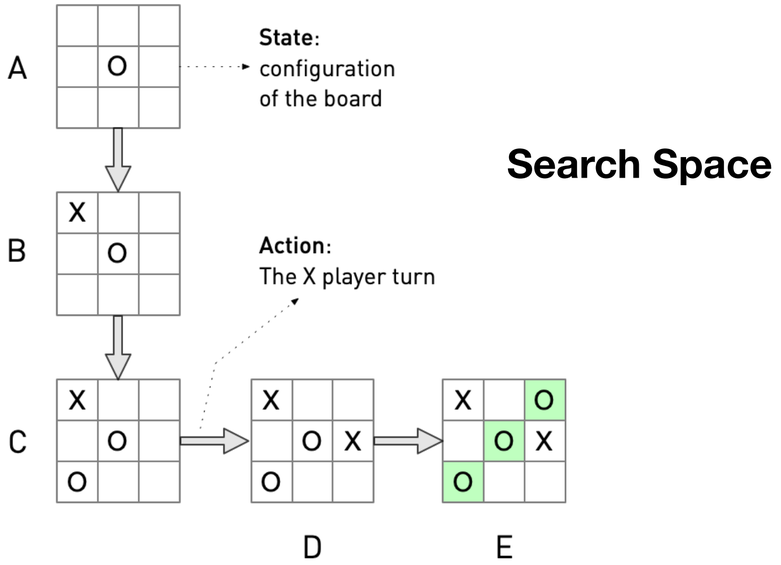
Search - Part I

AIMA Sections 3.1 - 3.4

Artificial Intelligence - Manuel Pita

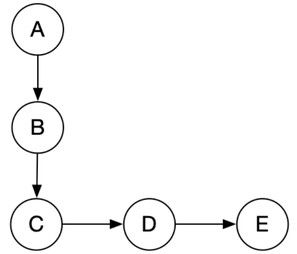
What is Search in AI?

- We need to **formalise** the problem first.
- The problem domain is represented as **states (nodes)** and **actions (links)**
- **Formalisation** of the **states** is crucial
- Actions make a state **transition** to another state
- From an **initial state** to a **goal state** via a **action sequences**
- We need to **interconnect** the entire **search space**



Search Space

Search Graph

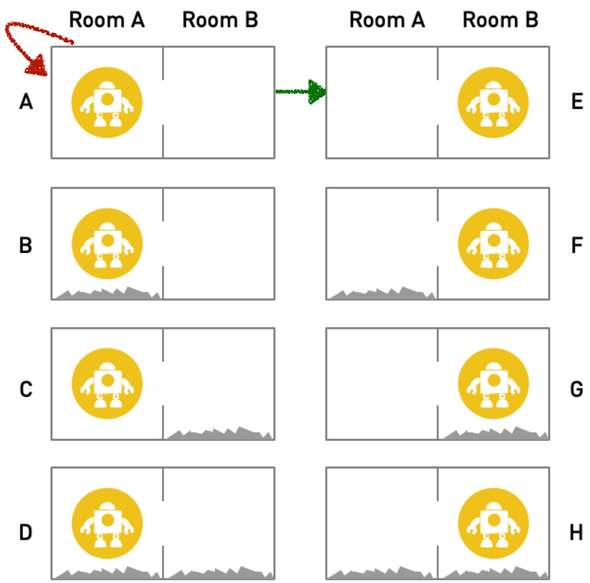


The Cleaning Robot

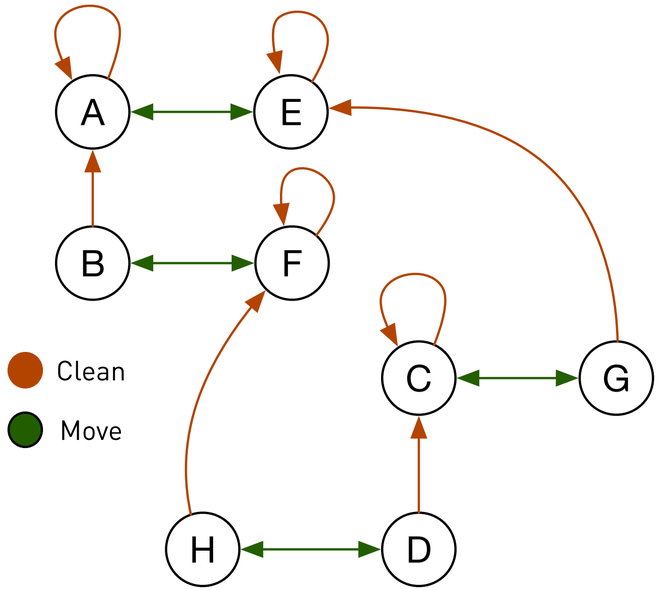
- **Two rooms connected** by an open door
- One **simple robot**:
  - It can **move** between rooms
  - It can **clean** the room where it is
  - It does not do **both actions** in the same instruction
- How to put this in a **computationally friendly** representation?

The Cleaning Robot

States



Search Graph



Adjacency List

```
g ={
(A,A),
(A,E),
(B,A),
(B,F),
(C,C),
(C,G),
(D,C),
(D,H),
…
}
```

Shorter Equivalent of adjacency List

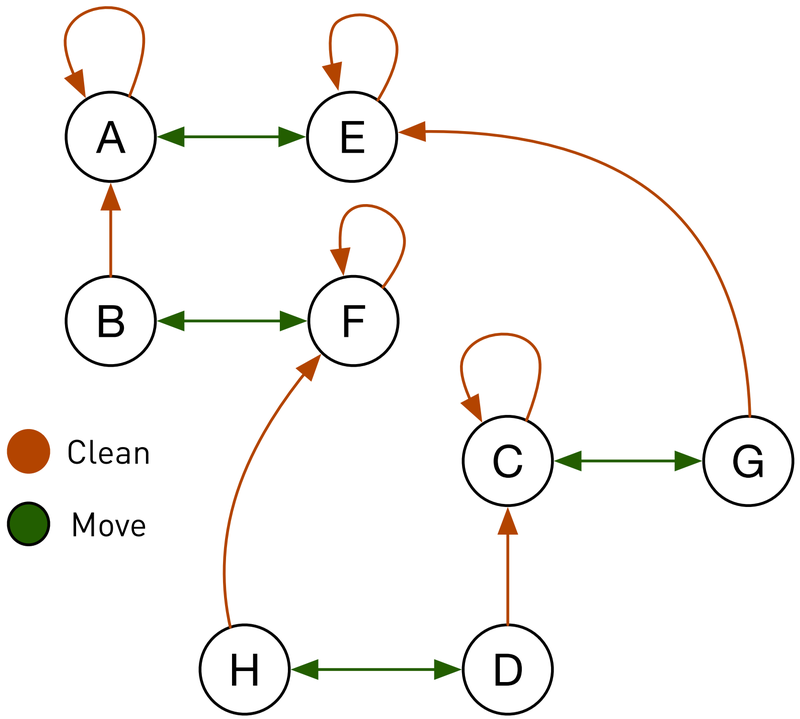
```
g = {
A: {A,E},
B: {A,F},
C: {C,G},
D: {C,H},
…
}
```

Robot Actions:

1) **Move**
2) **Clean**

The Cleaning Robot

Search Graph



What action sequence to go from D to A?

(D) Clean (C) : Move (G) : Clean (E) : Move (A)

Any other option?

(D) Move (H) : Clean (F) : Move (B) : Clean (A)

Notice we can do all this using the simpler **graph** representation instead of the explicit search space

Basic Search Algorithm

```
[1] g #search graph as adjacency list
[2] init #initial node from where we begin searching
[3] goal #destination node we want to find
[4] q = [init] #queue, data structure we use to traverse de search graph

[5] while q: #this is valid in python: while q is not empty
  [6] h = Head(q) #remember this is always the first element
  [7] r = Rest(q) #remember this is always a list of remaining elements
  [8] if h == goal:
     [9]  exit 😅
  [10] else:
     [11] e = Expand(h) #get nodes directly reachable from h (list)
     [12] q = Combine(e, r) #basically q contains the elements of e and r
```

What about loops?

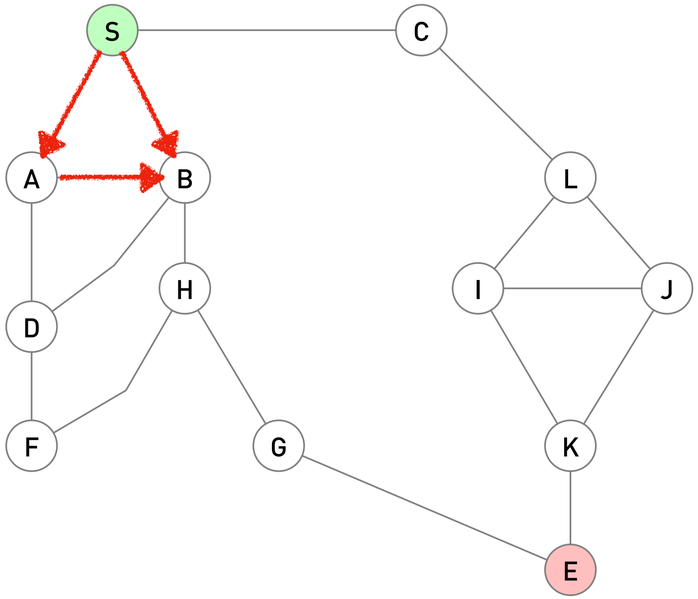
```
[1] g #search graph as adjacency list
[2] init #initial node from where we begin searching
[3] goal #destination node we want to find
[4] q = [init] #queue, data structure we use to traverse de search graph
[5] v = [ ]
[6] while q: #this is valid in python: while q is not empty
  [7] h = Head(q) #remember this is always the first element
  [8] r = Rest(q) #remember this is always a list of remaining elements
    [9] if h == goal:
       [10] exit 😅
    [11] else:
       [12] v.append(h)
       [13] e = Expand(h) - v #get nodes directly reachable from h
       [14] q = Combine(e, r) #put all past non visited nodes and future
```

Limitations of the *visited nodes* approach

```
g #search graph as adjacency list
init #initial node from where we begin searching
goal #destination node we want to find
q = [init] #queue, data structure we use to traverse de search graph
v = [ ]

while q: #this is valid in python: while q is not empty
    h = Head(q) #remember this is always the first element
    r = Rest(q) #remember this is always a list of remaining elements
    if h == goal:
       exit 😅
    else:
       v.append(h)
       e = Expand(h) - v #get nodes directly reachable from h
       q = Combine(e, r) #put all past non visited nodes and future
```

Look at this search graph



Take 5 minutes to think carefully about this.
HINT: our q could store more than the list of nodes to visit

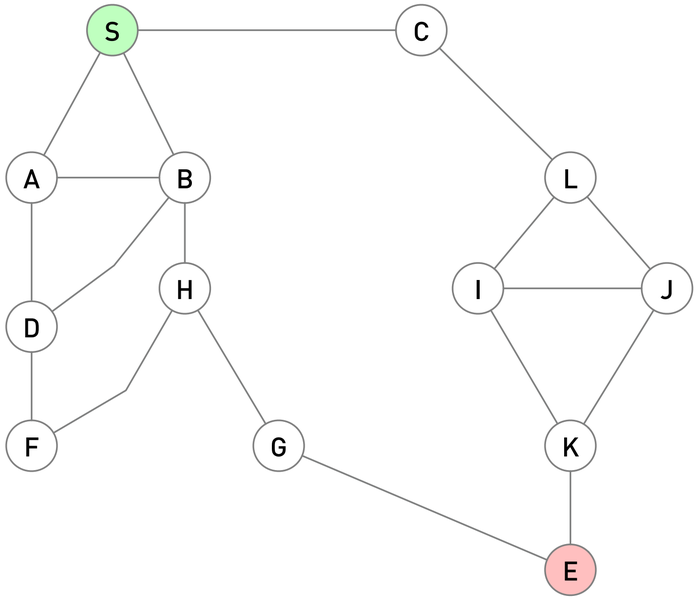
What about the different paths that go e.g. through B?

Keeping open paths in the queue

```
g #search graph as adjacency list
init #initial node from where we begin searching
goal #destination node we want to find
q = [(init, [init])] #keep next node to visit, and a list of the open path it is on
while q: #this is valid in python: while q is not empty
    h = Head(q) #remember this is always the first element
    r = Rest(q) #remember this is always a list of remaining elements
    if h[0] == goal: #we changed structure of q so the node is now on h[0]
       exit 😅
    else:
       e = Expand*(h) #get nodes directly reachable from h including the path
       q = Combine(e,r) #put all past non visited nodes and future
```

```
q = [(S, [S])]
h = (S, [S]), r = [ ]
h[0] is not the goal state
e = [(A, [S,A]), (B, [S,B]), (C, [S,C])]
q = [(A, [S,A]), (B, [S,B]), (C, [S,C])]
—
h = (A, [S,A]), r = [(B, [S,B]), (C, [S,C])]
h[0] is not the goal state
e = [~~(S, [S,A,S])~~, (B, [S,A,B]), (D, [S,A,D])]
…
```

But we have a problem!



What about the different paths that go through B? Or D?

Depth First and Breadth First

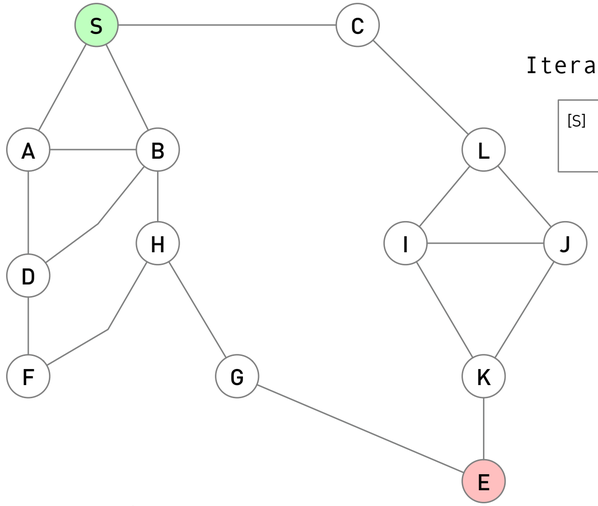
```
g #search graph as adjacency list
init #initial node from where we begin searching
goal #destination node we want to find
q = [{init: [init]}] #keep next node to visit, and a list of the open path it is on


while q: #this is valid in python: while q is not empty
    h = Head(q) #remember this is always the first element
    r = Rest(q) #remember this is always a list of remaining elements
    if h[0] == goal:
       exit 😅
    else:
       e = Expand*(h) #get nodes directly reachable from h
       q = Combine(e,r) #put all past non visited nodes and future
```

q = e + r : DEPTH FIRST

q = r + e : BREADTH FIRST

Depth-First Search: Example



Iteration 1 | Iteration 2 | Iteration 3 | Iteration 4 | Iteration 5 | Iteration 6 | Iteration 7 | Iteration 8

Iteration 1	Iteration 2	Iteration 3	Iteration 4	Iteration 5	Iteration 6	Iteration 7	Iteration 8
[S] S	[S:A] A	[S:A:B] B	[S:A:B:D] D	[S:A:B:D:F] F	[S:A:B:D:F:H] H	[S:A:B:D:F:H:G] G	[S:A:B:D:F:H:G:E] E
	[S:B]	[S:A:D]	[S:A:B:H]	~~[S:A:B:D:A]~~	[S:A:B:H]	[S:A:B:H]	[S:A:B:H]
	[S:C]	~~[S:A:S]~~	~~[S:A:B:A]~~	~~[S:A:B:D:B]~~	[S:A:D]	[S:A:D]	[S:A:D]
		[S:B]	~~[S:A:B:S]~~	[S:A:B:H]	[S:B]	[S:B]	[S:B]
		[S:C]	[S:A:D]	[S:A:D]	[S:C]	[S:C]	[S:C]
			[S:B]	[S:B]			
			[S:C]	[S:C]			

Open Path

h

[X] X — Node is current **HEAD** element (h)

[X...] (...) [X...] — Node(s) is (are) in current **REST** LIST (r)

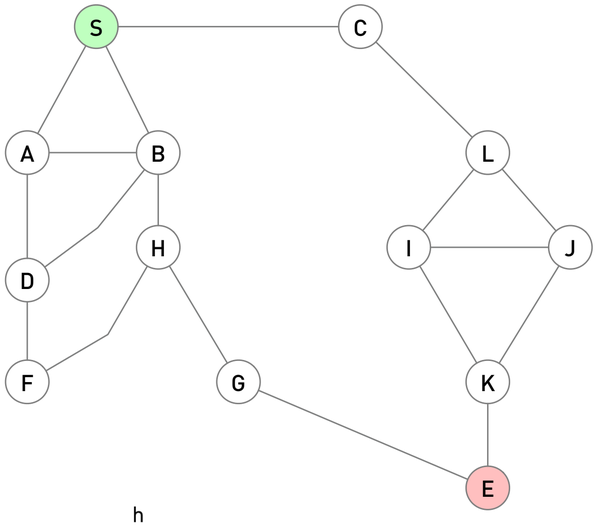
Node comes from EXPAND in previous iteration

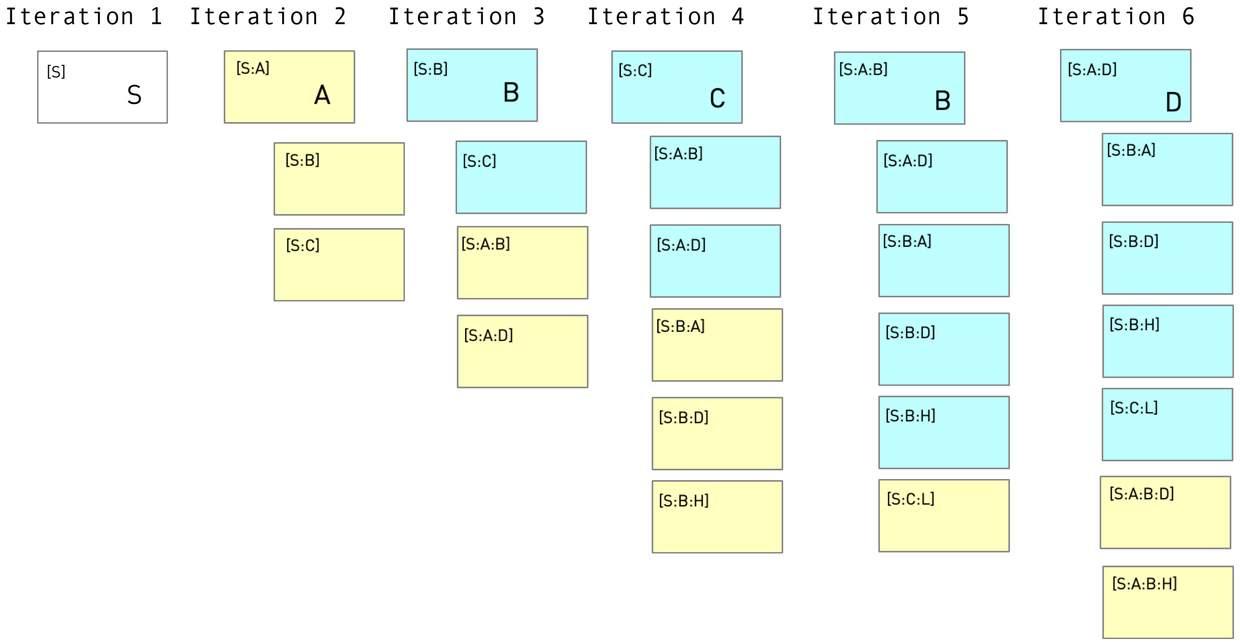
Node comes from REST in previous iteration

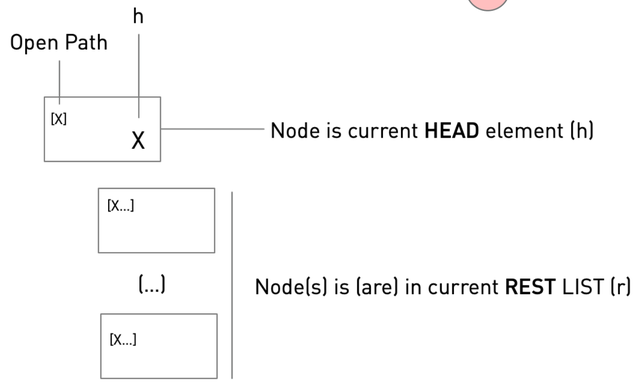
From now on we don't show the loops anymore

We found the goal state!
But notice that q is not empty. This means we can keep finding other paths

Breadth-First Search: Example







Now you have the knowledge and practise to continue on your own.

What happens to the lengths of paths in this algorithm?

Node comes from EXPAND in previous iteration

Node comes from REST in previous iteration

Differences Between BFS and DFS

Breadth-First Search (BFS)	Depth-First Search (DFS)
QUEUE data structure: First In First Out	STACK data structure: Last In First Out (LIFO)
Finds Shortest Path in terms of number of actions on unweighted graphs	May traverse a lot of nodes to find the node corresponding to the goal state
Best for situation in which we want to get to the goal with smallest number of actions	Best to use when we want to look ahead into the future as sequences of actions
Considers all sibling nodes before children nodes, which makes it not so good to decision making in tree structures like we do e.g. in games	Better for games and puzzles type problems. We make a choice, then explore all its consequences. And if the choice leads to win situation, we stop.
In general BFS has bigger memory requirements which may become problematic as the search graph becomes more complex	Depending on the nature of the search graph memory requirements are significantly less tan BFS, we keep in memory the chain of expansions of a currently explored path