Lecture Notes: Search & Planning in AI

Levi H. S. Lelis
Department of Computing Science
University of Alberta

January 30, 2023

Chapter 1

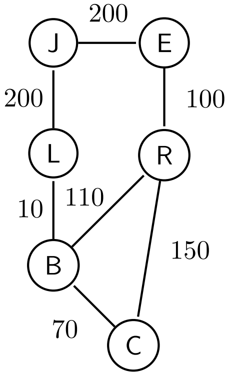
Uninformed Search Algorithms

1.1 State-Space Search Problems

In this lecture we will learn how to model real-world problems as state-space search problems. For example, you might use a GPS to find the shortest path between the university and the airport or maybe the fastest path depending on traffic. The task of finding a path on the city map is a search problem, where we have a starting location (university) and a goal location (airport) and one needs to find a sequence of streets one needs to drive through and the turns one needs to make to reach to goal location.

We call each location the agent assumes in the city map a **state**. The **initial state** is described as "the agent is at the university," while the **goal state** is described as "the agent is at the airport." The **path** from the university to the airport goes through multiple states (e.g., at 117th street), which are connected by **actions** (e.g., make a right turn on Groat road).

We represent the space of states and actions, i.e., the **state space**, as a graph, which is a general structure for representing the relation between entities. In a state space, two states s_1 and s_2 are connected with an edge if there is an action that takes the agent from s_1 to s_2. Consider the simplified map of Alberta in the graph below, where the circles (vertices) represent different states and the lines (edges) represent different actions the agent can take. For example, the state "the agent is in Edmonton" (see vertex E in the graph) is connected through a single action to Jasper (J) and to Red Deer (R). This is because, in this simplified environment, an action allows the agent to drive from Edmonton to Jasper and another action allows the agent to drive from Edmonton to Red Deer.



Each action has a cost, which is represented by the edge weights. In this case it could be the distance between the cities, the time taken in minutes, or the amount of gas the agent spends while driving from one city to the next. Assuming that the costs represent the distance between the cities, we can ask questions such as "What is the shortest path between Edmonton and Banff?" A solution path to this question is "drive to Red Deer, drive to Calgary, drive to Banff." Note that, in this case, we defined a path as a sequence of actions the agent needs to perform to start at the start state and reach the goal state. It is also common to define a path as a sequence of states. In this case, a solution path is "Edmonton, Red Deer, Calgary, Banff."

This solution path is not, however, the **optimal solution** (shortest path). The path above has a cost of $100+150+70 = 320$, while the path "drive to Red Deer, drive to Banff" has the cost of $100+110 = 210$. The path with cost 210 is optimal because there is no other path connecting the two cities whose cost is cheaper than 210. In the next few lectures we will study algorithms for finding optimal and suboptimal solutions to problems such as the one described in this section.

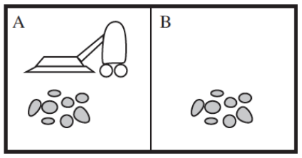
1.1.1 Definition of a State-Space Search Problem

A search problem is defined by a tuple (G, s_0, s_g, T, C), where

- $G = (S, A)$ is a graph with a set of states S and edges A defining the relation between states;
- s_0 in S is the initial state (e.g., "the agent is in Edmonton");
- s_g in S is the goal state (e.g., "the agent is in Banff");
- T is a **transition function** (often also called a successor function) that receives a state s in S and returns a set of states s' that are connected to s with an edge in A, i.e., (s, s') in A. Some state spaces are defined with directed graphs (e.g., one can drive from Edmonton to Red Deer, but not from Red Deer to Edmonton);
- C is a **cost function** that receives an edge in A and returns the cost of the action the edge represents. For example, $C(E, R) = 100$ in our example above, because this is the cost of the edge between Edmonton and Red Deer.

1.1.2 Vacuum Cleaner Example (from Russell & Norvig)

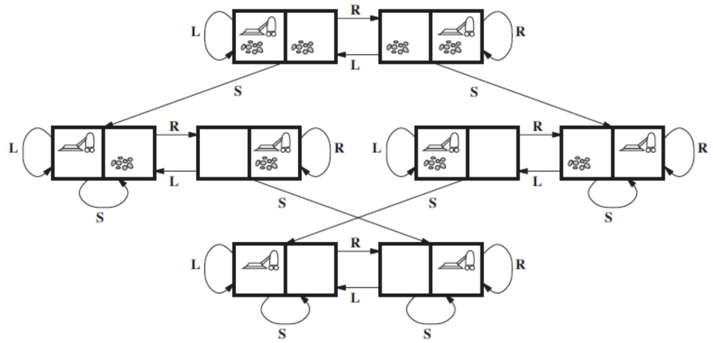
Consider the shortest path search problem depicted in the image below.



There are two rooms in this problem (A and B) and both rooms need to be cleaned. The vacuum robot starts in room A and it can perform the actions: move left (L), move right (R), and suck up dust (S). The effects of each action is what one would expect: if the robot moves right when in A, it will go to B, if it moves left in A, then nothing happens; when it sucks up the dust in a room that needs cleaning, the dust in the room disappears. The goal in this problem is to have both rooms free of dust; the position of the robot is not important as long as both rooms are free of dust.

The optimal solution path to this problem is S, R, S. Assuming unitary costs to each action, the solution path R, L, S, R, S is suboptimal because it requires more actions than the S, R, S solution.

The search space of this vacuum problem is small enough that we can fit it on a single page.

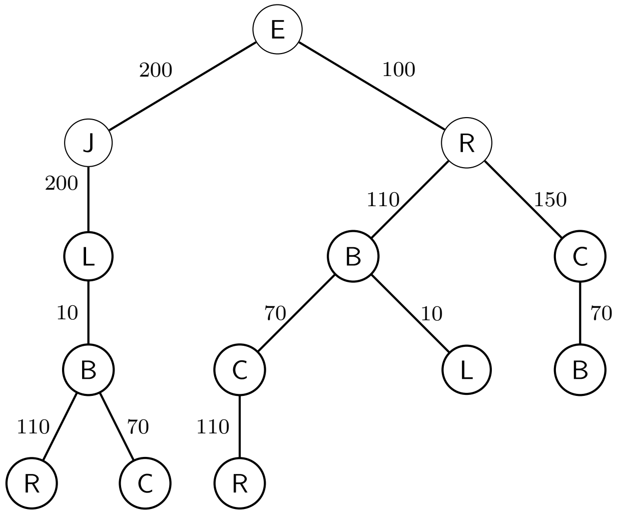


You will notice that this problem has two goal states—see the two states at the bottom of the figure above, they both represent states where both rooms are free of dust. We will see when studying Classical Planning that, for some problems, it is easier to specify a set of goal conditions as opposed to specify a set of goal states. This is because some problems can have a number of goal states that is so large that it can be hard to specify the goal states explicitly. In the example above we could specify the set of goal states by defining the condition that both rooms are clean. A state s is a goal as long as the goal conditions are satisfied at s.

1.2 Search Tree

The algorithms we will study generate a **search tree** to solve a search problem. The tree below shows an example.[1] The tree is rooted at the start state, which in this case is E, the city of Edmonton. The next level of the tree is given by the states that can be reached once the agent applies a single action at E. From E the agent can reach J and R with a single action. We say that J and R are the **children** of E in the tree, and E is the **parent** of J and R. We say that the children of a node n is **generated** when a search algorithm invokes the transition function for n. A node is **expanded** when their children are generated. This all might sound a bit confusing now, but we will get used to all these terms as we study different search algorithms.

[1] Note that this tree isn't the tree of a specific algorithm. The nodes in the tree are somewhat arbitrary to illustrate some of the concepts we will use in this course.



You probably noticed that a few states are repeated in the search tree above. For example, we can reach B with the path E, R, B and with the path E, J, L, B. Ideally the search tree will be as small as possible because the running time of search algorithms is well correlated with the size of the algorithm's tree: a larger tree means that the search algorithm runs for longer, while a smaller tree means that the algorithm is able to find a solution quicker. We will study different ways of handling repeated states in the tree.

The repeated states in the tree can be of two types: **cycles** or **transpositions**.

Cycles. A cycle occurs when the same state appears twice on the same path. For example, the branch R, B, C, R represents a cycle. In the search tree shown above we already eliminated the shortest cycle possible, which is when the parent of a node is generated among its children. For example, E is a neighbor of R in the state space, but it does not appear as a child of the R node in the second level of the tree. This is because E is R's parent. Here we performed **parent pruning**. Parent pruning is easy to implement, therefore it is normally implemented in practice. It is also possible to detect longer cycles, such as the R, B, C, R cycle above. In the worst case, when the cycle is formed by a leaf and the root of the tree, one needs to traverse the entire branch, from the leaf to the root, to detect the cycle. This operation has a non-trivial computational cost as it is linear on the length of the path. The detection of longer cycles isn't normally implemented by traversing the path backward. We will see in our next lecture that we can use memory to remember the states we have visited and thus avoid cycles.

Transpositions. Transpositions occur when multiple paths lead to the same state. For example, the node B on the path E, J, L, B is a transposition of the node B on the path E, R, B. Similarly to cycles, transpositions can be harmful to search because they represent duplicated work. As we will see in our next lecture, transpositions can also be avoided with the use of memory. That is, if the algorithm stores all states visited, then it can avoid visiting states that were visited before.

1.3 Uninformed Search Algorithms

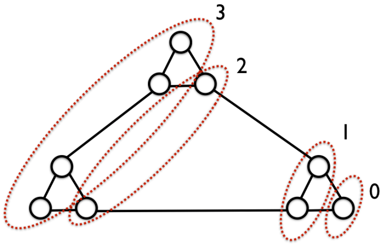
In this lecture we will study Breadth-First Search (BFS) and Dijkstra's algorithm. These algorithms are commonly studied in other courses and they might be a review for many of you. However, the way we present these algorithms is likely different from how you studied them in other courses. We present BFS and Dijkstra's algorithm in a way that makes it easier to learn heuristic search algorithms, a core topic of this course.

In addition to the pedagogical reasons, we will not assume that we can store the state space in memory, as is commonly done in other courses. We provide the search algorithms with start and goal states, a transition function, and a cost function. Note that we do not have to provide the graph encoding the state space. This is because the initial state and the transition function allow us to perform the search without necessarily storing all state space in memory. We begin the search in the start state and we can use the transition function to reach its neighbors, and the neighbors or neighbors, until a goal state is found and returned.

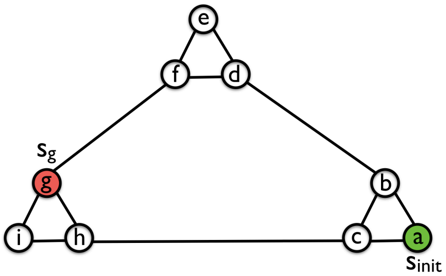
Unless otherwise stated, we will assume that the problems we are solving have a solution. That is, there exists a path connecting the start and goal states.

1.3.1 Breadth-First Search (BFS)

In BFS we expand all nodes X edges away from the start before expanding nodes $X + 1$ edges away from the start. The figure below shows an example, where the state at the bottom-right corner is the start state. The dashed lines show the states at different levels of the search tree. At level 0 we have the start state, at level 1 the children of start, at level 2 the grandchildren, and so on.



BFS enumerates all possible states, level by level, until a goal state is encountered. Let's see a concrete example of BFS. In the graph below s_{init} is the start state and s_g is the goal state.



We will use two data structures to implement BFS: `OPEN` and `CLOSED`. `OPEN` stores all states encountered in search but that were not expanded, while `CLOSED` stores all states encountered in search. `OPEN` and `CLOSED` are initialized with the start state. BFS then removes it from `OPEN` and adds its children to both `OPEN` and `CLOSED`. In every iteration we remove from `OPEN` a state from the level that is currently being expanded. We start expanding states of the next level once there are no more states of the current level in `OPEN`. The `CLOSED` structure serves two purposes. The first is to avoid expanding transpositions and cycles. Since it stores all states encountered in search, if it encounters a repeated state (transposition or cycle), BFS can simply not add it to `OPEN`. The second is to recover the solution path, once one is encountered. We store with each node n in `CLOSED` the information of n's parent in the BFS search tree. Once a goal node is encountered, we can simply follow all these "parent pointers" all the way back to the root of the tree to recover the solution path.

Each line of the figure below shows OPEN and CLOSED for the example above.

OPEN	CLOSED
(a, 0)	(a, 0)
(b, 1) (c, 1)	(a, 0) (b, 1) (c, 1)
(c, 1) (d, 2)	(a, 0) (b, 1) (c, 1) (d, 2)
(d, 2) (h, 2)	(a, 0) (b, 1) (c, 1) (d, 2) (h, 2)
(h, 2) (f, 3) (e, 3)	(a, 0) (b, 1) (c, 1) (d, 2) (h, 2) (f, 3) (e, 3)
(g, 3) (i, 3) (f, 3) (e, 3)	(a, 0) (b, 1) (c, 1) (d, 2) (h, 2) (f, 3) (e, 3) (g, 3) (i, 3)

OPEN and CLOSED start with the pair $(a, 0)$ representing the start state. The value of 0 denotes the level in which a is located. Node $(a, 0)$ is removed from OPEN and expanded; its children, b and c, are stored in OPEN and CLOSED. Next, we expand all nodes from OPEN whose level is 1; we start expanding nodes of level 2 once we finish with all nodes of level 1. CLOSED prevents us from expanding cycles and transpositions. For example, when $(b, 1)$ is expanded, we don't add $(a, 2)$ nor $(c, 2)$ to OPEN. This is because both a and c are in CLOSED, meaning that these states were already encountered at earlier levels of the search. The search stops once the goal s_g is generated, see the boldfaced node in OPEN. As we will see in the pseudocode of BFS, the goal doesn't need to be inserted in OPEN for the search to stop, as shown in the scheme above; BFS stops as soon as the goal is generated.

The following pseudocode shows BFS. If OPEN is implemented with a first-in first-out (FIFO) structure, we don't need to bother adding the level of the nodes, as shown in the example above. This is because the FIFO structure guarantees that the nodes at level X are expanded before BFS expands nodes at level $X + 1$. The CLOSED structure is better implemented as a hash table since it allows us to verify in constant time if a node is stored in CLOSED.

```
def BFS($s_0$, $s_g$, $T$):
  OPEN.append($s_0$)
  CLOSED.add($s_0$)
  while not OPEN.empty():
    $n$ = OPEN.pop()
    for $n'$ in $T(n)$:
      if $n'$ == $s_g$:
        return path between $s_0$ and $n'$
      if $n'$ not in CLOSED:
        OPEN.append($n'$)
        CLOSED.add($n'$)
```

Properties of BFS

When analyzing a search algorithms we will be interested in the following properties: completeness, optimality, and time and memory complexities. We say that an algorithm is **complete** if it is guaranteed to find a solution if one exists. An algorithm is **optimal** if it is guaranteed to find an optimal solution.

It is easy to see that this algorithm is complete because it checks all states encountered. Is it optimal? BFS is optimal as long as one is minimizing the solution length, i.e., the number of actions needed to achieve a goal state (this is the same as minimizing the solution cost when all actions have the cost of 1).

We will make two assumptions for deriving the time and memory complexity of BFS. We will assume that the number of children is fixed for all nodes, which we will denote as b. The number of children is also referred to as the **branching factor** of the problem. We will use the number of nodes generated as a proxy for the algorithm's running time and space requirements.

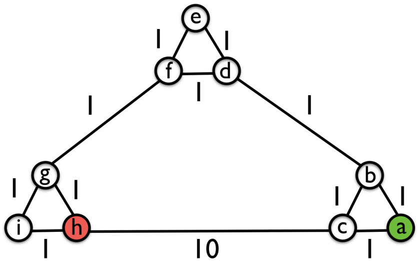
BFS generates b nodes at level 1, b^2 at level 2, b^3 at level 3, and so on. If the solution is encountered at level d of the search tree, then we have the following total number of nodes generated

$$b + b^2 + b^3 + \cdots + b^d \, .$$

The equation above is dominated by the b^d term. So, in terms of big-oh notation, the time complexity of BFS is $O(b^d)$. How about its memory complexity? Since every node BFS generates is added to both `OPEN` and `CLOSED`, the memory complexity is also $O(b^d)$.

BFS Fails to Find Optimal Solutions in General

BFS might fail to find optimal solutions when the action costs aren't unitary. Consider the example shown in the graph below, where a is the start state and h is the goal state.



BFS expands the space by its levels, as shown in the previous example. Thus, it finds the sub-optimal path a, c, h with cost 11. While the optimal solution is a, b, d, f, g, h costs 5.

1.3.2 Dijkstra's Algorithm

Dijkstra's algorithm, which is also referred to as Uniform-Cost Search, can be used to find optimal solutions to problems with varied action costs. Instead of using a FIFO structure to decide the next node to be expanded, Dijkstra's algorithm uses a priority queue and it expands, in each iteration, the node with cheapest cost that was generated but not yet expanded. We denote the cost of a path connecting the root of the tree to node n as the g-value of n, or $g(n)$.

The table below shows `OPEN` and `CLOSED` for Dijkstra's algorithm in the example with non-unitary action costs shown above. Here, each pair denotes a node and its g-value. In contrast with the BFS algorithm, Dijkstra's algorithm only stops when the goal node, h, is removed from `OPEN`. The goal node is first inserted in `OPEN` with a suboptimal cost in iteration 4 of Dijkstra's search. In iteration 9 the algorithm finds a better path to h and it updates its cost from 11 to 5. Like in BFS, we also use the `CLOSED` list to recover the path from the root to a goal state in Dijkstra's. That is why we need to update the information about the parent of a state in `CLOSED` when we encounter a better path to the state. For example, the parent of h was c in iteration 4 and it was updated to g in iteration 8.

We can summarize the differences between Dijkstra's algorithm and BFS as follows.

Iteration	OPEN	CLOSED
1	(a, 0)	(a, 0)
2	(b, 1) (c, 1)	(a, 0) (b, 1) (c, 1)
3	(d, 2) (c, 1)	(a, 0) (b, 1) (c, 1) (d, 2)
4	(d, 2) (h, 11)	(a, 0) (b, 1) (c, 1) (d, 2) (h, 11)
5	(f, 3) (e, 3) (h, 11)	(a, 0) (b, 1) (c, 1) (d, 2) (h, 11) (f, 3) (e, 3)
6	(f, 3) (h, 11)	(a, 0) (b, 1) (c, 1) (d, 2) (h, 11) (f, 3) (e, 3)
7	(g, 4) (h, 11)	(a, 0) (b, 1) (c, 1) (d, 2) (h, 11) (f, 3) (e, 3) (g, 4)
8	(i, 5) (h, 5)	(a, 0) (b, 1) (c, 1) (d, 2) (h, 5) (f, 3) (e, 3) (g, 4) (i, 5)
9	(i, 5)	(a, 0) (b, 1) (c, 1) (d, 2) (h, 5) (f, 3) (e, 3) (g, 4) (i, 5)

- BFS uses a FIFO structure; Dijkstra's algorithm uses a priority queue sorted by the node cost.
- BFS stops the search when the goal is generated; Dijkstra's algorithm stops when the goal is removed from `OPEN`.
- BFS always finds the shortest path (in terms of number of actions) to every node when it is first generated; Dijkstra's algorithm might need to update the path and the cost of nodes in `OPEN`.

Properties of Dijkstra's Algorithm

The algorithm is complete since it considers all nodes encountered during search. It is also optimal, as it expands all cheapest paths before moving on to more expensive ones. It is possible to show that every state, when expanded, it is expanded with its optimal cost (cheapest cost from start to the state).

In order to simplify the time and memory complexity analysis, we assume that the actions have unit costs, such that the optimal solution cost C^* equals the depth d in which the solution is encountered. Since Dijkstra's algorithm stops only when the goal state is removed from `OPEN`, Dijkstra's memory and time complexity is $O(b^{d+1})$. This is because, in the worst case, all nodes at depth d must be expanded as the goal can be the last state expanded at that depth, which means that all nodes at depth $d+1$ are generated.

Implementation Details of Dijkstra's Algorithm

The efficiency of Dijkstra's algorithm depends on the efficiency of two operations: find the node with minimum cost in `OPEN` and verify if a node was already encountered in search by checking whether the state the node represents is in `CLOSED`. `OPEN` is implemented as a heap and `CLOSED` as a hash table. As an example of a concrete implementation of a heap, the **`heapq`** library in Python implements a binary heap, which allows us to insert nodes in `OPEN` in $O(\log(n))$ time, where n is the size of heap. It also allows us to retrieve the cheapest node in $O(1)$ time. However, note that after finding the cheapest node n we also need to remove n from `OPEN` and the complexity for removing it is $O(\log(n))$. There exist other heap implementations such as Fibonacci heap that reduce the complexity for inserting elements in the heap from logarithmic to constant. The hash table allows us to verify if a state was already visited in $O(1)$ time.

A common mistake in implementations of Dijkstra's algorithm is to implement `OPEN` and `CLOSED` as lists. The time complexity for inserting and removing nodes would be constant, but the time required to find the cheapest state would be linear in the size of `OPEN`, which is quite inefficient compared to the logarithmic time offered by the heap. The time complexity for checking if a state was visited in search would also be linear in the size of `CLOSED`, which is much worse than the constant time offered by the hash table.

The pseudocode below shows Dijkstra's algorithm. This implementation adds the same copy of each node to `OPEN` and `CLOSED` (see lines 2 and 3 as well as lines 10 and 11). The trick of adding nodes simultaneously to `OPEN` and `CLOSED` offers an efficient way of checking if the algorithm found a better path to a given state (in our example we initially found a path with cost 11 to h and then later we found a cheaper path with cost 5 to the same state). Since we keep the same copy of the node in both structures,[2] we can use the hash table to check if a better path was found, which can be done in $O(1)$ (see line 13). If we had to implement this operation using the heap, we would have to scan the entire heap, in $O(n)$ time.

You might wonder what happens if the node n' is in `CLOSED` but not in `OPEN` when the check is performed in line 13. If n' is in `CLOSED` but not in `OPEN`, then it means that the node was already expanded. Dijkstra's algorithm guarantees that when a node is expanded, it is expanded with its cheapest value (otherwise it wouldn't guarantee optimal solutions). Thus, if n' is in `CLOSED` but not in `OPEN`, one cannot find a cheaper path to the state n' represents and the second part of the Boolean expression in line 13 will always be false.

```
 1 def dijkstra($s_0$, $s_g$, $T$):
 2   OPEN.append($s_0$)
 3   CLOSED.add($s_0$)
 4   while not OPEN.empty():
 5     $n =$ OPEN.pop()
 6     if $n == s_g$:
 7         return path from $s_0$ to $n$
 8     for $n'$ in $T(n)$:
 9       if $n'$ not in CLOSED:
10         OPEN.append($n'$)
11         CLOSED.add($n'$)
12       #if it has found better path
13       if $n'$ is in CLOSED and $g(n')$ < CLOSED[$n'$].g_value:
14         update $g(n')$ in OPEN and CLOSED
15         update parent of $n'$ in CLOSED
16         #reconstruct entire heap
17         heapify OPEN
```

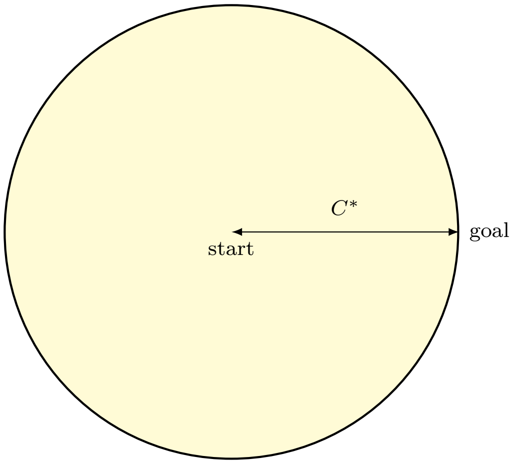
In line 14 we update the value of $g(n')$ in both `OPEN` and `CLOSED` simultaneously. Since we have both structures point to the same object in memory, we use the hash table to access such an object and update its cost. We also update the parent of n' in `CLOSED`. This is because we found a better path from the root of the tree to n', so we must update n''s parent, in case the solution goes through n' and we will need to recover the solution path. Note that updating the cost of n' does not cause any side effects in `CLOSED` as the hash table does not depend on the cost of the nodes. However, the update operation can invalidate the heap structure of `OPEN`. That is why is line 17 we reconstruct the heap structure from scratch, to ensure that the heap is still valid. The "re-heapify" operation is computationally expensive: linear in the size of the heap. However, we only pay this linear cost when a better path is found to a state, which happens much less frequently than the other operations we have discussed so far, such as checking the cost of a node in `CLOSED`.

1.4 Uninformed Bidirectional Search

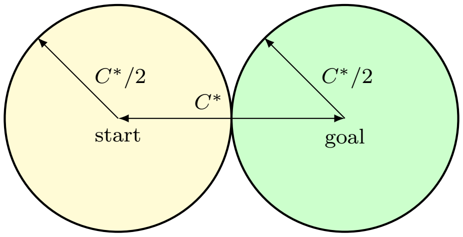
The search algorithms we have studied so far are known as unidirectional search algorithms. This is because the search is performed from the start state toward the goal state—a single direction of search. As a

[2] You can think of both `OPEN` and `CLOSED` storing a pointer to the same object in memory.

cartoonish representation of a unidirectional search algorithm, the circle in the image below represents the set of states Dijkstra's algorithm encounters while searching for an optimal path with cost C^* between start and goal. Depending on the search problem, the number states encountered can grow exponentially with the search depth. For example, if the solution depth $d = C^*$ the memory and time complexities of the algorithm can be $O(b^{C^*})$, where b is the problem's branching factor.

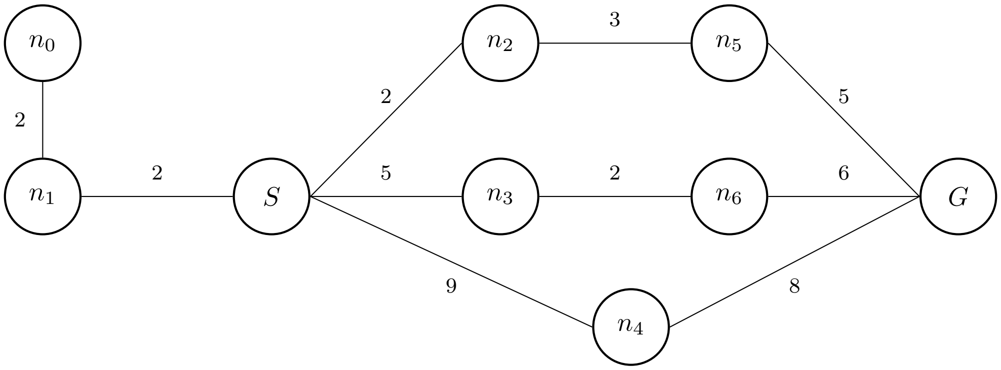


An intuitive idea that might provide exponential savings in terms of search tree size (and thus running time of the search algorithm) is to perform a bidirectional search. That is, we will search simultaneously from both ends of the search: from the start to goal (forward search) and goal to start (backward search). If the searches "meet in the middle," in the sense that both the forward and backward searches do not expand nodes with cost larger $C^*/2$, then we can replace the b^{C^*} unidirectional cost with the much smaller $2 \times b^{\frac{C^*}{2}}$ bidirectional cost. This intuitive idea is illustrated in the image below: once both forward and back searches encounter the same state s, then the search must have a path between start and goal through s.



1.4.1 Bidirectional Brute-Force Search (Bi-BS)

Let's derive a Bidirectional Brute-Force Search (Bi-BS) algorithm through the example below. We will start with a simple idea: we will perform a Dijkstra's search from both ends (start and goal). This means that we will maintain two `OPEN` lists and two `CLOSED` lists, denoted as $\texttt{OPEN}_f$ and $\texttt{OPEN}_b$, and $\texttt{CLOSED}_f$ and $\texttt{CLOSED}_b$—the subscripts f and b stand for "forward" and "backward." In each iteration of the algorithm, it expands the cheapest node in either `OPEN` lists. The `CLOSED` lists will be used as usual, for each of the searches.



Iteration	$\texttt{OPEN}_f$	$\texttt{OPEN}_b$	Notes
1	$(S, 0)$	$(G, 0)$	-
2	$(n_2, 2), (n_1, 2), (n_3, 5), (n_4, 9)$	$(G, 0)$	-
3	$(n_2, 2), (n_1, 2), (n_3, 5), (n_4, 9)$	$(n_5, 5), (n_6, 6), (n_4, 8)$	Solution path through n_4.
4	$(n_1, 2), (n_5, 5), (n_3, 5), (n_4, 9)$	$(n_5, 5), (n_6, 6), (n_4, 8)$	Solution path through n_5.
5	$(n_0, 4), (n_5, 5), (n_3, 5), (n_4, 9)$	$(n_5, 5), (n_6, 6), (n_4, 8)$	-
6	$(n_5, 5), (n_3, 5), (n_4, 9)$	$(n_5, 5), (n_6, 6), (n_4, 8)$	Solution path through n_5 is optimal.

In the graph above, S is the initial state and G is the goal. The table shows the state of both OPEN lists in every iteration of the algorithm; we omit the CLOSED lists for clarity, but we will assume that the nodes are inserted in both OPEN and CLOSED as they are generated, just like how we implemented Dijkstra's algorithm. The search starts by inserting S in $\texttt{OPEN}_f$ and G in $\texttt{OPEN}_b$ with the g-value of 0. We will break ties arbitrarily. For example, in iteration 1 we can expand either S or G because they both have the same cost of 0. In this example we expand S first and its children are added to $\texttt{OPEN}_f$ (see Iteration 2 in the table). In the next iteration we expand G because it is the cheapest in either OPEN lists.

Note that as soon as we expand G (see Iteration 3 in the table) we see the same state, n_4, in both searches (it is in both OPEN lists). This means that we know how to reach n_4 from both S and G. Therefore, we know how to reach G from S. The path S, n_4, G is not optimal, however. One can quickly see that the upper path S, n_2, n_5, G is cheaper than the path the Bi-BS just found. The search has not uncovered such path, however. How does it know then that the path S, n_4, G with cost $9 + 8 = 17$ is not optimal? The answer is in the states stored in both OPEN lists. Nodes n_1 and n_2 are the cheapest nodes in $\texttt{OPEN}_f$, with the cost of 2; node n_5 is the cheapest in $\texttt{OPEN}_b$ with the cost of 5. Bi-BS does not stop searching and return S, n_4, G as the optimal solution path because there could be a path connecting either n_1 or n_2 with n_5 whose cost is cheaper than 17. That is, the sum of the cheapest costs of nodes in $\texttt{OPEN}_f$ and $\texttt{OPEN}_b$ is smaller than the cost of the path found $2 + 5 < 17$, so it is too early to stop as we can still uncover a better path.

In iteration 3, we expand n_2 from $\texttt{OPEN}_f$, which generates n_5. This means that Bi-BS discovered a cheaper solution path as n_5 is present in both OPEN lists. Although the path uncovered, S, n_2, n_5, G, is the optimal solution path, Bi-BS has not proved it yet. This is because the sum of the cheapest g-values in both OPEN lists (2 for the forward search and 5 for the backward search) is less than the solution path, i.e., $2+5 < 2+3+5$. That is, it is possible that there is a path connecting n_1 (in the forward search) and n_5 (in the backward search) that is cheaper than 10, the cost of path S, n_2, n_5, G.

In iteration 4, we expand n_1 from $\texttt{OPEN}_f$, which generates n_0. Bi-BS still does not stop the search because the sum of smallest g-values in $\texttt{OPEN}_f$ and $\texttt{OPEN}_b$ is $4 + 5 = 9$, which is less than the cost of the cheapest path the search found thus far. In iteration 5, once n_0 is expanded, Bi-BS returns S, n_2, n_5, G as the optimal solution path. This is because the sum of the smallest g-values in the OPEN lists, $5 + 5 = 10$, is no longer smaller than the cost of S, n_2, n_5, G. No path that remains to be uncovered by continuing to expand nodes from the OPEN lists will be cheaper than 10. Thus, S, n_2, n_5, G must be optimal.

Bi-BS Stopping Condition

In the example above, Bi-BS stops searching and returns the cheapest solution path p encountered in search when the sum of the minimum g-values in both OPEN lists is no longer smaller than the cost of p. Bi-BS's stopping condition can then be written as follows. Bi-BS stops when:

1. It finds the same state n in both searches and
2. $g_f(n) + g_b(n) \leq g_{minf} + g_{minb}$

Here, $g_f(n)$ and $g_b(n)$ are the g-values of n in the forward and backward searches, respectively; g_{minf} and g_{minb} are the minimum g-values in $\texttt{OPEN}_f$ and $\texttt{OPEN}_b$, respectively. Once a solution path is encountered (Condition 1), the values of g_{minf} and g_{minb} might not allow Bi-BS to stop searching (as in our example above). After Bi-BS expands more nodes, the values of g_{minf} and g_{minb} will go up and Condition 2 might be satisfied. That is why we should check for the stopping condition in every node expansion.

Let us suppose that we know the cost ϵ of the cheapest action in a problem domain. In our example above, the cheapest action costs 2 (e.g., connection between S and n_2). Then we are able to implement a better stopping condition to Bi-BS. Note how in iteration 5 of our example $g_{minf} = 4$ (for n_0) and $g_{min_b} = 5$ (for n_5), while the cost of the cheapest solution found is 10. If Bi-BS is to find a path connecting n_0 to n_5, such a path would have to use at least one action, whose cost is at least 2. Since $g_{minf} + g_{minb} + \epsilon = 11$, then we know that there cannot exist a path cheaper than 10, so Bi-BS could stop and return S, n_2, n_5, G as the optimal solution path. Such a modified stopping condition would have saved us the last iteration of the algorithm in our example. In practice it can save many iterations of search.

In summary, once the cheapest action cost ϵ is known, one can replace Condition 2 above with the following.

$$g_f(n) + g_b(n) \leq g_{minf} + g_{minb} + \epsilon\,.$$

Bi-BS's Pseudocode

The pseudocode below shows an implementation of Bi-BS. We start by adding the initial state s and goal state g to $\texttt{OPEN}_f$ and $\texttt{OPEN}_b$, respectively. We keep in U the cost of the cheapest solution Bi-BS finds in search, which is initialized to ∞. If either $\texttt{OPEN}_f$ or $\texttt{OPEN}_b$ becomes empty before a solution is found, then the problem has no solution (see while loop in line 7). Once the stopping condition is satisfied, the algorithm returns the optimal solution cost (see lines 9 and 10). Note that the pseudocode does not recover the actual path, but only returns the cost of the optimal solution. In every iteration, Bi-BS expands the cheapest node in either list (if statement in line 10 and else statement in line 24). If a state n' is found in both searches (line 13), then Bi-BS updates the value of U, as Bi-BS has encountered a solution path going through n'.

1.5 Linear-Memory Uninformed Search Algorithms

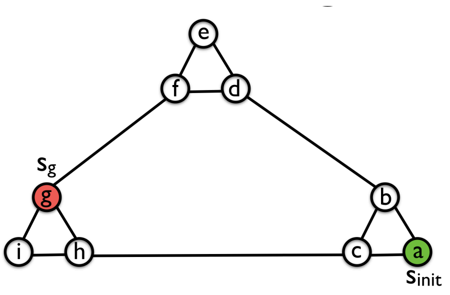
Both BFS and Dijkstra's algorithm store the entire state space in memory in the worst case. Their memory requirement can be prohibitive depending on the problem's size. In this lecture we will study search

```
def Bi-BS(s, g, T):
  OPEN_f.append(s, 0)
  CLOSED_f.add(s)
  OPEN_b.append(g, 0)
  CLOSED_b.add(g)
  U $\leftarrow \infty$
  while not OPEN_f.empty() and not OPEN_b.empty():
    # stopping condition
    if U <= OPEN_f[0].g + OPEN_b[0].g + $\epsilon$:
      return U
    # expanding forward search
    if OPEN_f[0].g < OPEN_b[0].g:
      $n \leftarrow$ OPEN_f.pop()
      for $n'$ in $T(n)$:
        # found a solution path going through $n'$
        if $n'$ is in CLOSED_b:
          U $\leftarrow$ min(U, $g(n')$+CLOSED_b[$n'$].g)
        if $n'$ not in CLOSED_f:
          OPEN_f.append($n'$)
          CLOSED_f.add($n'$)
        # if it has found better path
        if $n'$ is in CLOSED_f and $g(n')$ < CLOSED_f[$n'$].g_value:
          update $g(n')$ in OPEN_f and CLOSED_f
          update parent of $n'$ in CLOSED_f
          # reconstruct heap
          heapify OPEN_f
    else:
      # expanding backward search (exactly as above but with OPEN_b)
```

algorithms with a much lower memory requirement. In particular, we will see algorithms whose memory requirement is only linear in the solution depth—BFS and Dijkstra's algorithm memory requirement is exponential in the solution depth.

1.5.1 Depth-First Search

In Depth-First Search (DFS) we dive as quickly as possible into the search space. Consider the example below, where the initial state is s_{init} and the goal state is s_g. Since our goal is to have a memory-efficient algorithm, we will not employ a `CLOSED` list. In order to obtain to effect of "diving as quickly as possible" into the search space, we will use a stack, which is a "last-in first-out" (LIFO) structure, to implement `OPEN`.



As shown below, where we use parent pruning to eliminate some of the cycles, `OPEN` starts with the initial state with the cost of 0 and, in each iteration of DFS, we remove the last state inserted in the structure and we expand it. For example, once a is removed from `OPEN` we insert c and b. We are assuming b is inserted last (since they are both children of a, the order here is arbitrary) and is thus expanded before c. In the next iteration we expand d because it was the last node inserted in the structure. DFS quickly dives into the search space and finds the solution path a, b, d, f, g, which is suboptimal; the optimal path is a, c, h, g.

`OPEN`
(a, 0)
(b, 1) (c, 1)
(d, 2) (c, 2) (c, 1)
(f, 3) (e, 3) (c, 2) (c, 1)
(g, 4) (d, 4) (e, 3) (c, 2) (c, 1)

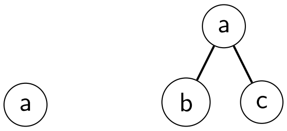
DFS is also not complete. This is because the algorithm will never return from an infinitely deep subtree. Despite not being optimal or complete, DFS's memory requirement is only linear in the search depth: it only stores the nodes on the expanded path and the siblings of those nodes. Can we make this search algorithm optimal and complete while retaining its memory complexity?

If we knew the solution depth we could perform DFS and prune a path that is deeper than the solution depth. For example, in the problem above, if we knew that the solution depth was 3, then we would prune the node g path a, b, d, f, g because that path "went too far." DFS would then backtrack and try a different path. This backtracking version of DFS would eventually find the optimal solution path. The problem is that we normally do not know the optimal solution depth for many problems of interest.

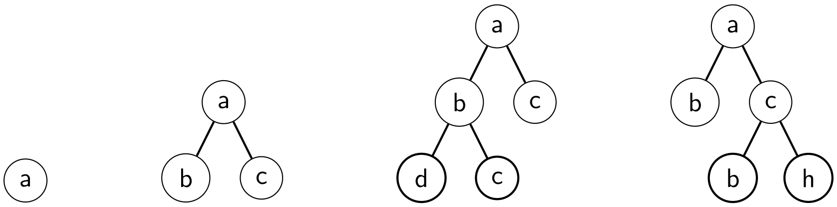
1.5.2 Iterative-Deepening Depth-First Search

Here is how we can discover the optimal solution depth. First we check if the initial state isn't the goal state (if it is, then we are done). If we are not done, then we guess that the solution depth is 1 and enumerate all paths with length 1 (i.e., the children of the start state); if we didn't find the goal, then we guess that the solution depth is 2 and enumerate all paths with length 2. The search stops once the search encounters the goal. This search strategy will find the solution path with its optimal length (i.e., number of actions) because it enumerates all paths with length X before enumerating all paths of length $X + 1$. This search procedure is known as **iterative-deepening depth-first search** (IDDFS).

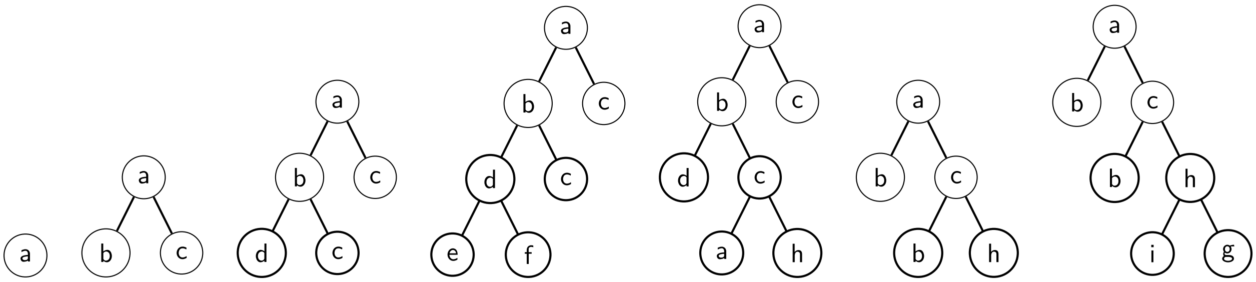
For our example, IDDFS expands the following sequence of paths, after checking that a isn't a goal state. IDDFS sets the bound to be $d = 0$ and it expands the root a, thus generating b and c, which are pruned because they are "too deep" for the current value of d. The two trees below represent the steps search, with only the root a (on the left), and once a is expanded thus generating its children b and c (on the right).



Since we haven't encountered the goal, we increase the bound $d = 1$ and repeat the search. Once again all nodes at levels deeper than 1 are pruned. Also note how we only keep a single path in memory (in addition to the siblings of the nodes on the path). For example, IDDFS first expands the path a, b (see the first three trees below) and it backtracks once it doesn't find the goal and it expands the path a, c, which also doesn't contain the goal. IDDFS has finished searching all paths with length 1, so the goal must be at a deeper depth.



We set the bound $d = 2$ and repeat the search; this time the goal g is encountered and the search terminates. Note that we can terminate as soon as the goal is generated. Thus, we found the goal at level 3 while searching with $d = 2$.



The pseudocode below shows IDDFS. IDDFS receives an initial state, a goal state, and a transition function as input. In this pseudocode the algorithm only returns true or false depending on whether it finds a goal state. In order to return the path we would have to collect the states on the solution path as we unroll the recursive calls once the goal is encountered; we leave this modification as an exercise.

IDDFS calls DFS for increasingly larger depth bounds d. It starts with $d = 0$. If the goal is not encountered, it increases the value of d to 1 so that DFS enumerates all paths of length of 1. The procedure is repeated until a goal state is generated. This algorithm is optimal with respect to the path length (i.e., if all actions cost 1) and complete. IDDFS's memory requirement is linear on the solution depth, $O(d)$. How about the algorithms time complexity? Do the repeated iterations hurt performance?

We will derive IDDFS time complexity while considering the number of nodes the algorithm generates. We will also assume a fixed branching factor b and that the solution is encountered at depth d. The children of start state is generated a number of times equal to the depth d (this can be easily seen in the example above, where b and c are generated 3 times, one for each iteration). The grandchildren of the start state are generated $d - 1$ times, the great-grandchildren $d - 2$ times and so on. The following equation summarizes the total number of nodes IDDFS generates as there are b^2 grandchildren, b^3 great-grandchildren, and so on.

$$db + (d-1)b^2 + (d-2)b^3 + \cdots + b^d$$

The summation above is dominated by b^d, so the time complexity of IDDFS is also $O(b^d)$. Although we are performing multiple iterations and re-generating many nodes, in terms of big-oh notation both BFS and

```
def Bounded-DFS($n$, $s_g$, $T$, $d$):
  if $n == s_g$:
    return True
  if $d < 0$:
    return False
  for $n'$ in $T(n)$:
    if Bounded-DFS($n'$, $s_g$, $T$, $d - 1$):
      return True
  return False

def IDDFS($s_0$, $s_g$, $T$):
  $d = 0$
  while True:
    if Bounded-DFS($s_0$, $s_g$, $T$, $d$):
      return True
    $d = d + 1$
```

IDDFS have the same time complexity. This happens because the search tree grows exponentially with the search depth and the last iteration dominates the algorithm's running time.

IDDFS has the same time complexity as BFS, but a much better memory complexity. While the memory complexity of the former is $O(d)$ the complexity of the latter is $O(b^d)$. Should we always use IDDFS then? The answer is no. The reader might have noticed that IDDFS expands transpositions, which can be a problem if the state space allows for a large number of transpositions in the search tree. This is because IDDFS doesn't use memory to remember the states it has already visited. In our big-oh notation we ignore the transpositions. In practice BFS might be better suited if the problem domain has many transpositions.

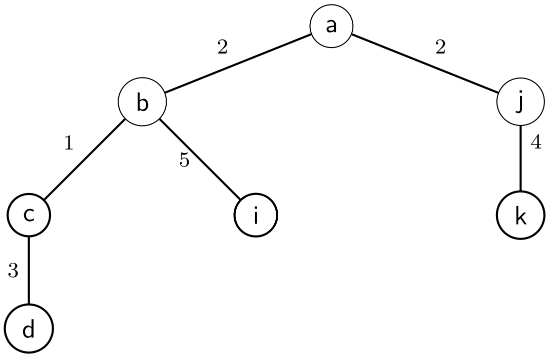
General Cost Functions

How about edge costs? If actions have different costs, then IDDFS doesn't necessarily find optimal solutions. Two small modification to IDDFS guarantees optimal solutions (we still call the algorithm IDDFS).

1. We increase the bound d to the smallest g-value of a node pruned in the previous iteration. This way we guarantee that our increments in cost are conservative and we find the optimal solution.

2. We stop when the goal is expanded, not when it is generated as we did in the unitary-cost case.

Let's consider the following state space where a is the initial state and k is the goal.



Initially the bound d is 0, so IDDFS expands the initial state a and generates b and j, which are pruned because they exceed the bound of 0. The bound is set to 2 for the next iteration because this is the smallest g-value pruned in the previous iteration. For bound $d = 2$, c, i, and k are pruned with g-values of 3, 7, and 6, respectively. The smallest value that is pruned is 3, which becomes IDDFS's next bound value. The search continues until a goal node is found.

Worst Case Due to Reexpansions

If N is the total number of states in the state space, IDDFS can perform N^2 expansions. In the worst case BFS generates "only" N states, which is the entire state space. IDDFS's worst case happens when the algorithm only expands a single new node in each iteration: in the first iteration it expands one node, in the second two nodes, and so on. The total number of nodes IDDFS expands can be written as

$$1 + 2 + 3 + \cdots + N \,,$$

which is $O(N^2)$.

The time complexity analysis that matched the time complexity of BFS implicitly assumed that the tree grows exponentially from iteration to iteration. In the worst-case scenario, the tree grows very slowly from one iteration to the next (exactly by one new state per iteration). This quadratic behavior of IDDFS was recently solved through an algorithm called IBEX, which is out of the scope of this course.

While the exact quadratic behavior is rare in practice, there are problems in which the re-expansion of nodes can be quite harmful for IDDFS. This is particularly true in problem domains with a large diversity of costs. For example, in map-based pathfinding it is common to assign the cost of 1 to moves in the four cardinal directions and of $\sqrt{2}$ to diagonal moves. It is then easy to see that the search will encounter large diversity of costs. In this setting the IDDFS search tree grows very slowly from iteration to iteration, which hurts the algorithm's performance due to the large number of re-expansions.

IDDFS is also easily hurt by a large number of transpositions in map-based pathfinding problems. Consider for example a grid of size 128×128 in a map-based pathfinding problem. There are at most $128 \times 128 = 16,384$ distinct states in the state space. Since IDDFS isn't able to detect transpositions, if we assume the branching factor to be 3, then the search tree with depth 50 has 7.17×10^{23} leaf nodes. Although the size of the space is quadratic with the size of the map, the search tree grows exponentially due to the duplicated nodes. We will see how to alleviate the problem of transpositions in the next lecture.

1.6 Transposition Tables

BFS and Dijkstra's algorithm have memory complexities that are exponential on the solution depth, while IDDFS's memory complexity is only linear on the solution depth. While BFS and Dijkstra's algorithm do not suffer from transpositions because they are eliminated in search, IDDFS can be affected by a large number of transpositions. A structure named **transposition table** can be used to alleviate the IDDFS problems with transpositions while using a limited amount of memory.

A transposition table is a hash table used to store states visited in search. The amount of memory used with the transposition table can be limited. If the search algorithm fills the memory devoted to the table, the search can either replace states already in the table with a new state s or simply ignore s. Due to the limited amount of memory, transposition tables might not be able to detect all transpositions encountered in search. The nature of depth-first search also prevents us from detecting transitions even if the transposition table has memory available to store nodes, as we will see in the implementations below. Nevertheless, a transposition table can still be quite effective to avoid re-expanding transpositions. We will consider three implementations of transposition tables, where each implementation fixes a problem in the previous implementation.

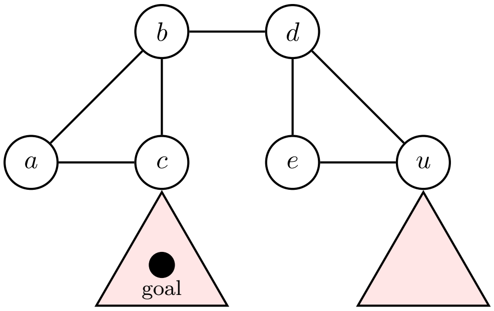
Implementation 1

In this implementation we assume that we start each iteration of IDDFS with an empty transposition table.

Let TT be a transposition table. Before expanding node n:

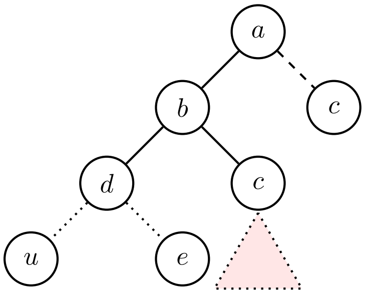
- if n is not in the TT:
  - add n to the TT
  - expand n

The implementation above can possibly prune nodes on the optimal solution path. Consider example below, where a is the initial state and the goal is somewhere in the subtree rooted at c, represented by the triangle.



The tree below shows the nodes IDDFS with a transposition table expands and prunes while searching with a cost bound of 2. In the tree, dotted edges represent nodes pruned due to the cost bound (e.g., u and e are pruned because $g(u) = g(e) = 3 > 2$); the dashed line represents a node pruned due to the transposition table. In this example, IDDFS expands a and adds it to the transposition table. Assuming IDDFS goes left first in this example, b is the next state to be expanded and added to the table; then d is expanded and added to the table; d's children are pruned because they exceed the cost bound; the algorithm backtracks to c, which is expanded and added to the table; c's children are pruned due to the cost bound. Once IDDFS backtracks to a to expand its child c, it checks that c already is in the transposition table and c is pruned. The path a, c leads to the optimal solution, but it is pruned from search. How can we fix this problem?

TT
a
b
d
c



Implementation 2

We fix the problem illustrated in the example above by adding the cost of the node in the transposition table. That way, when the first c is expanded we store its g-value of 2. Later, when we encounter the optimal path to c, we know that we have to expand it because it has a cheaper cost than the previously expanded path.

Implementation 2 can be written as follows:

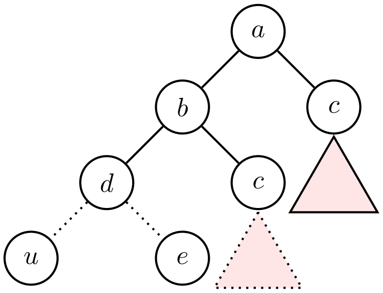
Before expanding node n:

- (if n is not in the TT) or (if n is in the TT and $g(n) < TT[n].g$):
  - add (or update) $[n, g(n)]$ to the TT
  - expand n

Here, the expression $g(n) < TT[n].g$ returns true if the g-value of the node we encountered is smaller than the g-value of the node n in the transposition table, denoted by $TT[n].g$.

The figure below illustrates the previous example with the new implementation. Again considering the cost bout of 2, the state c is added with the cost of 2. When IDDFS encounters c with g-value of 1, it expands it again as now it has found a cheaper path to c. The g-value of c is then updated in the transposition table from 2 to 1. The table below shows the state of the table after completing the search with cost bound of 2.

TT
a, 0
b, 1
d, 2
c, ~~2~~ 1



Note that if the goal is not found in the iteration with cost bound of 2, then IDDFS increases the bound to 3 and repeat the search. Once again the two nodes representing state c will be expanded. This is wasteful computation as we know from the iteration with cost bound of 2 that the optimal cost to c is of cost 1 and not 2. Can we avoid expanding the c with g-value of 2 in the iteration with cost bound 3?

Implementation 3

The iteration with cost bound 2 should inform the search algorithm that there exists a node representing state c whose g-cost is 1, so that the search does not expand the c node with g-value of 2 in later iterations. This can be achieved by making two modifications to Implementation 2. First, we will keep the information stored in the transposition table across the iterations. Second, we will add the bound value to the transposition table. That way, in the iteration with cost bound of 3, we can ignore the c node with g-value of 2 because the transposition table will contain the tuple $(c, 1, 2)$ indicating that a node representing state c was encountered with the g-value of 1 in iteration with cost bound of 2. Thus, we know that a c node with g-value of 1 will eventually appear in the iteration with cost bound of 3 (so the search can prune all nodes representing c whose cost is larger than 1). We update the tuple in the transposition table from $(c, 1, 2)$ to $(c, 1, 3)$ once IDDFS expands c with g-value of 1 in the iteration with cost bound of 3. That way we will know that c was expanded in the current iteration and the search can ignore other c nodes we encounter later in iteration.

Implementation 3 can be written as follows:

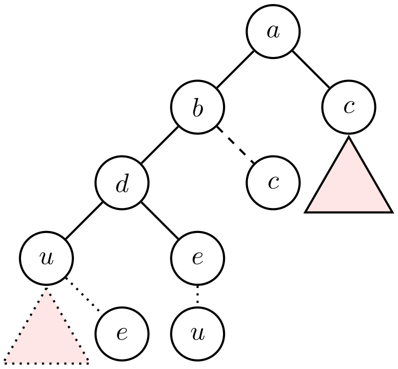
Before expanding node n in an iteration with cost bound d:

- if n is not in the TT or
  if n is in the TT and $g(n) < TT[n].g$ or
  if n is in the TT and $g(n) = TT[n].g$ and $d > TT[n].d$:
  - add/replace $[n, g(n), d]$ to the TT
  - expand n

Where the condition $d > TT[n].d$ returns true if the node n was not expanded in the current iteration.

TT (before search)
a, 0, 2
b, 1, 2
d, 2, 2
c, 1, 2

TT (after search)
a, 0, ~~2~~ 3
b, 1, ~~2~~ 3
d, 2, ~~2~~ 3
c, 1, ~~2~~ 3
u, 3, 3
3, 3, 3



In the image above, the table on the left shows the transposition table before starting the search with cost bound of 3 and the table on the right shows the transposition table after expanding node c. The node c with g-value of 1 is not expanded due to the transposition table. This is because all three conditions of Implementation 3 returns false: the state is in the table (first condition is false); the g-value of c in the table is smaller than $g(c)$ (second condition is false); the g-value of c is not equal to the g-value of c in the table (third condition is false). The node c with g-value of 1 is expanded because the third condition is true: c is in the table, g-value of c is equal to its g-value in the table, and the current cost bound 3 is larger than the cost bound in the table, which is initially 2.

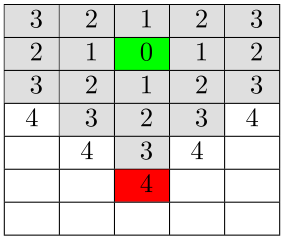
Chapter 2

Informed Search Algorithms

2.1 Heuristic Search

In the previous chapter we studied uninformed algorithms for solving state-space search problems. The algorithms are called uninformed because they do not use the information of the goal state (or goal conditions) to guide their search to a solution path.

The grid below shows an example of how Dijkstra's algorithm, an uninformed search, can be inefficient. The number in each cell shows the g-value of each state, with the agent starting at the green cell, with g-value of 0. The goal is the reach the red cell, marked with g-value of 4. The cells marked in gray denote the states Dijkstra's algorithm needs to expand to find a solution path. In this example we are showing the case where the first state with g-value of 4 expanded is the goal state. Dijkstra's algorithm expands states equally in all directions from the initial state; it forms a circle around the initial state that grows as the algorithm expands more states. This is clearly wasteful as the algorithm explores states that might be far from the goal. In the example above, Dijkstra's algorithm expands the states at the top-left and at top-right corners of the map, despite the goal being in the opposite direction. In this lecture we will see how a **heuristic function**, which is a function that estimates the cost-to-go from a state to the goal, can be used to speed up the search by preventing the search from exploring potentially unpromising parts of the search space.



3	2	1	2	3
2	1	0	1	2
3	2	1	2	3
4	3	2	3	4
	4	3	4	
		4		

2.1.1 Heuristic Function

The heuristic function does need to be perfect (i.e., provide the exact cost-to-go to the goal) to speed up the search. Often inaccurate heuristic functions are able to dramatically reduce the search time. For grid-based pathfinding problems such as the one above, we can define a heuristic function by ignoring all obstacles on the map. The grid below computes the distance between the cell highlighted in red and all the other cells in the space (assuming the four cardinal moves). We denote the heuristic value of state s as $h(s)$.

3	2	1	2	3
2	1	0	1	2
3	2	1	2	3
4	3	2	3	4
5	4	3	4	5
6	5	4	5	6

The heuristic shown in the grid is known as Manhattan distance and can be easily computed given two states on the grid. For example, if the goal state is the cell with coordinates $(5, 4)$ and we want to estimate the distance the agent needs to traverse from state $(1, 2)$ to the goal, then we simply compute $|1-5|+|2-4| = 6$, which is the sum of the absolute differences between the two coordinates in the x and y coordinates of the map. Note that if the map had obstacles, the distance could be larger than 6; the heuristic provides only an estimate of the cost-to-go.

2.1.2 A*

We now have two values we can use to guide the search: $g(s)$, which is the cost from the start to state s, and $h(s)$, which is the estimated cost-to-go from s to the goal. In every iteration, Dijkstra's algorithm expands the state in `OPEN` with smallest g-value. What should we do with h? We will add g and h resulting in the f-value of a state: $f(s) = g(s) + h(s)$. The value f of a state provides an estimate for the cost of a solution that goes through s. The algorithm that sorts the nodes in `OPEN` by their f-value is known as A*. Let's consider the example below.

		6		
	6	4	6	
	6	4	6	
	6	4	6	
	6	4	6	
	6	4	6	

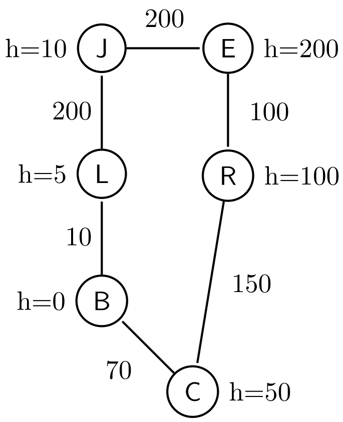
This is the same search problem shown in the first grid of this lecture. The numbers in the cells are the f-values of the states. The initial state s_0 is marked in green and it has the f-value of $f(s_0) = g(s_0) + h(s_0) = 0 + 4 = 4$. Like Dijkstra's algorithm, the A* search starts with only the initial state in `OPEN` and, in every iteration, it expands the state with the smallest f-value. Once s_0 is expanded, we add to `OPEN` all four states that can be reached from s_0 with a single action. You will notice in the grid above that the states above, to the right and to the left of s_0 have the f-value of 6 (their f-value is $1 + 5 = 6$), while the state below s_0 has the f-value of 4 (its f-value is $1 + 3 = 4$). Naturally, in the next iteration, A* will expand the state with f-value of 4, which generates two states with f-value of 6 and another state with f-value of 4. This process continues until the goal is expanded and a solution path is returned.

You should now compare the gray cells in the example for Dijkstra's algorithm with the gray cells in the example for A* algorithm. You will notice that A* focuses its search in the direction of the goal. This is possible thanks to the heuristic function. Dijkstra's algorithm treats equally the states above and below s_0 as they both have the g-value of 1. By contrast, A* prefers the state below s_0 because the estimated cost of a solution going through that state is smaller than the estimated cost of a solution going through the state above s_0. This makes sense. If the agent moves to the state above s_0, it will be moving away from the goal.

In this example A* goes directly to the goal because it is employing a perfect heuristic function: the heuristic function assumes that there are no obstacles on the map and indeed there are no obstacles on the map. What if the heuristic function is not perfect? This is what we consider in the example below, where the solid cell represents a wall that the agent cannot traverse. A* expands all states with f-value of 4, thus making quick progress towards the goal, until it reaches the wall. This is when A* expands the states with f-value of 6, until it finds a path around the wall. Even in this example where the heuristic function is not perfect, we can see by the pattern of gray cells that A* is focusing its search downwards. In particular, A* does not expand states at the top corners of the map, as those states are not deemed as promising by the f-function.

	8	6	8	
8	6	4	6	8
8	6	4	6	8
8	6	4	6	8
8	6	■	6	8
8	6	6	6	8
	8	8	8	

Let us consider an example where we see how `OPEN` and `CLOSED` are used in the A* search. The numbers by the edges in the graph below represent the cost of each action and the h-values are written by the nodes in the graph. The start state is E and the goal state is B.



We start by inserting E in `OPEN` and `CLOSED`. Then, in every iteration we remove the node from `OPEN` with the cheapest f-value. Like Dijkstra's algorithm, the search stops when the goal state is expanded (see B in boldface below).

`OPEN`	`CLOSED`
(E, 200)	(E, 200)
(R, 200), (J, 210)	(E, 200), (R, 200), (J, 210)
(J, 210), (C, 300)	(E, 200), (R, 200), (J, 210), (C, 300)
(C, 300), (L, 405)	(E, 200), (R, 200), (J, 210), (C, 300), (L, 405)
(B, 320), (L, 405)	(E, 200), (R, 200), (J, 210), (C, 300), (L, 405), **(B, 320)**

A* is a complete algorithm because it inserts into OPEN all states generated in search, thus trying all paths encountered in search.

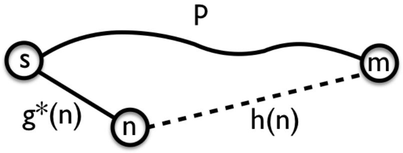
Admissibility of Heuristic Functions

The algorithm is guaranteed to be find optimal solutions if the heuristic function never overestimates the optimal solution cost. That is,

$$h(s) \leq h^*(s) \text{ for all states } s\,.$$

Here, $h^*(s)$ is the optimal solution cost of state s. If the inequality above is satisfied, then we say that the heuristic is **admissible**. The admissibility condition for optimal solutions is sufficient, but not necessary. This means that A* might still find optimal solutions even if using an inadmissible heuristic.

Let us see a sketch of a proof for the admissibility property. Consider the general scenario depicted in the image below. Here, s is in the initial state and m is the goal state. The path at the top, s to m, is suboptimal and has the cost of P. The path at the bottom, going through n, is optimal. We denote the cost of the optimal path between s and the state n as $g^*(n)$. Since the optimal path between s and m goes through n, then the cost between s and n must also be optimal, thus the cost is $g^*(n)$.



Now suppose that at a given iteration A* has both n with the f-value of $g^*(n)+h(n)$ and m with the f-value of P in OPEN. A* retrieves the optimal path only if n is expanded before m, as the algorithm terminates as soon as the goal is expanded. Thus, A* returns the optimal path if

$$\begin{aligned} g^*(n) + h(n) &< P \\ h(n) &< P - g^*(n) \end{aligned}$$

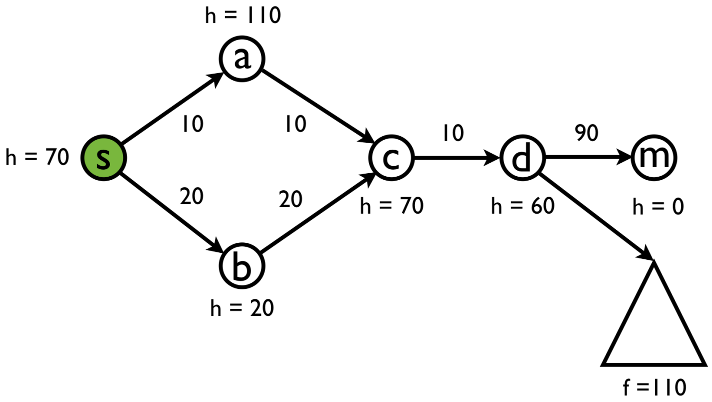
This condition is not helpful because it depends on the value of P. In order to obtain a meaningful property, we will use the fact that the path going through n is optimal, which allows us to write the following.

$$\begin{aligned} g^*(n) + h^*(n) &< P \\ h^*(n) &< P - g^*(n) \end{aligned}$$

If $h(n) \leq h^*(n)$, then we have that $h(n) < P - g^*(n)$, which is what we were looking for. The property $h(n) \leq h^*(n)$ is admissibility. We can now use the reasoning explained above with an inductive argument for all nodes n along the optimal path. That is, if the heuristic is admissible, then all nodes along the optimal path will be expanded before the goal node m is expanded through a suboptimal path.

Consistency of Heuristic Functions

Let us consider the following search problem, where s is the start state and m the goal state. The triangle represents a large subtree where all nodes have f-value of 110. A* behaves as described in the table below.



OPEN	CLOSED
(s, 70)	(s, 70)
(b, 40) (a, 120)	(s, 70) (b, 40) (a, 120)
(c, 110) (a, 120)	(s, 70) (b, 40) (a, 120) (c, 110)
(d, 110) (a, 120)	(s, 70) (b, 40) (a, 120) (c, 110) (d, 110)
(subtree, 110) (a, 120) (m, 140)	(s, 70) (b, 40) (a, 120) (c, 110) (d, 110) (subtree, 110) (m, 140)
(a, 120) (m, 140)	(s, 70) (b, 40) (a, 120) (c, 110) (d, 110) (subtree, 110) (m, 140)
(c, 90) (m, 140)	(s, 70) (b, 40) (a, 120) **(c, 90)** (d, 110) (subtree, 110) (m, 140)
(d, 90) (m, 140)	(s, 70) (b, 40) (a, 120) (c, 90) **(d, 90)** (subtree, 110) (m, 140)
(subtree, <110) (m, 140)	(s, 70) (b, 40) (a, 120) (c, 90) (d, 90) **(subtree, <110)** (m, 140)
(m, 120)	(s, 70) (b, 40) (a, 120) (c, 90) (d, 90) (subtree, <110) **(m, 120)**

A* expands the bottom path s, b, c, d as well as the subtree with nodes with f-value of 110. The goal is inserted in OPEN with the cost of 140, which is suboptimal (the optimal path goes through a, not b). Only after node a is expanded that A* discovers the optimal path. In this example this means that c, d and the subtree represented by the triangle need to be re-opened (re-inserted in OPEN) and re-expanded. The nodes that are re-opened are highlighted in bold in the table. The re-expansion of nodes can significantly affect A*'s performance (e.g., the subtree with nodes with f-value of 110 could be very large). In the worst case A* can expand 2^N nodes, where N is the number of nodes Dijkstra's algorithm expands.

The re-expansion of nodes from the example above points to another modification that we need to make to Dijkstra's pseudocode to transform it into an A* pseudocode. In addition to changing the cost function in which OPEN is sorted, we need to re-open nodes if we find a better path to them.

There is a property of the heuristic function that prevents node re-opening entirely. If the heuristic function is **consistent** A* will never re-open nodes. For any pair of nodes n and n', let $\text{cost}(n, n')$ be the cost of the cheapest path connecting n and n' in the search space. We say that a heuristic function h is consistent if the following inequality holds

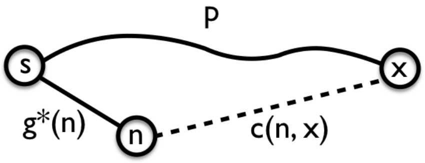
$$h(n) - h(n') \leq \text{cost}(n, n') \, .$$

Let us now look at the example above and look for a pair of nodes for which this inequality does not hold. The cheapest path between a and c is the edge connecting them with the cost of 10, which gives us the following

$$\begin{aligned} h(a) - h(c) &\leq \text{cost}(a, c) \\ 110 - 70 &\leq 10 \\ 40 &\leq 10 \text{ (this is false)} \end{aligned}$$

Since the inequality does not hold, we say that the heuristic is inconsistent. Here is an intuitive way of thinking about inconsistency. When A* visited node a, the heuristic informed the algorithm that the estimated cost-to-go was 110. Then, A* reaches node c while paying a cost of 10 for the action connecting a to c and the heuristic informs the algorithm that the estimated cost-to-go is 70. The heuristic is being inconsistent because in the previous node it estimated a cost-to-go of 110 and the algorithm paid a cost of only 10 to receive a new estimate of 70. If the previous estimate was consistent, then we know that an admissible estimated cost-to-go from c should be at least $110 - 10 = 100$.

Let us see a sketch of a proof showing that if the heuristic is consistent, then A* never re-opens nodes. We will consider the general case shown in the image below, where s is the start state, x is a node encountered during search through a suboptimal path with cost P (path at the top) as well as an optimal path that goes through node n with the cost $g^*(n) + c(n, x)$ (path at the bottom).



A* will not have to re-open x if it is removed from `OPEN` with its optimal g^*-value, which is given by the path at the bottom. If `OPEN` contains n with f-value of $g^*(n) + h(n)$ and x with the f-value of $P + h(x)$, in order to avoid re-opening x we need to have that

$$\begin{aligned} g^*(n) + h(n) &< P + h(x) \\ h(n) - h(x) &< P - g^*(n) \end{aligned}$$

We also have that $g^*(n) + c(n, x) < P$, where $c(n, x)$ is the optimal cost between n and x, because the path at the bottom is the optimal path. This inequality can be rewritten as $c(n, x) < P - g^*(n)$. Thus, if the heuristic is consistent, namely, that $h(n) - h(x) \leq c(n, x)$, then we have that $h(n) - h(x) < P - g^*(n)$, which is the property we needed. Similarly to the admissibility proof, we can make an inductive argument that the reasoning discussed above applies to all nodes along the optimal path between s and x. That way, if h is consistent, then x is guaranteed to be expanded with its optimal $g^*(x)$ value, which avoids re-opening x.

We will end this subsection by showing that consistency implies in admissibility. We start with the consistency definition. For any pair of states n and n'.

$$\begin{aligned} h(n) - h(n') &\leq c(n, n') \\ h(n) - h(s_g) &\leq c(n, s_g) \text{ (consistency also holds for the goal } s_g\text{)} \\ h(n) &\leq h^*(n) \text{ because } h(s_g) = 0 \text{ and } c(n, s_g) = h^*(n) \end{aligned}$$

Summary

In summary, A* algorithm can be implemented by changing the pseudocode of Dijkstra's algorithm as follows.

- Sort the nodes in `OPEN` according to the f-value of the nodes.
- If a node is expanded with a suboptimal cost, we need to re-open the node when a better path is found.

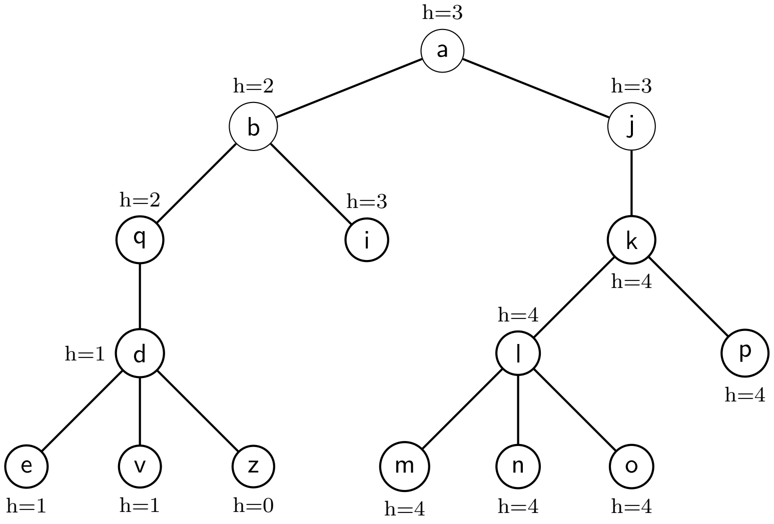
The second item in the list above does not have to be implemented if the heuristic employed is consistent.

2.2 Iterative Deepening A*

Similarly to how Dijkstra's algorithm can be modified to employ a heuristic function to guide the search, we will also modify IDDFS to employ a heuristic function to speed up its search. The resulting algorithm is known as Iterative-Deepening A* (IDA*).

In IDDFS we start with the cost bound d of zero, which is the smallest value possible. If an admissible heuristic is available, then we can safely use the initial cost bound of $h(s_0)$ for start state s_0. Recall that the f-value of a node n estimates the cost of a solution going through n. Thus, if $f(n) > d$, then n can be pruned if the cost bound of a given iteration is d. Similarly to IDDFS, the next d-value of IDA* is given by the smallest f-value of a node that was pruned in the previous iteration. This is it for IDA*!

Like A*, IDA* will go deeper in the branches of the tree with nodes with smaller f-values as they are deemed as promising by the heuristic function. Branches with larger f-values will be pruned as they exceed the cost bound d. Like IDDFS, IDA*'s memory requirement is also linear on the solution depth and it might suffer from a large number of transpositions. We can also implement IDA* with a transposition table, as we have discussed in the IDDFS lecture. Let's see an example of IDA* in action.



The tree above shows the search space where a is the start state and z is the goal state. The initial cost bound is set to 3 and let's assume that the children of a node are searched from left to right. All actions cost 1 in this example. Naturally, node a is expanded because $f(a) = 0 + 3 \leq 3$; b is also expanded because $f(b) = 1 + 2 \leq 3$; q and i are pruned because their f-value exceeds the cost bound of 3. That is when IDA* backtracks from the left subtree and it visits j, which is also pruned because $f(j) = 1 + 3 = 4 > 3$. At this time IDA* has finished the first iteration and it sets the next cost bound to the smallest f-value of a node pruned in the previous iteration. Such a value is given by the f-value of either q or j, which is 4. This procedure is repeated until IDA* uncovers the solution path a, b, q, d, z.

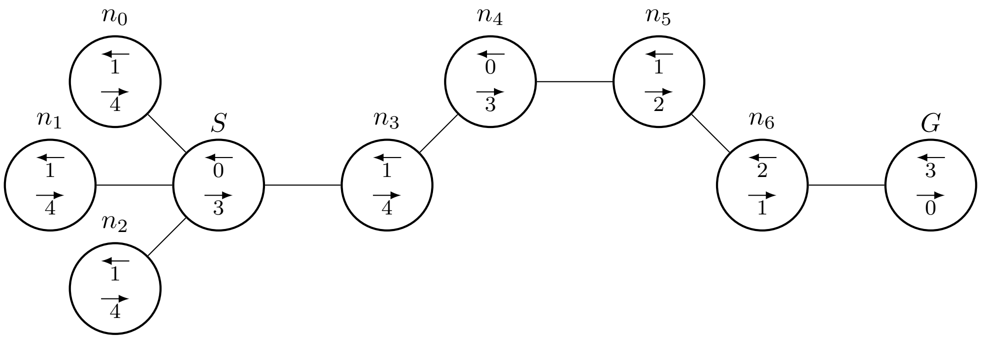
The admissibility proof we showed for A* also applies to IDA*. Naturally, IDA* doesn't suffer from node re-openings because it does not employ an OPEN list. In fact, in practice, IDA* can be quite effective with inconsistent heuristics. This is not, however, a topic covered in this course.

2.3 Bidirectional A*

We are going to combine two ideas we have already explored into a single algorithm: search from both ends of the search problem (start and goal) and use a heuristic function to guide these searches. Our initial strategy will be similar to what we did for deriving the Bi-BS algorithm. Instead of using a Dijkstra's search from both ends, we will use an A* search from both ends. This algorithm is known as Bidirectional A* (Bi-A*). In Bi-A*, $\texttt{OPEN}_f$ and $\texttt{OPEN}_b$ are initialized with the initial state and the goal state, respectively. The f-values used to sort the nodes in $\texttt{OPEN}_f$ will be computed with a heuristic function that measures the cost-to-go from a given state to the goal; the f-values used to sort the nodes in $\texttt{OPEN}_b$ will be computed with a heuristic function that measures the cost-to-go from a given state to the start. In every iteration of search we expand the state with lowest f-value in either OPEN list.

Similarly to Bi-BS, Bi-A* encounters a solution path once a state is visited in both searches. Recall that the f-value of a node n is an estimated cost of a solution going through n. If the heuristic function is admissible, which we assume it to be, the f-value is a lower bound on the cost of a solution going through n. If the cheapest solution path Bi-A* finds in search has cost less or equal to the minimum f-value in either list, then no solution going through any of the open nodes (in either direction) can be cheaper than the cost of Bi-A*'s path, which must be optimal.

Let us consider the example shown in the graph below, where S is the initial state and G the goal state.[1] The numbers under the left-pointing arrows show the backward heuristic value of each node; the numbers under the right-pointing arrows show the forward heuristic value of each node.



The table above shows the state of $\texttt{OPEN}_f$ and $\texttt{OPEN}_b$ during the execution of Bi-A* for this problem. The start and goal states are inserted in $\texttt{OPEN}_f$ and $\texttt{OPEN}_b$, respectively, in iteration 1. Let us assume that ties are broken by favoring the forward search, so that S is expanded in iteration 2. In the following iterations, Bi-A* expands the chain G, n_6, n_5 and n_4 through the backward search. Bi-A* stops once n_3 is generated in the backward search as it was also generated in the forward search. The path going through n_3 has cost of $1 + 4 = 5$, which is optimal because the lowest f-value in $\texttt{OPEN}_f$ matches the value of 5.

In the example above you will notice that the forward search expanded nodes with g-value up to 0, while the backward search expanded nodes with g-value up to 3. You might recall that the bidirectional search's

[1] This example is a modified version of the example from the paper "MM: A bidirectional search algorithm that is guaranteed to meet in the middle" by R. Holte et al.

Iteration	$\texttt{OPEN}_f$	$\texttt{OPEN}_b$	Notes
1	$(S, 3)$	$(G, 3)$	-
2	$(n_0, 5)$, $(n_1, 5)$, $(n_2, 5)$, $(n_3, 5)$	$(G, 3)$	-
3	$(n_0, 5)$, $(n_1, 5)$, $(n_2, 5)$, $(n_3, 5)$	$(n_6, 3)$	-
4	$(n_0, 5)$, $(n_1, 5)$, $(n_2, 5)$, $(n_3, 5)$	$(n_5, 3)$	-
5	$(n_0, 5)$, $(n_1, 5)$, $(n_2, 5)$, $(n_3, 5)$	$(n_4, 3)$	-
6	$(n_0, 5)$, $(n_1, 5)$, $(n_2, 5)$, $(n_3, 5)$	$(n_3, 3)$	Optimal solution through n_3.

promise of exponential speed ups depends on dividing the usual unidirectional search with cost b^d with two searches with cost $b^{d/2}$ each. The example above shows that Bi-A* might divide the searches asymmetrically, where one of the searches can potentially perform many more expansions than the other search.

One way of attempting to remedy this search asymmetry is by enforcing that the two searches "meet in the middle". The forward and backward searches meet in the middle if they only expand nodes whose g-values are never larger than $C^*/2$, where C^* is the optimal solution cost. Bi-A* does not meet in the middle, as shown in our example. The backward search expanded n_4, whose g-value is 3, which is larger than $5/2 = 2.5$.

In case you are wondering, the Bi-BS algorithm we have seen is guaranteed to meet in the middle. The table below shows the states of $\texttt{OPEN}_f$ and $\texttt{OPEN}_b$ of Bi-BS for the problem above. In the execution below we assume that ties are broken favoring the backward search. The node with largest g-value the forward search expands is n_3, with the cost of 1; while the node with largest g-value the backward search expands is n_5, with the cost of 2. It is possible to prove that Bi-BS meets in the middle in general.

Iteration	$\texttt{OPEN}_f$	$\texttt{OPEN}_b$	Notes
1	$(S, 0)$	$(G, 0)$	-
2	$(n_0, 1)$, $(n_1, 1)$, $(n_2, 1)$, $(n_3, 1)$	$(G, 0)$	-
3	$(n_0, 1)$, $(n_1, 1)$, $(n_2, 1)$, $(n_3, 1)$	$(n_6, 1)$	-
4	$(n_0, 1)$, $(n_1, 1)$, $(n_2, 1)$, $(n_3, 1)$	$(n_5, 2)$	-
5	$(n_1, 1)$, $(n_2, 1)$, $(n_3, 1)$	$(n_5, 2)$	-
6	$(n_2, 1)$, $(n_3, 1)$	$(n_5, 2)$	-
7	$(n_3, 1)$	$(n_5, 2)$	-
8	$(n_4, 2)$	$(n_5, 2)$	-
9	$(n_4, 2)$	$(n_4, 3)$	Optimal solution through n_4.

2.4 The Meet in the Middle Algorithm (MM)

A small modification to Bi-A* allows it to provably meet in the middle. Instead of sorting the `OPEN` lists by f-value, we will sort the nodes according to the following cost function of nodes n.

$$p(n) = \max\big(f(n), 2 \times g(n)\big) .$$

The bidirectional search algorithm that uses the p-function is known as Meet in the Middle (MM). The second term of the max function acts as a "guard" that does not allow searches go "too far." Even if a node n has a promising (i.e., small) f-value, MM might still place it far back in its priority queue if it has a large g-value. This is to prevent the searches from crossing the mid point of search. The table below shows the execution of MM for the problem above.

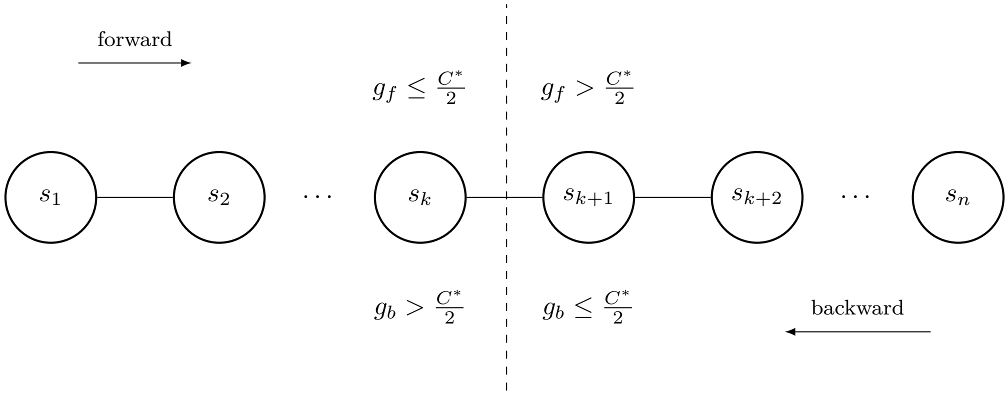
Iteration	$\texttt{OPEN}_f$	$\texttt{OPEN}_b$	Notes
1	$(S, 3)$	$(G, 3)$	-
2	$(n_0, 5)$, $(n_1, 5)$, $(n_2, 5)$, $(n_3, 5)$	$(G, 3)$	-
3	$(n_0, 5)$, $(n_1, 5)$, $(n_2, 5)$, $(n_3, 5)$	$(n_6, 3)$	n_6 is generated and $f(n_6) > 2 \times g(n_6)$
4	$(n_0, 5)$, $(n_1, 5)$, $(n_2, 5)$, $(n_3, 5)$	$(n_5, 4)$	n_5 is generated and $f(n_5) < 2 \times g(n_5)$
5	$(n_0, 5)$, $(n_1, 5)$, $(n_2, 5)$, $(n_3, 5)$	$(n_4, 6)$	n_4 is generated and $f(n_4) < 2 \times g(n_4)$
6	$(n_1, 5)$, $(n_2, 5)$, $(n_3, 5)$	$(n_4, 6)$	-
7	$(n_2, 5)$, $(n_3, 5)$	$(n_4, 6)$	-
8	$(n_3, 5)$	$(n_4, 6)$	-
9	$(n_4, 5)$	$(n_4, 6)$	Optimal solution through n_4

MM behaves exactly like Bi-A* for the first 3 iterations. This is because $p(n) = f(n)$ for all nodes n generated in these iterations. It is in iteration 4 that things start to be different between MM and Bi-A*. In iteration 4, MM generates n_5 with the p-value of 4. If MM had only accounted for the f-value of n_5, the node would have the priority of 3 instead of 4 in the backward search. Note that node n_5 still has the lowest p-value in either direction and it is expanded next, thus generating n_4 with the p-value of 6. If MM had accounted only for f, the priority of n_4 would be 3 and it would be expanded next. Since MM accounts for p, the priority of n_4 is 6 in the backward search, which makes MM switch to the forward direction. The p-function is "telling the search" that it went too far with the backward search and it should now expand from the forward list. The optimal solution is found in iteration 9 and the searches indeed met in the middle.

2.4.1 Why Does MM Meet in the Middle?

Why does MM work in general? That is, how does it guarantee that neither the forward search nor the backward search will expand nodes whose g-value is larger than $C^*/2$? Let us consider the optimal solution path $P = \{s_1, s_2, \cdots, s_k, s_{k+1}, s_{k+2}, \cdots, s_n\}$ shown below. We denote the g-values in the forward and backward searches as g_f and g_b, respectively, and we assume that the heuristic function is admissible. "The middle" is between s_k and s_{k+1}. Thus, $g_f(s_j) \leq C^*/2$ for $j \leq k$ and $g_b(s_j) \leq C^*/2$ for $j \geq k + 1$. In order to show that MM meets in the middle, we need to show that the forward search does not expand states $s_{k+1}, s_{k+2}, \cdots, s_n$. Due to the symmetry of our arrangement, the arguments we will discuss below can be used to show that the backward search does not expand states $s_k, s_{k-1}, \cdots, s_1$.

Let p_f and p_b be the MM cost function for nodes in the forward and backward searches, respectively. Also, let f_f and f_b be the f-value of nodes in the forward and backward searches, respectively. We will show that (i) $p_f(s_j) > C^*$ and (ii) $p_b(s_j) \leq C^*$ for all s_j in $\{s_{k+1}, s_{k+2}, \cdots, s_n\}$. If conditions (i) and (ii) are satisfied, then all states s_j in $\{s_{k+1}, s_{k+2}, \cdots, s_n\}$ must be expanded in the backward search before they are expanded in the forward search (their MM cost is cheaper in the backward search than in the forward search).



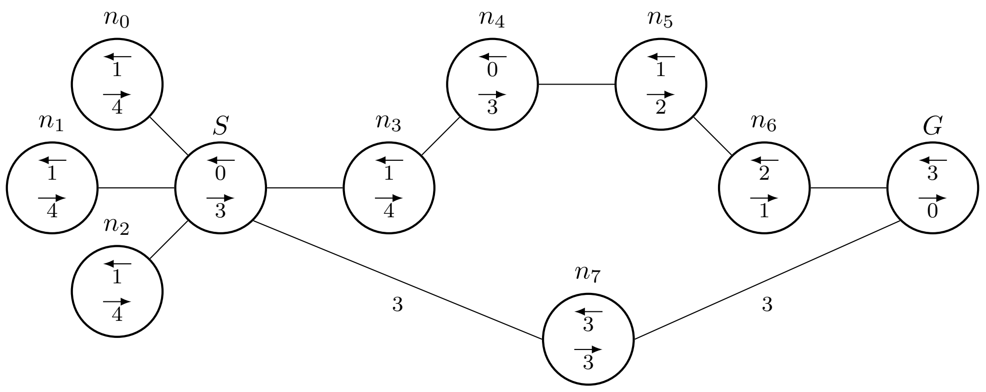
We have that $2g_f(s_j) > f_f(s_j)$ because $2g_f(s_j) > C^*$ (the s_j nodes are on the righthand side of the dashed line) and $f_f(s_j) \leq C^*$ (the heuristic function is admissible). Thus, $p_f(s_j) = 2g_f(s_j) > C^*$. We also have that $p_b(s_j) = \max(f_b(s_j), 2g_b(s_j))$. Here, $f_b(s_j) \leq C^*$ because the heuristic function is admissible and $2g_b(s_j) \leq C^*$ because all s_j are on the righthand side of the dashed line. Thus, $p_b(s_j) \leq C^*$. We just showed that the conditions (i) and (ii) are satisfied. One can now use the same arguments to show that the backward search does not expand states $s_k, \cdots, s_2, s_1$.

2.4.2 MM's Stopping Condition

You will recall that Bi-BS stops searching when the cost U of the cheapest solution it finds in search satisfies $U \leq g_{minf} + g_{minb} + \epsilon$, where g_{minf} and g_{minb} are the minimum g-values in the forward and backward searches, respectively, and ϵ is the cheapest action in the search problem. MM has access to more information than Bi-BS so it can use a more powerful stopping condition. In addition to Bi-BS's stopping condition, MM also verifies for Bi-A*'s stopping condition, i.e., it stops if either $U \leq f_{minf}$ or $U \leq f_{minb}$ is true. Here, f_{minf} and f_{minb} are the minimum f-values in the forward and backward searches.

Another stopping condition MM uses is $U \leq \min(p_{minf}, p_{minb})$, where p_{minf} and p_{minb} are the smallest p-values in the forward and backward searches. Each inequality represents a lower bound on the cost of the solutions we can find should we continue searching. The lower bound $\min(p_{minf}, p_{minb})$ is perhaps the least intuitive one. Recall that $p(n) = \max(f(n), 2 \times g(n))$. If the $f(n) \geq 2 \times g(n)$ for either p_{minf} or p_{minb}, then $\min(p_{minf}, p_{minb})$ must be a lower bound. This is because, if one of the values of a min operation is a lower bound (the f-value is a lower bound), then the result of the min must also be a lower bound.

The trickier situation happens when $f(n) < 2 \times g(n)$ for both p_{minf} and p_{minb}. Each p is not individually a lower bound. This is because the minimum p-value in one direction can be the p-value of a node n that is beyond the mid point (i.e., $g(n) > C^*/2$ and thus $2 \times g(n)$ cannot be a lower bound on the optimal solution cost C^*), but is on an optimal solution path that is yet to be discovered. If n is on an optimal solution path yet to be discovered, then there must be a node n' in `OPEN` of the other search direction whose g-value is less than $C^*/2$ (i.e., the other direction did not reach the middle yet). In this case, $2 \times g_{min}$, where g_{min} is the smallest g value in the `OPEN` list that contains n', must be a lower bound on the cost of solutions yet to be discovered. We show below an example illustrating why $\min(p_{minf}, p_{minb})$ is a lower bound, but p_{minf} and p_{minb} are not. In this problem, all actions cost 1, except the actions connecting n_7, which costs 3 each.



Iteration	$\texttt{OPEN}_f$	$\texttt{OPEN}_b$	Notes
1	$(S, 3)$	$(G, 3)$	-
2	$(n_0, 5)$, $(n_1, 5)$, $(n_2, 5)$, $(n_3, 5)$, $(n_7, 6)$	$(G, 3)$	-
3	$(n_0, 5)$, $(n_1, 5)$, $(n_2, 5)$, $(n_3, 5)$, $(n_7, 6)$	$(n_6, 3)$, $(n_7, 6)$	Suboptimal solution path through n_7
4	$(n_0, 5)$, $(n_1, 5)$, $(n_2, 5)$, $(n_3, 5)$, $(n_7, 6)$	$(n_5, 4)$, $(n_7, 6)$	-
5	$(n_0, 5)$, $(n_1, 5)$, $(n_2, 5)$, $(n_3, 5)$, $(n_7, 6)$	$(n_4, 6)$, $(n_7, 6)$	p_{minb} is not a lower bound
6	$(n_1, 5)$, $(n_2, 5)$, $(n_3, 5)$, $(n_7, 6)$	$(n_4, 6)$, $(n_7, 6)$	-
7	$(n_2, 5)$, $(n_3, 5)$, $(n_7, 6)$	$(n_4, 6)$, $(n_7, 6)$	-
8	$(n_3, 5)$, $(n_7, 6)$	$(n_4, 6)$, $(n_7, 6)$	-
9	$(n_4, 5)$, $(n_7, 6)$	$(n_4, 6)$, $(n_7, 6)$	$\min(p_{minf}, p_{minb})$ is a lower bound

The table shows the state of $\texttt{OPEN}_f$ and $\texttt{OPEN}_b$ for the problem. In iteration 3, MM finds a suboptimal path with cost 6 going through n_7. The optimal solution path goes through the upper path and it costs 5. In iteration 5 we have that $p_{minb} = 6$, which is not a lower bound on the cost of solutions yet to be discovered (the optimal solution is yet to be discovered). Node n_4 is beyond the middle of the backward search and is on the optimal solution path. The forward `OPEN` list contains n_3, a node on the optimal solution path whose g-value is 1, which is less than $C^*/2$. Once n_3 is expanded, MM discovers the optimal solution and immediately returns it, because $\min(p_{minf}, p_{minb}) = 5$, which is equal to the cost of the path MM has found.

In summary, MM uses four lower bounds in its stopping condition.

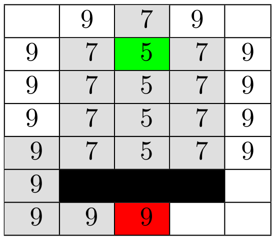
1. $g_{minf} + g_{minb} + \epsilon$
2. f_{minf}
3. f_{minb}
4. $\min(p_{minf}, p_{minb})$

In practice, however, MM can be implemented using only stopping condition (4). This is because the nodes in the `OPEN` lists are sorted according to the p-values, thus p_{minf} and p_{minb} readily available by querying the top of the heap structures. One would have to implement special data structures to efficiently access the minimum g and f-values during search.

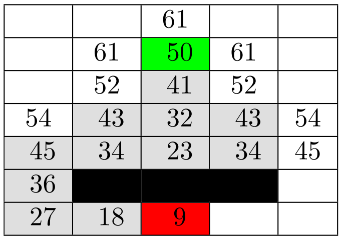
2.5 Weighted A* and Weighted IDA*

In Weighted A* (WA*) and Weighted IDA* (WIDA*) we inflate the heuristic values by using the cost function $f(s) = g(s) + w \cdot h(s)$ with $w > 1$. WA* and WIDA* give more importance to the heuristic function due to $w > 1$. Let us see an example comparing the behavior of A* and WA* in a simple problem.

The left grid shows the search information for A* with Manhattan distance as heuristic function, while the grid on the right shows the information for WA* with Manhattan distance multiplied by 10. In this problem the start state is in green (second row and third column) while the goal state is in red (last row and third column). The solid cells denote walls that cannot be traversed. The numbers in the cells show the f and inflated-f-values for A* and for WA*, respectively. The cells highlighted in gray denote the expanded states.



	9	7	9	
9	7	5	7	9
9	7	5	7	9
9	7	5	7	9
9	7	5	7	9
9	■	■	■	
9	9	9		



		61		
	61	50	61	
	52	41	52	
54	43	32	43	54
45	34	23	34	45
36	■	■	■	
27	18	9		

WA* is greedier with respect to the heuristic function. For example, while A* expands the states around the start state, WA* only expands the state below the start state. This is because WA* will prefer to expand states that are closer to the goal according to the heuristic function. In this example WA* expands fewer states than A*, but this is not necessarily always the case. Since it is very easy to implement WA* and WIDA* once you have A* and IDA* implemented, it is usually worth trying to apply a weight $w > 1$ to the heuristic function to see if the inflated heuristic values allow for a faster search.

It is likely that the heuristic is no longer admissible once we multiply the heuristic values by $w > 1$. Although WA* and WIDA* are not guaranteed to find optimal solutions, if the heuristic employed is admissible (before multiplying by w), then WA* and WIDA* are guaranteed to find bounded suboptimal solutions. That is, the solutions they find are guaranteed to be no larger than $w \cdot C^*$, where C^* is the optimal solution cost. This is an important property because it allows us to find solutions more quickly while knowing that the solutions will not be arbitrarily suboptimal.

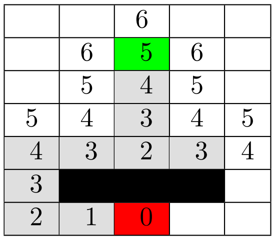
Let us see a sketch of a proof for WA*'s bounded suboptimal solutions. Let h be an admissible heuristic function and f_{min} be the cheapest value of a node in OPEN at a given iteration of WA*. Also, let $f(n_{opt})$ be the inflated f-value of the node in OPEN that is on the optimal solution path (you should try to convince yourself that OPEN must always have a node that is on the optimal solution path). Then we have the following.

$$\begin{aligned} f_{min} \leq f(n_{opt}) &= w \cdot h(n_{opt}) + g^*(n_{opt}) \\ &\leq w \cdot h^*(n_{opt}) + g^*(n_{opt}) \text{ (since h is admissible we have that } h(n_{opt}) \leq h^*(n_{opt})) \\ &\leq w(h^*(n_{opt}) + g^*(n_{opt})) \\ &= W \cdot C^* \end{aligned}$$

The inequality above shows that the node with minimum f-value in OPEN is bounded above by $w \cdot C^*$, which is also true for the goal state when WA* expands it. So the solution cost is bounded suboptimal. The bound above is loose, meaning that we tend to encounter solutions with costs much closer to C^* than to $W \cdot C^*$.

2.6 Greedy Best-First Search

WA* allows us to change the importance of the heuristic function depending on the value of w. The extreme case is when we consider only the heuristic function in the cost function used to order the nodes in `OPEN`, i.e., $f(s) = h(s)$. This algorithm is known as Greedy Best-First Search (GBFS). GBFS is complete but suboptimal. The solutions GBFS finds are not bounded suboptimal, meaning that the solution paths can be arbitrarily costly. GBFS is still an algorithm that is often used in practice. Let us see an example of GBFS.



		6		
	6	5	6	
	5	4	5	
5	4	3	4	5
4	3	2	3	4
3				
2	1	0		

The problem is the same we discussed for A* and WA*. The numbers shown are the f-values according to GBFS. The algorithm expands fewer states than WA* in this particular example, but this is not guaranteed to always happen in practice.

2.7 Heuristic Functions: Where do They Come From?

We have seen an easy way of defining an effective heuristic function for map-based pathfinding problems. How about other problems? How do we define effective heuristics to guide the search? In this section we will discuss another heuristic for map-based pathfinding and we will discuss heuristics for the sliding-tile puzzle.

2.7.1 Map-Based Pathfinding

We have seen the Manhattan distance heuristic function, which can be used when the agent can move in the four cardinal directions. The Manhattan distance of a state s for a goal state s_g can be computed as follows.

$$h(s) = \Delta x + \Delta y\,,$$

where $\Delta x = |s.x - s_g.x|$ and $\Delta y = |s.y - s_g.y|$, with $s.x$ and $s.y$ being the x and y coordinates of state s.

We might also allow diagonal moves, in addition to the four cardinal moves, in map-based pathfinding problems. Here, we consider that diagonal moves cost 1.5 and moves in one of the four cardinal directions cost 1.0. The heuristic function **Octile distance** accounts for the eight possible moves on the grid. Intuitively, if we are considering a map free of obstacles, the agent will perform as many diagonal moves as possible because a diagonal move allows one to progress in both the x and y coordinates toward the goal. The maximum number of diagonal moves we can perform is given by $\min(\Delta x, \Delta y)$; the remaining values can be corrected by equation $|\Delta x - \Delta y|$. Octile distance can then be written as follows.

$$h(s) = 1.5 \min(\Delta x, \Delta y) + |\Delta x - \Delta y|\,,$$

2.7.2 Combinatorial Spaces

Let us consider the 3×3 sliding-tile puzzle (9-puzzle) shown in the figure below. The left grid shows an arbitrary state, while the right grid shows the goal state of the 9-puzzle. In this puzzle an action is given by sliding a tile onto the empty space. For example, for the grid on the left we could slide tiles $2, 6, 3$ or 5 into the empty space. The goal is to find a sequence of actions that transforms an initial state into the goal state, where the tiles are ordered from left to right and top to bottom as shown on the grid on the right.

7	2	4
5		6
8	3	1

	1	2
3	4	5
6	7	8

The 9-puzzle has 9!/2 states in its state space (it is only half of the total number of permutations because only half of the permutations can reach the goal state with the sliding actions). If we increase the size of the grid, the size of the space grows very quickly: the 4×4 and 5×5 puzzles have 16!/2 and 25!/2 states, respectively. The growth of the space of the sliding-tile puzzle is different from map-based pathfinding problems. While the former grows according to a factorial function on the size of the grid, the latter grows only according to a polynomial function on the size of the map. Uninformed search algorithms such as Dijkstra's algorithm and BFS run in polynomial time with respect to the size of the search space. This is not good news if the space is exponential on the size of the input. That is why we need good heuristic functions to guide the search in such spaces. How can we derive a heuristic function for the sliding-tile puzzles?

Tiles out of Place

A simple admissible heuristic is to simply count the number of tiles out of place. For example, in the state shown on the left of the figure above we have that all tiles are not in their goal locations. Thus, we know that we will have to spend at least one move to fix each tile, which adds to an h-value of 8. We do not count the empty space as a tile out of place because that would render the heuristic inadmissible. Consider the situation where only tile 1 is out of place. We can transform that state into the goal state with a single action. That is why we only count 1 as out of place and not both 1 and the empty space.

Manhattan Distance

A heuristic functions that is far better (i.e., the function produces better estimates of the cost-to-go) than Tiles out of Place is the sum of the Manhattan distance of all tiles. We can see each tile as an agent that needs to move to its goal location. An effective heuristic sums the Manhattan distance value of all tiles. For the state on the left, if we add the Manhattan distance value of the tiles from left to right and top to bottom (the first 3 in the summation below is the distance tile 7 has to traverse), then we have the following.

$$h(n) = 3 + 1 + 2 + 2 + 3 + 2 + 2 + 3 = 18\,.$$

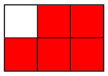
Both Tiles out of Place and Manhattan distance are admissible and consistent heuristic functions. Both heuristic functions are derived from a simplified version of the problem (similar to how we assumed in map-based pathfinding that the map had no obstacles). In Tiles out of Place we assumed that we can remove the tile from the grid and place it in its goal location. In Manhattan distance we assumed that there is only one tile present at a time on the grid, so that the tile can move freely to its goal location.

Pattern databases form a family of heuristic functions that are also created based on the ideia of simplifying the problem. This is the topic of the next section.

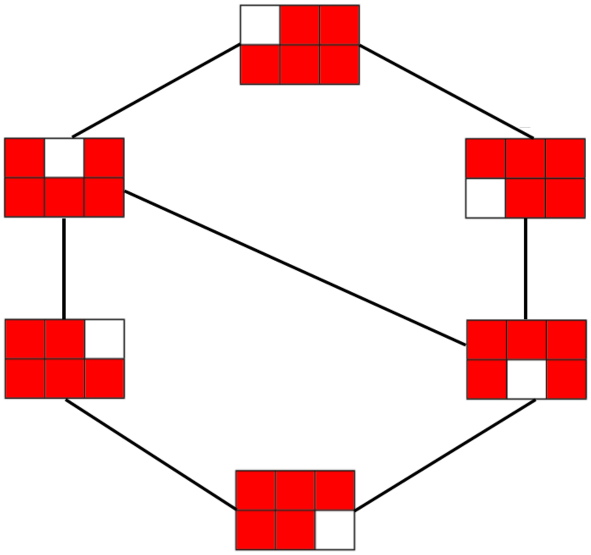
Pattern Databases

In a pattern database (PDB) heuristic we construct a simplified version of the problem, which we call an abstraction of the state space. For example, in the 2×3 puzzle we can treat all numbered tiles as a “red tile,” as shown below. By doing so we are creating an abstracted version of the original space that is smaller than the original one. The 2×3 puzzle has $6!/2 = 360$ states, while the abstracted space has only 6.

	1	2
3	4	5

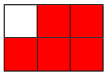
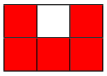
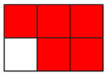
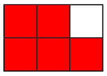
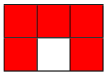
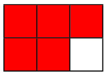


The graph representing the abstracted search space is so small that we can draw it on this page; see the graph below. The abstract goal state is at the top. If we slide the two adjacent red tiles onto the empty position we obtain the two state on the next row of the graph. The other states in the search space can be reached as shown in the graph.



The idea of creating smaller abstract spaces is that we can enumerate all abstract states as a preprocessing step and compute the distance between all states and the goal. The values of the distances in the abstract space serve as a heuristic function to guide the search in the original space.

For example, we can compute the following table for the abstract space shown above. This table, which is known as a pattern database (PDB), contains each state s in the abstract space and the distance between s and the abstract goal state. The values in this table are normally computed with Dijkstra’s algorithm where the initial state is the abstract goal state.

0	1	1	2	2	3
					

Then, we map each state encountered in the search in the original space to an entry in the PDB. This mapping is performed according to the abstraction that is being used. Let us consider the example below.

7	2	4
5		6
8	3	1

🟥	🟥	🟥
🟥		🟥
🟥	🟥	🟥

Here we are considering an abstraction that maps all numbers to the red color. Then, if the state shown on the left is encountered in search, we simply map all its numbers to the red color to obtain its corresponding abstract state, which is shown on the right. Once we obtain the abstract state we simply look in the PDB the optimal solution cost for that state (computed ahead of time with Dijkstra's algorithm); this value is then used as a heuristic function for the state in the original space.

There are many different ways of defining abstractions. For example, we could map tiles $1, 2, 3, 4$ to the green color and tiles $5, 6, 7, 8$ to the red color in the 9-puzzle. The resulting abstract search space will be larger than the space induced by the abstraction in which we map all tiles to the same color. The PDB would require a larger table to store all values, but the heuristic function would likely be more effective. The creation of effective abstractions for deriving PDBs is still an active area of research.

Chapter 3

Local Search Algorithms

3.1 Combinatorial Search Problems

In the previous chapters we studied algorithms for finding solution paths, i.e., a sequence of actions leading to a goal state. In the next two lectures we will study algorithms for solving state-space search problems that do not require a solution path, finding a "goal configuration" is enough. For example, in the n-queens problem we need to place n queens on a $n \times n$ board such that none of the queens attack one another.[1]

The grids below show (i) a candidate solution to the 4-queens problem (left) and (ii) a candidate that is not a solution (right). None of the queens on the left grid attack one another. For the grid on the right, the queen on the first and second columns attack each other; the queens on the third and fourth columns are also attacking one another.

		Q	
Q			
			Q
	Q		

	Q		
Q			
		Q	
			Q

Another example of a search problem where the solution path is not needed is the Traveling Salesman Problem (TSP). In the TSP, a person needs to visit a set of cities once and return to the initial city. The solution must minimize the distance traveled. The TSP is a classic computationally hard problem. A candidate solution for the TSP is a permutation of the cities, where the first city in the permutation is visited first, the second is visited second and so on. An optimal solution is a permutation with the shortest distance traveled.

Problems such as the n-queens problem are known as **pure-search problems**. This is because any candidate that solves the problem is good enough. The task is to find any solution. The TSP is a **pure-optimization problem** because any permutation is a valid solution and the task is to look for the optimal one.

A combinatorial search problem is defined by a tuple (C, S, v, opt), where

- C is a set of candidates;
- $S \subseteq C$ is a subset of the candidates that are solutions;

[1] In chess a queen can attack any pieces on the same row, column, or diagonal line.

- $opt \in \{\min, \max\}$ is the type of optimization (either maximization or minimization);
- v is an objective function mapping a candidate to a real value.

The optimal solution s^* of a combinatorial search problem can be obtained by solving the following equation.

$$s^* = \begin{cases} \min_{c \in C} v(c) & \text{if } opt = \min \\ \max_{c \in C} v(c) & \text{if } opt = \max \end{cases}$$

In pure-search problems we have that the objective function v is either 0 (not a solution) or 1 (a solution) and the problem is treated as a maximization problem ($opt = \max$). In pure-optimization problems the subset of solutions is equal to the set of candidate solutions ($C = S$).

In pathfinding problems we talked about states representing the environment in which the agent acts. In combinatorial search problems we talk about candidate solutions, which is a similar concept used to describe a possible solution to the problem. The main difference is that we no longer have an agent deciding which action to take in the environment. In combinatorial search problems we do not consider a set of legal actions, but we consider a set of **neighborhood candidates**. For example, given a candidate c for the n-queens problem, we can generate a set of neighbors of c by changing the position of each queen in a given column. The neighborhood offers a way of navigating through the space of candidates, while the actions in pathfinding problems have the semantics of an agent taking actions.

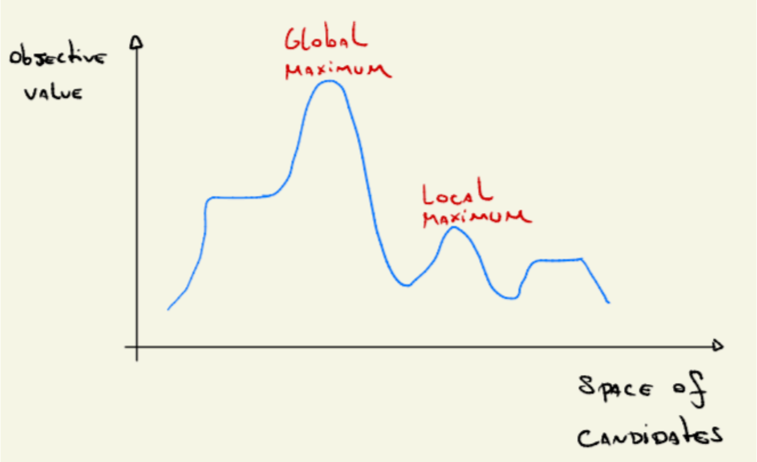
In pathfinding problems we had a heuristic function to guide the search. Some of the combinatorial search problems also have a heuristic function that helps with the search, as we will see an example later for the n-queens problem. In some of the combinatorial search problems we might use the objective function to help guide the search. For example, in the TSP, a candidate solution with total distance traveled of 100 seems to be more promising than a candidate solution with total distance traveled of 200.

Many interesting problems can be cast as a combinatorial optimization problem, including TSP, vertex cover, knapsack problem, program synthesis and even the task of decoding musical notes from neural models. In this lecture we will use the toy n-queens problem because it is a problem that is easy to understand, which is helpful for learning new algorithms. The ideas we will discuss are applicable to a much larger family of problems, such as the ones mentioned above. The algorithms we will study are called **local search**. This is because we use the information of the neighbors of a candidate to decide where to go next in search.

3.2 Hill Climbing

The first algorithm we will study for solving combinatorial search problems is hill climbing (HC). In HC we start with an arbitrary candidate (e.g., queens are randomly placed on the grid) and we greedily improve the initial candidate. This is achieved by generating the neighbors of the initial solution and "accepting" the neighbor with best heuristic value. The process is repeated from the newly accepted candidate. The search continues until the algorithm finds a candidate solution whose neighbors are not better than the current candidate. The current candidate is then returned as HC's solution to the problem.

The process is known as hill climbing because it resembles a hiker going up hill: at every iteration the hiker makes a step toward the top of the hill. HC stops when it reaches the top of the hill, which can be a **local maximum** or **global maximum**. The image below illustrates how the search topology might look like. On the y-axis we have the objective function (or heuristic value depending on the problem) and on the x-axis we have the space of candidates. Depending on where we start the HC search, we might end on a local maximum, on a global maximum, or on a plateau (see the flat area on the righthand side of the plot).



Let us consider an example of HC trying to solve the 4-queens problem. In order to make the problem easier, we will only allow candidates with one queen per column of the grid, as a solution must have at most one queen per column. The neighbors of a candidate are all the possible changes one can make to the position of each queen in her column. We will consider the heuristic function that counts the number of attacking queens on the grid. Intuitively, if a candidate c_1 has fewer attacking queens than c_2, then c_1 is likely closer to a solution and it should be preferred. The grid below shows a possible initial candidate.

$h = 4$

4	Q	2	5
5	3	Q	3
4	5	3	Q
Q	3	4	2

Each number in the grid shows the heuristic value of the candidate obtained should we move the queen of a column to that position. For example, if we move the queen from the first column to the first row, then the resulting candidate will have the heuristic value of 4. The initial candidate also has 4 attacking queens (see h-value at the top of the grid). If HC starts with the candidate above, then it will choose the neighbor obtained by moving either the queen of the third column to the first row or the queen of the fourth column to the last row, since these two candidates have the better h-value of 2.

The grids below show a complete execution of HC. The leftmost grid shows the initial candidate and the solution is obtained with a single move in the rightmost grid. That is, we need to move the queen in the first column to the third row (see the rightmost grid).

$h = 6$

4	3	3	4
4	4	4	4
4	5	5	4
Q	Q	Q	Q

$h = 3$

3	Q	2	3
3	4	3	1
1	5	2	3
Q	6	Q	Q

$h = 1$

1	Q	2	3
3	2	3	Q
0	3	1	3
Q	4	Q	3

HC is not able to solve the problem depending on the initial candidate. The example below illustrates a case in which HC stops at a local minimum ($h = 1$). The grid on the righthand side has the h-value of 1, which is also the value of its best neighbor (obtained by moving the queen in the third column to the first row).

$h = 3$

2	2	4	2
4	4	Q	Q
Q	3	5	3
4	Q	3	2

$h = 2$

Q	3	3	2
4	4	Q	Q
3	2	4	1
4	Q	2	2

$h = 1$

Q	4	1	2
2	3	Q	2
3	3	3	Q
3	Q	2	2

3.2.1 Avoiding Local Minima

How can we avoid local minima (or maxima)? Here are a few strategies that can be effective in practice.

Sideway Moves

In sideway moves we allow the next candidate to have the same value as the current candidate. In the example above in which HC failed to solve the problem we would have the following result.

$h = 3$

2	2	4	2
4	4	Q	Q
Q	3	5	3
4	Q	3	2

$h = 2$

Q	3	3	2
4	4	Q	Q
3	2	4	1
4	Q	2	2

$h = 1$

Q	4	1	2
2	3	Q	2
3	3	3	Q
3	Q	2	2

$h = 1$

Q	4	Q	3
0	3	1	3
3	2	3	Q
1	Q	2	3

By allowing a sideway move we obtain another candidate with $h = 1$ who has a neighbor that is a solution (i.e., $h = 0$). The issue with sideway moves is that they could cause an infinite loop. For example, HC could be stuck in a plateau in the optimization landscape. The solution to avoid an infinite number of moves is to limit the number of sideway moves. The algorithm stops and returns the best solution once HC reaches the limit of sideway moves.

Stochastic Hill Climbing

Another approach for avoiding getting stuck in local minima is to employ a stochastic rule for deciding which neighbor to select. We can define a probability distribution over the neighbors that is proportional to their h-values. For example, if *opt* = max and the h-values of the neighbors of a candidate is given by $(4, 5, 2, 10)$, then we will select the last neighbors with higher probability and the third with lowest probability. That way the search will have a higher chance of accepting the candidates that look more promising according to the heuristic function. Namely, we can define a probability distribution for the neighbors in our example by dividing their h-value by the sum of h-values: $(4/21, 5/21, 2/21, 10/21)$.

Random Restarts

HC will achieve different parts of the space depending on the initial candidate solution. Instead of running HC only once, we can run it multiple times while starting from a different random location each time. This is a powerful idea known as random restarts. Random restarts can be used with regular HC with or without sideway moves and with Stochastic HC. We often employ random starts in practice.

The importance of HC is also theoretical. If all candidates have a non-zero probability of being selected, then as the number of restarts grow large, the probability of finding a solution approaches 1.0.

3.2.2 Example (from Russel & Norvig)

In this example we are trying to solve the 8-queens problem while minimizing the number of HC steps. We will consider two approaches:

1. HC with random restarts;
2. HC with random restarts and a maximum of 100 sideway moves.

We ran the two approaches many times and collected the following data about each method. Approach (1) succeeds in 14% of the attempts while performing 4 steps (it solves the problem in 4 steps); when the algorithm fails it performs 3 steps (it reaches a local minimum in 3 steps). Approach (2) succeeds in 94% of the attempts while performing 21 moves; when it fails it performs 64 moves.

We are trying to understand which method is more effective: the one that fails quickly and has a low success rate or the one that takes longer to fail and has a high success rate? We measure effectiveness in terms of expected number of search steps. What is the expected number of steps for the two approaches?

Let p be the probability of succeeding in a trial (one run of the algorithm from a random initial candidate). The expected number of trials is $1/p$. For example, if $p = 0.5$, then the expected number of trials is 2. The number of failures is given by the total number of expected trials minus 1, the successful trial: $\frac{1}{p} - 1 = \frac{1-p}{p}$.

The expected number of steps is given by the expected number of failures $\frac{1-p}{p}$ times the average number of steps performed in each failure plus the number of successful trials, which is always 1 because we stop as soon as we succeed, times the average number of steps performed in each successful trial.

$$1 \cdot 4 + \frac{0.86}{0.14} \cdot 3 \approx 22, \text{ for approach (1)}$$
$$1 \cdot 21 + \frac{0.06}{0.94} \cdot 64 \approx 25, \text{ for approach (2)}$$

Approach (1) is more effective than approach (2). Unfortunately, in practice we often do not know the expected number of trials nor the average number of search steps for success and failures. Nevertheless, this example illustrates what we are trying to balance in practice (despite not knowing the exact numbers). This example also shows that sometimes it is better to fail and restart more quickly than to spend more time searching in each attempt.

3.3 Random Walks

Hill climbing greedily follows the heuristic function while trying to solve combinatorial search problems. Random walks (RW) is an approach that disregards the heuristic function entirely. In RW we start at a random candidate c and we randomly select one of the neighbors c' of c. For pure search problems, if c' solves the problem, then the search stops and it returns it; if c' isn't a solution, then the process is repeated until reaching a time limit. For pure optimization problems, RW keeps track of the best solution it has encountered in search; once it reaches a time limit, RW returns the best solution found. RW can also be seen as the stochastic version of HC where the neighbors are chosen according to a uniform probably.

RW can be used with and without restarts. The user needs to specify a number of search steps l RW performs when used with restarts. If a solution is not encountered in l steps, then the search is restarted from a random candidate. Note that generating a random candidate is not the same as selecting a random neighbor of the current candidate. The neighbors of a candidate tend to be similar to the current candidate, while a random candidate can be completely different from the last candidate of a RW attempt.

The RW method is asymptotically complete and optimal. That is, if any candidate can be reached with a random walk, then as the number of search steps grows large, the probability of encountering an optimal solution approaches 1.0. The issue with the RW method is that it can be very slow in practice.

RW is often used when a heuristic function is not available or when the function is computationally expensive. RW can also be combined with other search algorithms. For example, one can run HC and, when it encounters a local minimum, the search spawns an RW search that tries to escape the local minimum. Once the RW encounters a candidate c that is better (in terms of h-value) than the candidate at the local minimum, the algorithm starts a new HC search from c. The algorithm we just described was successfully used to solve pathfinding problems in classical planning.[2] As you can see, the local search algorithms we have been discussing can also be applied to solve pathfinding problems.

3.4 Simulated Annealing

While HC makes good use of the heuristic function, its performance depends on the region of the space from which the search is initialized. RW explores the space and does not suffer from local minima because it ignores the heuristic functions entirely, which also means that RW misses the opportunity to use the guidance the heuristic provides.

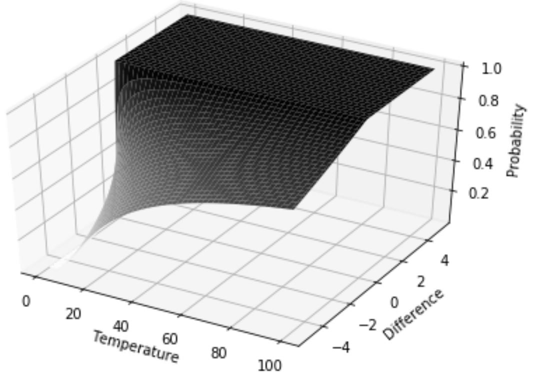
In this section we will study Simulated Annealing (SA), an algorithm that mixes HC and RW by changing its behavior during search. SA behaves similarly to RW in the beginning of search and behaves more and more similar to HC as it performs more search iterations. The idea behind SA is that the RW-like behavior allows the algorithm to explore the space of candidates in the beginning of search. Later, as SA has potentially encountered a more promising region of the candidate space, it behaves more like HC to better use the information the heuristic function provides.

The behavior described above is obtained by accepting a neighbor candidate according to a probability value. SA generates a random neighbor c' of c and it accepts c' according to the probability given in the equation below. We assume $opt = \min$, so that SA prefers neighbors c' for which $h(c) > h(c')$.

$$\min \left\{ 1, e^{(h(c)-h(c'))\frac{\beta}{T_i}} \right\} .$$

Where T_i is a temperature parameter at iteration i and β is an input parameter that adjusts the greediness of the algorithm; larger values of β makes it less likely to accept a worse neighbor c'. If c' represents an improvement over c, then the probability is 1.0 $(h(c) - h(c') > 0)$. If $h(c) - h(c') \leq 0$, then c' is equal or worse than c so the probability of accepting c' depends on the parameters T_i and β. Large values of T_i makes the algorithm give less importance to the difference $h(c) - h(c')$. Thus, even if c' is worse than c, SA might accept it if T_i is large. The figure below shows the the probability values for different temperatures T_i and differences $h(c) - h(c')$; β is fixed to 5 in this plot.

[2] In classical planning one specifies a pathfinding problem in a logical language, which is given to an automated planner. The planner automatically derives a heuristic function for solving the problem and searches in the problem's space.

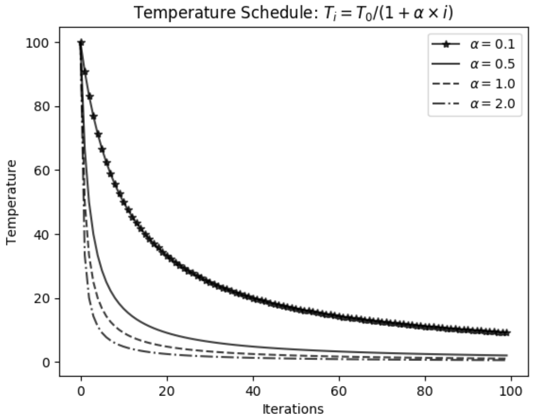


SA starts the search with a high temperature and the search cools it off as the number of iterations i increases. We see in the plot above that, if the temperature is high, the probability of accepting a neighbor is high even if the neighbor is much worse than the current candidate. As we decrease the temperature value, the probability of accepting candidates with negative differences drops quickly. When the temperature is high SA behaves similarly to RW because it accepts almost any neighbor; when the temperature is low SA behaves similarly to HC because it only accepts neighbors that are better than the current candidate.

The temperature decreases according to a pre-defined schedule. A schedule that is often used in practice is given by the following equation, where α is an input parameter that affects the speed in which the temperature drops.

$$T_i = \frac{T_0}{(1 + \alpha \cdot i)}$$

The plot below shows how the temperature drops for different values of α; larger values of α make the temperature drop more quickly.



The pseudocode below shows SA implemented with the acceptance function and temperature schedule as we have just described. SA receives a heuristic function h, an initial temperature T_0, a β value that is used in the acceptance probability, an α value that is used in the temperature schedule, and an ϵ value that determines the stopping condition of the algorithm. SA returns the best candidate encountered in search when the temperature T_i drops below the value of ϵ. The value of ϵ is set experimentally to a small number.

In practice SA is often also implemented with restarts.

```
def SA($h$, $T_0$, $\beta$, $\alpha$, $\epsilon$):
    i = 0
    c = random_candidate()
    best = s
    while True:
        c' = random_neighbor(c)
        with probability $\min\left\{1, e^{(h(c)-h(c'))\frac{\beta}{T_i}}\right\}$:
            c = c'
        if h(c') < h(best):
            best = c'
        $i = i + 1$
        $T_i = T_0/(1 + \alpha \cdot i)$
        if $T_i < \epsilon$: return best
```

3.5 Beam Search

Beam Search can be seen as a variant of HC where, instead of keeping in memory a single candidate c, it keeps a set of B candidates. The set of B candidates is called a beam. In every iteration of Beam Search we generate all neighbors of the B candidates we have in memory and evaluate all of them according to their heuristic value. Let us suppose that each candidate has M neighbors. So we generate and evaluate $B \cdot M$ candidates. We then select the top B out of the $B \cdot M$ candidates. The process is repeated with this new set of B candidates. Beam Search performs multiple HC searches simultaneously.

One might wonder why we would use Beam Search with B candidates if we can use HC with B restarts. Beam Search can be better than HC in this setting because the former does not trust the heuristic function as the latter does. Let us suppose that the heuristic function is always misleading in the first step of search. HC will always fall into the trap of choosing the suboptimal candidate because it greedily chooses the next candidate according to the heuristic function. Beam Search with $B = 2$ selects the best and the second best candidate according to the heuristic function. In this case Beam Search will be able to solve the problem because it considered the candidate that was not the most promising according to the heuristic function.

3.6 Genetic Algorithms (GAs)

GAs perform search by mimicking the biological process of evolution. Namely, GAs keep a set of candidates solutions in memory. The candidates are referred to as **individuals** and the set as a **population**. The GA selects individuals from the population and it pairs them up for procreation. The children of individuals of the current population will form a new population, which is referred to as the next **generation**. The process is repeated with the new generation. The chances of being selected for procreation is related to the individuals' likelihood of survival, which is given by a **fitness function**. The fitness function is essentially a heuristic function, as we have used in other algorithms such as Beam Search and HC. GAs also define a "procreation operator," which is known as **crossover**. Crossover takes two (or more) individuals as input and returns a set of individuals that are made by mixing the "genes" of the individuals provided as input. The individuals can also suffer **mutation**, where a candidate is changed often only slightly and at random.

When the GA selects a pair of individuals according to their fitness value, the GA is focusing its search on more promising candidates. The crossover operator allows the search to mix the features of different promising candidate solutions, while the mutation allows the search to evaluate a neighbor of a promising candidate. Let us see an example of a GA algorithm for the 4-queen problem.

Solving 4-Queens with a Genetic Algorithm

The example we show in this section is inspired in an example from Russell and Norvig (2010). Our population of individuals is composed by the four individuals below. In this formulation the fitness function (h-values at the top of the grids) measures the number of non-attacking queens. Thus, an individual is deemed more fit for survival if it has a large fitness value.

$h = 4$

			Q
Q			
	Q	Q	

$h = 0$

Q	Q	Q	Q

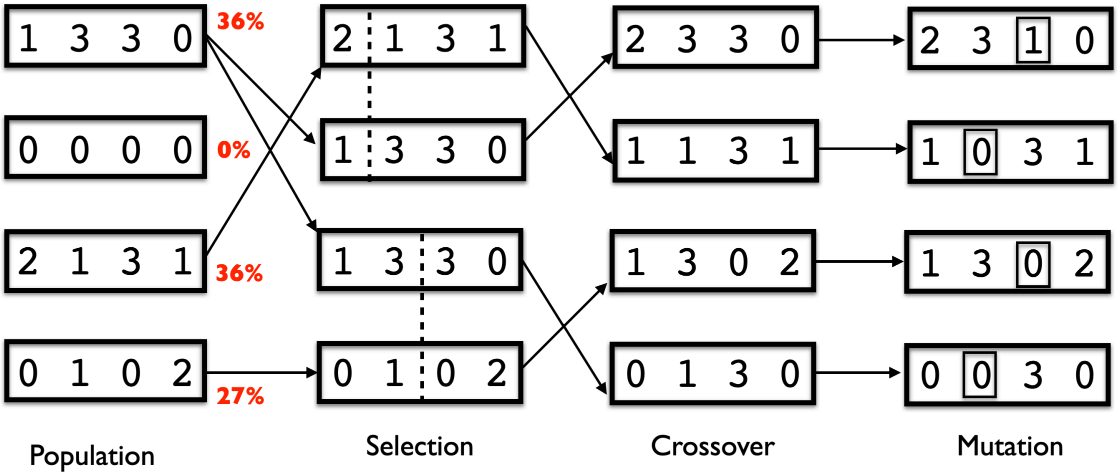
$h = 4$

	Q		Q
Q			
		Q	

$h = 3$

Q		Q	
	Q		
			Q

The scheme below shows all the steps for producing the next generation of individuals. We use a more compact representation for the individuals so that our scheme fits more easily on the page. In our representation we have one integer for each column representing the row in which the queen is located. For example, for the first grid above we have the representation 1, 3, 3, 0 and for the second we have 0, 0, 0, 0.



The column on the left shows the population of the current generation of the GA. In our GA we select two pairs of individuals for performing crossover. The probability in which the individuals are selected is proportional to the fitness value of the individuals. We obtain a probability distribution for the individuals by dividing each fitness value by the total sum of fitness values. For example, the probability of selecting the first individual is $4/11 \approx 0.36$. The individuals in the second column are those sampled from the population according to the probability distribution given by the fitness function. Note that some individuals might not be selected in the process, while others might be selected more than once.

The crossover splits the vector representing each individual into two parts and it generates two new individuals by mixing the parts thus obtained. For example, the first individual in the third column ("Crossover") is composed of the first part of its first parent and of the second part of its second parent. Notice how the combination of individuals 1, 3, 3, 0 and 0, 1, 0, 2 resulted in the individual 1, 3, 0, 2, which is a solution to the 4-queen problem. The crossover allows for the combination of partially correct solutions (first two digits of 1, 3, 3, 0 and the last two digits of 0, 1, 0, 2). The search stops here, as we have encountered a solution. If we had not encountered a solution, we would perform the mutation step. In the mutation step we randomly choose one of the numbers (often referred as genes) in the vector and we change its value.

The Representation Can Matter

The way that we represent the candidate solutions matters for the effectiveness of the search. Let us suppose we have two options for representing the candidates in the 4-queens problem:

1. represent the row of each queen in base 10: 0, 1, 2, 3
2. represent the row of each queen in base 2: 00, 01, 10, 11.

Which one would you choose? Let us consider the case from the example above, where we are to perform crossover of the individuals 1, 3, 3, 0 and 0, 1, 0, 2. Here we can split the vectors after the first, the second, or the third number. If we split after the second we obtain 1, 3, 0, 2, which is a solution to the problem. That is, with 1/3 chance we solve the problem. If we were using the base 2 representation we would have a 1/7 chance of finding the solution because we would have more options for splitting the individuals.

It is true that we could force the crossover to not split the binary numbers in the middle, but that is equivalent to using the base 10 representation. The lesson here is that the way of represent the individuals can matter, as the genetic operators (selection, mutation, crossover) depend on the representation.

3.7 Program Synthesis

In program synthesis one is interested in generating computer programs that satisfy the user's intent. Defining a set of training examples is perhaps the most common and applicable form of expressing intent. The user provides a set of training input-output pairs and the synthesizer searches for a program that satisfies the training set. For example, the user could provide the following examples with input x, y and output o.

$$(x = 1, y = 2, o = 1)$$
$$(x = 10, y = 2, o = 2)$$

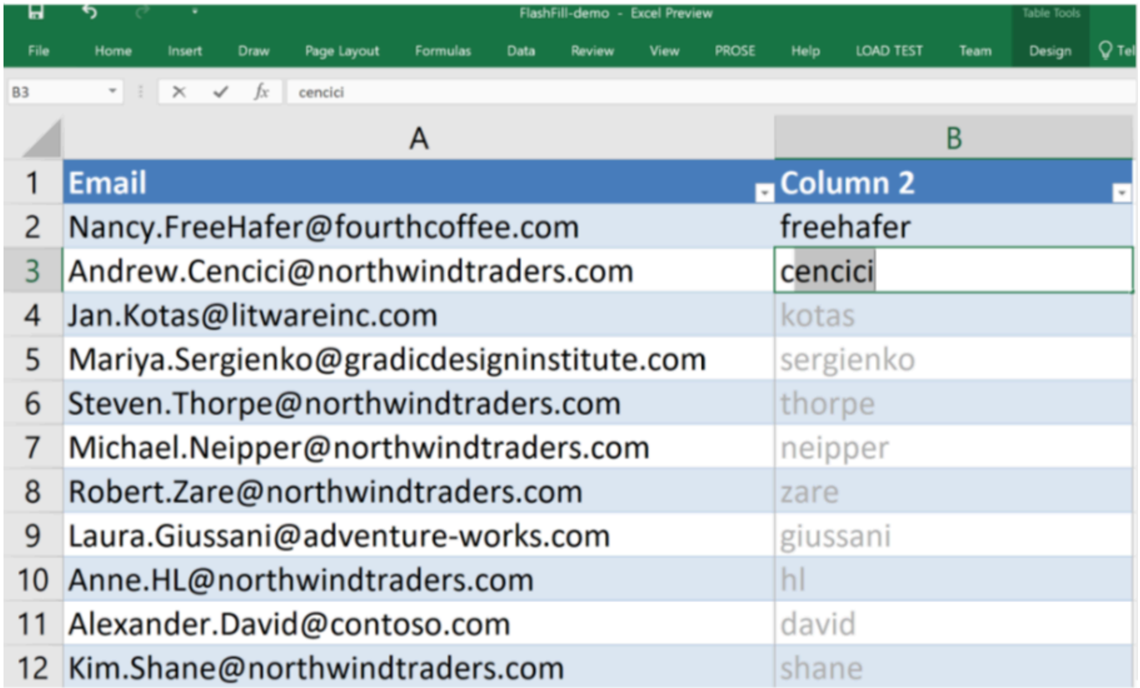
A program that satisfies the constraints imposed by the training set is `if x < y then x else y`.

3.7.1 Program Synthesis and Machine Learning

In the discussion of the user's intent we considered scenarios that are common in supervised learning, where one trains a function based on a set of input-output pairs. What is the difference between program synthesis and machine learning? One can argue that machine learning is a special case of program synthesis, where the program space tends to be more restricted for machine learning algorithms. For example, the space one searches for the weights of a neural network is much more restricted than the space induced by general purposes languages such as Python and C++.

In machine learning we normally assume that the data provided for training is noisy and part of the difficulty of training such models is related to dealing with noisy data. Program synthesis traditionally considers perfect input data (e.g., a logical expression that exactly expresses the user's intent). However, this is changing in the recent years and researchers in the program synthesis are also considering scenarios where the specification is only partially provided or it is noisy.

A good example of partial specification is FlashFill, a Microsoft Excel tool that uses a small number of examples to synthesize programs to manipulate strings. The Figure below shows an example of FlashFill. In FlashFill the training examples are often underspecified, which could result in a synthesis problem for which many programs satisfy the user's intent. This is a challenging also faced in the supervised learning literature where a very large number of weight sets of a neural network minimize the training error.



cencici

	A	B
1	Email	Column 2
2	Nancy.FreeHafer@fourthcoffee.com	freehafer
3	Andrew.Cencici@northwindtraders.com	cencici
4	Jan.Kotas@litwareinc.com	kotas
5	Mariya.Sergienko@gradicdesigninstitute.com	sergienko
6	Steven.Thorpe@northwindtraders.com	thorpe
7	Michael.Neipper@northwindtraders.com	neipper
8	Robert.Zare@northwindtraders.com	zare
9	Laura.Giussani@adventure-works.com	giussani
10	Anne.HL@northwindtraders.com	hl
11	Alexander.David@contoso.com	david
12	Kim.Shane@northwindtraders.com	shane

Figure 3.1: Extracted from Guwlani's ECML'19 Slides.

The FlashFill example points to two key challenges in program synthesis: (1) find a solution; (2) choose a solution that generalizes better to unseen examples. We will see how Simulated Annealing (SA) can be used for solving (1).

3.8 What Are Programs and How to Represent Them?

One normally does not employ general purposes languages for solving program synthesis problems. This is because such languages encode a space that is too large to be practical. Normally one defines a domain-specific language (DSL) that is expressive enough for solving a given problem, but constrained enough to make the search feasible. For example, if we are interested in solving arithmetic problems, then it would not make sense to use a general purpose language, if all we use are operations such as addition and multiplication.

If we wanted to be formal, we would need to specify the syntax and semantics of a domain-specific language. We will be informal with respect to the semantics and will define the syntax of DSLs with context-free grammars (CFGs). The following CFG, whose initial symbol is E, specifies a DSL that allows additions and multiplications of four constants.

$$\begin{aligned} &E \to E + E \mid E * E \mid N \\ &N \to 1 \mid 2 \mid 3 \mid 4 \end{aligned}$$

Strings accepted by the CFG are valid programs in our toy language. For example, the program `4 + 2 * 3` can be written in this DSL. This can be verified by using the following transformations.

$$\begin{aligned} &E \to \\ &E + E \to \\ &4 + E \to \\ &4 + E * E \to \\ &4 + 2 * E \to \\ &4 + 2 * 3 \end{aligned}$$

How about the program `(4 + 2) * 3`? This program can also be represented by this language as we build a structure known as *Abstract Syntax Trees* (AST). The AST is an abstracted version of a Parse Tree. The AST is abstract because we can ignore information that is used to make the program more readable, such as brackets and semi-colons.

3.8.1 Implementing an AST in Python

Let us write Python code to implement the AST structure for our running example.

```python
class Node: # defines the nodes of the AST
  """
     Implementation will depend on the search algorithm we will use and on whether we will build an interpreter for this language.
  """
  ...
```

```python
class Sum(Node):
  def __init__(self, left, right):
    self.left = left
    self.right = right
```

```python
class Times(Node):
  def __init__(self, left, right):
    self.left = left
    self.right = right
```

```python
class Num(Node):
  def __init__(self, val):
    self.val = val
```

Once we have this structure in memory, we can create ASTs representing different programs. For example, program `4 + 2 * 3` is instantiated with `Plus(Num(4), Times(Num(2), Num(3)))`. When resolving the program we will compute the value of the right branch of the node Plus, which is the multiplication `2 * 3`.

Program `(4 + 2) * 3` is instantiated with `Times(Plus(Num(4), Num(2)), Num(3))`. Here, the addition will be resolved before the multiplication. As an exercise, draw the ASTs representing these programs.

3.9 Simulated Annealing for Program Synthesis

The problem of synthesizing programs that satisfy a specification can be seen as an optimization problem. Let $G = (V, \Sigma, R, S)$ be a context free grammar defining a domain-specific language (DSL), where V is the set of non-terminal symbols, Σ is the set of terminal symbols, R are the rules in which one can transform non-terminals, and S is the initial symbol. Also, let $[\![G]\!]$ be the set of programs that can be synthesized with grammar G.

Given a set of training input-output pairs D, we can define an error function J for a program $p \in [\![G]\!]$ and D that counts the number of pairs for which the output produced by p for input x does not match the true output y.

$$J(p, D) = \sum_{(x,y)\in D} [p(x) \neq y]$$

Then, program synthesis can be formulated as the following problem,

$$\operatorname*{argmin}_{p\in[\![G]\!]} J(p, D) \tag{3.1}$$

How can we apply the SA algorithm to solve program synthesis problems (i.e., approximate a solution to Equation 3.1). We will use function J as the objective function used in the acceptance function of SA.

Here is a summary of SA for program synthesis.

1. Generate a random program p using the rules of the grammar defining the language.
2. While the temperature is greater than a threshold ϵ:
   (a) Generate a mutation p' of p and accept it according to SA's acceptance rule for function J.
   (b) Decrease temperature
3. Return the program with smallest J-value encountered in search.

A powerful enhancement to SA for solving program synthesis tasks is the use of restarts. The procedure above should be run as many times as there is time available for searching.

While we have already discussed most of the steps of the procedure above, we still need to discuss how to generate a random program from a grammar and how to mutate a program. We will describe these operations while considering an example, which is modified from from the paper 'Program Synthesis' by S. Gulwani, O. Polozov, and R. Singh. The language below allows simple operations.

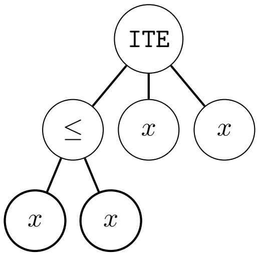
$$\begin{aligned}
&S \rightarrow x \mid y \mid 0 \mid 1 \mid (S + S) \mid \text{if } B \text{ then } S \text{ else } S \\
&B \rightarrow (S \leq S) \mid (S == S) \mid (S \geq S)
\end{aligned}$$

The specification is given by the following set of training input-output pairs, drawn from a max function.

A random program can be generated by starting from the start symbol and randomly applying rules from the grammar. This procedure can be implemented as a recursive function similar to depth-first search (DFS).

For example, starting at initial symbol S, we choose one of the valid production rules for S according to the grammar. Since there are 6 production rules for S, each can be chosen with probability 1/6. Let us suppose that we randomly choose the if-then-else (ITE) production rule, so that S is transformed into `if B then S else S`. Next, we randomly replace B with one of its valid production rules. This process continues until the program only has terminal symbols. One might want to ensure a maximum size for the program by forcing it to choose a terminal symbol if the program reaches a size limit.

A program that can be randomly obtained is `if x` $\leq$ `x then x else x`, whose AST is shown below.



The root of the tree specifies an if-clause with three children: the Boolean expression, the programs that are executed in case the expression evaluates to true or to false. The operator, $\leq$, has two children, one for each side of the operation. The AST defines the initial program p_0 of a SA procedure, whose J-value is 2 (can you verify?). A neighbor can be generated by selecting one of the nodes in the AST to be mutated; the node is selected uniformly at random. Since the AST of p_0 has 6 nodes, each can be selected with probability 1/6. The subtree of the selected node is then replaced by a subtree that is randomly generated according to its non-terminal symbol and the rules in the grammar. If the node $\leq$ is selected, then it can be replaced by one of the following rules (with equal probability): $\leq, ==, \geq$. This is because node $\leq$ was derived from the non-terminal symbol B. The subtrees of the Boolean node are also replaced by randomly generated subtrees. For example, the expression $x \leq x$ can be replaced by a new expression such as $y == x$.

If the neighbor is the program `if x` $\leq$ `x then y else x`, then its J-value is 1 (can you verify?) and SA would accept it with 1.0 probability as it represents an improvement over the current program.

Chapter 4

Multiagent Search

Many examples from these notes on Game Theory are adapted from the course Game Theory by Ben Polak.[1]

4.1 Thinking Strategically

Until now we have studied several algorithms for solving *single-agent problems*, i.e., problems where we have one agent trying to find a solution path or optimize a given objective function. In the next two lectures we will study scenarios in which two or more agents interact while each tries to maximize their own objective function, which we refer to as the *utility function.* A classic example is a pair of agents playing a match of chess. Each player wants to maximize their chances of winning the game, i.e., their utility value.

Game theory provides the machinery we need to formalize how rational agents might act in scenarios where they interact with other agents. We will then study how to derive strategies that optimize for the agent's utilities. We call a problem where multiple agents interact a *game.* Let us start by considering an example.

Example. *Consider the Guessing Game, a game in which players choose a number between 1 and 100. The player who chooses the number x that is closest to m, which is the average of the chosen numbers multiplied by $\frac{2}{3}$, receives a number of points minus a penalty value:* $500 - |x - m|$. *All the other players receive* 0.

A *strategy* specifies exactly how a player acts in the game. Let us suppose that 3 agents are playing the game described above. The strategy of each player is the number each player chooses to play and the *strategy profile* is the set of strategies of all players. An example of a strategy profile for this game is the tuple $(25, 5, 60)$ specifying the strategies of each player. The value of m given by this strategy profile is $\frac{2}{3} \times 30 = 20$. The player whose strategy was 25 wins $500 - 5 = 495$; the other two players receive 0.

What would be your strategy in this game?

4.2 Formalization

We define a game as follows.

- A set of players, which we denote with letters i and j (e.g., all students in class for the guessing game).

[1] urlhttps://oyc.yale.edu/economics/econ-159

- Strategies available for players (e.g., $1, 2, \cdots, 100$). We denote the strategy of player i as S_i.
- Utility of player i is denoted as $U_i(S_1, S_2, \cdots, S_i, \cdots, S_n)$ (e.g., $U_i(S) = 500$ minus penalty if win, 0 otherwise, where S is the strategy profile).

We use S_{-i} to denote all strategies of a player but player i. And we use $U_i(S_i, S_{-i})$ to denote the strategy of player i when i plays S_i and all the other players play S_{-i}.

4.3 Normal-Form Game and Strictly Dominance

A way of representing games is with a table known as the *normal form.* Consider the following example.

			2	
		L	C	R
1	T	$5, -1$	$11, 3$	$0, 0$
	B	$6, 4$	$0, 2$	$2, 0$

The table specifies the players 1 (row player) and 2 (column player) and their strategies: player 1's strategies are T and B, while player 2's strategies are L, C, and R. Each cell of the table specifies the utility value of each player should they play the corresponding row and column strategies; the first number is the row player's utility, while the second number is the column player's utility. For example, if player 1 chooses T and player 2 chooses C, then $U_1(T, C) = 11$ and $U_2(T, C) = 3$.

Considering the game above, player 2 will never choose R. This is because C is always better than R, independently of what player 1 chooses. We say that C *strictly dominates* R. An agent who is trying to maximize their own utility will never play a dominated strategy.

4.4 Weakly Dominance

Let us consider another game. Suppose you are playing a tabletop role-play game and you need to defend your kingdom from an attack. You can either defend an attack that comes through the mountains (difficult access, denoted as d) or through the sea (easy access, denoted as e). The attacker should choose to attack either through the mountains (D) or through the sea (E). See the game below. Which strategy would you play as defender? To answer this question we need to first think how the attacker will act. The attacker prefers E because it cannot be worse than D, independently for the defender's strategy. Given that the attacker chooses E, the defender chooses e. Here, we say that E *weakly dominates* D. This is because E and D are equally good if the defender plays e, but E is better than D if the defender plays d.

		Attacker	
		E	D
Defender	e	$1, 1$	$1, 1$
	d	$0, 2$	$2, 0$

4.5 Back to the Guessing Game

Let us now use the concept of weak dominance to analyze the Guessing Game. Assuming the players are interested in maximizing their own utility, we will iteratively remove the dominated strategies from the set of strategies a player can play.

- Strategy 67 weakly dominates all strategies $S > 67$. Why? The closest the strategies $S > 67$ will be from matching the value of m is if everybody plays 100. In that case, $m = 67$, which makes the strategy 67 the winner. Can you explain why 67 weakly dominates and not strictly dominates $S > 67$?

- If we remove the weakly dominated > 67 strategies from the pool os strategies, then with the same reasoning we conclude that 45 weakly dominates all strategies $S > 45$.

- If we remove $S > 45$, then 30 weakly dominates $S > 30$.

- $\cdots$

- The only non-dominated strategy is 1.

Despite being the only non-dominated strategy, it is unlikely we see human players playing the strategy 1 in this game. Studying how humans behave in strategic settings is an active and exciting area of research, but it is outside the scope of this course.

4.6 Best Response

The games we have analyzed so far have an obvious answer because we can simplify the game by removing dominated strategies. Many games are not as simple, however. Let us consider another example.

		2	
		e	d
	A	$5,1$	$0,2$
1	M	$1,3$	$4,1$
	B	$4,2$	$2,3$

There is no single strategy that dominates another strategy in this game. Which strategies will the players choose? It depends on the belief a player has on the other player's strategy. For example.

- if 2 plays e, then 1 should play A.

- if 2 plays d, then 1 should play M.

We say that A is a *best response* to e, and M is a best response to d.

What if player 2 plays e and d 50% of time? We denote such a strategy with a tuple with the probability in which each strategy is played: $\left(\frac{1}{2}, \frac{1}{2}\right)$.

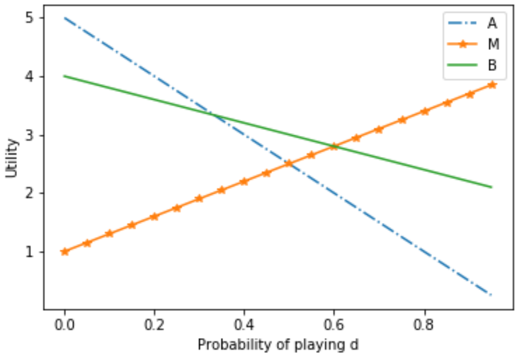
$$\begin{aligned} U_1(A) &= \frac{1}{2} \cdot 5 + \frac{1}{2} \cdot 0 = 2.5 \\ U_1(M) &= \frac{1}{2} \cdot 1 + \frac{1}{2} \cdot 4 = 2.5 \\ U_1(B) &= \frac{1}{2} \cdot 4 + \frac{1}{2} \cdot 2 = 3 \end{aligned}$$

B is a best response to $\left(\frac{1}{2}, \frac{1}{2}\right)$ because it maximizes player 1's utility.

We can write the equations above more generally by using $p(d)$ as the probability of player 2 playing d.

$$\begin{aligned} U_1(A) &= (1 - p(d)) \cdot 5 + p(d) \cdot 0 = 5 - 5p(d) \\ U_1(M) &= (1 - p(d)) \cdot 1 + p(d) \cdot 4 = 1 + 3p(d) \\ U_1(B) &= (1 - p(d)) \cdot 4 + p(d) \cdot 2 = 4 - 2p(d) \end{aligned}$$

Then we can plot the lines from the equations above.



The point where the curves of A and B intersect is given by $-5p(d) + 5 = -2p(d) + 4$, or $p(d) = \frac{1}{3}$. Thus, A is a best response for values of $p(d)$ smaller than $\frac{1}{3}$, while B is the best response for values of $p(d)$ greater than $\frac{1}{3}$ and smaller than the point where lines M and B intersect, which is given by $3p(d) + 1 = -2p(d) + 4$, or $p(d) = 3/5$.

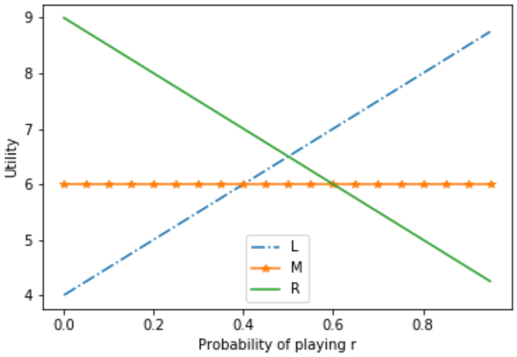
4.6.1 The Penalty Kick Problem

Let us consider a simplified version of the penalty kick problem in soccer. In this game the utility can be seen as the probability of the goal being scored (see the normal form game below). For example, if the goalie jumps to the right and striker shoots to the left, then there is a 90% chance of the goal being scored.

As you can verify, there is no single dominated strategy in this game.

The plot below shows the utilities for a given probability $p(r)$ that the goalie jumps to the right. M is not a best response for any belief (value of $p(r)$) because the Striker will always maximize their utility by choosing either L or R, depending on the value of $p(r)$. A player trying to maximize their own utility should never choose a strategy that is not a best response to any belief. Thus, the Striker should never choose M.

		Goalie	
		l	r
	L	$4, -4$	$9, -9$
Striker	M	$6, -6$	$6, -6$
	R	$9, -9$	$4, -4$



4.7 Nash Equilibrium

We have been saying that "player trying to maximize their own utility will never do this or that," but we never defined this kind of player. So here it is. We say that a set of agents play optimally if they play a *Nash equilibrium* (NE) profile. In a NE profile all players play best responses to the other players strategies.

Let us see a couple of examples.

			2	
		l	m	r
	L	$0, 4$	$4, 0$	$5, 3$
1	M	$4, 0$	$0, 4$	$5, 3$
	R	$3, 5$	$3, 5$	$6, 6$

By looking at the best responses for each strategy we can find the Nash equilibria. In the example above we have that (R, r) is a Nash equilibrium profile because R is a best response to r and r is also a best response to R. The table below shows another example, where (M, m) is the only Nash equilibrium.

			2	
		l	m	r
	L	$0, 2$	$2, 3$	$4, 3$
1	M	$11, 1$	$3, 2$	$0, 0$
	R	$0, 3$	$1, 0$	$8, 0$

We denote a NE profile as $(S_1^*, S_2^*, \cdots, S_n^*)$. In such a profile no player can improve their utility by deviating from S_i^* while S_{-i}^* is fixed. That is the idea behind an *equilibrium*—no one has incentives to deviate from it.

The NE profile for the guessing game is the profile where all players play 1. Although a NE profile can be used to predict how rational agents act in some games, in practice, agents might not play optimally. Some games are too complex for us to be able to derive optimal strategies (e.g., chess and Go), so we will try to approximate an optimal strategy with search algorithms.

4.8 Dominance and Nash Equilibrium

In the game below α strictly dominates β.

		2	
		α	β
1	α	0, 0	3, -1
	β	-1, 3	1, 1

Strictly dominated strategies are not best responses to any strategy, thus they cannot be part of any Nash equilibrium. The game above is known as the Prisoner's Dilemma and the only NE profile we have for this game is the profile (α, α). I know some of you might be disappointed with the fact that the players could both choose β and maximize their utilities. Note, however, that both players have an incentive to unilaterally deviate from the (β, β) profile as they would increase their utility by doing so.

4.9 Mixed Strategies

Some games do not have a single strategy each player can choose to form a NE profile. Let us consider the game of Rock, Paper, and Scissors (RPS) as an example. In this game Rock beats Scissors, Scissors beat Paper, and Paper beats Rock. There is no strategy profile using single strategies (we call them pure strategies) that is a Nash equilibrium. We can, however, define strategies that mix pure strategies, i.e., each strategy is played according to a probability. In the case of RPS, a NE profile is given when both players choose each strategy with 1/3 probability, i.e., the profile $\left[\left(\frac{1}{3}, \frac{1}{3}, \frac{1}{3}\right), \left(\frac{1}{3}, \frac{1}{3}, \frac{1}{3}\right)\right]$ is a Nash equilibrium.

		2		
		R	P	S
	R	$0, 0$	$-1, 1$	$1, -1$
1	P	$1, -1$	$0, 0$	$-1, 1$
	S	$-1, 1$	$1, -1$	$0, 0$

We can verify the profile to be an equilibrium by computing the utility of the players and verifying how much they can gain by deviating from it.

$$U_1\Big(R, (\frac{1}{3}, \frac{1}{3}, \frac{1}{3})\Big) = \frac{-1}{3} + \frac{1}{3} = 0$$
$$U_1\Big(P, (\frac{1}{3}, \frac{1}{3}, \frac{1}{3})\Big) = \frac{1}{3} + \frac{-1}{3} = 0$$
$$U_1\Big(S, (\frac{1}{3}, \frac{1}{3}, \frac{1}{3})\Big) = \frac{-1}{3} + \frac{1}{3} = 0$$

If player 2 plays $\left(\frac{1}{3}, \frac{1}{3}, \frac{1}{3}\right)$, then any strategy for player 1 is a best response to player 2 as all three strategies have a utility of 0 and so would any mixture of these strategies. Thus, if both players play $\left(\frac{1}{3}, \frac{1}{3}, \frac{1}{3}\right)$, then no player would increase their utility by deviating unilaterally from their strategy, which characterizes a NE.

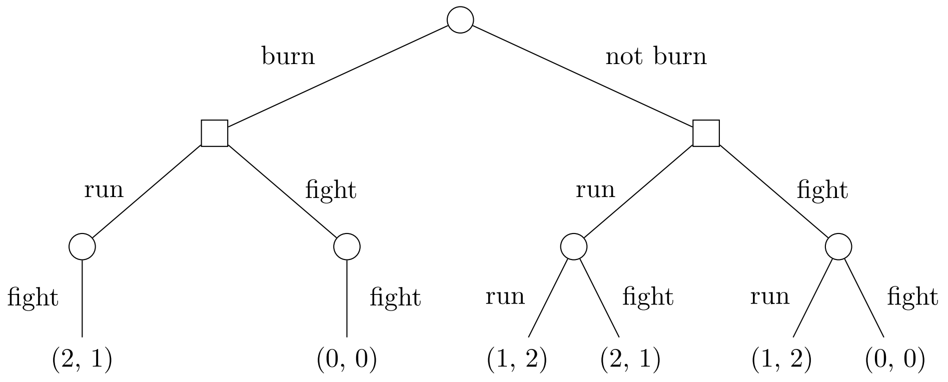
Note that the strategy $\left(\frac{1}{3}, \frac{1}{3}, \frac{1}{3}\right)$ is not a best response to many other strategies. For example, if player 1 plays Rock, then the best response is Paper, and so on. In tournaments of RPS, often the winning strategy is not the NE strategy, but something that attempts to exploit other players. We can see the NE strategy as a contract that offers the guarantee that the player will never receive a utility value that is less than 0.

4.10 Perfect-Information Extensive-Form Games

Extensive-form games allows us to represent games where the players perform a sequence of decisions. These games with a sequence of decisions can also be represented in normal-form, but representing them as a table is not convenient nor efficient because it would require one entry in the table (row or column, depending on the player) for each possible sequence of decisions. Extensive-form games represent games as a tree, which is a much more efficient and compact form of representation for this kind of game. As an example, in the game of chess the players make a sequence of decisions, which are alternated, before deciding the utility of the game, i.e., who wins the match. In extensive-form, the game of chess can be represented as a tree, where each node in the tree represents a configuration of the board where one of the players must act.

We will consider only perfect-information games, which are the games where the players know exactly the current state of the game while making a decision. For example, chess is a perfect-information game because both players know exactly the current state of the game by looking at the pieces on the board; Poker, by contrast, is an imperfect-information game because a player does not know the cards the other player is holding, so they are not able to exactly pinpoint the current state of the game. Algorithms for solving imperfect-information games are outside the scope of this course.

Let us consider an example of an extensive-form game. In the battle of 1066 the Normans invaded the Saxons and decided to burn their own boats! Why? The game is represented in its extensive form below, where circles represent states of the game where the Normans act and squares represent states where the Saxons act. The leaf nodes show the utility values of each player, where the first value in the pair shows the utility of the Normans and the second the utility of the Saxons. For example, if Normans do not burn their boats, Saxons fight, and Normans run away, then the utility is $(1, 2)$, 1 for Normans and 2 for Saxons.



In a procedure known as *backward induction* we compute a Nash equilibrium profile in the game by computing the value of each node in the tree. We start at the leaves and propagate their values up the tree, until we reach the root of the tree. Assuming that the root of the tree is at level 1, we start at the leaf nodes at level 4. Each leaf node already has its value, which is given in the game's definition. What are then the values of the nodes in level 3? Starting from left to right, the values are: $(2, 1), (0, 0), (2, 1), (1, 2)$. The first two values are not interesting because the Normans only have the option to fight, so the value of the leaf nodes are propagated up the tree. The third value is $(2, 1)$ because the Normans are given the option to either run or fight and the latter maximizes their utility (2 for fighting and 1 for running). The forth value is computed similarly, with the Normans choosing to run.

The values in level 3 of the tree are then propagated up, to level 2. From left to right the values of nodes in level 2 are: $(2, 1), (1, 2)$. Finally, the value of the game is $(2, 1)$, which is given by the Normans burning their own boats; the option of not burning their boats results 1 for the Normans (not burn, fight, and run).

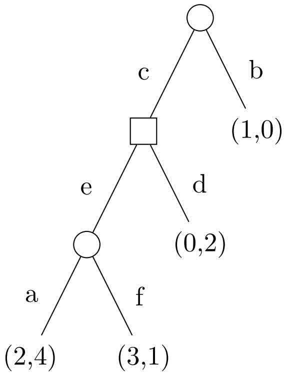
By burning their boats the Normans force the Saxons to run away. The Normans make a commitment to fight, and knowing that the Normans are committed to fight makes the Saxons run away.

Perfect-Information Extensive-form games can be defined by a tuple (P, N, Z, A, J, T, U, I), which defines a tree as follows:

- P is the set of p players.
- N is the set of non-terminal nodes; we call them the decision nodes of the tree.
- Z is the set of terminal nodes, where the utility of the players can be computed directly.
- I in N is the initial node of the game.
- $A(n)$ is a function receiving a node n in the tree and returning a set of actions.
- $P(n)$ is a function that returns the player who is to act in n.
- $T(n, a)$ is a transition function mapping a node and an action to another node.
- $U(n) = (u_1, u_2, \cdots, u_n)$ is a function that returns a utility vector with one entry for each player given a terminal node $n \in Z$.

The extensive-form tree is defined implicitly with the initial node I and the transition function $T(n, a)$. The root of the tree is I and its children is given by the transition function $T(n, a)$, with one child for each a in $A(n)$. The grandchildren are generated similarly by applying the transition function from each child of I.

A strategy in extensive-form games is a complete plan of how the player acts in every decision node of the tree. Let us consider the following two-player extensive-form game, where circles represent decision nodes of Player 1 and squares represent decision nodes of Player 2.



Here are the possible strategies Player 1 and Player 2 can choose.

- Player 1: $[ca]$, $[cf]$, $[ba]$, $[bf]$.
- Player 2: $[e]$, $[d]$.

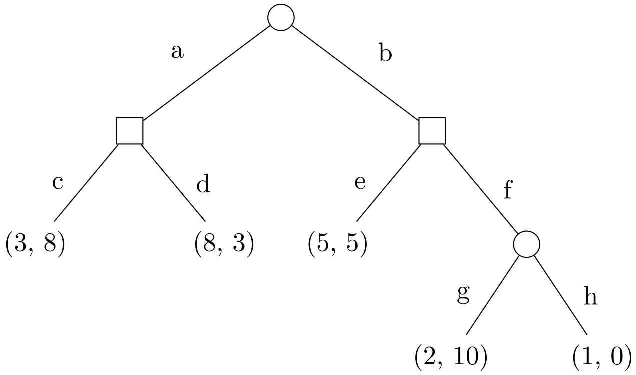
If we apply the Backward induction procedure we described in the example of the Normans and Saxons, we will see that it returns the profile $([b, f], [d])$ as the Nash equilibrium profile of the game. Similarly to normal-form games, we can see that neither player has an incentive to unilaterally deviate from their strategy in a Nash equilibrium. For example, if Player 1 deviates from b to c, then they will reduce their utility from 1 to 0 (Player 1 chooses c and Player 2 chooses d). Player 2 also does not increase their utility by deviating from d to e as they will still receive the utility of 0, which is given by Player 1 choosing b at the root of the tree.

4.11 Algorithms for Backward Induction

The optimal strategies for a game can be computed with Backward Induction. Before writing the pseudocode of Backward Induction, let us write a simpler algorithm that simply traverses the tree and visits each node n a number of times that matches the number of children n has.

```
Algorithm 1 Traverse
Require: Node $n$
Ensure: Visit all nodes rooted at $n$.
if $n \in Z$ then
    visit(n)
    return
for each $a \in A(n)$ do
    Traverse($T(n, a)$)
    visit(n)
```

Let us verify the algorithm above in the following extensive-form game. The procedure is called with node n being the root of the tree. The procedure checks if n is a terminal node (it is not) and it proceeds to call the procedure recursively for each child of n. Once the procedure returns from the recursive call of the first child, it "visits" n; the same procedure is then repeated for the second child. We will replace "visit" instructions with the computation of the value of the current node n to obtain a Backward Induction algorithm, which is shown in Algorithm 2.



Algorithm 2 Backward Induction

```
Require: Node $n$
Ensure: Visit all nodes rooted at $n$.
if $n \in Z$ then
    return $U(n)$
$b \leftarrow \{-\infty, \cdots, -\infty\}$
for each $a \in A(n)$ do
    $v \leftarrow \text{BI}(T(n, a))$
    if $v[P(n)] > b[P(n)]$ then
        $b \leftarrow v$
return b
```

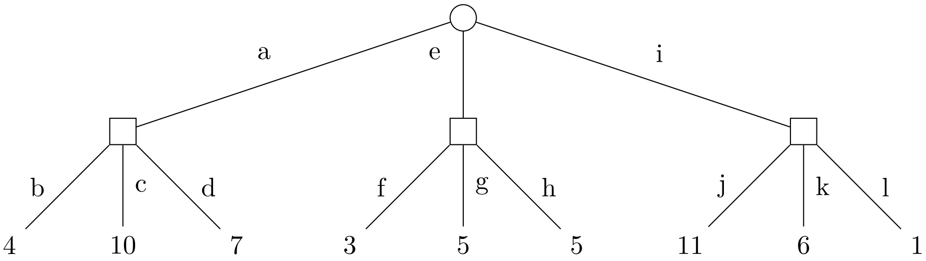
In Algorithm 2 we create a vector b to store the utility value of all players at node n. The utility values are initialized with $-\infty$ for all players. The procedure then computes the value v of each child of n and it replaces b with v if the value returned in v is larger than the value in b for the player who is to act in n, denoted as $P(n)$. In the pseudocode we use $b[P(n)]$ and $v[P(n)]$ to denote the utility value of player $P(n)$ in vector b and v, respectively. The Backward Induction procedure eventually returns the value of the root of the tree.

The time complexity of backward induction is $O(b^d)$, where b is number of options the players have in each node and d is the height of the tree. We can do better for zero-sum games with a clever pruning scheme.

4.12 Zero-Sum Games

In zero-sum games gains and losses of a player is perfectly balanced with the gains and losses of the other players. That is, if the utility of player i is U_i, then the utility of the remaining players has to be $U_{-i} = -U_i$. We will focus on two-player zero-sum games. Examples of this class of game includes rock, paper, and scissors (imperfect information) as well as chess, checkers, and Go (perfect information).

We can now use a single value to denote the utility of a terminal node in the tree because the utility of one player will be the negative of the other. Consider the example below where one player is trying to maximize the utility value and the other is trying to minimize it. The circles represent decision nodes of the player who wants to maximize the value of the game (max player) and the squares the nodes of the player who wants to minimize the value of the game (min player).



The backward induction procedure above is known as Minimax if applied to two-player zero-sum games. What is the value of the game returned by Minimax? The best the min player can do is to play b, f, and l for the left, middle, and right subtrees. Given that, max should play a, as that will maximize their utility.

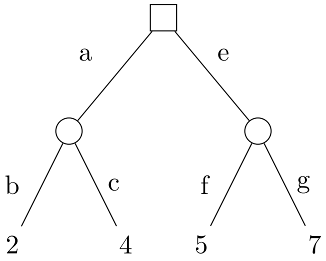
4.12.1 Alpha Beta Pruning

Let us still consider the example above. Let M be the minimum value we obtain while searching in the subtrees rooted at the children of the node reached with action e in the tree. Initially we set $M = \infty$ and, after searching in the subtree given by action f, we assign $M = \min(M, 3) = 3$. At this point we have not searched in the subtrees rooted at g and h yet. However, we already know that the value of M cannot be larger than 3 by the time we finish searching in all these subtrees. That is, $M = 3$ is an upper bound of the final value of M. We also know that the max player can choose action a at the root to obtain the utility of 4 as we have already searched that subtree. Thus, we know that the max player will never play e (if they did, they would receive a utility value of at most 3, which is worse than the guaranteed value of 4 the player has for playing a). This means that the search can skip (prune) the subtrees rooted at g and h. This scheme of skipping nodes during backward induction is known as Alpha Beta pruning.

Note that when we prune the subtrees rooted at g and h, the value of 4 is a lower bound on the value of the game for the max. That is, if we continue to search in the subtrees rooted at e and i the value of 4 cannot decrease. We will call this lower bound α and define the following pruning rule based on the example above.

$$\text{Given } \alpha \text{ and } M, \text{ we can prune all children of a min node if } \alpha \geq M\,. \tag{4.1}$$

We can derive a similar pruning rule for max nodes. Let us consider the example below, where the squared node is a min node and the circle nodes are max nodes and the nodes are explored from left to right.



After exploring the subtree rooted at a we find that the min player can secure the utility of 4 by choosing a. The value of 4 is an upper bound of the value of the min node that we denote as β. Similarly to the example above, once we start searching the subtree rooted at e, we find that the max player can obtain at least the value of $M = 5$ with action f. Since $M = 5$ cannot decrease if we search under g and the min player can secure the utility of 4 by choosing a, we can prune g. We are now ready to write another pruning rule.

$$\text{Given } \beta \text{ and } M, \text{ we can prune all children of a max node if } \beta \leq M\,. \tag{4.2}$$

Pruning rules 4.1 and 4.2 are known as the Alpha-Beta pruning rules. Algorithm 3 shows Minimax with Alpha Beta pruning. The algorithm keeps track of the best solution encountered for the max player (denoted as α) and the best solution encountered for the min player (denoted as β) at a given node of the tree. At a min node n, while searching the subtrees of the children of n, if each minimum value M among all children already searched is smaller than α, then we stop searching under n. In the pseudocode below this is shown in lines 6-8, where variable M stores the minimum value of all children of n. If $M \leq \alpha$ (line 7), then the procedure returns the value of M (line 8). By retuning M the procedure stops searching in the subtrees rooted at the children of the current node n. Note that once the value of M is returned due to pruning, it will be ignored as it will be consider either in a max operation where there is a value larger than M or in a min operation where there is a value smaller than M. The value of β represents the best solution encountered along the path from the root of the tree to n for the min player. Thus, we update the β value if $M < \beta$ (line 9). The updated β will passed in the recursive calls of the algorithm that will search the subtrees of the children of the current node. The β value can be used to prune the children of max nodes in these subtrees.

Similarly to the computation performed at min nodes, at a max node, if the value M of the best child of a node n is larger or equal to the value of β, then we can stop searching under that max node.

Algorithm 3 AlphaBeta

```
Require: Node $h$, $\alpha = -\infty$, $\beta = \infty$
Ensure: The value of node $h$.
 1: if $h \in Z$ then
 2:     return $U(h)$
 3: $M = \begin{cases} -\infty, & \text{if } P(h) = \max \\ \infty, & \text{if } P(h) = \min \end{cases}$
 4: for each $a \in A(h)$ do
 5:     if $P(h) = \min$ then
 6:         M ← min(AlphaBeta($T(h,a), \alpha, \beta$), $M$)
 7:         if $M \leq \alpha$ then
 8:             return $M$
 9:         $\beta = \min(\beta, M)$
10:     if $P(h) = \max$ then
11:         M ← max(AlphaBeta($T(h,a), \alpha, \beta$), $M$)
12:         if $M \geq \beta$ then
13:             return $M$
14:         $\alpha = \max(\alpha, M)$
15: return $M$
```

As an exercise, you should trace the behavior of Minimax with Alpha-Beta pruning in the examples of this section. You will see that the procedure returns the optimal strategies without expanding the entire tree.

Time Complexity of Alpha Beta Pruning

The **worst case** for Alpha Beta pruning is when the max nodes are ordered with increasing value and the min nodes are ordered in decreasing value. In this case Minimax with and without Alpha Beta pruning have the same complexity: $O(b^d)$. The **best case** happens if the max nodes ordered with decreasing value and the min nodes with increasing value. In this case the time complexity is of $O(b^{\frac{d}{2}})$. The **average case** happens if nodes are ordered randomly in the tree, which results in a time complexity of $O(b^{\frac{3d}{4}})$.

Limited Resources

For many practical problems we do not have enough time to search the entire tree, so that the values of the leaf nodes can be propagated up the tree. Here is what we do in such cases.

- Search to a certain depth and use a heuristic function to approximate the value of the reached nodes.
- No guarantee that the procedure will find the actual value of the game.
- Usually the deeper you search, the stronger the returned strategy is (some pathological cases have been reported in the literature).
- One should use iterative deepening to have an anytime algorithm.

Evaluation Functions

Evaluation functions should order nodes in the same way that the actual values of the game would order them (e.g., victory > draw > defeat in game of chess). Moreover, evaluation functions must be efficient, as they will take up time that could be used for searching deeper.

Here is an example of an effective evaluation function for chess. Each piece has a value related to it (queen = 4, rook = 5, bishop = knight = 3, pawn = 1) and for a given node one just sums the total value for each player. Intuitively, the player with more important pieces will have a higher score according to this heuristic.

4.12.2 Limitations of Minimax Search

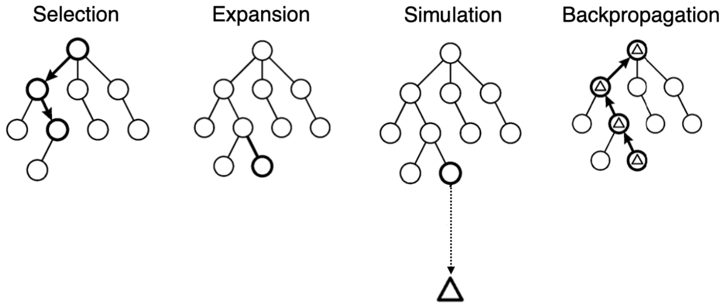
The quality of the strategy Minimax search (with or without Alpha Beta pruning) returns will depend on the quality of the heuristic function employed. If the heuristic ranks the leaf nodes incorrectly, then there is not much hope that the procedure will be able to derive a strong strategy of the game. The game of Go is an example where it is hard for us to come up with good heuristic functions to guide the search. That is why Minimax search with Alpha Beta pruning performs so poorly in Go. There are, however, other very successful alternatives for these games. For example, AlphaZero uses a different search algorithm (Monte Carlo tree search) to learn good heuristic functions to guide the search for games such as Go and chess.

4.13 Monte Carlo Tree Search

Minimax search, with or without Alpha Beta pruning can be used to approximate a Nash equilibrium strategy for large zero-sum extensive-form games. This can be achieved by using an iterative-deepening approach with a heuristic function to evaluate the leaves of the expanded trees.

We will study now a family of search algorithms, known as Monte Carlo tree search (MCTS), that approximate the minimax value of games through sampling, which bypasses the need of iterative deepening and the use of a heuristic function. Eliminating the use of heuristic function is particularly important in domains such as Go where it is hard to encode domain knowledge in a function.

The figure below illustrates the four steps of the MCTS procedure.[2]



An MCTS algorithm expands a sub-tree of the tree describing the game. In each iteration the algorithm selects a node to be expanded (i.e., to be added to the MCTS tree, which initially starts only with the current state of the game as the root). This **selection** is done through a **tree policy**. The procedure starts at the root n of the tree and it chooses one of n's child nodes. This procedure is repeated recursively from n's chosen child until reaching a node that has one or more child nodes that are not part of the tree yet. One of such child nodes is added to the tree. This is called the **expansion** step.

Once a node c is added to the MCTS tree, we evaluate it with a **default policy**, by playing out the game from c until a terminal node. This is known as a **simulation** step. The utility of that terminal node is used to approximate the value of node c (utility depicted as a triangle in the figure) and the value of all nodes on the path from the root of the tree to c. This last step is known as **backpropagation**.

These four steps: selection, expansion, simulation, and backpropagation are repeated while there is time available for searching. Once a time limit is reached, the algorithm returns the action that leads to the child node with highest value in the tree. In the case of games, the returned action would be played in the actual game. Once it is again the turn of the player, we repeat the the MCTS procedure to choose the next action. MCTS acts as an anytime algorithm as it is able to return an action for any reasonable time bound.[3]

By choosing a different tree policy and default policy we instantiate different MCTS algorithms. Here is a possible instantiation.

- ϵ-greedy policy as tree policy: select best action with probability $1-\epsilon$; select a random action otherwise.
- random policy as default policy: randomly choose an action at every node.

4.13.1 Upper Confidence Bounds Applied to Trees

Another instantiation of a MCTS algorithm is upper confidence bounds applied to trees (UCT), which uses UCB1, an algorithm for **multi-armed bandit** problems, as the tree policy.

In a bandit problem one is given a set of slot machines (arms) and a limited amount of k samples (number of times one can play with a slot machine). One then tries to maximize their profit given that we do not have previous information about the machines. A way of maximizing profits is by minimizing the **regret**.

[2] Image is from the paper "A Survey of Monte Carlo Tree Search Methods" by C. Browne et al. (2012).

[3] It will not be able to make an informed decision in real-time games where the branching factor can be very large and the time available for searching is in the order of milliseconds. In this kind of games the search might fail even to evaluate each action of the root node at least once.

- The reward for playing machine i at time t is given by $X_{i,t}$.
- The regret for policy $\pi(t)$, which is a function receiving a time step and returning the machine i that should be chosen at t, in a sequence of k rounds if given by,

$$\max_i \mathbb{E}[\sum_{t=1}^{k} X_{i,t}] - \mathbb{E}[\sum_{t=1}^{k} X_{\pi(t),t}]$$

The first term above computes what is the expected utility value if one plays in the best machine all k iterations; the second term gives the expected utility given a policy for choosing which machines to play. The difference between these two quantities gives the regret: how much one could have gained if they had always played the best machine versus playing the machines they actually played.

Here is an example of a multi-armed bandit problem with four machines.

If you play each machine once and obtain the reward values of 10, 0, -5, -10 for M1, M2, M3, and M4, respectively. Which machine should be played next? One needs to balance **exploitation** (take advantage of the best known machine) and **exploration** (discover better machines).

UCB1

UCB1 provides a policy that balances exploration and exploitation, meaning that the regret grows logarithmically with the number of trials. UCB1 is given by the following.

- At each step choose the machine j that maximizes the following

  $$\bar{X}_j + \sqrt{\frac{2 \cdot \ln(n)}{n_j}},$$

  where $\bar{X}_j$ is the average empirical reward given by machine j (exploitation term), n_j is the number of times machine j has been chosen, and n is the total number of trials; the last term is known as the exploration term.

If a machine j provides good rewards (large $\bar{X}_j$ value), then it will have a high UCB1 value, allowing it to be chosen often. However, even if j does not provide good rewards, the exploration term of j grows as j isn't chosen for a long time. UCB1 ensures that all machines are eventually explored.

UCT

UCT is a MCTS algorithm that uses UCB1 as the tree policy. The formula used in UCT is slightly different from the UCB1 formula.

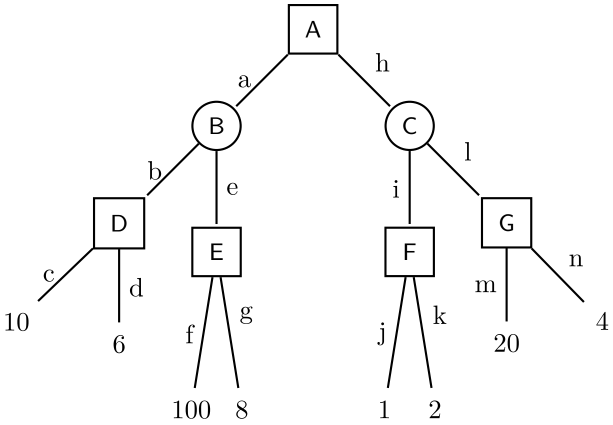
$$\bar{X}_j + c \times \sqrt{\frac{\log(n)}{n_j}}$$

Here, c is a tunable exploration parameter: larger values of c will allow the algorithm to explore more. The formula is different because the probability of sampling nodes below a node n in the tree changes over time. The use of an appropriate constant c allows the theoretical bound to hold.

In order to implement UCT we need to store the following values in memory:

1. $N(n)$: number of times n was visited.
2. $N(n, a)$: number of times a was chosen at n.
3. $Q(n, a)$: average utility value of the simulations that went through n and that a was chosen.

We are going to run UCT with $c = 2$ in the following two-player zero-sum extensive form game, where squares represent max nodes and circles represent min nodes.



1. First iteration:
   (a) Tree policy selects a.
   (b) Expand node B.
   (c) Simulation reaches leaf node with utility of 6.
   (d) Update $N(A, a) = 1$, $Q(A, a) = 6$ e $N(A) = 1$
2. Second iteration:
   (a) Tree policy selects h.
   (b) Expand C.
   (c) Simulation reaches leaf node with utility of 1.
   (d) Update $N(A, h) = 1$, $Q(A, h) = 1$ e $N(A) = 2$
3. Third iteration:
   (a) Tree policy selects a, b.

$$6 + 2 \times \sqrt{\frac{log(2)}{1}} = 7.09 \text{ (for a)}$$

$$1 + 2 \times \sqrt{\frac{log(2)}{1}} = 2.09 \text{ (for h)}$$

   (b) Expand D.
   (c) Simulation reaches leaf node with utility of 10.
   (d) Update:
      i. $N(B,b) = 1$, $Q(B,b) = 10$, $N(B) = 1$.
      ii. $N(A,a) = 2$, $Q(A,a) = 8$, $N(A) = 3$.

4. Fourth iteration:

   (a) Tree policy selects a, e.

$$8 + 2 \times \sqrt{\frac{log(3)}{2}} = 8.97 \text{ (for a)}$$
$$1 + 2 \times \sqrt{\frac{log(3)}{1}} = 2.38 \text{ (for h)}$$

   (b) Expand E.
   (c) Simulation reaches leaf node with utility of 100.
   (d) Update:
      i. $N(B,e) = 1$, $Q(B,e) = 100$, $N(B) = 2$.
      ii. $N(A,a) = 3$, $Q(A,a) = 38.67$, $N(A) = 4$.

5. Fifth iteration:

   (a) Tree policy selects a, b.

$$38.66 + 2 \times \sqrt{\frac{log(4)}{3}} = 39.55 \text{ (for a)}$$
$$1 + 2 \times \sqrt{\frac{log(4)}{1}} = 2.55 \text{ (for h)}$$

$$10 - 2 \times \sqrt{\frac{log(2)}{1}} = 8.90 \text{ (for b)}$$
$$100 - 2 \times \sqrt{\frac{log(2)}{1}} = 98.90 \text{ (for e)}$$

   (b) $\cdots$

If the search finished in this iteration, then UCT would return action a for the max player.

UCT Pseudocode

```
def UCT(Node $n$):
create-node($n$)
while not finished:
  search($n$)
return select-action($n$)
```

```
def search(Node $n$):
  $[(n,a),\cdots,(n_t,a_t)] =$ select-path($n$)
  $u =$ default-policy($T(n_t,a_t)$)
  backpropagate($[(n,a),\cdots,(n_t,a_t)],u$)
```

```
def default-policy(Node $n$):
  while $n$ not in $Z$:
    $a =$ default($n$)
    $n = T(n,a)$
  return $U(n)$
```

```
def select-path(Node $n$):
  $c$ exploration constant
  path $= []$
  while $n$ not in $Z$:
    if $n$ not in tree:
      create-node($n$)
      return path
    $a =$ select-action-UCB1($n,c$) # tree policy
    path.append($(n,a)$)
    $n = T(n,a)$
  return path
```

```
def select-action-UCB1(Node $n$, $c$):
  if $P(n) = \max$:
    $a = \arg\max_{a \in A(h)}(Q(n,a) + c\sqrt{\frac{\log(N(n))}{N(n,a)}})$
  if $P(n) = \min$:
    $a = \arg\min_{a \in A(n)}(Q(n,a) - c\sqrt{\frac{log(N(n))}{N(n,a)}})$
  return $a$
```

```
def backpropagate(Path $H$, $u$):
  for $n,a$ in $H$:
    $N(n) += 1$
    $N(n,a) += 1$
    $Q(n,a) = Q(n,a) + \frac{u - Q(n,a)}{N(n,a)}$ # online algorithm for computing averages
```

```
def create-node(Node $n$):
  insert $n$ in tree
  $N(n) = 0$
  for $a$ in $A(n)$:
    $N(n, a) = 0$
    $Q(n, a) = 0$
```

Chapter 5

Constraint Satisfaction

5.1 Constraint Satisfaction Problems

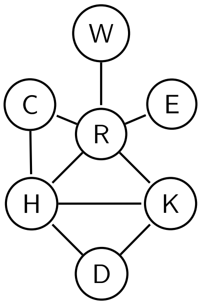
In the previous lectures we studied heuristic search algorithms for solving pathfinding problems and local search algorithms for solving combinatorial search problems. The heuristic search and local search algorithms we studied perform some reasoning in the sense that they use the information a heuristic function provides to guide their search. The heuristic function can be provided by an opaque model and the search algorithm isn't responsible for the reasoning the heuristic function performs.

In this lecture we will study Constraint Satisfaction Problems (CSPs), which are problems with **factored states**. A state is factored if we can access its parts and reason about them. The search algorithm we will study in the next two lectures can perform lots of inference and reasoning while searching for a solution.

CSPs can be applied to many important problems such as scheduling tasks (timetabling, courses to offer), hardware configuration, model checking, and many others. We will see mostly toy problems in this course because they are easy to understand, which is important when we are learning new algorithms. Nevertheless, the algorithms we study are general and can be used to solve any problem that can be formulated as a CSP.

5.1.1 Example: Map-Coloring Problem

Let's describe CSPs with a map-coloring example. The graph below represents the kingdoms from the novel "A Game of Thrones: A Song of Ice and Fire". In this graph, W represents the kingdom of Winterfell and K King's Landing, and so on. Two kingdoms are connected by an edge if they share a border. Our task is to assign a color to each kingdom such that neighboring kingdoms have different colors.



A CSP is defined as a set of variables (kingdoms in our example), a domain for each variables (the colors we can assign to each kingdom), and a set of constraints (neighboring kingdoms can't be assigned the same color). As a more concrete example for the coloring problem described above, we have:

- Variables: W, R, E, C, H, K, D;
- Domain: {red, green, yellow};
- Constraints: {(W, R) $\neq$}, {(R, E) $\neq$}, {(C, R) $\neq$}, {(C, H) $\neq$}, {(H, K) $\neq$}, {(R, H) $\neq$}, {(K, R) $\neq$}, {(H, K) $\neq$}

A solution to a CSP is an assignment of values to each variable such that none of the constraints are violated. Here is a solution to our map-coloring problem: W = green, R = red, C = green, E = green, H = yellow, K = green, D = red.

5.1.2 Example: Sudoku

Sudoku is a puzzle where we are given a 9 × 9 grid with only a few cells filled with a number. In this lecture we will use the smaller 4 × 4 Sudoku because they are easier to draw and understand. Let's see an example.

	1	2	3	4
1	2	1		
2	3			
3				3
4			1	

Each empty cells has to be filled with one of the following values: $1, 2, 3, 4$, such that no row, column, or unit have repeated numbers. A unit is a square of size 2 × 2; there are four units in the 4 × 4 Sudoku grid. The top-left corner of the 2 × 2 units are the cells (1, 1), (1, 3), (3, 1), and (3, 3). To illustrate the constraint posed on units, cell (2, 2) can only assume the value of 4 because the values of $1, 2$ and 3 were already used in that unit. The grid below shows a solution to the problem above.

	1	2	3	4
1	2	1	3	4
2	3	4	2	1
3	1	2	4	3
4	4	3	1	2

The solution for this problem can be easily obtained by filling the empty cells that are easy to fill, such as (2, 2). After assigning 4 to (2, 2) we know that cell (2, 3) must be assigned a 2. This is because its row already has 3 and 4 and its column has a 1. All cells of this puzzles can be filled this way. The algorithm we study in this lecture uses exactly this strategy for solving (or at least simplifying) CSPs.

Sudoku can be formulated as a CSP as follows.

- Variables: One $X_{i,j}$ for each cell in the matrix.
- Domain: $\{1, 2, 3, 4\}$

- Constraints:
  - All different: $\{X_{1,1}, X_{1,2}, X_{1,3}, X_{1,4}\}$
  - All different: $\{X_{1,1}, X_{2,1}, X_{3,1}, X_{4,1}\}$
  - All different: $\{X_{1,1}, X_{1,2}, X_{2,1}, X_{2,2}\}$
  - $\dots$

5.1.3 Types of Constraints

The constraints that involve two variables are known as binary constraints. The map-coloring problem was formulated with a set of binary constraints (e.g., $\{(\text{W, R}) \neq\}$). The Sudoku puzzle was formulated with global constraints. Despite the name, global constraints don't have to involve all variables, but more than 3 variables. Sudoku could also be formulated with binary constraints (e.g., $\{(X_{1,1}, X_{1,2}) \neq\}$).

The last type of constraint are the unary constraints. As you might have guessed, unary constraints involve a single variable. As an example, maybe the king of Winterfell does't like the color yellow and they set a unary constraint to variable W that it can't be assigned the value of yellow. Unary constraints are easy to deal with in a preprocessing step. We just need to remove from the domain of a variable the values that cannot satisfy the unary constraints of that variable.

5.2 Arc Consistency

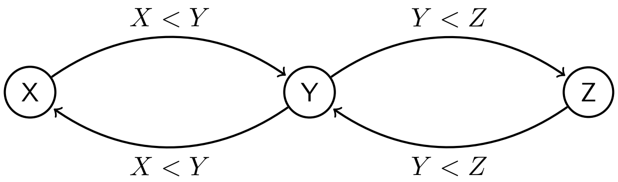
We say that a variable is arc consistent if it satisfies its binary constraints. Let's consider the following example. We have two variables Y and X and they must satisfy the constraint $Y = X^2$. The domain of both variables is the set of integers $\{0, 1, 2, \cdots, 9\}$. The graph describing this problem has two vertices (one for each variable) and they are connected by a single edge representing the constraint $Y = X^2$.

The variables aren't arc consistent. For example, if we assign $X = 9$, there won't be a value in Y that satisfies the constraint $Y = X^2$ (81 is not in the domain of Y). For this reason we can remove the values $4, 5, 6, 7, 8, 9$ from the domain of X without affecting the solution of the problem, as these values cannot be used. Similarly, if we assign $Y = 2$, we don't find a value in X's domain that will satisfy the constraint ($\sqrt{2}$ isn't in the domain of X). We can remove $2, 3, 5, 6, 7, 8$ from the domain of Y, which will leave the following domains for X and Y: $\{0, 1, 2, 3\}$ and $\{0, 1, 4, 9\}$, respectively. Both variables are now arc consistent.

By enforcing arc consistency we were able to simplify the problem while not changing its solution. In some cases we are able to solve the problem. This is exactly what we did we our 4×4 Sudoku puzzle. We made the variable $X_{2,2}$ arc consistent by removing from its domain the values that would violate one of the constraints in which the variable was involved. In the case of the Sudoku puzzle we managed to reduce the domain of all variables to a single value. In this case, we can assign each value to its variable and the problem is solved.

5.2.1 AC3

The algorithm AC3 can be used to enforce arc consistency in CSP problems. We will study AC3 through an example. Our problem has variables X, Y, Z with domain $\{0, 1, 2, 3\}$ and they must satisfy the constraint $X < Y < Z$. AC3 considers the arc in two directions, that is why we have arcs going both ways in the graph below. When considering arc (X, Y) AC3 simplifies the domain of X, while considering arc (Y, X) AC3 simplifies the domain of Y. Despite both arcs representing exactly the same constraint, they are treated differently during AC3's execution, as we will see in the example below.



Before we show the example of AC3, let's look at the domains of the variables and convinced ourselves that the variables aren't arc consistent. If we make $X = 3$, then there isn't any value in Y that will satisfy the constraint $X < Y$. Similarly, if we make $Z = 0$, then there is no value in Y that will satisfy $Y < Z$. The variables are not arc consistent, something we will fix with AC3.

AC3 keeps a set of arcs in memory and, in every iteration, it removes an arc from the set (the order in which the arcs are removed from the set isn't important) and it tries to simplify the domain of the outgoing variable, e.g., X if the arc is (X, Y) and Y if the arc is (Y, X). The variables are arc consistent once the set becomes empty. The table below shows the execution of AC3 for our example.

Arc to Process	`Q`	`Action`
-	$(X, Y), (Y, X), (Y, Z), (Z, Y)$	-
(X, Y)	$(Y, X), (Y, Z), (Z, Y)$	Remove 3 from X
(Y, Z)	$(Y, X), (Z, Y)$	Remove 3 from Y; add (X, Y) to `Q`
(X, Y)	$(Y, X), (Z, Y)$	Remove 2 from X
(Y, X)	(Z, Y)	Remove 0 from Y; add (Z, Y) to `Q`
(Z, Y)	$\{\}$	Remove 0 and 1 from Z

AC3 starts by initializing `Q` with all arcs in the graph. In its first iteration it removes (X, Y) from Q and it removes from X's domain all values that do not satisfy the constraint with Y. Here, there is no value of Y that would allow us to satisfy the constraint $X < Y$ if we assign 3 to X, so 3 is removed from the domain of X. Similarly, in the next iteration, AC3 removes (Y, Z) and it removes 3 from the domain of Y.

Notice that once we remove 3 from the domain of Y the value of 2 can no longer be assigned to X as $Y = 3$ was the assignment that made $X = 2$ consistent. If $Y = 3$ is no longer possible, 2 has to be removed from the domain of X. AC3 achieves this by reinserting (X, Y) into `Q`. The general rule for reinsertions is the following: if we change the domain of a variable X_i through arc (X_i, X_j), then we need to reinsert all arcs (X_k, X_i) with $k \neq j$ into `Q`. This is exactly what we are doing in this step of the execution of AC3. The arc (X, Y) is reinserted in `Q` because we reduced the domain of Y through arc (Y, Z). Note that we don't need to reinsert (Z, Y) in `Q` (if it wasn't already there) because the modification that we made to the domain of Y was based on the same constraint represented by arc (Z, Y). Therefore, removing 3 from Y must not affect the domain of Z as 3 was removed because there was no value in the domain Z that would satisfy the constraint $Y < Z$.

The next arc AC3 processes is again (X, Y). Now it removes 2 from X's domain. In the next iteration AC3 processes (Y, X), which allows it to remove 0 from Y. Note that we would need to reinsert (Z, Y) in `Q` if it wasn't already there. Finally, the last arc processed is (Z, Y), which allows us to remove 0 and 1 from Z's domain. Once `Q` becomes empty the variables must be arc consistent, as can be verified.

The pseudocode below presents AC3.

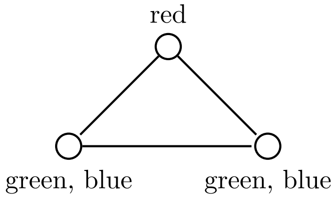
Once we finish running AC3 we can witness three different outputs. First, one of the variables might have an empty domain. When this happens the CSP doesn't have a solution. Second, all variables have a single value in their domain. This means that AC3 managed to solve the CSP (similarly to how we did for the small Sudoku puzzle). Third, variables have one or more values in their domain. This means that we didn't

```
def AC3(CSP):
  Initialize set Q with all arcs
  while $Q \neq \emptyset$:
    $(X_i, X_j)$ = Q.pop()
    if revise($X_i, X_j$):
      for $X_k$ in {neighbors of $X_i$} - $X_j$:
        $Q$.add($X_k, X_i$)
```

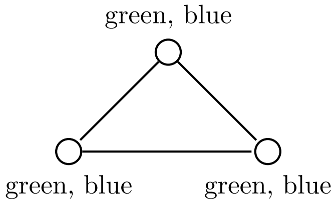
```
1 def revise(Arc $(X_i, X_j)$):
2   removed = False
3   for $d$ in domain of $X_i$:
4     if no value $d'$ in the domain of $X_j$ satisfies $(X_i = d, X_j = d')$
5       remove $d$ from $X_i$
6       removed = True
7   return removed
```

find the solution yet, but we might have significantly simplified the problem, which will make it easier to solve the problem with the backtracking approach we will study in our next lecture.

The graph below shows an example of a possible output of AC3 for a map-coloring problem, where the variables are arc consistent, but the solution still needs to be derived.



The next example shows another possible output of AC3 for a different map-coloring problem, where all variables are also arc consistent, but the problem has no solution (AC3 is unable to detect that the problem has no solution).

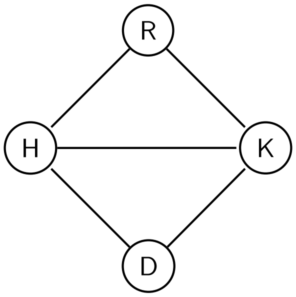


The time complexity of AC3 is $O(cd^3)$ if the problem has c arcs and d values in the domain of each variable. The cost of processing each arc is d^2 as we must evaluate all d values of variable X_j in the if statement of line 4 of the `revise` function. This adds up to a complexity of $O(cd^2)$. However, each arc can be reinserted in `Q`. The maximum number of times an arc (X_i, X_j) can be reinserted in `Q` is d. This is because we reinsert (X_i, X_j) only after we removed one or more values from the domain of X_j. Thus the complexity is $O(cd^3)$.

5.3 Backtracking for Solving CSPs

AC3 is able to solve CSP problems by ensuring arc consistency of the variables in the problem. In many cases, however, AC3 will not be able to solve the problem, but only simplify it. Next we will see how backtracking (an algorithm similar to DFS) can solve CSPs by enumerating the possible variable assignments. We will also see how to combine the reasoning that AC3 performs with the Backtracking search.

Let us consider a map-coloring problem with variables R, H, K, D and domains $1, 2, 3$, where each integer represents a color. The following graph shows the binary ($\neq$) constraints of the problem.



How many different variable assignments does this problem support? In general, if we have n variables with d different values in the domain of each variable, then there are d^n possible assignments. Thus, in the worst-case scenario we should expect backtracking to not evaluate more than d^n assignments.

5.3.1 Backtracking - Attempt 1

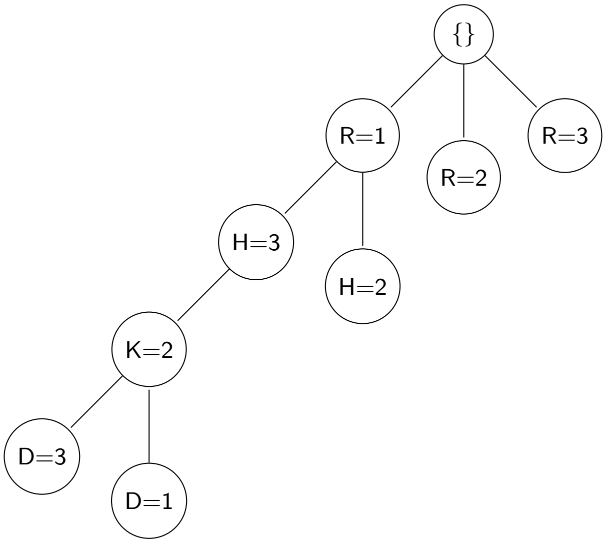
Our first attempt to search over the assignment space is to define a search tree where the root node represents an empty assignment for the variables and, in the next level of search, we consider all possible assignments for each variable in the problem. In our example, the root of the tree would have one child for each of the following assignments: $R = 1$, $R = 2$, $R = 3$, $H = 1$, $H = 2$, $H = 3$, $K = 1$, $K = 2$, $K = 3$, $D = 1$, $D = 2$, $D = 3$. The root of this tree has $n \cdot d$ children. The next level we would have to consider $n - 1$ variables, as we have set of the value of one of the variables in the previous level, which would account for $(n - 1)d$ grandchildren for each child of the root. The total number of nodes in this tree is

$$nd(n-1)d(n-2)d \cdots d = n!d^n \,.$$

This tree is much larger than the d^n total number of distinct assignments for the problem.

5.3.2 Backtracking - Attempt 2

Our first attempt to design a backtracking algorithm is far from ideal as we have to search more assignments than the total number of assignments for the problem—our first attempt must be considering repeated assignments during search. We can ensure that each assignment is searched at most once by assigning a value to a single variable at every level of the tree. We present below part of the search tree for our example.



The first level of the tree assigns a value to R, while the second assigns a value to H, and so on. Every node in the tree has d children, which adds to d^n leaf nodes—the exact number of different value assignments that we have in the problem. The pseudocode below shows the backtracking procedure.

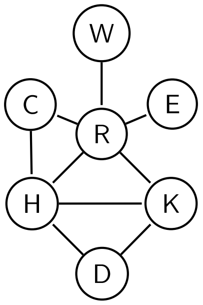
```
 1 def Backtracking(A):
 2   if A is complete: return A
 3   var = select-unassigned-var(A)
 4   for d in select-domain-value(var, A):
 5     if d is consistent with A:
 6       {var = d} in A
 7       rb = Backtracking(A)
 8       if rb is not failure:
 9         return rb
10       {var = $\emptyset$} in A
11   return failure
```

Backtracking is implemented as a recursive function that receives a partial assignment A and tries all possible values d in the domain of one of the variables in the CSP. If Backtracking encounters a solution, then it stops searching and returns the solution (see lines 2 and 7–9). If Backtracking finds that assigning a given value d to a variable results in failure (the partial assignment renders the problem unsolvable), then it erases that value (line 10) in the assignment and tries another one (another iteration of the for-loop). In line 5, Backtracking checks if it can assign the value of d to the variable. As an example, this if statement prevents the search from assigning the same color to two neighboring kingdoms in a map-coloring problem.

The functions `select-unassigned-var` and `select-domain-value` decide which variable and which value are attempted next in search. As we will discuss in the next sections, the choice of variable and value can substantially influence how fast we can find a solution to the CSP.

5.3.3 Which Variables to Try Next?

Let us consider the coloring problem on the graph below, where the colors allowed are red, green, and orange.



If we assign red to K and green to H, then we will have the option of assigning orange to R and the options of assigning either orange or red to C. If we only had the option of choosing either R or C as the next variable in our backtracking procedure, then choosing R instead of C as the next variable will likely reduce the size of the search tree. This happens because R's domain has fewer values than C's domain. The heuristic that chooses the variable with smallest domain is known as the **Minimum Remaining Value (MRV)**. We will now try to understand why this heuristic works so well in practice.

In our example Backtracking has produced the partial assignment $K = \text{red}$ and $H = \text{green}$. There are two possible scenarios for a partial assignment: it either renders the problem unsolvable or it is part of a solution to the problem. Let us consider both cases as Scenario 1 (unsolvable) and Scenario 2 (part of a solution).

Scenario 1: If the partial assignment renders the problem unsolvable, then ideally we will prove the problem to be unsolvable as quickly as possible so we can try other assignments to variables K and H. In terms of search, we want to minimize as much as possible the size of the subtree rooted at the partial assignment $K = \text{red}$ and $H = \text{green}$. If we choose R as the next variable, then there is only one subtree for us to search over, which is the subtree given by the assignment of orange to R. If we discover that the domain of any variable becomes empty while searching in this subtree, then we know that the partial assignment $K = \text{red}$ and $H = \text{green}$ cannot lead to a solution and we would then backtrack. Let us suppose that we choose C instead of R. Since there are two values in the domain of C, we would have to search through two subtrees to prove that the initial assignment $K = \text{red}$ and $H = \text{green}$ cannot lead to a solution. The MRV heuristic uses the number of children of a node n representing a partial assignment as a proxy for the time that it takes for Backtracking to prove that the partial assignment renders the problem unsolvable.

Scenario 2: If the partial assignment is part of a solution, then we would like to find a complete assignment representing a solution as quickly as possible. And it is much easier to guess the right assignment if we have fewer options than if we have many options. In our example, because R has only one option, we will choose the right value for R. Variable C has two options and, if only one of the options can lead to a solution, then we have $1/2$ chance of choosing the right one. Note that once we choose R because our chances of choosing the right value are larger, we are then able to further simplify the domain of C, so when the search chooses C at a deeper level of tree, we have better chances of immediately assigning the right value to C.

In summary, the MRV heuristic tries to prove a partial assignment to be unsolvable as quickly as possible, if the partial assignment cannot lead to a solution, and it tries to find a solution as quickly as possible if the partial assignment can lead to a solution.

You might have noticed that the example we used with variables K, H, C, and R was carefully designed to illustrate MRV in action. What if we start from an empty assignment? What would MRV do? All variables start with the same domain in the map-coloring problem, so there would be a tie among all variables in the problem according to the MRV heuristic. Here is an effective heuristic for breaking these ties.

The degree heuristic (DH) chooses the variable with largest degree in the constraint graph as the next variable to be searched in the Backtracking procedure. In our coloring-map problem DH chooses R because it is the variable associated with the largest number of binary constraints (its degree is 5). DH attempts to reduce the branching factor of the tree by assigning a value to the variable with largest the degree. In our example, if we assign red to R, then we can remove red from the domain of W, E, K, H, C, which reduces the number of subtrees rooted at each of these variables from 3 to 2. Choosing variables with smaller degrees would cause smaller reductions to the branching factor of the tree. For example, if we choose W and assign the color red to it, then we would only reduce the domain of R. As an implementation of the function `select-unassigned-var`, we often use a combination of MRV and DH, where the former is the main heuristic and the latter is used as a tie-breaker.

5.3.4 Which Values to Try Next?

The choice of which values to try first in search can also dictate how fast we can find a solution to a CSP. A heuristic that works well in practice is the Least-Constraining-Value (LCV) heuristic, which chooses the value for a variable that will leave more options to other variables. For example, if we assign $K =$ red and $H =$ green, we can choose either red or orange for variable C. If we choose to assign orange to C then we will leave the domain of R empty; if we choose red instead, then we leave the domain of R with orange.

The LCV heuristic is only effective for partial assignments that can still lead to a solution. LCV chooses a value that is least constraining because it hopes to find a solution more quickly that way. If the partial assignment renders the problem unsolvable, then it is unlikely that LCV will affect the speed of search. This is because, once we choose a variable `var` (see line 3 of the pseudocode for the Backtracking procedure), we have to iterate through all possible values d in the domain of `var` before returning failure (see line 11); the order in which we go through the values d will not affect the search. Thus, LCV can only speed up the search when the partial assignment in which the heuristic is applied can lead to a solution.

5.4 Search and Inference

In the previous lecture we saw that AC3 can be used as a pre-processing step to simplify a CSP, or possibly even solve the CSP. If AC3 cannot find a solution, then we can use Backtracking on the simplified problem to either find a solution or prove that the problem has no solution.

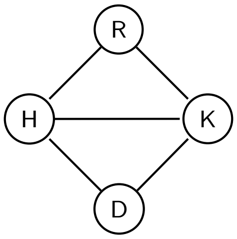
The kind of inference AC3 performs can be even more powerful if it is interleaved with search. That is, every time we assign a value to a variable during search we can use AC3 to try to further simplify the problem or even solve it. The pseudocode below shows a modified version Backtracking that performs reasoning whenever the search assigns a value to a variable (see lines 7 and 8). If the inference step (e.g., running AC3 on the partial assignment) returns failure, then we do not need to recursively call Backtracking to search in the subtree, we can backtrack right away and try a different value d to variable `var`. If the inference is able to solve the problem, then Backtracking is called recursively and the solution is recognized in line 2 of the new call; the search unrolls the recursive calls and returns the solution.

We will see examples of two inference methods: Forward Checking and AC3 as implementations of the method `inference` in line 7.

5.4.1 Forward Checking

In Forward Checking, whenever Backtracking assigns a value to a variable X, it ensures that all the neighboring variables in the constraint graph are arc consistent with X. Let's consider the following map-coloring problem where we can assign colors represented by the numbers $1, 2, 3$.

```
def Backtracking(A):
  if A is complete: return A
  var = select-unassigned-var(A)
  for d in select-domain-value(var, A):
    if d is consistent with A:
      {var = d} in A
      ri = inference(var, A)
      if ri is not failure:
        rb = Backtracking(A)
        if rb is not failure:
          return rb
      {var = ∅} in A
  return failure
```



The table below shows how Backtracking with Forward Checking simplifies the domains of the variables during search. Before the search starts all domains are complete. Backtracking first assigns a value to R and Forward Checking makes H and K arc consistent with R by removing 1 from their domains (see the row for assignment $R = 1$ in the table). In the next level Backtracking assigns $H = 2$ and Forward Checking tries to simplify the domains of the neighboring variables R, K, D by making them arc consistent with H. Once Backtracking assigns $D = 3$, Forward Checking finds that K has an empty domain. Thus the inference returns failure and it allows Backtracking to move on to the next value for variable D, which is 1 (the assignment $D = 1$ isn't shown in the table).

Domains				Assignment
R	H	K	D	
$\{1, 2, 3\}$	$\{1, 2, 3\}$	$\{1, 2, 3\}$	$\{1, 2, 3\}$	{}
$\{1\}$	$\{2, 3\}$	$\{2, 3\}$	$\{1, 2, 3\}$	{R = 1}
$\{1\}$	$\{2\}$	$\{3\}$	$\{1, 3\}$	{R = 1, H = 2}
$\{1\}$	$\{2\}$	$\{\}$	$\{3\}$	{R = 1, H = 2, D = 3}

Forward Checking is able to simplify a partial assignment and potentially reduce the amount of search required to solve the problem. However, Forward Checking is unable to detect failure in some obvious cases. Let's see an example in the table below, where Backtracking assigns $R = 1$ and then $D = 2$. Forward Checking leaves both H and K with a non-empty domain, while the partial assignment clearly does not have a solution since H and K cannot be assigned the same color.

Domains				Assignment
R	H	K	D	
$\{1,2,3\}$	$\{1,2,3\}$	$\{1,2,3\}$	$\{1,2,3\}$	{}
$\{1\}$	$\{2,3\}$	$\{2,3\}$	$\{1,2,3\}$	{R = 1}
$\{1\}$	$\{3\}$	$\{3\}$	$\{2\}$	{R = 1, D = 2}

5.4.2 Maintaining Arc Consistency

Maintaining Arc Consistency (MAC) is a reasoning method that runs AC3 whenever Backtracking assigns a value to a variable. Specifically, whenever the search sets a value to X_i, MAC runs AC3 with the set `Q` initialized with all arcs (X_j, X_i). That way, AC3 can ensure arc consistency based on the newly assigned value of X_i. MAC correctly detects that $R = 1$ and $D = 2$ cannot lead to a solution. This is because AC3 recursively adds other arcs to `Q` while ensuring arc consistency of neighboring variables. The table below shows this example in detail.

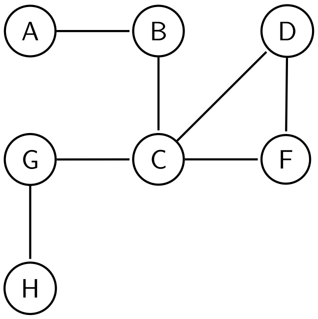
Domains				Assignment
R	H	K	D	
$\{1,2,3\}$	$\{1,2,3\}$	$\{1,2,3\}$	$\{1,2,3\}$	{}
$\{1\}$	$\{2,3\}$	$\{2,3\}$	$\{1,2,3\}$	{R = 1}
$\{1\}$	{}	$\{3\}$	$\{2\}$	{R = 1, D = 2}

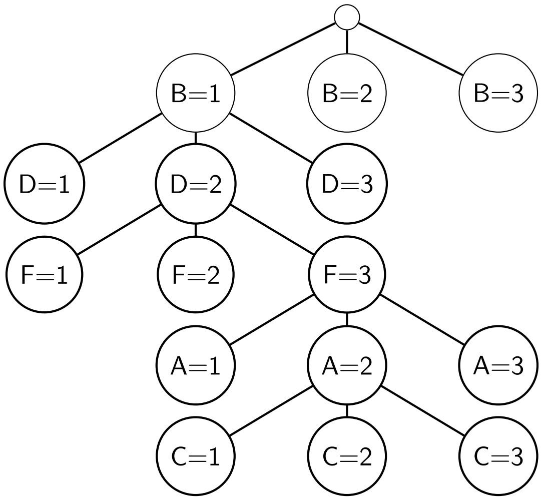
Once Backtracking assigns $R = 1$, MAC runs AC3 with `Q` initialized with (K, R) and (H, R). AC3 then simplifies the domains of K and H to $\{2, 3\}$. In the next step of search, Backtracking assigns $D = 2$ and MAC runs AC3 with `Q` initialized with (K, D) and (H, D). Once AC3 modifies the domain of K to $\{3\}$, it also adds the arcs (R, K), (H, K), and (D, K) to `Q`. Then, when (H, K) is removed from `Q` and processed, AC3 removes 3 from H's domain, which flags the original assignment to be unsolvable.

5.5 Intelligent Backtracking

The backtracking procedure we have studied for solving CSPs is known as *chronological backtracking.* This is because, after fully exploring a subtree rooted at a node X_i, if the search does not find a solution, it will backtrack to the parent of X_i and search in the subtrees of the siblings of X_i.

For example, consider the map coloring problem defined by the graph below; here we consider three colors, which are represented by numbers: 1, 2, and 3. The search tree on the righthand side shows a path where we perform the following partial assignment: $B = 1, D = 2, F = 3, A = 2$. Each of the C-nodes at the end of the path violates one of the constraints. C cannot be 1 because B is 1; it cannot be 2 because D is 2; it cannot be 3 because F is 3. In chronological backtracking, we would backtrack to node $F = 3$ so that the search can try the other assignments to A (i.e., 1 and 3). This is wasteful because the culprit for C having an empty domain are the assignments to B, D, and F. Instead, the search could skip the subtrees rooted at $A = 1$ and $A = 3$ and backtrack directly to $D = 2$, so the search can try new values for F. This is what we call *backjumping.* Backjumping works similarly to a pruning scheme, since the subtrees rooted at $A = 1$ and $A = 3$ for the partial assignment $B = 1, D = 2, F = 3$ are pruned from search.

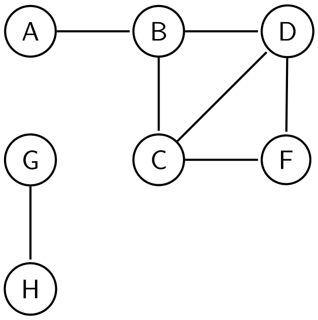


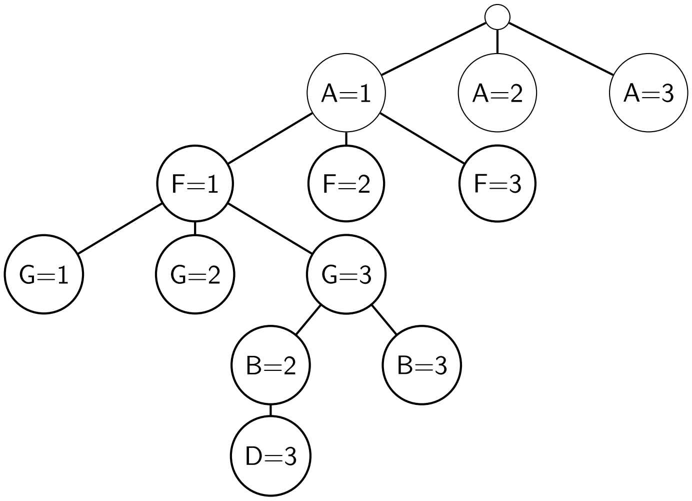


Let us reconsider the example above while using forward checking. The domain of C starts the search with the values of $\{1, 2, 3\}$. Once we assign $B = 1$, we remove 1 from C's domain; we remove 2 once we make $D = 2$; and as soon as we assign 3 to F we know that this partial assignment cannot lead to a solution because the domain of C becomes empty and the search must backtrack; we do not even attempt to assign values to A and C. In this case, backjumping would not be helpful if used with forward checking. Next we will see an example where intelligent backtracking can be helpful even when using forward checking.

5.5.1 Conflict-Directed Backjumping

Let us consider the map-coloring problem below where we assign 1 to both A and F. Before moving forward you should convince yourself that this partial assignment is a *no good*, i.e., there is no solution to the problem once we assign $A = F = 1$. For example, if $B = 2$, then D must be 3, so C has an empty domain. Forward checking is unable to detect this no good because the domains of B, C and D is $\{2, 3\}$, which is arc consistent.





In conflict-directed backjumping, we maintain one set of "conflicts" for each variable, as the search traverses a path of the tree. In the conflict set for variable V, we store all the variables that were responsible for reducing the domain of V. When backtracking from the level in the tree responsible for V, we backtrack to the nearest variable on the current path that is in the set of V. In our first example, once we discover that there is no valid assignment for C, we backtrack to F, as A would not be in the set of conflicts for C.

Let us consider the example above where the path the search expands leads to the following partial assignment: $A = 1, F = 1, G = 3, B = 2, D = 3$ (see the search tree above). The table below shows the set of conflicts for each variable as the search traverses the path that leads to the partial assignment. In the first iteration, when the root of the tree is expanded, all sets are empty. In the second iteration, when the search assigns 1 to A, forward checking removes the value of 1 from the domain of B (B is the only variable connected to A in the graph representing the problem), so A is added to the set of conflicts of variable B. This process is repeated for the first six iterations, until the search reaches the assignment for node D. Once the search sets $D = 3$, the domain of C becomes empty due to forward checking and the search backtracks. In chronological backtracking, the search would attempt to set $B = 3$, which would not lead to a solution as the no-good $A = F = 1$ remains. Then, the search would attempt $G = 1$ and $G = 2$; only after searching in these subtrees, it would attempt to resolve the no-good with a different value for F.

Iterations	Conflicts						
	A	B	C	D	F	G	H
1	$\{\}$	$\{\}$	$\{\}$	$\{\}$	$\{\}$	$\{\}$	$\{\}$
2	$\{\}$	$\{A\}$	$\{\}$	$\{\}$	$\{\}$	$\{\}$	$\{\}$
3	$\{\}$	$\{A\}$	$\{F\}$	$\{F\}$	$\{\}$	$\{\}$	$\{\}$
4	$\{\}$	$\{A\}$	$\{F\}$	$\{F\}$	$\{\}$	$\{\}$	$\{G\}$
5	$\{B\}$	$\{A\}$	$\{F, B\}$	$\{F, B\}$	$\{\}$	$\{\}$	$\{G\}$
6	$\{B\}$	$\{A, D\}$	$\{F, B, D\}$	$\{F, B\}$	$\{D\}$	$\{\}$	$\{G\}$
7	$\{B\}$	$\{A, D, F^*\}$	$\{F, B, D\}$	$\{F, B\}$	$\{D\}$	$\{\}$	$\{G\}$
8	$\{B\}$	$\{A, D, F^*\}$	$\{F, B, D\}$	$\{F, B\}$	$\{D, A^*\}$	$\{\}$	$\{G\}$

In conflict-directed backjumping, once the domain of C becomes empty by assigning $D = 3$, we check in D's set of conflicts for the closest variable on the path leading to $D = 3$, which is B. Thus, in the seventh iteration, the search backtracks to B. However, as it backtracks to B, it augments the set of conflicts of B with the variables that are in D's set of conflict and are not in B's. Naturally, B is not added to the conflict set of B as the variable cannot be in conflict with itself. The set of conflicts of B contains A, D and F, with the last being marked with a star in the table to highlight that is was added from D during backtracking. Although F did not reduce the domain of B, F is in conflict with D, which is in conflict of B. This propagation of variables in the conflict sets as the search backtracks (e.g., F is propagated from D's set of conflicts to B's set of conflicts) allows the search to keep track of this chain of conflicts and thus backtrack to variables that matter.

Searching for other values for B does not solve the problem, so conflict-directed backjumping backtracks to the closest variable in the conflict set of B, which is F. As it backtracks to F, in the eighth iteration, the variable A is inserted in the conflict set of F. The search is finally able to find a solution once it attempts a different value for F (e.g., 2). A valid solution is $A = 1, B = 2, D = 1, F = 2, C = 3, G = 1, H = 2$. Note that conflict-directed backjumping skipped other assignments to the variable G; these subtrees were pruned given the partial assignment $A = F = 1$.

To summarize, conflict-directed backjumping is performed as follows.

1. Perform backtrack search with forward checking while keeping sets of conflict variables, one set for each variable in the domain. Let us denote the set of conflict variables for X_i as $\texttt{Conf}(X_i)$.

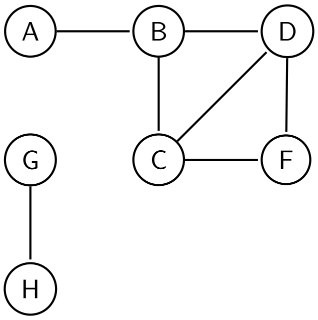
2. Once an assignment is inconsistent with the constraints (e.g., the domain of a variable becomes empty) while searching for a value for X_i, then do the following:
   (a) Backjump to the closest variable X_j in $\texttt{Conf}(X_i)$.
   (b) Propagate X_i's conflicts to X_j: $\texttt{Conf}(X_j) \leftarrow \texttt{Conf}(X_j) \cup \texttt{Conf}(X_i) \setminus X_j$.

5.6 Problem Structure

The CSP can be easy to solve depending on the structure of the graph defining the problem. In this section we will study two approaches for exploring the structure of the CSP to ease the search process.

5.6.1 Independent Subproblems

Consider the CSP below, where variables G and H are independent of the assignment of values to the other variables. This problem can be divided into two independent subproblems and the union of the solution to each subproblem is a solution to the original problem.



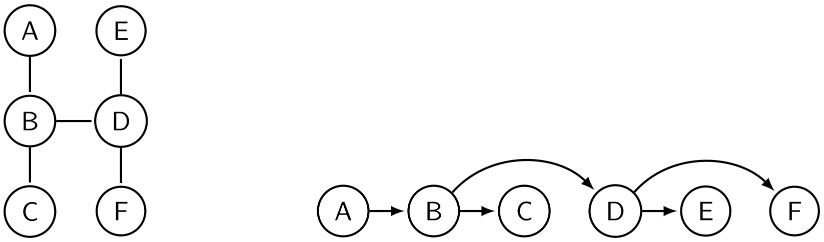
One can identify independent subproblems by obtaining the connected components of the graph representing the CSP. This can be achieved by running breadth-first search from any vertex of the graph; all vertices reached in this search must be in the same connected component. The search should then be repeated from a vertex that was not reached in the first search; all vertices reached in the second search are in another connected component. This procedure is repeated until all vertices are visited by one of the searches.

Dividing the problem into independent subproblems can substantially reduce the amount of search one needs to perform. In the worst case the search needs to search over all possible assignments to a CSP, which is d^n, for a problem with n variables and d values in the domains of the variables. If each subproblem has $c < n$ variables, then the number of assignments one needs to check is $(n/c) \cdot d^c$ (n/c subproblems with the cost of d^c each). Consider a problem where $n = 15$ and $d = 20$. If there are 3 independent subproblems with 5 variables each, then we reduce the number of assignments the search needs to verify from $20^{15} = 3.27 \times 10^{19}$ to $3 \cdot 20^5 = 9.6 \times 10^6$. The difference is so large that it could mean that we are able to solve in a fraction of a second a problem that we would not be able to solve in our lifetime.

5.6.2 Tree Structure

If the graph representing the CSP is a tree, then we can solve the problem in polynomial time. This is achieved by first performing a topological sort of the tree. Let us consider the example below, where the tree on the left represents the problem and the directed graph on the right is obtained after performing a topological sort of it.

Topological sort can be obtained by performing a depth-first search from any node of the tree. Then we add a directed edge from nodes a and b if a is the parent of b in the depth-first search. In our example, if we start the depth-first search from A, then B is its only child; C and D are the children of B and E and F are the children for D. The relationship of the parent and children for all nodes in the tree are shown in the directed graph on the righthand side. The order in which the nodes are visited will determine the ordering from left to right in the graph below: A, B, C, D, E, F.



The ordering given by the topological sort allows us to greedily select the value for each variable by going left to right, as long as the variables are arc consistent. Let us consider the following example, where the graph represents a map coloring problem and the domain of each variable starts with the values $1, 2, 3$ (Iteration 1 in the table below), where each number represents a color.

Iterations	Domains					
	A	B	C	D	E	F
1	{1, 2, 3}	{1, 2, 3}	{1, 2, 3}	{1, 2, 3}	{1, 2, 3}	{1, 2, 3}
2	{1}	{1, 2, 3}	{1, 2, 3}	{1, 2, 3}	{1, 2, 3}	{1, 2, 3}
3	{1}	{2}	{1, 2, 3}	{1, 2, 3}	{1, 2, 3}	{1, 2, 3}
4	{1}	{2}	{1}	{1, 2, 3}	{1, 2, 3}	{1, 2, 3}
5	{1}	{2}	{1}	{3}	{1, 2, 3}	{1, 2, 3}
6	{1}	{2}	{1}	{3}	{1}	{1, 2, 3}
7	{1}	{2}	{1}	{3}	{1}	{2}

We first select the value of 1 for A; this choice is arbitrary because A is the first node in our ordering. In the next iteration we select a value for B, which must be consistent with its parent (A in our example), so we choose 2. We must choose a value for C that is consistent with its parent B, so we choose 1. This process is applied to all variables in the graph, so we choose a value to all of them. Note how the tree structure allows us to decide on the value for a variable X_i while considering only the parent X_j of X_i. We know that this assignment will work for the children of X_i because the problem is arc consistent. Thus, no matter which value we choose for X_i, there must be a value that will be consistent for the children of X_i.

The table below shows an example where the values in the domains of the variables are not arc consistent. For example, if we assign the value of 3 to D, then F will not have any valid assignment available. If we apply the same greedy procedure described above, we will quickly run into trouble. In the table below we first select $A = 1$ and then $B = 2$, which is already a no-good as C is left with no valid assignment.

The solution is to first ensure that the problem is arc consistent by iterating through all arcs (X_i, X_j) from right to left and removing from the domain of X_i all values that are not consistent with the constraint of (X_i, X_j). In our example, we would remove 3 from the domain of D when processing the arc (D, F) and the

Iterations	Domains					
	A	B	C	D	E	F
1	{1, 3}	{2, 3}	{2}	{1, 2, 3}	{2, 3}	{3}
1	{1}	{2, 3}	{2}	{1, 2, 3}	{2, 3}	{3}
1	{1}	{2}	{2}	{1, 2, 3}	{2, 3}	{3}

value of 2 from the domain of B when processing the arc (B, C); these values are crossed off in the table below. The greedy selection from left to right solves the problem as shown in the table.

Iterations	Domains					
	A	B	C	D	E	F
1	{1, 3}	{~~2~~, 3}	{2}	{1, 2, ~~3~~}	{2, 3}	{3}
2	{1}	{2, 3}	{2}	{1, 2, 3}	{2, 3}	{3}
3	{1}	{2}	{2}	{1, 2, 3}	{2, 3}	{3}
4	{1}	{2}	{2}	{1}	{2, 3}	{3}
5	{1}	{2}	{2}	{1}	{2}	{3}

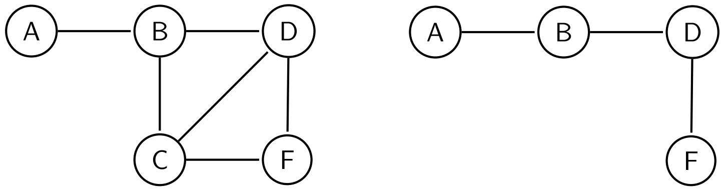
The process can be summarized as follows.

1. Perform topological sort; given that the n variables are now sorted, then:
2. for j in range$(n, 2)$:
   (a) Remove inconsistencies in $(\texttt{Parent}(X_j), X_j)$ by removing values from the domain of $\texttt{Parent}(X_j)$.
3. for j in range$(1, n)$:
   (a) $X_j \leftarrow v$, where v is consistent with $\texttt{Parent}(X_j)$.

Each operation in both for-loops above performs d^2 operations. This is because we need to verify all pairs of values for two variables in both cases. Since we have n variables (the for-loops above iterate over all variables), then the complexity of this procedure is $O(n \cdot d^2)$, which is vastly superior to the $O(d^n)$ complexity.

Tree Decomposition

The time complexity for solving tree-structured CSPs is so much better than the time complexity for solving CSPs in general that it is worth attempting to use such a solver even in cases where the CSP is not represented by a tree. Let us consider the example below, where the graph on the lefthand side is not a tree, but it becomes one if we remove vertex C (see graph on the right). We can use the tree solver to efficiently solve the problem on the right. How can we use the solution to the tree to obtain a solution to the original problem?



We need to consider all possible assignments for the variables removed from the graph—these variables are known as the cutset. Assuming a map coloring problem with the domain $\{1, 2, 3\}$, in our example we have to consider all three assignments for C: 1, 2, and 3 and solve the tree-structured CSP for all three possibilities. For $C = 1$, we make all variables connected to C arc-consistent with the assignment, i.e., the domains of B, D, and F become $\{2, 3\}$. That way, the tree solver will find a solution that is consistent with the assignment we used for C. Since we are attempting all possible assignments for C, if there is a solution, we will find it.

What if the cutset has two variables? We would have to solve one tree-structure problem for each combination of values for the two variables. If the domains of the variables have d values, then in the worst case (when the search processes all assignments for the variables in the cutset) we would have to solve d^2 problems. In general, if the cutset has c variables, we need to solve d^c tree-structured problems in the worst case. In practice we would like the cutset to be as small as possible, as the overall algorithm is exponential in the size of this set. Finding the minimum cutset is NP-Hard, which is known as the Feedback Vertex Set problem. In practice one could employ approximation algorithms or even a naïve approach where one attempts to remove each one of the variables from the graph and verify if one obtains a tree by removing a single variable; if no tree is obtained, then one could try to remove all possible pairs of variables. This naïve approach can work well in graphs that are "almost trees" (i.e., the cutset is small), as in our example above.

Chapter 6

Classical Planning

6.1 Classical Planning

We studied how to write computer programs for solving pathfinding problems such as grid-based pathfinding on video game maps and sliding-tile puzzles. How about logistic problems? The approach one can take is to write the required code to define the search problem: definition of a state, the transition function (given a state and an action the function returns the set of generated child), and goal checks. This is to be able to use uniformed algorithms such as Dijkstra's algorithm. If one is interested in using informed algorithms such as A*, then we need to define a heuristic function. This is a process that needs to be repeated for each new problem we are interested in solving.

We will consider a more general solution, one that will allow us to write a single program for solving any classical search problem. By classical search problems we mean problems with the following features.

- Pathfinding – the solution to the problem is a sequence of actions that transforms the initial state into a goal state; sliding-tile puzzles and grid-based pathfinding are examples of pathfinding problems.
- Deterministic actions – the transition function is deterministic, i.e., given a state and an action, the transition function always returns the same state.
- Discrete and finite spaces – the state and action spaces are discrete and finite.
- Fully observable – the agent has access to all information at a given state.
- Static – the environment does not change while the agent is executing actions.
- Single agent – there is a single agent in the problem, so the agent does not have to reason how other agents might reason about the problem.

The field that considers general solvers for this type of problem is known as classical planning. Although classical planning is restricted to classical search problems, many interesting and important problems such as logistic problems can be solved with classical planners. Moreover, many non-classical planning problems can be compiled into classical planning problems.

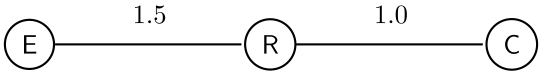
In classical planning we follow an approach similar to CSPs in the sense that the solvers are general. As long as we are able to specify the problem as a CSP, we are able to use a general solver to tackle the problem. It will be similar here, we will use a language—Planning Domain Definition Language (PDDL)—to specify classical planning problems. Planners parse the PDDL file and attempt to come up with a solution to the problem. Current state-of-the-art planners use heuristic search. That is, they automatically derive a heuristic function and use heuristic search algorithms to solve the problem.

We will focus on a family of heuristic functions based on a relaxation of the original problem known as delete relaxation. In order to study this family of heuristic functions and to formally define classical planning problems, we will use the STRIPS formalism, which we explain with an example below.

Many of the examples from these notes are based on the examples from the slides of the AI Planning course by Joerg Hoffman, Álvaro Torralba, and Cosmina Croitoru.[1]

6.2 STRIPS Formalism

Let us consider a simplified version of the traveling salesman problem (TSP) for a simplified map of Alberta. In this problem, the salesman is in Red Deer (R) and it needs to visit Edmonton (E) and Calgary (C). Consider the roadmap below, where R is connected to E and C, with the cost of traveling between the cities is shown as the edge value.



In the STRIPS formalism we define a set of propositions, which are the things that can be true in different states of the problem. For our simplified TSP we have the following propositions.

$$\{\texttt{at}(x), \texttt{visited}(x), \texttt{connected}(x, y) | x, y \in \{E, R, C\}\}$$

A state is formed by a collection of propositions; if a proposition is not in a state, then it means the proposition is not true at that state. The propositions `connected(E, R)`, `connected(R, E)`, `connected(R, C)`, and `connected(C, R)` are true in all states because they define the structure of the underlying map. The following state defines a scenario where the salesman has already visited R and is currently in R.

$$\{\texttt{at(R)}, \texttt{visited(R)}, \texttt{connected(E, R)}, \texttt{connected(R, E)}, \texttt{connected(R, C)}, \texttt{connected(C, R)}\}$$

The state above is also the initial state of our example, while the goal state is defined as follows.

$$\{\texttt{visited}(x) | x \in \{E, R, C\}\}$$

This is a simplified TSP because we do not require the salesman to be back at R; we need to add `at(R)` to the goal propositions to transform it into a TSP. The actions available at a state are given by the following scheme.

$$\begin{aligned}\texttt{drive}(x, y) :& (\\ \text{Pre}_a :& \{\texttt{at(x)}, \texttt{connected(x, y)}\}\\ \text{Add}_a :& \{\texttt{at(y)}, \texttt{visited(y)}\}\\ \text{Del}_a :& \{\texttt{at(x)}\})\end{aligned}$$

Each action is defined by a triple Pre_a, Add_a, and Del_a, defining the preconditions, addition, deletion sets of an action. The precondition set of an action a defines the propositions that must be true at a state s so

[1] `http://fai.cs.uni-saarland.de/teaching/winter18-19/planning.html`.

that a can be applied in s. For example, it is not possible to apply action `drive(E, C)` in the initial state of the problem because the action requires the propositions `at(E)` and `connected(E, C)`. Action `drive(R, E)` can be applied in the initial state since it contains both `at(R)` and `connected(R, E)`.

The addition set defines the propositions that are added to a state s once action a is applied in s. For example, once we apply action `drive(R, E)` to the initial state, we add the propositions `at(E)` and `visited(E)` to the state. The deletion set determines the propositions that are removed from the state. In our example, once we apply action `drive(R, E)` in the initial state, the agent is no longer in Red Deer, so we remove `at(R)` from the state. Each action scheme can also define a cost function; whenever we omit the cost function we will be assuming that all actions cost 1. In our example, we have the following.

$$c(\texttt{drive}(x, y)) = \begin{cases} 1 & \text{if } x, y \in \{R, C\} \\ 1.5 & \text{if } x, y \in \{R, E\} \end{cases}$$

6.2.1 Formal Definition of a STRIPS Planning Problem

We can now formally define a STRIPS planning task as a tuple (P, A, c, I, G), where

- P is a finite set of propositions.
- A is a finite set of actions, where each a in A is a triple $(\text{pre}_a, \text{add}_a, \text{del}_a)$; let s be a state:
  - pre_a are the propositions that are required in s so that a is applicable at s.
  - add_a are the propositions that are added to s once a is applied in s.
  - del_a are the propositions that are removed from s once a is applied in s.
- $c : A \to \mathbb{R}_0^+$ is a cost function mapping actions to a non-negative real value.
- $I \subseteq P$ is the initial state.
- $G \subseteq P$ is the goal.

We say that a planning problem is solvable if there exists a path $p = (a_1, a_2, \cdots, a_n)$ that transforms I into a state s such that $G \subseteq s$. The sequence of actions is called a path or a plan. The cost of a path is the sum of the cost of the actions in the path. The path p is said be optimal if there is no other path that solves the problem and whose cost is lower than the cost of p. "Optimal planning" refers to the problem of finding optimal solutions to planning problems; "satisficing planning" is the setting in which we are after any solution, optimal or otherwise.

6.2.2 STRIPS State Space

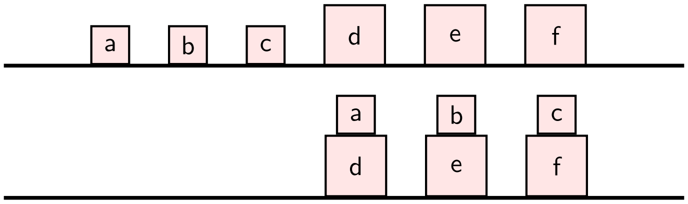
A STRIPS planning problem defines a state space. The set of possible states are given by all possible combinations of propositions (all possible subsets of the set P). There is an outgoing edge from state s to s' if there is an action a that, when applied at s, it transforms s into s'. Note that the state space is defined implicitly, i.e., we can build the space as needed. A STRIPS planning problem gives us the initial state I of the problem and the actions available at I. The actions available at I allow us to reach the children of I; with the actions available at the children of I we are able to reach the grandchildren of I and so on.

How many states does the state space of the simplified TSP have? The only propositions that can be added and removed from a state are `at(x)` and `visited(x)` for x in $\{E, R, C\}$ (the `connected` propositions are fixed and thus do not contribute to the increase in the size of the space). Since each proposition can either be present or not, the state space has 2^6 states. However, many of these states cannot be reached from the initial state of the problem. For example, $\{$`at(E)`, `visited(R)`$\}$ cannot be reached because `visited(E)` is added to the state whenever `at(E)` is added. State $\{$`at(E)`, `visited(E)`, `visited(R)`$\}$ is reachable.

Once a state space is defined, we can do search to find a path that transforms I into a state that satisfies the goal conditions. We can immediately apply uninformed search algorithms such as Dijkstra's algorithm. We will see later how to derive heuristic functions given the STRIPS formalism for classical planning, so we can also use informed algorithms such as A*. We note that there exist other formalisms in addition to STRIPS, such as such the finite-domain representation, which might be more suitable for describing some heuristic functions and search algorithms. We focus on the STRIPS formalism.

6.3 Planning Domain Definition Language (PDDL)

The interface between planning systems and users is through the Planning Domain Definition Language (PDDL). The user defines the classical planning problem in PDDL. The PDDL file is provided as input to the planning system, which attempts to solve the problem. Let us consider the Blocks World (BW) example shown in the figure below. In BW one needs to find a sequence of actions to transform an initial configuration of a set of blocks, which are on a table (upper part of the figure) into a goal configuration (bottom part of the figure). In this example, we consider two types of blocks: small (a, b, c) and large (d, e, f). In this domain, all one can do is to move a small block from the table on top of a large block. We can solve the problem by moving a onto d, b onto e, and c onto f.



In PDDL, we need to write two files to define a planning problem. The first is the domain file, where we specify the objects present in the problem, the predicates (e.g., block a is on the table), and the actions one is able to apply in a given state of the problem. The second file defines a specific problem instance. For example, the image above defines a specific instance of BW by defining the initial and goals states. In this example, the initial state has all blocks (small and large) on the table. A different second file could define a different problem by defining a different initial and/or goal states.

6.3.1 Domain File

For this particular version of BW we have the following domain file.

```
(define (domain BlocksWorld)
   (:requirements: typing)
   (:types block - object
         small - block)
   (:predicates (on ?x - small ?y - block)
                  (ontable ?x - block)
                  (clear ?x - block))
   (:action movefromtable
         :parameters (?x - small ?y - block)
         :precondition (and (not (= ?x ? y))
                        (clear ?x)
                        (ontable ?x)
                        (clear ?y))
         :effect (and (not (ontable ?x))
                        (not (clear ?y))
                        (on ?x ?y))))
```

In the file above we define the domain with the name `BlocksWorld`, which is used later in the problem file. The `requirements` keyword allows us to specify which features of PDDL our file uses. For example, among other features, `typing` allows us to define types to the objects in the domain. In `BlocksWorld` we have objects of two types: `small` and `block`. Note, however, that objects of type `small` are also of type `block`.

The file also defines the predicates, which are the things that can be true at a given state of the problem. This file specifies that a `small` block can be `on` a `block` (the `?x` and `?y` parameters can be replaced by objects of type `small` and `block` respectively); a `block` can be on the table (`ontable`); a `block` can be `clear`. Note that a `small` object can be `on` another `small` object, a `small` object can be `ontable` and be `clear`. This is because objects that are of the `small` type are also of the `block` type.

The domain file also defines the action schema. In this case we have one action, `movefromtable`. The action takes two inputs as parameters: `x` and `y` of type `small` and `block`, respectively. The `precondition` defines what needs to be true at a given state s so that the action can be applied at s; this is equivalent to the pre_a in the STRIPS formalism. The `effect` defines what is added and removed to a state s once the action is applied in s. The `effect` contains the add_a and del_a in the STRIPS formalism.

The preconditions to apply the `movefromtable` action are the following (in the order given in the definition above): we cannot move a block onto itself, the block that is being moved (parameter `x`) has to be `clear` and `ontable`, while the block where we are moving `x` to (parameter `y`) has to be `clear`. As effects of the action, the block `x` is no longer `ontable`, `y` is no longer `clear` and `x` is `on` `y`.

6.3.2 Problem File

We present below the problem file for the instance of the `BlocksWorld` we presented in the image above.

```
(define (problem p1)
   (:domain BlocksWorld)
   (:objects a b c - small
             d e f - block)
   (:init (clear a) (clear b) (clear c) (clear d) (clear e) (clear f)
          (ontable a) (ontable b) (ontable c) (ontable d) (ontable e) (ontable f))
   (:goal (and (on a d) (on b e) (on c f)))
)
```

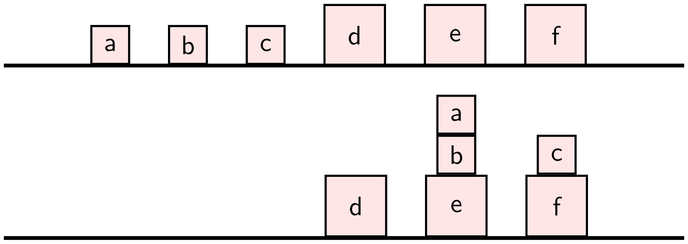
The problem file specifies which domain we use, which is `BlocksWorld`, and it defines that the set of objects: `a`, `b`, `c` are of type `small` and `d`, `e`, `f` are of type `block`. The `init` keyword shows the predicates that are true in the initial state and the `goal` shows the predicates that need to be true in a state s so s is goal state.

The table below shows a sequence of action that solves the `BlocksWorld` instance (column on the right) and how the state changes after each action is applied to the current state (column on the left). For example, after action `movefromtable(a, d)` is applied to the initial state, the predicate `(on a d)` is added to the state and predicates `(clear d)` and `(ontable a)` are removed from the state (see second line of the table).

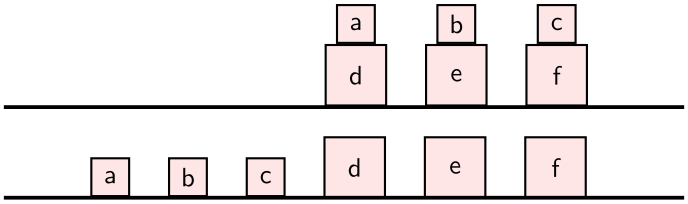
Current State	**Actions**
(clear a) (clear b) (clear c) (clear d) (clear e) (clear f) (ontable a) (ontable b) (ontable c) (ontable d) (ontable e) (ontable f)	movefromtable(a, d)
(on a d) (clear a) (clear b) (clear c) (clear e) (clear f) (ontable b) (ontable c) (ontable d) (ontable e) (ontable f)	movefromtable(b, e)
(on b e) (on a d) (clear a) (clear b) (clear c) (clear f) (ontable c) (ontable d) (ontable e) (ontable f)	movefromtable(c, f)
(on c f) (on b e) (on a d) (clear a) (clear b) (clear c) (ontable d) (ontable e) (ontable f)	-

6.3.3 More Blocks World Examples

Consider the following instance of the same `BlocksWorld` domain, where the configuration at the top shows the initial state and the configuration at the bottom the goal state. Are we able to solve this problem with the actions we have available in this domain? The answer is 'yes'. First, we move `b` from the table onto `e` and then `a` onto `b`. The action `movefromtable` takes an object of type `block` as the second parameter `y`. This is fine in the actions listed above because both `b` and `a` are of type `small`, which is also of type `block`.



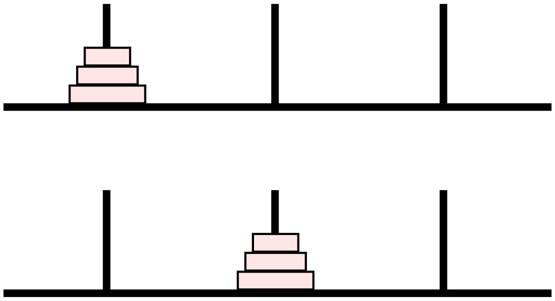
Consider now the following problem instance where again the top configuration shows the initial state and the bottom one shows the goal configuration. Are we able to solve it with the actions we have available? The answer is 'no'. This is because the precondition for `movefromtable` requires the block that is being moved to be on the table. The problem becomes solvable if we add the following action scheme to the domain.



```
(:action move-from-block-to-table
      :parameters (?x - small ?y - block)
      :precondition (and (clear ?x)
                     (on ?x ?y))
      :effect (and (ontable ?x)
                     (clear ?y)
                     (not (on ?x ?y))))
```

6.3.4 Example - Towers of Hanoi

In the problem of Towers of Hanoi, a number of discs of different sizes are stacked in a peg, as shown in the figure below. The goal is to transfer all discs to another peg, as shown in the image at the bottom. One can move a disc from a peg to another as long as the disc is at the top of its pile and it goes on top of a larger disc. The following PDDL files model the problem below where the figure at the top represents the initial state and the figure at the bottom the goal state.



```
(define (domain hanoi)
   (:requirements :strips)
   (:predicates (clear ?x)
             (on ?x ?y)
             (smaller ?x ?y))

   (:action move
      :parameters (?disc ?from ?to)
      :precondition (and (smaller ?to ?disc)
                   (on ?disc ?from)
                   (clear ?disc)
                   (clear ?to))
      :effect  (and (clear ?from)
              (on ?disc ?to)
              (not (on ?disc ?from))
              (not (clear ?to)))
))
```

In the domain file we have `requirements :strips`, which allows us to use basic add and delete effects from the STRIPS formalization. In our BW example, the requirement of types also includes the basic STRIPS operations. The file below specifies the instance shown in the image above. The smallest disc is named `d1`, while the second smallest is named `d2`, and the largest `d3`. The leftmost peg is `peg1`, the one in the middle `peg2`, and the one on the righthand side is `peg3`.

```
(define (problem hanoi-3)
  (:domain hanoi)
  (:objects peg1 peg2 peg3 d1 d2 d3 )
  (:init (smaller d1 peg1) (smaller d2 peg1) (smaller d3 peg1) (smaller d1 peg2)
         (smaller d2 peg2) (smaller d3 peg2) (smaller d1 peg3) (smaller d2 peg3)
         (smaller d3 peg3) (smaller d1 d2) (smaller d1 d3) (smaller d2 d3)
         (clear peg2) (clear peg3) (clear d1) (on d3 peg1) (on d2 d3) (on d1 d2))
  (:goal (and (on d3 peg2) (on d2 d3) (on d1 d2))))
```

6.4 Delete Relaxation Heuristics

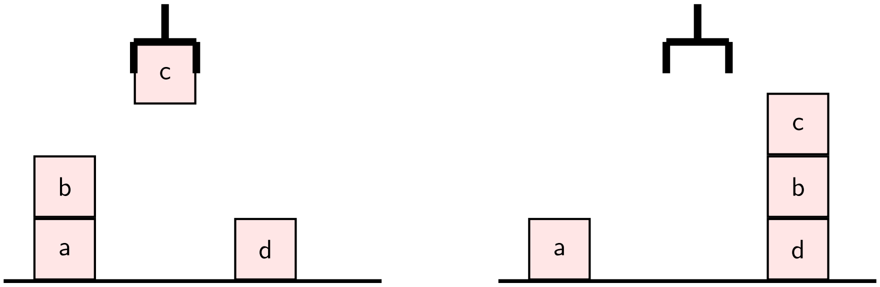
The STRIPS formalism allows us to define a search space that can be used with uninformed search algorithms such as Dijkstra's algorithm to find a solution path to a classical planning problem. In order to use more powerful algorithms such as A*, we need to be able to automatically derive heuristic functions from the classical planning problem. We will study a family of heuristic functions that solve a relaxed version of the original problem, similar to how we did with pattern databases. We will relax the planning problem by removing the delete effects of the actions—once something becomes true, it remains true.

Given the set of actions $a = (\text{pre}_a, \text{add}_a, \text{del}_a)$, with delete relaxation we define a new planning problem where all actions are of the form $a^+ = (\text{pre}_a, \text{add}_a, \emptyset)$. During an A* search, we compute the h-value of a state s, denoted $h^+(s)$, by finding an optimal solution for s in the relaxed space, where we use a^+ instead of a; the optimal solution cost of the relaxed problem is the h^+-value for s. We can solve the relaxed planning problem s represents with an uninformed search algorithm such as Dijkstra's algorithm.

6.4.1 Blocks World Example

Let us consider the Blocks World (BW) example shown below. In addition to the blocks and the table, we have a mechanical hand that is used to pick up the blocks and place them on top of another block or on the table; the hand can hold a single block at a time. The set of actions available in this domain are as follows.

- `Putdown`: places a block from the hand on the table
- `Unstack`: picks a block from the top of the stack.
- `Stack`: places a block from the hand at the top of a stack.
- `Pickup`: picks a block from the table.



The initial state (figure on the left) can be described with the following propositions:

`(hand c)`, `(ontable a)`, `(ontable d)`, `(on b a)`, `(clear b)`, `(clear d)`

While the goal conditions are defined as follows:

`(ontable a)`, `(ontable d)`, `(on b d)`, `(on c b)`, `(clear a)`, `(clear c)`

The table below shows the optimal plan transforming the initial state into a state satisfying the goal conditions. The first row shows the initial state and the action applied at it; the last row shows a goal state.

Current State	**Actions**
`(hand c)`, `(ontable a)`, `(ontable d)`, `(on b a)`, `(clear b)`, `(clear d)`	`putdown`(c)
`(clear hand)`, `(ontable a)`, `(ontable d)`, `(ontable c)`, `(on b a)`, `(clear b)`, `(clear d)`, `(clear c)`	`unstack`(b)
`(hand b)`, `(ontable a)`, `(ontable d)`, `(ontable c)`, `(clear a)`, `(clear d)`, `(clear c)`	`stack`(b, d)
`(clear hand)`, `(on b d)`, `(ontable a)`, `(ontable d)`, `(ontable c)`, `(clear a)`, `(clear b)`, `(clear c)`	`pickup`(c)
`(on b d)`, `(ontable a)`, `(ontable d)`, `(clear a)`, `(clear b)`, `(hand c)`	`stack`(c, b)
`(clear hand)`, `(on b d)`, `(on c b)`, `(ontable a)`, `(ontable d)`, `(clear a)`, `(clear c)`	-

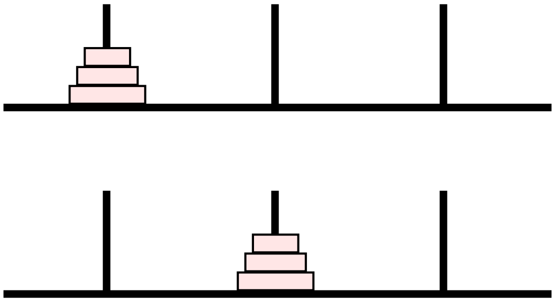
What about an optimal plan in the relaxed space where the delete effects are removed from the actions? The sequence of actions in the table below shows the optimal plan in the relaxed space. Note how removing the delete effects creates strange (yet valid in the relaxed space) scenarios. Once we stack `c` on `b`, the former remains in the hand and the latter remains clear, and the propositions `(on c b)` and `(clear hand)` are added to the state. This allows us to unstack `b` (both `b` and the hand are clear!) and stack it onto `b`, thus solving the problem. The h^+-value for the initial state of our example is 3, while its h^*-value is 5.

6.4.2 Towers of Hanoi Example

What is the h^+-value for the following initial state of the Towers of Hanoi (initial state is shown at the top and goal state at the bottom)? The optimal plan in the relaxed space involves "clearing" the bottom disc,

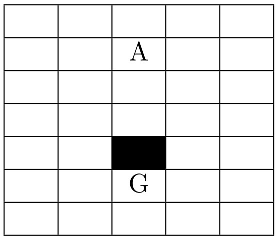
Current State	Actions
(hand c), (ontable a), (ontable d), (on b a), (clear b), (clear d)	stack(c, b)
(hand c), (ontable a), (ontable d), (on b a), (clear b), (clear d), (on c b), (clear b), (clear hand)	unstack(b)
(hand c), (ontable a), (ontable d), (on b a), (clear b), (clear d), (on c b), (clear b), (clear hand), (hand c)	stack(b, d)
(hand c), (ontable a), (ontable d), (on b a), (clear b), (clear d), (on c b), (clear b), (clear hand), (hand c)	-

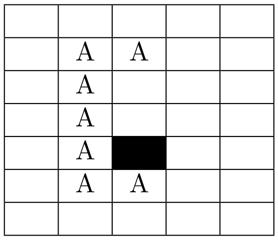
so that we can move it to its goal location. We need to clear one disc at a time. In the initial state only the smallest disc is clear, so we move it to any other peg (e.g., the peg in the middle). Note that the peg in middle remains clear and the small disc also remains on top of the stack in the first peg as we do not have delete effects. Then, we move the middle disc to another peg thus clearing the bottom disc. Once the bottom disc is clear, we move it to the goal peg. This sequence of actions achieve a goal state as the propositions indicating that the middle disc is on the largest disc and that the smallest disc is on the middle disc are already true in the initial state. The only proposition that need to be added is the one stating that the largest disc is on the middle peg, which is achieved by clearing the discs as explained above. Thus, the h^+-value of this initial state is 3. More generally, if the state had n discs the h^+-value would be n.



6.4.3 Grid-Based Pathfinding Example

The grid below (lefthand side) shows a pathfinding problem where the agent starts in the cell marked with an “A” and the goal is marked with a “G”; the agent can only move in one of the four cardinal directions. The dark cell represents a wall that the agent cannot traverse. What is the h^+-value for this state I? If we remove the delete effects, the agent will simultaneously be in two cells once they move from one cell to the next. The removal of the delete effects does not affect the solution cost in the relaxed space: the solution cost in both the original and relaxed spaces is the same, 6. The grid on the lefthand side shows the relaxed goal state. The agent is simultaneously in all cells along the path. Since those cells are never revisited in search, it does not matter whether the agent is simultaneously in multiple places or not, $h^+(I) = h^*(I)$.



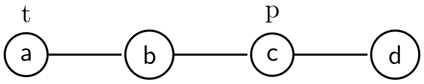


6.4.4 Properties of h^+

The heuristic function h^+ is consistent and admissible. Recall that A* using such a heuristic function is guaranteed to find optimal solutions and WA* is guaranteed to find bounded-suboptimal solutions. On a negative note, computing the value of h^+ requires one to solve the relaxed planning task and this is NP-Complete in general. Thus, the use of h^+ in practice requires one to solve an NP-Complete problem for every state expanded in search. Instead of using such a computationally expensive heuristic function, we will study three heuristic functions that approximate the h^+ value: h^{add}, h^{max}, and h^{FF}.

6.5 Approximations of h^+: h^{max} and h^{add}

The approximation used in both h^{max} and h^{add} of h^+ is performed by assuming that the subgoals in the goal conditions can be solved independently. In our BW example, we assume that the each proposition in the goal, $\{$`(ontable a)`, `(ontable d)`, `(on b d)`, `(on c b)`, `(clear a)`, `(clear c)`$\}$ can be solved independently of each other. h^{max} approximates h^+ by returning the cost of the most expensive subgoal; h^{add} returns the sum of the cost of the solutions of each subgoal. Let us consider the following logistic problem as an example.



In this logistic problem, we have four cities: a, b, c, d, a truck that starts in city a (denoted by t), and a package that starts in city c (denoted by p). The actions are `drive(x, y)`, `load(x, y)`, and `unload(x, y)`. The preconditions and effects are the obvious ones: you can only load a package into a truck, if both the package and the truck are in the same city; you can only unload a package from the truck if the package is in the truck. Once you load the truck, the package is no longer in the city, but in the truck; once you unload a package from the truck the package is no longer in the truck, but in the city where it was unloaded. All actions cost one. The initial state is $I = \{$`(at t, a)`$, (at, p, c)\}$ and the goal conditions are $G = \{$`(at t, a)`, `(at p, d)`$\}$. The optimal solution cost for this problem h^* is 8: `drive(a, b)`, `drive(b, c)`, `load(c, p)`, `drive(c, d)`, `unload(p, d)`, `drive(d, c)`, `drive(c, b)`, `drive(b, a)`. The h^+-value for the initial state is 5: $\{$`drive(a, b)`, `drive(b, c)`, `load(c, p)`, `drive(c, d)`, `unload(p, d)`$\}$. The solution in the relaxed space does not require the truck to drive back to a as `(at t, a)` is true in the initial state.

We assume that h^{max} approximates h^+ by assuming that the subgoals can be solved independently and it returns the most expensive subgoal. Thus, we can write h^{max} as follows: $h^{\text{max}}\,(I, G) = \max\big(h^{\text{max}}(I,$ `(at t, a)`$)$,

$h^{\max}(I, \texttt{(at p d)})\big)$. That is, the computation of $h^{\max}$ is reduced to the computation of two other $h^{\max}$ calls.

$$
\begin{aligned}
h^{\max}(I, G) &= \max\big(h^{\max}(I, \texttt{(at t a)}), h^{\max}(I, \texttt{(at p d)})\big) && (6.1)\\
&= h^{\max}(I, \texttt{(at p d)}) && (6.2)\\
&= 1 + \max\big(h^{\max}(I, \texttt{(at t d)}), h^{\max}(I, \texttt{(in p t)})\big) && (6.3)\\
&= 1 + \max(3, 3) = 4 && (6.4)
\end{aligned}
$$

Step 6.2 in the derivation above is because `(at t a)` is already present in the initial state, so $h^{\max}(I, \texttt{(at t a)})$ is zero. Step 6.3 is obtained through the action that achieves the subgoal `(at p d)`, which is `unload(p, d)`. The +1 in step 6.3 is due to the cost of using action `unload(p, d)` and the max term accounts for the preconditions of action `unload(p, d)`: `(at t d)` and `(in p t)`. Similarly to step 6.1, in step 6.3, $h^{\max}$ assumes that the problem of computing $h^{\max}(I, \{\texttt{(at t d)}, \texttt{(in p t)}\})$ can be simplified by computing $h^{\max}$ for the subgoals `(at t d)` and `(in p t)` independently. Step 6.4 is obtained by repeating the same recursive procedure to obtain $h^{\max}(I, \texttt{(at t d)}) = 3$ and $h^{\max}(I, \texttt{(in p t)}) = 3$. We detail the computation of $h^{\max}(I, \texttt{(at t d)})$ below and we leave the computation of $h^{\max}(I, \texttt{(at t d)})$ as an exercise.

$$
\begin{aligned}
h^{\max}(I, \texttt{(at t d)}) &= 1 + h^{\max}(I, \texttt{(at t c)}) && \text{(use action } \texttt{drive(c, d)}\text{)}\\
&= 1 + 1 + h^{\max}(I, \texttt{(at t b)}) && \text{(use action } \texttt{drive(b, c)}\text{)}\\
&= 1 + 1 + 1 + h^{\max}(I, \texttt{(at t a)}) && \text{(use action } \texttt{drive(a, b)}\text{)}\\
&= 3 && \text{(because } h^{\max}(I, \texttt{(at t a)}) = 0\text{)}
\end{aligned}
$$

We follow a similar procedure for h^{add}. However, instead of taking the max over the h-values, we add the h-values, as shown in the derivation below.

$$
\begin{aligned}
h^{\text{add}}(I, G) &= h^{\text{add}}(I, \texttt{(at t a)}) + h^{\text{add}}(I, \texttt{(at p d)})\\
&= h^{\text{add}}(I, \texttt{(at p d)})\\
&= 1 + h^{\text{add}}(I, \texttt{(at t d)}) + h^{\text{add}}(I, \texttt{(in p t)})\\
&= 1 + 3 + 3 = 7
\end{aligned}
$$

The formulae below generalize the derivation we wrote above for $h^{\max}$ and h^{add}.

$$
h^{\max}(s, g) = \begin{cases}
\max_{g' \in g} h^{\max}(s, g') & \text{if } |g| > 1\\
\min_{a \in A, g \in \text{add}_a} c(a) + h^{\max}(s, \text{pre}_a) & \text{if } |g| = 1\\
0 & \text{if } g \subseteq s
\end{cases}
$$

$$
h^{\text{add}}(s, g) = \begin{cases}
\sum_{g' \in g} h^{\text{add}}(s, g') & \text{if } |g| > 1\\
\min_{a \in A, g \in \text{add}_a} c(a) + h^{\text{add}}(s, \text{pre}_a) & \text{if } |g| = 1\\
0 & \text{if } g \subseteq s
\end{cases}
$$

The first case of the equations above is the same one used to compute $h^{\max}(I, G)$ and $h^{\text{add}}(I, G)$, since $|G| > 1$ as $G = \{\texttt{(at t a), (at p, d)}\}$. When the goal is of size one, then we need to solve an optimization problem. For example, for $h^{\max}$ we have the following problem: $min_{a \in A, g \in \text{add}_a} c(a) + h^{\max}(s, \text{pre}_a)$. Here, we are after the action a that adds g to the state as one a's addition effects and that minimizes the sum $c(a) + h^{\max}(s, \text{pre}_a)$. Solving this optimization problem was trivial in our logistic example as we only had a single action that achieves the subgoals. We can use dynamic programming to solve this problem in general.

6.5.1 Computing $h^{\max}$ and h^{add} with Dynamic Programming

Dynamic programming can be used to solve optimization problems that (i) can be decomposed into subproblems and (ii) the optimal solution to the problem can be obtained by using the optimal solution of the subproblems. This is exactly the kind of optimization problem we are solving while computing $h^{\max}$ and h^{add}. For example, if we knew the $h^{\max}$ value for all possible preconditions needed to achieve the subgoal `(at p d)`, then it becomes easy to compute the value for $h^{\max}(I, \texttt{(at p d)})$. This is because, all we need to do is to choose the action that minimizes the $c(a) + h^{\max}(s, \text{pre}_a)$.

Dynamic programming iteratively computes the optimal solution of all subproblems and store their solutions in a table. For computing $h^{\max}$ (or h^{add}), we maintain a table with the $h^{\max}$-value (or h^{add}-value) for all possible propositions in the planning problem. Initially we only know the h-value of the propositions present in the initial state (they are zero because they are present in the initial state). Then, we use the equations $h^{\max}(s, g)$ and $h^{\text{add}}(s, g)$ to fill up the table. Once the values in the table do not change, we know that we have converged to the optimal solution to all subproblems (i.e., all propositions). Then, the h-value for the state used to build the table can be easily extracted from the table, as we explain in the next sections.

Computation of $h^{\max}$

The table below shows the values the algorithm computes for $h^{\max}$ for the logistic example.

Iteration	`ta`	`tb`	`tc`	`td`	`pt`	`pa`	`pb`	`pc`	`pd`
0	0	∞	∞	∞	∞	∞	∞	0	∞
1	0	1	∞	∞	∞	∞	∞	0	∞
2	0	1	2	∞	∞	∞	∞	0	∞
3	0	1	2	3	3	∞	∞	0	∞
4	0	1	2	3	3	4	4	0	4
5	0	1	2	3	3	4	4	0	4

In the table above we write `ta` instead of `(at t a)`, `pt` instead of `(in p t)`, and finally, `pa` instead of `(at p a)`. In iteration 0, we initialize a table with all possible propositions in the planning problem. The propositions present in the initial state are initialized with the value of 0, while all the other propositions are initialized with the value of ∞ (i.e., we initially assume they cannot be reached). The values we initialize with 0 are those matching with the third case in the formulae for $h^{\max}(s, g)$ and $h^{\text{add}}(s, g)$, where $g \subseteq s$. Let $T[i][p]$ be the value of the proposition p in the i-th iteration of the algorithm.

In every iteration of the algorithm, we attempt to find a value for a given proposition p that is smaller than p's value in the previous iteration. For example, the value of $T[1][\texttt{tb}]$ receives the smallest value between the following two options: $T[0][\texttt{tb}] = \infty$ (the value of `tb` in the previous iteration) and $\min_{a \in A, \texttt{tb} \in \text{add}_a} c(a) + T[0][\text{pre}_a] = 1 + 0 = 1$ (the cost of driving from a to b added to the cost of achieving `ta`, the precondition to drive from a to b). That is how the value of 1 is assigned to `tb` in iteration 1. Note that the computation of $T[1][\texttt{tb}]$ is performed using the second case in the formula for $h^{\max}(s, g)$, because $|g| = 1$.

One might wonder why `tc` is still ∞ in iteration 1. For `tc` we assign the minimum between its previous value, which is ∞, and the value of $\min_{a \in A, \texttt{tc} \in \text{add}_a} c(a) + T[0][\text{pre}_a]$. The only action that adds `tc` to a state is to drive from b to c and its precondition is `tb`. Since $T[0][\texttt{tb}] = \infty$, we have that $T[1][\texttt{tc}] = \infty$.

Let us consider $T[4][\texttt{pd}]$. The previous value for the proposition is $T[3][\texttt{pd}] = \infty$. The action that adds `pd` to a state is to unload the truck at d, which has two preconditions: `td` and `pt`. We use the second case of the formula $h^{\max}(s, g)$, where we add 1 (cost of unloading the truck) with the $h^{\max}$-value of the action's preconditions. The $h^{\max}$-value of the preconditions matches the first case of the $h^{\max}(s, g)$ formula, which translates into $\max(T[3][\texttt{td}], T[3][\texttt{pt}]) = \max(3, 3) = 3$. Thus, we assign 4 to $T[4][\texttt{pd}]$.

The algorithm stops if the i-th row is equal to the $i-1$-th row. This means that the algorithm has converged on the optimal value for all propositions in the problem. At this point, the $h^{\max}$-value for the state can be easily extracted from the table. Since $G = \{\texttt{ta}, \texttt{pd}\}$, we have $h^{\max}(I, G) = \max(T[5][\texttt{ta}], T[5][\texttt{pd}]) = 4$.

Computation of h^{add}

The table below shows the values the algorithm computes for h^{add} for the logistic example.

Iteration	ta	tb	tc	td	pt	pa	pb	pc	pd
0	0	∞	∞	∞	∞	∞	∞	0	∞
1	0	1	∞	∞	∞	∞	∞	0	∞
2	0	1	2	∞	∞	∞	∞	0	∞
3	0	1	2	3	3	∞	∞	0	∞
4	0	1	2	3	3	4	5	0	7
5	0	1	2	3	3	4	5	0	7

The dynamic programming procedure for filling the h^{add} table is very similar to the one we described for $h^{\max}$. The difference is that we follow the equations of $h^{\text{add}}(s, g)$ instead of $h^{\max}(s, g)$. For example, when computing $T[4][\texttt{pd}]$, we add the cost of unloading 1 with the h^{add}-value of the preconditions for unloading the truck: $1 + T[3][\texttt{td}] + T[3][\texttt{pt}] = 7$. Similarly to $h^{\max}$, the value of $h^{\text{add}}(I, G)$ can be easily extracted from the table: $h^{\text{add}}(I, G) = T[5][\texttt{ta}] + T[5][\texttt{pd}] = 7$. What if the goal was $G = \{\texttt{td}, \texttt{pd}\}$? Then, we have $h^{\text{add}}(I, G) = T[5][\texttt{td}] + T[5][\texttt{pd}] = 10$.

6.5.2 Properties of $h^{\max}$ and h^{add}

Let us suppose that there are 100 packages in c that need to be delivered in d with the truck returning to a. What is the h^{add} value for this initial state? Since we are assuming that the problem can be solved independently for each subgoal and the cost to deliver a single package at d is 7, h^{add} estimates that the cost for delivering 100 packages is $100 \times 7 = 700$. The optimal solution cost to delivering 100 packages is $2 + 100 + 1 + 100 + 3 = 206$ (cost of driving to c, loading 100 packages, driving to d, unloading 100 packages, and driving back to a). As this example shows, $h^{\max}$ can overestimate h^* by a large margin. Note that $h^{\max}$ would still estimate the cost-to-go to be 4 as it approximates the total cost as the cost of the most expensive subgoal. This example also shows that $h^{\max}$ can underestimate h^* by a large margin.

$h^{\max}$ is admissible as it satisfies $h^{\max} \leq h^+ \leq h^*$. For h^{add} we have that $h^{\text{add}} \geq h^+$, and for some states, as we have seen in the example with 100 packages, we can have that $h^{\text{add}} \geq h^*$, so it is inadmissible.

6.5.3 Pseudocode for $h^{\max}$ and h^{add}

The function `build_table` below defines the dynamic programming procedure described in the previous sections for building the table for $h^{\max}$ and h^{add}. The function returns the table T from which the h-value can be easily computed, as we explained above. If one is interested in using $h^{\max}$ to compute the h-value of state s, then first we need to build the table by calling `build_table`$(s, G, P,$ `cost_max`$)$, where G is the goal of the problem, P the set of propositions, and `cost_max` is $h^{\max}$'s cost function.

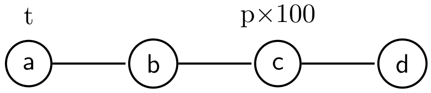
```
def cost_add($g, T$):
  if $|g| = 1$: return $T[g]$
  else return $\sum_{g' \in g} T[g']$

def cost_max($g, T$):
  if $|g| = 1$: return $T[g]$
  else return $\max_{g' \in g} T[g']$

def build_table($s, g, P$, cost):
  create line $T[0]$ of size $|P|$
  for $p$ in $P$:
    if $p$ in $s$: $T[0][p] = 0$
    else: $T[0][p] = \infty$
  $i = 0$
  while True:
    create line $T[i+1]$ of size $|P|$
    for $p$ in $P$:
      $T[i+1][p] = \min(\texttt{cost}(p, T[i]), \min_{a \in A, p \in \texttt{add}_a} c(a) + \texttt{cost}(\texttt{pre}_a, T[i]))$
    if $T[i] = T[i+1]$: return $T$
    $i = i + 1$
```

6.6 Relaxed-Plan Heuristic - $h^{\mathbf{FF}}$

The approximation h^{add} of h^+ can overestimate by a large the h^*-value of states, as we showed in the logistic problem below where a truck needs to move 100 packages from c to d and return to a. On the other hand, the approximation $h^{\max}$ of h^+ can underestimate the h^*-value for the same problem by a large margin. Next, we will consider h^{FF}, another approximation of h^+ that extracts a plan—known as the relaxed plan—from the table the dynamic programming algorithm produces while computing either $h^{\max}$ or h^{add}.



The table below shows the dynamic programming table of $h^{\max}$ for the logistic problem where we need to move a single package from c to d and return to a. The bottom row shows the action used to achieve each proposition. For example, while filling the table, in iteration 1, we set `tb` to 1 because we can use the precondition `ta` to apply the action `drive(a, b)` with the cost of 1. These actions are known as the best supporter actions for the relaxed problem. h^{FF} extracts a plan from the best supporter actions by initializing an `OPEN` list with the propositions of the goal and iteratively removing an arbitrary proposition p from `OPEN` and adding to the relaxed plan the best supporter action a that achieved p in the table. We then add to `OPEN` the preconditions of a. The process stops when `OPEN` is empty. This procedure retrieves the actions needed to achieve the goal propositions and it recursively retrieves the actions needed to achieve the preconditions of the actions needed to achieve the goal. All actions obtained in this process form the relaxed plan and the sum of their costs is the h^{FF}-value of the state for which the table was built.

The table below shows the state of `OPEN` for extracting the h^{FF}-value for the initial state of the logistic problem with a single package. h^{FF} also uses a `CLOSED` list to avoid inserting into `OPEN` propositions that were already accounted for in the computation. Note that `OPEN` is a set of propositions that allows us to

Iteration	`ta`	`tb`	`tc`	`td`	`pt`	`pa`	`pb`	`pc`	`pd`
0	0	∞	∞	∞	∞	∞	∞	0	∞
1	0	1	∞	∞	∞	∞	∞	0	∞
2	0	1	2	∞	∞	∞	∞	0	∞
3	0	1	2	3	3	∞	∞	0	∞
4	0	1	2	3	3	4	4	0	4
5	0	1	2	3	3	4	4	0	4
Actions	-	drive(a, b)	drive(b, c)	drive(c, d)	load(c)	unload(a)	unload(b)	-	unload(d)

remove an arbitrary proposition from it in every iteration of the algorithm. This is different from the `OPEN` lists we use in algorithms such as A*, where the list is a priority queue. We use the names `OPEN` and `CLOSED` with h^{FF} for historical reasons. Do not be confused with the data structures used in search algorithms.

Iterations	`OPEN`	`CLOSED`	Action Added
0	`pd`	-	-
1	`pt`, `td`	`pd`	`unload(p, d)`
2	`tc td`	`pt`, `pd`	`load(p, c)`
3	`tc`	`td`, `pt`, `pd`	`drive(c, d)`
4	`tb`	`tc`, `td`, `pt`, `pd`	`drive(b, c)`
5	-	`tc`, `td`, `pt`, `pd`	`drive(a, b)`

`OPEN` starts with `pd`. Note that although `ta` is part of the goal in this problem, we do not initialize `OPEN` with it because `ta` is already part of the initial state—the state for which we are computing the h^{FF}-value. The rightmost column in the table shows the actions added to the relaxed plan as we process propositions removed from `OPEN`. For example, in iteration 1, we remove `pd` from `OPEN` and add its best supporter action to the relaxed plan: `unload(d)`; the preconditions `pt` and `td` of `unload(d)` are added to `OPEN`. Similarly to the initialization of `OPEN`, in iteration 5, when we process `tb`, we do not add the precondition `ta` in `OPEN` because it is already present in the initial state. The h^{FF}-value is the sum of the cost of the actions `unload(p, d)`, `load(p, c)`, `drive(c, d)`, `drive(b, c)`, `drive(a, b)`, which is 5.

How about the problem with 100 packages in c? The process shown above would add to the relaxed plan one load and one unload action for each package. In this case we have $h^{\mathrm{FF}}(I) = 100 + 100 + 3 = 203$, where the 3 refers to the drive actions shown in the table above. Recall that the h^*-value for 100 packages is 206. h^{FF} offers a much more accurate estimate of the true cost than h^{add}, which estimates a cost of 700, and h^{max}, which estimates 4. Although this is only an example, it is representative of what we observe in practice: h^{FF} tends to offer more accurate estimates of the cost to go than h^{max} and h^{add}.

The pseudocode below shows how the h^{FF}-value is computed.

The function `hff` receives a state s and a set of goal propositions G; the function returns $h^{\mathrm{FF}}(s)$. First, h^{FF} computes the set of best supporter actions `BS`. In the pseudocode above it uses h^{max}, but it could also use h^{add}. `BS` is a dictionary mapping a proposition p to p's best supporter action. `BS` is passed to `extract_plan` along with s and G. The `extract_plan` function initializes `OPEN` with all propositions in G that are not in s. In every iteration, it removes a proposition p from `OPEN` and adds it to `CLOSED`. Then, it adds to variable `RPlan` the best supporter action a of p and it augments `OPEN` with the preconditions of a while removing what was already seen (i.e., what is in `CLOSED`) and what is in s. The `RPlan` is returned to `hff`, which simply sums the cost of all actions in `RPlan`. This sum is returned as the h^{FF}-value for s.

```
def extract_plan(BS, s, G):
  OPEN = G \ s
  CLOSED = {}
  RPlan = {}
  while OPEN != {}:
    remove g from OPEN
    CLOSED = CLOSED ∪ g
    RPlan = RPlan ∪ (BS[g])
    OPEN = OPEN ∪ (pre$_{BS[g]}$ \ (s ∪ CLOSED))
  return RPlan

def hff(s, G):
  BS = hmax(s, G)
  RPlan = extract_plan(BS, s, G)
  h = 0
  for a in RPlan:
    h += c(a)
  return h
```

6.6.1 h^{FF} is Inadmissible

h^{FF} tends to be much more accurate than both $h^{\max}$ and h^{add}. Although it does not happen often in practice, h^{FF} can overestimate the cost-to-go for some states, thus it is an inadmissible heuristic function. Let us consider a logistic problem with a single city, a, where a truck and two packages are located. To goal is to load both packages in the truck. The domain offers two actions for loading packages: `load` and `load2`. The first allows one to load one package at a time and it costs 1.0; the second allows two packages to be loaded at once and it costs 1.5. Clearly, the optimal solution uses action `load2` once with the cost of 1.5.

The computation of $h^{\max}$ is as follows.

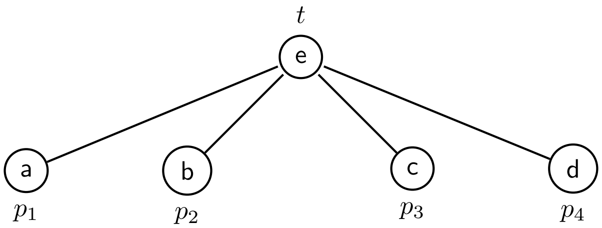
Iteration	`(at p1 a)`	`(at p2 a)`	`(at t a)`	`(in p1 t)`	`(in p2 t)`
0	0	0	0	∞	∞
1	0	0	0	1	1
Actions	-	-	-	load(p1, t)	load(p2, t)

If we run the `extract_plan` function described above, it will return the plan: `load(p1, t)` and `load(p2, t)` for a h^{FF}-value of 2, which is an overestimate of the optimal solution cost of 1.5.

6.7 Summary of Delete Relaxation Heuristics

The delete relaxation scheme offers a way of deriving the h^+ heuristic function. The problem of this approach is that the computation of h^+ involves solving the relaxed planning problem, which is NP-Complete in general. Instead of using h^+ directly, we approximate the h^+-values by assuming that the subgoals in the relaxed problem can be solved independently. The function $h^{\max}$ approximates h^+ by returning the most expensive subgoal, while h^{add} approximates h^+ by returning the sum of the cost of the subgoals. We have seen examples showing that $h^{\max}$ can underestimate the cost-to-go by a large margin, while h^{add} can overestimate the cost-to-go by a large margin.

We also showed that h^{FF} uses the relaxed plan obtained from the table constructed from $h^{\max}$ and h^{add} to approximate h^+. h^{FF} is an inadmissible heuristic function that often provides very accurate estimates of the cost-to-go. However, h^+ and its approximations fail to provide good guidance to the search algorithm in several problems. Let us consider the following logistic problem whose graph is "star-shaped". That is, the truck is located at e and each of the other cities, a, b, c, and d, has a package that needs to be transported to e. In this problem, one needs to perform 4 actions for each package, for a total of 16.



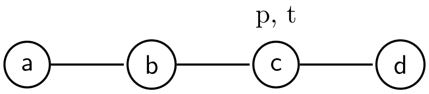
In the relaxed plan, the truck does not need to return to e (since we do not have delete effects, it is as if the truck never left e). Thus, $h^+(I) = 3 \times 4 = 12$. If we increase the number of cities and packages to 100 we have that $h^*(I) = 400$ while $h^+(I) = 300$; the larger the problem, the less accurate h^+ becomes.

Delete relaxation heuristics fail in these "star-shaped" problems that require back-and-forth actions. They also tend to fail to capture useful information in problems involving resources. For example, delete relaxation heuristics assume that one can fill the tank with gas and then drive forever. Other heuristic functions for classical planning attempt to overcome some of these problems. For example, Red and Black planning offers a way of deriving heuristic functions where some of the delete effects are taken into account. These heuristic functions are out of the scope of these notes. Nonetheless, the delete relaxation heuristic functions we covered in these notes are largely used in classical planners. In the next section we consider another way of using the information gathered while computing the h^{FF} heuristic to speed up the search.

6.8 Helpful Actions

Another powerful idea around the computation of h^{FF} is the one of helpful actions. h^{FF} computes a relaxed plan and the actions used in such a relaxed plan are known as helpful actions. Instead of searching with all actions available in the domain, one could substantially reduce the branching factor of the problem by considering only the set of helpful actions for each state encountered in search.

Let us consider the following logistic problem where the truck is in c and the package is in the truck. The goal is to have both the truck and the package in d. The table below shows the computation of $h^{\max}$ for the problem.



We can then run the procedure to extract the relaxed plan as shown in the table below.

The actions `drive(c, d)` and `unload(d)` are the helpful actions for this state, as they form the relaxed plan. Instead of considering all possible actions at the state {`tc`, `pt`}, which are: `drive(c, b)`, `drive(c,`

Iteration	`ta`	`tb`	`tc`	`td`	`pt`	`pa`	`pb`	`pc`	`pd`
0	∞	∞	0	∞	0	∞	∞	∞	∞
1	∞	1	0	1	0	∞	∞	1	∞
2	2	1	0	1	0	∞	2	1	2
3	2	1	0	1	0	3	2	1	2
4	2	1	0	1	0	3	2	1	2
Actions	drive(b, a)	drive(c, b)	-	drive(c, d)	-	unload(a)	unload(b)	unload(c)	unload(d)

Iterations	`OPEN`	`CLOSED`	Action Added
0	`td`, `pd`	-	-
1	`pd`	`td`	drive(c, d)
2	-	`pd`, `td`	unload(d)

`d)`, and `unload(c)`, we only consider `drive(c, d)` during search (we do not consider `unload(d)` because it cannot be applied at $\{\texttt{tc}, \texttt{pt}\}$). The use of helpful actions can substantially reduce the branching factor of the search and it allows the search algorithm to focus its search on the actions that are deemed important in the derivation of the h^{FF}-value for each state.

6.8.1 Helpful Actions Can Make the Problems Unsolvable

Helpful actions filter out some of the actions available to the planner. While they can substantially speed up the search, they can also render the problem unsolvable. Let us consider the following example.

Propositions: $\{\texttt{package\_intact}, \texttt{loaded}, \texttt{locked}, \texttt{transported}\}$
Initial State: $\{\texttt{package\_intact}\}$
Goal: $\{\texttt{package\_intact}, \texttt{transported}\}$

The domain has the following action schema.

`load` : (Pre_a :$\{\texttt{package\_intact}\}$
Add_a :$\{\texttt{loaded}\}$
Del_a :$\{\}$)

`lock` : (Pre_a :$\{\texttt{package\_intact}, \texttt{loaded}\}$
Add_a :$\{\texttt{locked}\}$
Del_a :$\{\}$)

`drive_locked` : (Pre_a :$\{\texttt{package\_intact}, \texttt{drive\_locked}\}$
Add_a :$\{\texttt{transported}\}$
Del_a :$\{\}$)

`drive` : (Pre_a :$\{\texttt{package\_intact}, \texttt{loaded}\}$
Add_a :$\{\texttt{transported}\}$
Del_a :$\{\texttt{package\_intact}\}$)

In order to solve this problem we need to load the package, lock it, and then "drive locked". If we only load and drive, then the package will no longer be intact (see the delete effect of `drive`) and the goal requires that the package is intact. If we run $h^{\max}$ for the initial state of this problem we obtain the following.

Iteration	`package_intact`	`locked`	`loaded`	`transported`
0	0	∞	∞	∞
1	0	∞	1	∞
2	0	2	1	2
2	0	2	1	2
Actions	-	lock	load	drive

Note how `drive_locked` is not a best supporter action for `transported`. If we extract a plan from the table above, `drive_locked` will not be a helpful action. This is also true for the state where the truck is loaded as the `drive` action achieves `transported` with a cheaper cost than `drive_locked`. A search algorithm using helpful actions would not be able to solve this problem as `drive_locked` would not be available in search.

6.8.2 Handling Lack of Completeness

Different planners use different strategies for handling the lack of completeness once we search in an action space that only accounts for helpful actions. For example, the Fast Forward planner first runs the search in the action space defined by the helpful actions. If the space is exhausted and the planner did not find a solution, then it runs the search again in the original action space.

The planner LAMA performs search with GBFS with two priority queues: one for states generated with any action, as we normally do, and another for states generated with helpful actions. The pseudocode below shows the part of the GBFS algorithm where it generates a state and pushes it into the priority queues.

```
...
Q = decide_queue()
n = Q.pop()
for a in A(n):
  Qa.push(T(n, a))
  if a is a helpful action:
    Qh.push(T(n, a))
...
```

In the pseudocode, `Qa` is the priority queue receiving all states generated in search; `Qh` is the queue receiving only the states generated from helpful actions. The function `decide_queue` encodes a policy for deciding which priority queue is expanded next in search. If this policy chooses to expand states from `Qh` more often than `Qa`, then the search will focus on the states reachable through helpful actions. The `Qa` ensures completeness as the policies employed in search eventually expands states from `Qa`.

Policies for Choosing Priority Queue

Perhaps the simplest policy one can employ is to alternate the queues: in one iteration the algorithm expands a node from one of the queues, in the other iteration, it expands a state from the other queue. In practice planners use “more aggressive” policies, where the priority queue of helpful actions is chosen more often.

For example, LAMA implements a policy that uses a priority score for each queue. Both queues are initialized with the score of zero, and in each iteration, the search expands a state from the queue with highest priority (ties are broken arbitrarily). If “progress” is made with the priority queue storing states generated with helpful actions, then the score of it is incremented by $1,000$. We say that an expansion “makes progress” if it generates a state whose heuristic value is the lowest h-value observed in search. Once the score of the helpful actions queue is incremented by $1,000$, the search will expand at least $1,000$ states from it before expanding a state from the other queue.

6.9 Deferred Evaluation of Heuristic Values

Classical planners implement many enhancements that are not necessarily related to deriving more accurate heuristic functions. The use of helpful actions is one of such enhancements. Another enhancement is known as deferred evaluations, which attempts to reduce the overall computational cost of evaluating states with heuristic functions. This enhancement can be helpful because the computation of the heuristic values dominates the running of search in classical planning. Note that this is not true for computationally cheap heuristics such as octile distance in grid-based pathfinding.

Planners using deferred evaluation inserts every state s in `OPEN` with the h-value of s’s parent in the tree. The heuristic value of s is computed only when s is expanded. The goal with deferred evaluation is to save the computation of the h-value of all states in `OPEN` that are never expanded (i.e., they are in `OPEN` when the search terminates). The drawback of using deferred evaluation is that the heuristic function is not as accurate as it could be as the h-value used is not of the states themselves, but of the parents of the states.

The pseudocode below shows how deferred evaluation can be implemented in GBFS.

```
...
while Q is not empty:
  n = Q.pop()
  h = compute_h(n)
  for a in A(n):
    Q.push(T(n, a), h) # insert n's children in OPEN with n's h-value
...
```

Let us consider the following example to gain some intuition of how deferred evaluations can speed up the search. In this example we assume that the computational cost of generating a state is much smaller than the cost of computing the h-value of a state. Let t_h and t_g be the expected running time for computing the h-value and for generating a state, respectively. So we have that $t_h \gg t_g$. Also, let n be the number of children a state s has. The total running time for expanding s is the following (the first term is due to the cost of generating the children and the second due to the computation of the heuristic value of the children).

$$t_h \cdot n + t_g \cdot n \approx t_h \cdot n$$

The total expansion running time can be approximated as $t_h \cdot n$ because $t_h \gg t_g$. If deferred evaluation is used, then all n children are added to `OPEN` with the h-value of the parent s. The cost of this operation is $t_g \cdot n$ (we ignore the evaluation of s itself when it is expanded as our goal is to compare the regular evaluations with deferred evaluations and s is evaluated in both approaches).

Let c be the child of s with the smallest h-value, i.e., $h(c) < h(c')$ for any child $c' \neq c$ of s. Without deferred evaluation, GBFS expands c right after s because c is inserted in `OPEN` with its h-value. Thus, the running for expanding c after expanding s can be approximated as $t_h \cdot n$, as we discussed above. By contrast, GBFS expands the children of s in an arbitrary order as all children are inserted with the h-value of s (i.e., the search is unable to distinguish c from the other children). Let $n' - 1$ be the expected number of children of

s that are expanded before c (i.e., c is the n'-th child of s to be expanded in search). Then the expected running time for expanding c is the following.

$$t_h \cdot n' + t_g(n + n \cdot n') \approx t_h \cdot n'$$

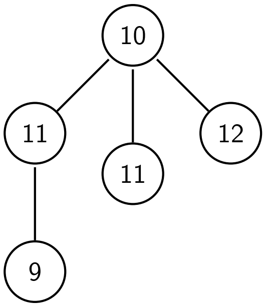
The term $t_h \cdot n'$ accounts for the computation of the h-value of the n' children of s (recall that in deferred evaluation the h-value is computed as the state is expanded); the term $t_g(n+n \cdot n')$ accounts for the generation of all n children of s and of the n children of each of the n' children of s that is expanded (many of the children of s can potentially be expanded before c is expanded). Since $t_h \gg t_g$, the expected running time to expand c can be approximated as $t_h \cdot n'$.

In the worst case for deferred evaluations, we have that $n = n'$ (i.e., c is the last child of s to be expanded). However, often $n' < n$, especially in problems with large branching factor. Although this example is perhaps overly simplified (e.g., we do not account for the computational cost of managing the priority queue and we simply ignore the computational cost of generating nodes), it shows how deferred evaluations can reduce the overall running time of the search. Despite reducing the accuracy of the heuristic function and forcing the algorithm to expand more states than what it would with GBFS's regular expansion scheme, the total running time can be reduced by reducing the total number of heuristic evaluations performed.

6.10 Enforced Hill Climbing

Local search algorithms such as variants of Hill Climbing are also commonly used in classical planners with the goal of reducing the number of states expanded (and thus heuristic evaluations) one needs to perform in order to find a solution to the problem. One of such algorithms is Enforced Hill Climbing (EHC), which is used the Fast Forward planner. EHC uses Hill Climbing to make quick progress in the search and Breadth-First Search for escaping local minima. Instead of only looking at the neighbors of the current state, as Hill Climbing does, EHC runs a Breadth-First Search from the current state and it stops as soon as it encounters a state with h-value smaller than the current state.

Let us consider the example below where the number in the nodes in the tree represent the h-values of the states. Hill Climbing stops searching as soon as it expands the children of the current state as none of the children have smaller h-value than the current state. By contrast, EHC performs a Breadth-First Search from the current state and is able to find the state with h-value of 9 in the tree. The state with h-value of 9 becomes the next current state and the same procedure is repeated. The pseudocode below shows EHC.



In the pseudocode above we check whether the state s is a goal state; if it is, then EHC returns the solution path. If s is not a solution, EHC runs Breadth-First Search (BFS) from s, which returns the next state s' where $h(s') < h(s)$. BFS can also return 'failure' as it can run out of memory while attempting to encounter a state s' with $h(s') < h(s)$. In that case, EHC also returns 'failure'. If the BFS is successful, then we assign

```
def enforced_hill_climbing(s, h):
  while True:
    if s is a goal: return path
    s', failure = BFS(s, h)
    if failure: return failure
    s = s'
```

s' to s and repeat the same procedure. EHC is a complete search algorithm if $h(s) = 0$ if and only if s is a goal state and the problem has no undetectable dead-end states (i.e., state from which one cannot derive a solution). If there are undetectable dead-ends, the search could assign such a dead-end as the state s and the search will not be able to find a solution from s.