Cairo University
Faculty of Computers & Artificial Intelligence
Department of Computer Science
Artificial Intelligence Course

Lab 4: Uninformed Search

In many tasks, we know what a solution looks like, but we do know the exact steps that produce a valid solution. Specifically, these tasks need an agent that starts with the environment in a given state and acts upon the environment until the altered state fits the goal of the task. These tasks (called **"Search Problems"**) **can be solved by search algorithms.**

Before we explain the search algorithms, let's look at the nature of search problems. In search problems, we need to have the following information:

- **Initial/Current state** of the environment
- **Goal state** (desired state) that fits some goal criteria
- **Actions** that transition from one state to another
- **Transition model** that defines what each action does
- **Costs** of actions/paths *(optional)*
- **Rules** of the problem/environment *(optional)*

Note: Each search problem has a ***search/state space*** *which is the set of all possible states and transitions between them. A tree/graph representation of the search space is called a* ***search tree/graph.*** *The root of the search tree/graph corresponds to the initial state and each branch at some level corresponds to an action.*

Search algorithms aim at finding a sequence of actions that can transform the initial state into a goal state. Search algorithms can be categorized into 2 main categories:

- **Uninformed Search** *(has no additional information about the closeness of the goal)*
- **Informed Search** *(has additional information about the closeness to the goal)*

A Simple Example:

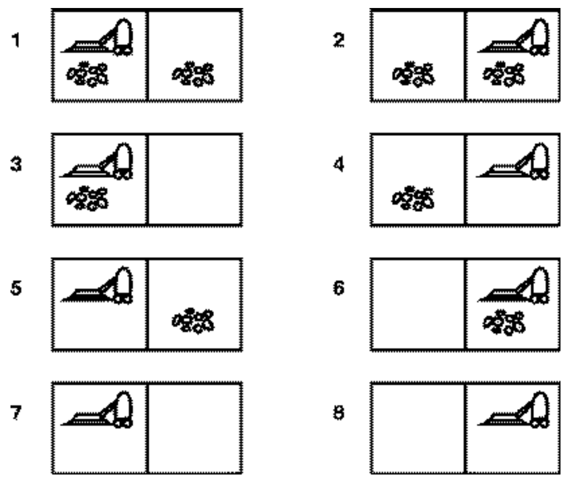
Suppose we have 2 tiles, each is either clean or dirty. We want to reach a state where both tiles are clean. So, we have a vacuum cleaner that can only be on one tile at a time. The vacuum cleaner can either move from one tile to the other or clean the tile it is on.

The current situation is that only tile 1 is dirty and the vacuum cleaner is on the second tile. **How can this vacuum cleaner use "search"** to find the steps it should take to reach the goal?

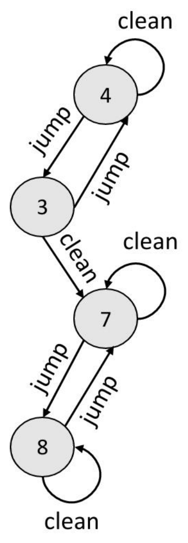
First, let' extract some information about the problrem in order to apply search:

- **Initial/Current state:** Tile 1 is dirty, tile 2 is clean, the vacuum cleaner is on tile 2.
- **Goal state:** Tile 1 is clean and tile 2 is clean *(the vacuum cleaner's location does not matter).*
- **Actions:** Move and clean
- **Costs**: None
- **Rules**: None

The state space is:



The search graph is:



Since we have no additional information about the closeness to the goal, we will apply **uninformed search.**

If we think about what uninformed search should do to get the sequence of actions, ***the simplest steps we can think of are:***

1. **Check whether the current state is a goal state.** If yes then stop the search, otherwise go to step 2.
2. Try an action and **get the next state.**
3. Check that **the next state has not been already explored/visited.** If it **has been visited, backtrack** and go to step 2.
4. Check that **the next state does not violate the rules** of the problem.
5. Add the current state to the list of visited states and go to step 1 with current = next.

So, for this problem, we start at state #4:

a. *Is state #4 a goal?* No.
b. *Try "move" action.* The next state is #3.
c. *Is state #3 in the visited states list?* No.
d. *Is state #3 okay?* Yes.
e. Visited list is [state #4], the current state is state #3.

Repeat

a. *Is state #3 a goal?* No.
b. *Try "move" action.* The next state is #4.
c. *Is state #4 in the visited states list?* Yes. (Go to step 2)
d. *Try "clean" action.* The next state is #7.
e. *Is state #7 in the visited states list?* No.
f. *Is state # okay?* Yes.
g. Visited list is [state #4, state #3], the current state is state #7.

Repeat

a. *Is state #7 a goal?* Yes. The search is complete, the steps are "move" then "clean".

Notice that in these steps, **we explored the states in the search tree by going as far as possible along each branch** (each action) before backtracking. This type of uninformed search is called **"Depth-first Search" (DFS).**

Due to the power of backtracking of Prolog, we can simply write the depth-first search steps as follows:

```
search(CurrentState, Visited, Goal):-
    CurrentState = Goal, !,     % Step 1
    write("Search is complete!"), nl,
    write(Visited), nl, write(CurrentState).

search(CurrentState, Visited, Goal):-
    move(CurrentState, NextState),     % Step 2
    not(member(NextState, Visited)),  % Step 3
    isOkay(NextState),     % Step 4
    append(Visited, [CurrentState], NewVisited),    % Step 5.1
    search(NextState, NewVisited, Goal).         % Step 5.2
```

For the "search" rule to run correctly, we need to be able to **represent the state** and to **implement "move" and "isOkay".** These 3 points are needed for the search algorithm to run, however they are **problem-specific.**

For the given vacuum cleaner problem:

```
% Representation can be a list [Tile1State, Tile2State, VacuumLoc]
% Example: state #4 is represented as [dirty, clean, 2]

move([X,Y,1], [X,Y,2]).     % move
move([X,Y,2], [X,Y,1]).     % move

move([dirty,Y,1], [clean,Y,1]).     % clean tile 1
move([X,dirty,2], [X,clean,2]).     % clean tile 2

isOkay(_):- true.     % This problem has no rules
```

Run using the query:

```
?- search([dirty,clean,2], [], [clean,clean,_]).
```

Question:

Is depth-first search the only method of uninformed search that exists?

Answer:

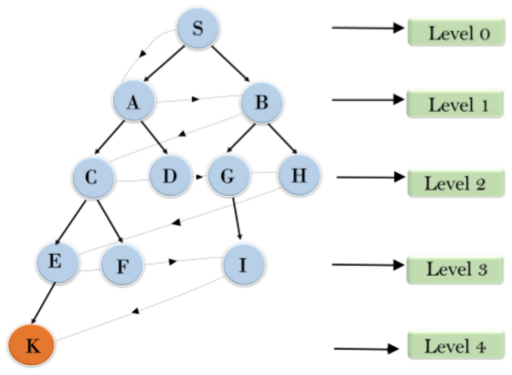
No, other algorithms exist as well.

Uninformed Search Algorithms:

- **Breadth-first Search:**

  Breadth-first search (BFS) starts searching from the root state (node) of the search tree and expands all nodes at the current level before moving to nodes of the next level **(i.e., it goes through the tree level by level).**

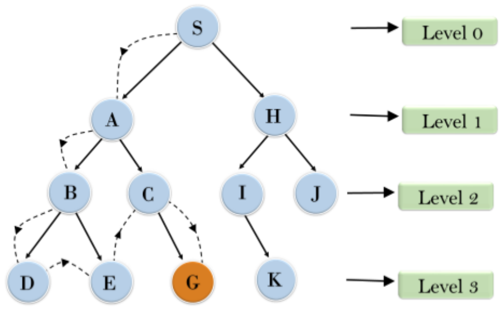
  It can be implemented using a **FIFO queue** where **new nodes are added at the end.**



- **Depth-first Search:**

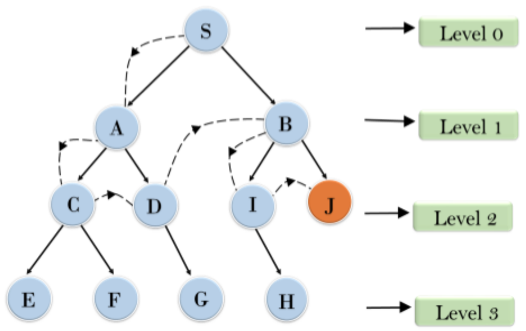
  Depth-first search (DFS) starts from the root node and **follows each path to its greatest depth** node before moving to the next path.

  It can be implemented using a **LIFO queue** where **new nodes are added at the beginning.**



- **Depth-limited Search:**

  Depth-limited search (DLS) is a variant of DFS that works exactly **like DFS but with a maximum depth** to solve the drawback of the infinite paths in the DFS.



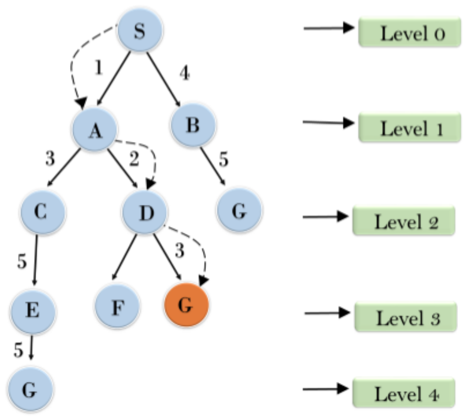
- **Iterative Deepening Search:**

  This search algorithm performs **depth-first search up to a certain "depth limit",** and it **keeps increasing the depth limit** after each iteration until the goal node is found.

- **Uniform-cost Search:**

  Uniform-cost search (UCS) comes into play when a different **cost is available for each edge.** The primary goal of the uniform-cost search is to find a path to the goal node which has the lowest cumulative cost. Uniform-cost search **expands nodes according to their path costs** from the root node.

  It can be implemented using a **priority queue.**



- **Bidirectional Search:**

  This search algorithm **runs two simultaneous searches**, one from the initial state and another from the goal node, and stops when these two search graphs intersect. It can use search algorithms such as BFS, DFS, DLS, etc.

Evaluation Criteria:

- **Completeness** *(Does the technique always find a solution if it exists?)*
- **Optimality** *(Does the solution have the minimum cost?)*
- **Time complexity** *(How long does it take to find a solution?)*
- **Space complexity** *(How much space is occupied during algorithm run?)*

Time complexity is usually measured in terms of the nodes explored and space complexity is usually measured in terms of the maximum size that the nodes list becomes during the search.

Algorithm	***Completeness***	***Optimality***	***Time***	***Space***
BFS	Yes	Yes (if costs of moves at the same level are the same)	$O(b^d)$	$O(b^d)$
DFS	No	No	$O(b^m)$	O(bm)
DLS	No	No	$O(b^l)$	O(bl)
IDS	Yes	Yes	$O(b^d)$	O(bd)
UCS	Yes	Yes	$O(b^{d+1})$	$O(b^{d+1})$
Bidirectional Search	Yes	Yes	$O(b^{d/2})$	$O(b^{d/2})$

where b is the branching factor, d is the solution depth, m is the maximum depth, and l is the depth limit.

Question:

What if, in DFS, we did not have the power to automatically backtrack if a move fails, or what if we wanted to implement a different search algorithm like BFS for example, how can we modify the simple uninformed search steps to work?

Answer:

We would need to **keep track of the generated states** that need to be explored/expanded, that is why we will need another list called the **"frontier"** or the "**open** list".

The Formal Uninformed Search Algorithm:

1. **Pop a state from the open list** so that it becomes the current state.
2. **Check whether the current state is a goal state.** If yes then stop the search, otherwise go to step 3.
3. **Get the list of all valid next states** (children) from the current state.
4. Add the next states to the **open list**.
5. Add the current state to the list of visited states **(closed list)** and go to step 1.

Notice that **"valid"** here in **step 3** means that the state is **not repeated** and **does not violate the rules** of the problem.

☆ **The decision of which state to pop or where to add the children usually determines the search algorithm used.**

☆ **For simplicity, a node can be represented as a list containing the state and its parent.**

We can now write the formal uninformed search algorithm in "search.pl" as follows:

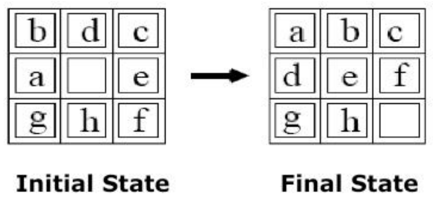
```
search(Open, Closed, Goal):-
    getState(Open, [CurrentState,Parent], _),     % Step 1
    CurrentState = Goal, !,      % Step 2
    write("Search is complete!"), nl,
    printSolution([CurrentState,Parent], Closed).

search(Open, Closed, Goal):-
    getState(Open, CurrentNode, TmpOpen),
    getAllValidChildren(CurrentNode,TmpOpen,Closed,Children), % Step3
    addChildren(Children, TmpOpen, NewOpen),     % Step 4
    append(Closed, [CurrentNode], NewClosed),     % Step 5.1
    search(NewOpen, NewClosed, Goal).          % Step 5.2


% Implementation of step 3 to get the next states
getAllValidChildren(Node, Open, Closed, Children):-
    findall(Next, getNextState(Node, Open, Closed, Next), Children).
```

```
getNextState([State,_], Open, Closed, [Next,State]):-
    move(State, Next),
    not(member([Next,_], Open)),
    not(member([Next,_], Closed)),
    isOkay(Next).


% Implementation of getState and addChildren determine the search
alg.

% BFS
getState([CurrentNode|Rest], CurrentNode, Rest).

addChildren(Children, Open, NewOpen):-
     append(Open, Children, NewOpen).


% Implementation of printSolution to print the actual solution path
printSolution([State, null],_):-
     write(State), nl.

printSolution([State, Parent], Closed):-
     member([Parent, GrandParent], Closed),
     printSolution([Parent, GrandParent], Closed),
     write(State), nl.

% How can we implement DFS, DLS and UCS?
```

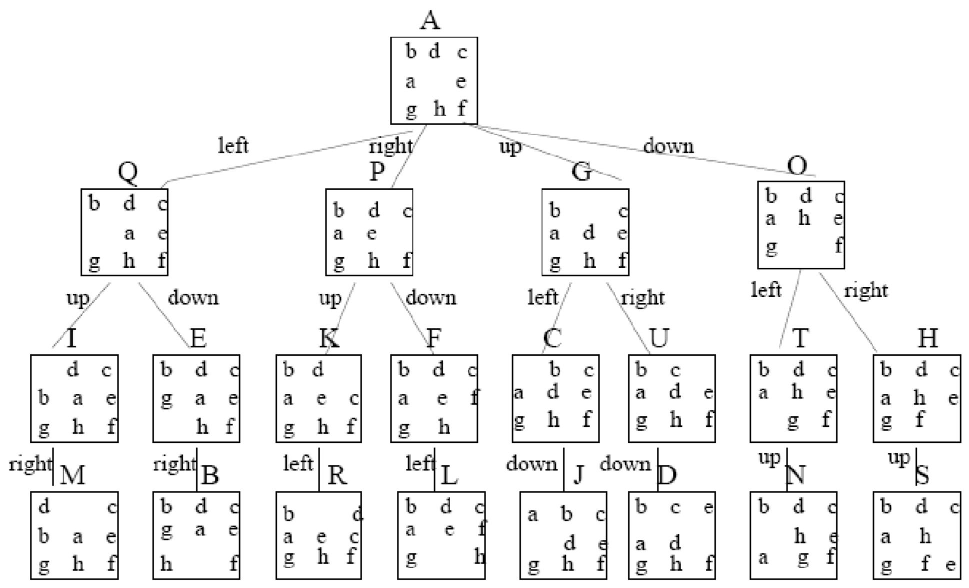
Notice that sometimes we do not know what a goal state looks like exactly but we know some criteria that fit a goal state, in that case we can implement the rule **"isGoal(CurrentState)"** and use it in **step 2** to check.

The 8-Puzzle Problem:

Given a 3×3 board with 8 tiles (every tile has one letter from a to h) and one empty space. The objective is to place the letters on the tiles to match the final configuration using the empty space. We can slide a tile into the adjacent empty space in one of four directions (left, right, up and down).



How can we use and implement search to solve this problem?



Implementation:

We can use the same code from the search module and just determine the state representation and implement the problem-specific predicates "move" and "isOkay".

1. **State Representation:**

   We can represent the state as a list that contains 9 elements. Each element represents a tile in the 8-puzzle. We can use any symbol, say #, to define the empty tile.

   For example the initial and goal states of 8-puzzle may be as follows:

   Initial: [b, d, c, a, #, e, g, h, f]

   Goal: [a, b, c, d, e, f, g, h, #]

2. **Moves:**

```
move(State, Next):-
     left(State, Next); right(State, Next);
     up(State, Next); down(State, Next).

left(State, Next):-
     nth0(EmptyTileIndex, State, #),
     not(0 is EmptyTileIndex mod 3),
     NewIndex is EmptyTileIndex - 1,
     nth0(NewIndex, State, Element),
     % Swap
     substitute(#, State, x, TmpList1),
     substitute(Element, TmpList1, #, TmpList2),
     substitute(x, TmpList2, Element, Next).

right(State, Next):-
     nth0(EmptyTileIndex, State, #),
     not(2 is EmptyTileIndex mod 3),
     NewIndex is EmptyTileIndex + 1,
     nth0(NewIndex, State, Element),
     % Swap
     substitute(#, State, x, TmpList1),
     substitute(Element, TmpList1, #, TmpList2),
     substitute(x, TmpList2, Element, Next).

up(State, Next):-
     nth0(EmptyTileIndex, State, #),
     EmptyTileIndex > 2,
     NewIndex is EmptyTileIndex - 3,
     nth0(NewIndex, State, Element),
     % Swap
     substitute(#, State, x, TmpList1),
     substitute(Element, TmpList1, #, TmpList2),
     substitute(x, TmpList2, Element, Next).

down(State, Next):-
     nth0(EmptyTileIndex, State, #),
     EmptyTileIndex < 6,
     NewIndex is EmptyTileIndex + 3,
     nth0(NewIndex, State, Element),
     % Swap
     substitute(#, State, x, TmpList1),
     substitute(Element, TmpList1, #, TmpList2),
     substitute(x, TmpList2, Element, Next).

substitute(Element, [Element|T], NewElement, [NewElement|T]):- !.
substitute(Element, [H|T], NewElement, [H|NewT]):-
     substitute(Element, T, NewElement, NewT).
```

3. Rules:

```
isOkay(_):- true.    % This problem has no rules
```

Run using the query:

```
?- search([[[b,d,c,a,#,e,g,h,f],null]], [],  [a,b,c,d,e,f,g,h,#]).
```