\theta, p_k)$, which gives us:

$$\text{SUBST}(\theta, p_1) \wedge ... \wedge \text{SUBST}(\theta, p_k) \implies q$$

8.4 Forward Chaining with FOL

8.4.1 First Order Definite Clauses

Before we get into performing forward chaining for FOL, we shall first introduce the concept of first order definite clauses. These are clauses that are either atomic, or are disjunctions of literals of which exactly one is positive. An implication statement in FOL is also a result of this, and hence a first oder definite clause. With first order definite clauses, the forward chaining algorithm is sound and complete. However in general, this is not always the case, especially with functions, as the algorithm may recursively generate infinitely many new sentences.

8.4.2 Forward Chaining Algorithm

Algorithm 18 FOL-FC-ASK(KB, α) **returns** substitution θ or $fail$

inputs: KB, the
knowledge base in FOL
α, the query
local variables:
new, set of new sentences inferred on each iteration.

while new is not empty **do**
    $new \leftarrow \{\}$
    for each $rule$ **in** KB **do**
        $(p_1 \wedge ... \wedge p_n \implies q) \leftarrow$ STANDARDIZE-VARIABLES($rule$)
        for each θ such that SUBST$(\theta, p_1 \wedge ... \wedge p_n)$ = SUBST$(\theta, p'_1 \wedge ... \wedge p'_n)$ for some $p'_1 \wedge ... \wedge p'_n$ in KB **do**
            $q' \leftarrow$ SUBST(θ, q)
            if q' does not unify with some sentence already in KB or new **then**
                add q' to new
                $\phi \leftarrow$ UNIFY(q', α)
                if ϕ is not $fail$ **then return** ϕ
                end if
            end if
        end for
        add new to KB
    end for
end while
return $false$

8.4.3 Efficient Forward Chaining

Observe that the forward chaining algorithm performs large number of operations, many of which may be unnecessary due to repeated searches. We can improve the efficiency of the FC algorithm for FOL by performing the following tricks:

Matching rules against known facts: We match rules that we want to infer, and it's premise against the facts that we already know in KB. For example, if we wish to infer.

$$Missile(x) \implies Weapon(x)$$

Then we don not need to loop over all facts in KB, but rather only those that unify with $Missile(x)$ or implies $Missile(s)$. Furthermore, consider the rule:

$$Missile(x) \wedge Owns(Nono, x) \implies Sells(West, x, Nono)$$

We can choose the order in which we want to unify the sentences in the premise of the implication rule. This is known as conjunct ordering, and optimal ordering is in NP-Hard. However, we can use good heuristics that we have seen before, such as $MinimumRemainingValues$. If we realise that there are many things that $Nono$ owns, but very little objects that are missiles, then we can choose to select values for $Missiles(x)$ first before unification on $Owns(Nono, x)$.

Incremental Forward Chaining: The FC algorithm still performs many redundant and repeated rule matchings. For example, we have that we know $Missiles(M_1)$ and $Missile(x) \implies Weapon(x)$. In the first round of forward chaining, using the substitution $x \leftarrow M_1$ we can infer $Weapon(M_1)$, which is added to our KB. However, because the algorithm loops over all rules, in future iterations of the algorithm, it will arrive at these 2 sentences and again re-infer what we already know, which is that M_1 is a weapon.

We can avoid such redundant rule matching by making the following observation: *Every new fact inferred on time t must be derived from at least one new fact inferred on iteration $t-1$.* This must be true, because if we use facts that were inferred before $t-1$, then the same information would have be available at time $t-1$ and hence the information would have been inferred then.

This observation leads naturally to an incremental forward-chaining algorithm where at iteration t, we check a rule only if it's premise includes a conjunct p_i that unifies with a newly inferred fact p_i' in the previous iteration. This generates the same facts as the original algorithm, without looping over already known facts over and over again.

Rete Algorithm: When we perform substitutions that only substitute some of the variables in the conjunct, the original algorithm does not keep the partially substituted conjuncts but discards it. This is wasteful because we may possibly substitute in the same values again in future iterations. The rete algorithm rectifies this by keeping partially matched rules to avoid duplicate work in the future.

8.5 Backward Chaining

However, similar to forward chaining for propositional logic, it also suffers from checking of irrelevant facts not related to the goal. Backward chaining solves this problem by only propagating premises backward that are related the goal. The following backward chaining algorithm comprises of 2 functions:

Algorithm 19 Backward Chaining

```
1: procedure FOL-BC-ASK((KB, query)) returns a generator of substitutions
2:     return FOL-BC-OR(KB, query, {})
3: end procedure
4:
5: procedure FOL-BC-OR((KB, goal, θ)) yields a substitution
6:     for each rule (lhs ⟹ rhs) FETCH-RULES-FOR-GOAL(KB, goal) do
7:         (lhs, rhs) ⟹ STANDARDIZE-VARIABLES(lhs, rhs)
8:         for each θ' in FOL-BC-AND(KB, lhs, UNIFY(rhs, goal, θ)) do
9:             yield θ'
10:        end for
11:    end for
12: end procedure
13:
14: procedure FOL-BC-AND((KB, goal, θ)) yields a substitution
15:    if θ = failure then return
16:    else if LENGTH(goals) = 0 then yield θ
17:    else
18:        first, rest ← FIRST(goals), REST(goals)
19:        for each θ' in FOL-BC-OR(KB, SUBST(θ, first), θ) do
20:            for each θ" in FOL-BC-AND(KB, rest, θ') do
21:                yield θ"
22:            end for
23:        end for
24:    end if
25: end procedure
```

Backward chaining functions exactly just as in propositional logic: each each step, we retrieve the rules that is relevant to the premises or query we wish to prove, and find a valid substitution that brings us one step to the goal. The FOL-BC-OR function allows us to find *any* rule that leads to the given query under unification, while in FOL-BC-AND we we wish to substitutions such that all of the premise conjuncts are true.

Similar to DFS, if the search tree for BC-FOL has infinite depth, then the algorithm is not guaranteed to terminate, making it incomplete. Additionally, we may run into repeated searches in cases like biconditionality.

8.6 Resolution in FOL

8.6.1 CNF for First Order Logic

As in the propositional case, first order resolution requires that the sentences be in conjunctive normal form(CNF). The only exception now, is that previously we had literals for propositional

logic sentences, we now have variables in the context of first order logic. We shall see later how to unify CNF sentences containing variables. For example, the sentence:

$$\forall x, [\forall y, Animal(y) \implies Loves(x, y)] \implies [\exists y, Loves(y, x)]$$

Eliminate implications: We transform implications into disjunctions to produce:

$$\forall x, [\neg \forall y, \neg Animal(y) \vee Loves(x, y)] \vee [\exists y, Loves(y, x)]$$

Reduce negations: Using De Morgan's law, we can further reduce the sentence:

$$\forall x, [\neg(\forall y, \neg Animal(y) \vee Loves(x, y))] \vee [\exists y, Loves(y, x)]$$
$$\forall x, [\exists y, \neg\neg Animal(y) \wedge \neg Loves(x, y)] \vee [\exists y, Loves(y, x)]$$
$$\forall x, [\exists y, Animal(y) \wedge \neg Loves(x, y)] \vee [\exists y, Loves(y, x)]$$

Standardize variables: To tell apart from the the variables y in the implication premise and implicant represent different things, we need to standardize them apart by renaming them. Replacing the y in the implicant with z, we get:

$$\forall x, [\exists y, Animal(y) \wedge \neg Loves(x, y)] \vee [\exists z, Loves(z, x)]$$

Skolemization: As before, we cannot merge universal instantiation with existential quantification together directly. To do so, we skolemize variables under existential quantification. Previously, we have seen that for a sentence like:

$$\exists x, \forall y, y \geq x$$

We can skolemize it to become:

$$\forall y, y \geq x_0$$

However, we cannot do the same for something like:

$$\forall y, \exists x, y \geq x$$

Does not equate to $\forall y, y \geq x_0$. Previously when the existential quantifier is outside the universal, the Skolem constant applies for every variable in the universal quantification. However, when the existential quantifier lies inside the universal instantiation, then the variable x may vary and be different depending on y. Hence, we need to replace the existential quantifiable variable with a **Skolem function**. This is exactly the case in the our original example, and replacing y, z with Skolem functions and removing universal instantiation, we get:

$$Animal(F(x)) \wedge \neg Loves(x, F(x)) \vee Loves(G(x), x)$$

by setting $y \leftarrow F(x), z \leftarrow G(x)$. Think of Skolem functions outputting the respective y, z values for the above sentence depending on the variable x.

Distribute disjunctions over conjunctions: Finally, using the distribution rule, we can obtain the sentence in CNF form represented by:

$$[Animal(F(x)) \land Loves(G(x), x)] \land [\neg Loves(x, F(x)) \lor Loves(G(x), x)]$$

Which concludes the process. In fact, every sentence of first-order logic can be converted into an inferentially equivalent sentence, allowing us to then perform resolution on these sentences.

8.6.2 Resolution Inference on FOL sentences

Just as modus ponens, we perform resolution on FOL CNF sentences as we do in propositional logic, with the except of lifting. Firstly, we say that propositional literals are **complementary** if one is the negation of the other, while in first order logic, 2 literals are complementary if one unifies with the *negation* of the other. In other words, we have:

$$\frac{l_1 \lor ... \lor l_k \lor a; m_1 \lor ... \lor m_n \lor \neg b}{\text{SUBST}(\theta, l_1 \lor ... \lor l_k \lor m_1 \lor ... \lor m_n)}$$

where a, b are complementary literals that unify under the substitution θ, and $\text{SUBST}(\theta, a) = \text{SUBST}(\theta, \neg b)$. For example, we resolve:

$$[Animal(F(x)) \land Loves(G(x), x)]; [\neg Loves(u, v) \lor Kills(u, v)]$$

With the substitution $\theta = [u \leftarrow G(x), v \leftarrow x]$ we can see that $Loves(G(x), x)$ and $\neg Loves(u, v)$ are first order complementary literals, and can resolve to form just $Animal(F(x)) \lor Kills(G(x), x)$. This is also known as first order factoring. The rest of the FOL resolution procedure proceeds as per in propositional logic: To prove that $KB \implies \alpha$, where KB is a knowledge base in FOL, we first convert KB to CNF form, and resolve $KB \land \neg\alpha$ to derive a contradiction.