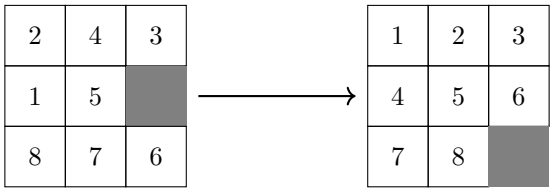
CS 217: Artificial Intelligence and Machine Learning **Jan-Apr 2024**

Lecture 24: Classical Methods of AI

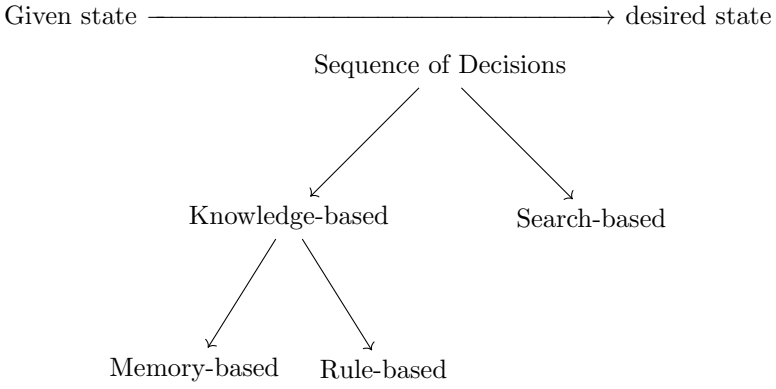
Lecturer: Swaprava Nath *Scribe(s): SG47 & SG48*

AI for problem solving

Previously we had discussed the reasoning of decision making under rational agents. In this lecture, we will see how AI is used for problem-solving. For example, solving a puzzle or finding the optimal path to a desired destination. A sequence of decisions are made to move from a given state to a desired state. Unlike the multi-agent problems seen previously, this is a single-agent (1-player) problem involving steps like moving right, left, up, or down. For example, to solve a Rubik's cube we need to reach the final state where every face has tiles of the same colour. At each state, there are a finite number of moves (18 different possible moves). Though solving the Rubik's cube is a hard problem, it is fairly doable if one follows a step-by-step method to solve the cube in a layer-wise fashion. These methods were developed through past experiences at solving the rubiks cube. Another example of a single-agent problem is the game where a board with movable tiles is given. Each tile has a number on it. Initially, the board is given with jumbled numbers and we are required to order those numbers in increasing order.



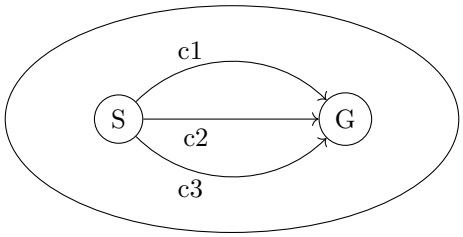
Methods of Decision making



- **Knowledge based approach**: This approach involves exploiting additional knowledge gained from experiences.
  - **Memory based**: Experiences are stored so that the next time a problem instance occurs, we can start from the nearest stored experience to solve it.
  - **Rule based**: A domain expert formulates a rule based on past experiences.
- **Search based**: This is the first approach used by any intelligent agent when no past experiences are available.

State Space Search

There is a set of all possible states that the given problem can be in. Some of the states are defined as goal states, which are the desired states to be reached. Given the start states, we connect them to all possible other states that can be achieved by using actions available at start state. This process is repeated recursively over each state until goal states are achieved. This leads to a network where each node represents a state and edges represent actions taken to move from one state to another.



State Space is the set of all possible states.
In the above diagram S is the Start State and G is the Goal State. An action taken at a state brings us to another state. Each action has an associated cost c_i. There could be multiple paths from a given state to a goal state. States as well as actions that define the transition between states are defined in the problem statement. After this reduction, we just need to find a path from start state to one of the goal states.

Some important terms:

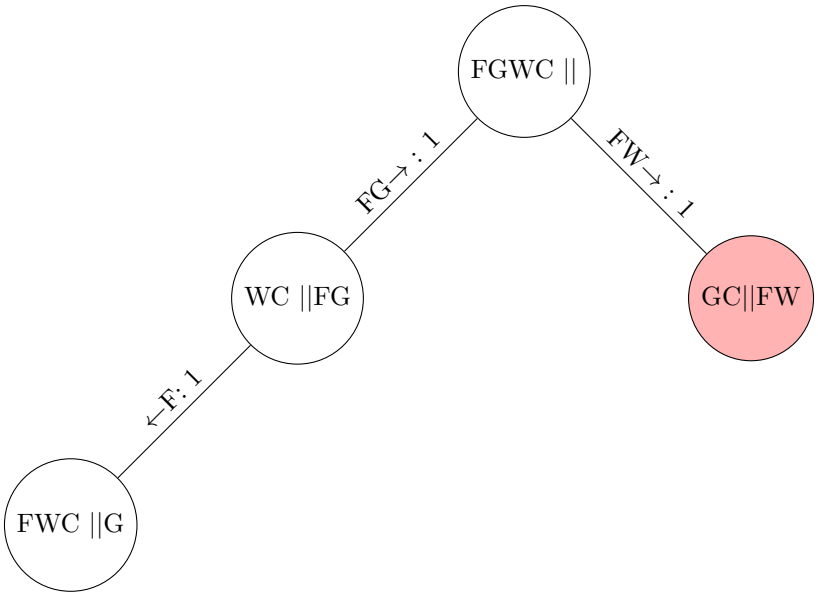
- Set of states : $S_1, S_2, \ldots, S_m$
- Action(s) : possible actions at state S
- Neighbours(s) : States which can be reached from a given state S
- isGoal(s) : whether a given state is the desired state or not, that is whether we have reached the final state to finally terminate the algorithm

Here no player function is present as it is a single-player game (single agent). Also, there is no utility function, the only goal is reach the final desired state possibly with the minimum cost.

River Crossing Puzzle

This puzzle is the classical puzzle with a wolf, a goat, a cabbage, and a farmer on one side of the river with a single boat. While solving the puzzle, the wolf and the goat should not be on the same side of the river in the absence of the farmer. Similarly, the goat and the cabbage also should not be on the same side of the river in the absence of the farmer. While the boat is moving from one bank of the river to another, a maximum of 2 out of the 4 elements can be transported. Also, the farmer has to be in the boat as he is rowing the boat. The question here is: what is the minimum number of rounds for which all 4 elements move from one bank to another following all the conditions? We are going to construct a state space search diagram for it.

State Space Search



- `F` stands for Farmer, `G` for Goat, `C` for Cabbage, `W` for Wolf, and `D` stands for Boat.
- The state colored in red indicates that the game ended in an undesirable outcome, as it violated the rule of not leaving the goat and cabbage together without the farmer's supervision.
- Our goal state is to have the configuration `||FGCW`, and the cost of each movement is `1`, as given on each edge.
- Here, the model attempts to search for the tree to reach the goal in the least number of steps.

The minimum cost path is 7 (trips).

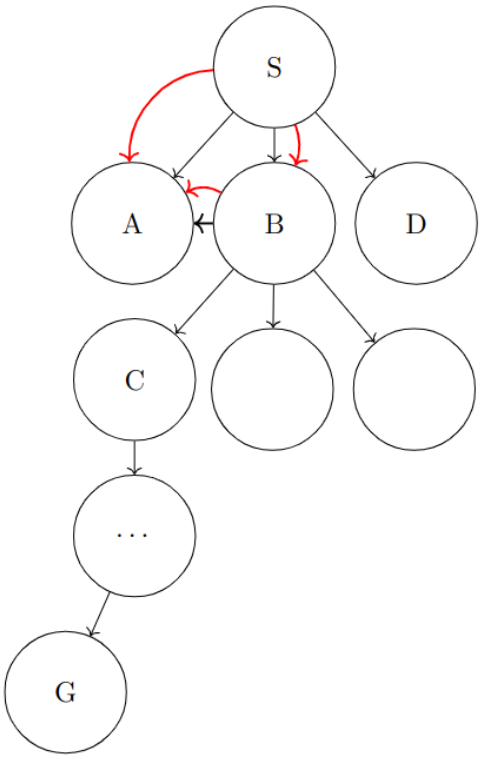
Search method requirements

- Abstraction
  - Consists of states, actions, neighbours, goal and cost . Abstraction depends on the problem instance

  - This process defines the objective to be achieved in terms of minimization of incurred cost

- Algorithm development
  - Once abstraction is done, algorithm development remains oblivious of actual problem statement
  - Finds the path from start state to goal state minimizing the cost can be done by various algorithms like BFS, DFS, Uniform cost search Informed search - A* search

General Anatomy of a Search Algorithm



The above tree representation shows how a search algorithm operates. Its possible to return to an already visited state under the algorithm so we have to maintain a set called Visited Set which includes all the visited nodes, so we don't search them again. Here, G is one of the goal states.

Visited States

S

Current State

B

Lets denote the current state by N, which is initialized to S when we start the algorithm. Suppose now, our current state is B and so will search for all B's neighbours.

Algorithm:

for state $c \in$ **neighbours** (N)
    if isGoal (c) **then**
        do something
    else
        do something else
    endif

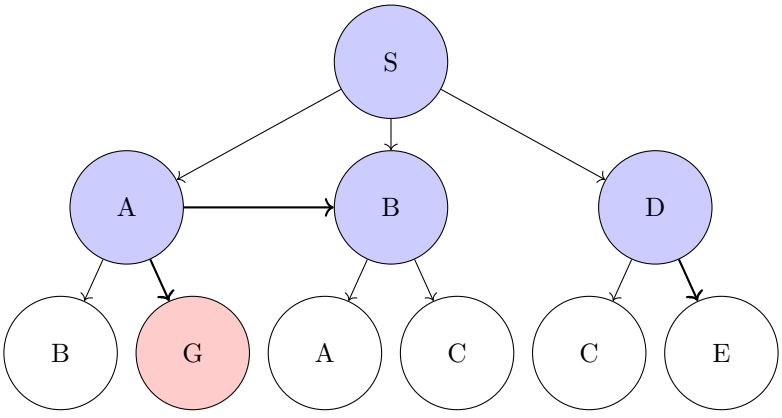
Search Algorithms

There are in general two types of search algorithms

- **Uninformed Search** - These search algorithms have no additional information on the goal node other than the one provided in the problem definition. The plans to reach the goal state from the start state differ only by the order and/or length of actions. Examples include Breadth First Search, Depth First Search etc.
- **Informed Search** - These algorithms have information on the goal state, which helps in more efficient searching. This information is obtained by something called a heuristic Examples include Greedy Search, A* Tree Search etc.

Breadth First Search

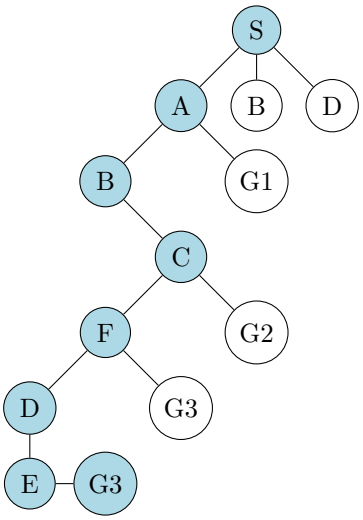
Breadth first search (BFS) is an algorithm for searching a tree data structure for a node that satisfies a given property. It starts at the tree root and explores all nodes at the present depth prior to moving on to the nodes at the next depth level. Extra memory, usually a queue, is needed to keep track of the child nodes that were encountered but not yet explored.



BFS finds the shortest path to the goal, though it need not be the least cost path. Here, we will have a set of visited vertices. The search can be explained with the help of the above example,

- First, node S is visited, so Visited = $\{S\}$
- Then, we search for the next depth level, which includes the node A, B and D, so Visited = $\{S, A, B, D\}$
- Following that, in the next depth level, the first node B is already in the visited set, so we won't include B again.
- G is our goal, so as soon as we reach G we finish our search.

Depth First Search



The colored nodes represent the path chosen using depth first search. It treats the newly explored nodes as stacks. It finds the goal quickly. which is neither shortest not minimum cost.
The overall algorithm is that we put S first into the stack and mark it as visited when we visit it and push all of its children into the stack. Also while visiting we check if that node is visited or not. It is just a normal dfs search algorithm. In the given example :-

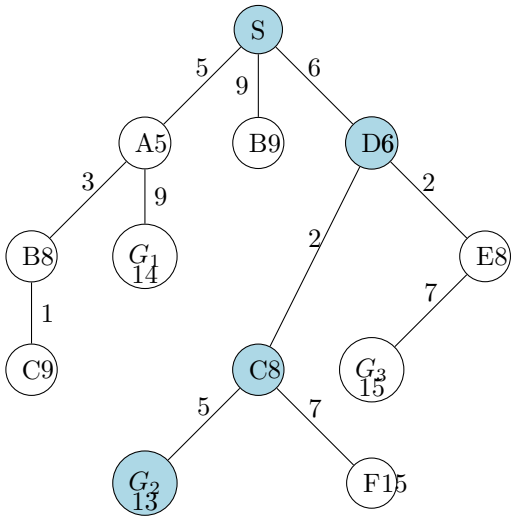
1. S is marked visited and A,B,D are pushed onto the stack.
2. Then A is marked visited and B,G1 are pushed onto the stack.
3. Then B is marked visited and C is pushed onto the stack.
4. Similarly we will do these steps and always visit nodes that are not visited.
5. Eventually this algorithm reaches G3 and stops and finds the path to goal in shortest time.

Issues With BFS and DFS

BFS and DFS find the goal nodes quickly however, it is possible that the path found is neither the shortest path nor the path with minimum cost. Hence, the Uniform Cost Search algorithm was developed for Uninformed Search.

Uniform Cost Path

It is basically Dijikstra's algorithm for finding minimum cost path from a source to a vertex. As we know that Dijikstra's algorithm picks the path with the least cost at any given iteration (it works greedily).



The colored path is the path chosen by the uniform search algorithm in this case. Here it will be the same like the previous one but just we will not use stack like functionality, i.e. just explore along the depth, instead we will explore greedily.
For the given example :-

1. Firstly mark S as visited and move all its children to the priority queue (as we use it for dijisktra). So, A,B,D move into the priority queue with the values (i.e. 5,9,6).

2. Then mark A as visited as it is the least value in the priority queue and then push B,G1 in the priority queue with values $(3 + 5 = 8)$,$(5 + 9 = 14)$ respectively.

3. Then mark D as visited and push C in the priority queue with value 8 and E with value 8.

4. At this step, it could have chosen C, E also but just to come to this example, B was chosen in the class(you can choose any one arbitrarily and try). Mark B as visited and push C with a value of 9.

5. Mark E as visited and push G3 in the priority queue with a value of 15.

6. Mark C as visited and push G2 and F with values 13 and 15 respectively.

7. Now we get G2 as the least value and it is a goal state so we got a path with minimum cost to a goal. We can just use backtracking to trace out the path that is shown in blue color.

So far, we have been looking at uninformed search methods, where we had no prior information to weigh our decisions and compare them, so the only option was an exhaustive search. Now we will look at informed (heuristic) searches, where we have some estimated costs of the decisions.

Informed Search: The A* algorithm

The goal of A* search is to find the shortest path from a starting node to a goal node in a graph, where each node has an estimated non-negative cost associated with it. This cost is an estimate of the shortest distance from that node to the nearest goal node.

Let's understand how the A* algorithm works with an example. Consider the graph:

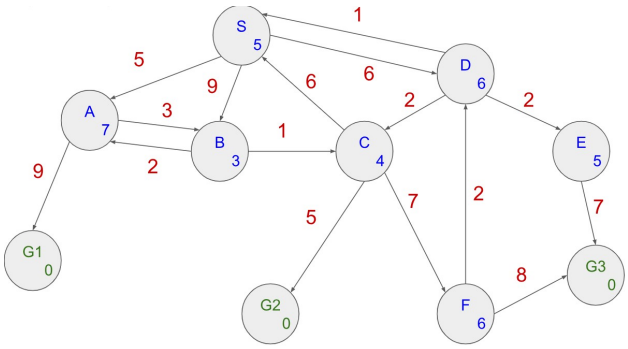


Figure 24.1: Example Graph for A* Search

In the above graph, S is the start node, and G1, G2, G3 are the goal nodes. The subscripts in a node denote the estimated (heuristic) shortest distance from that node to the nearest goal node.

A* score of node V = distance from source to V + estimated distance from V to nearest goal

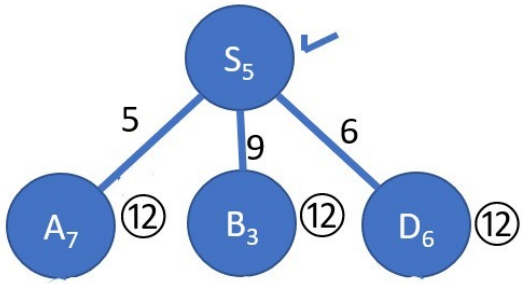
A walkthrough of the algorithm

We start off at S, and look at all the possible states we can reach from S. Whenever we explore a node (i.e., look at what all states it can take us to), we mark the node as 'visited'. Thus, currently, our set of visited nodes is {S}. We never explore a visited node again.

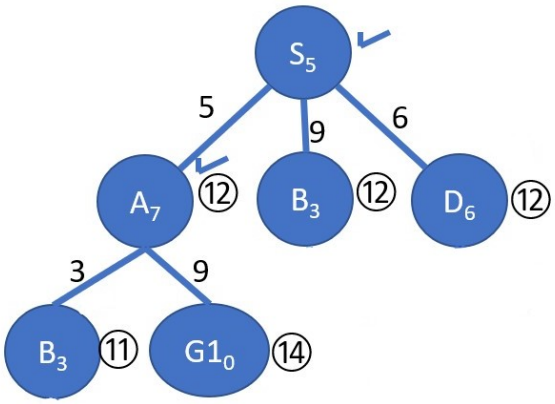
We assign an **A* score** to the nodes we have encountered but not yet explored. We call the set of such nodes the *frontier*. The A* score of a node is calculated as follows:

A* score of node V = distance from source to V + estimated distance from V to the nearest goal

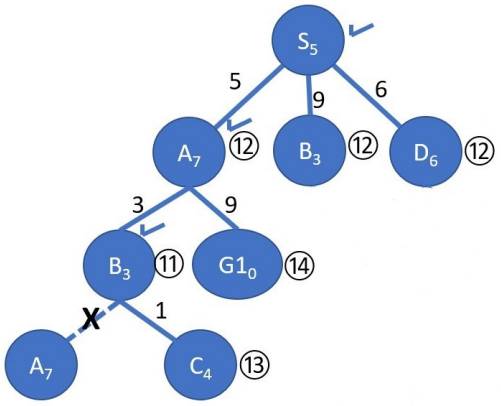
The A* scores for the nodes in our frontier are shown in black below:



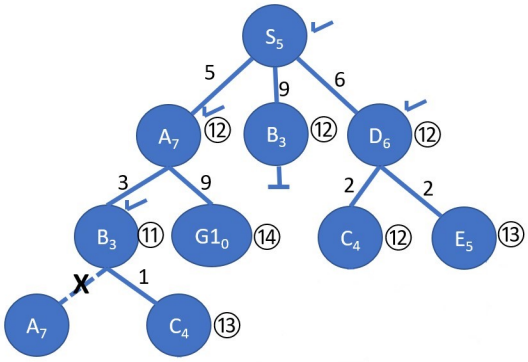
Now the next node we select to explore is **the node with the minimum A* score among the nodes in the frontier**. Ties are broken arbitrarily. In our case, the nodes A, B, and D have the same A* score, and supposing we choose to break ties from left to right, we choose A to explore next.



The visited set is now {S,A}. Now the node with the least A* score in the frontier is B, with an A* score of 11. Hence, we explore B next.



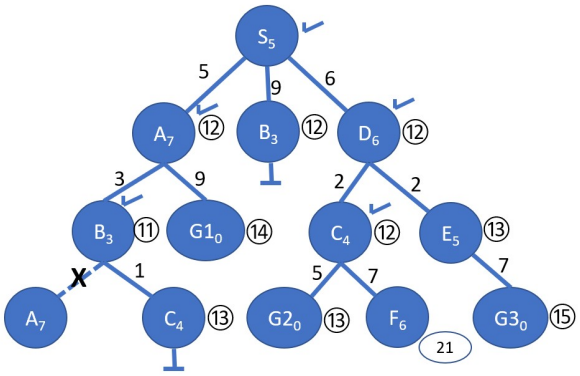
The visited set is now {S,A,B}. Observe that we will never choose A again as it is already visited.
The lowest A* score in the frontier is a tie between B and D, but we don't explore B as it is in our visited set. Thus, we visit D next.



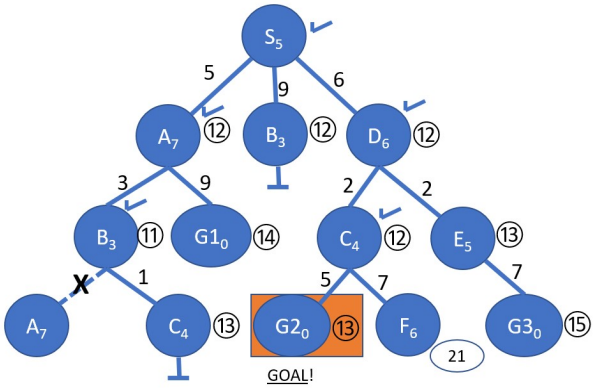
The visited set is now {S,A,B,D}. The lowest A* score in the frontier is now that of node C, hence we visit C next. This process continues.

Eventually, we arrive at the following situation[1]:

[1]Assuming at an intermediate step the tie between E and G2 was broken in favour of E.



Now the lowest A* score in the frontier is of G2, hence we visit G2.



Now that we have reached a goal state, the search ends.

Hence, the minimum distance between the start state and a goal state in the given graph is 13.

- If the estimate is smaller than the actual cost, then the algorithm is always optimal and finds the least cost path.
- If the estimate is zero then the given algorithm is similar to Dijkstra's algorithm.
- When estimates are perfect (i.e., they exactly match the actual costs), the least cost path is quickly found.
- Overestimating costs typically leads to sub optimal solutions in this algorithm.