Search (Basics)

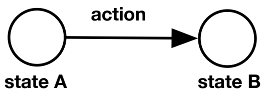
Agents that work using search strategies must rely on our ability to formalise the problem in very specific ways. The most important thing is to be able to **represent** the problem with a **search space.**

A search space (SS) is a network made of **states**, where we can transition from one state to another via an **action.**

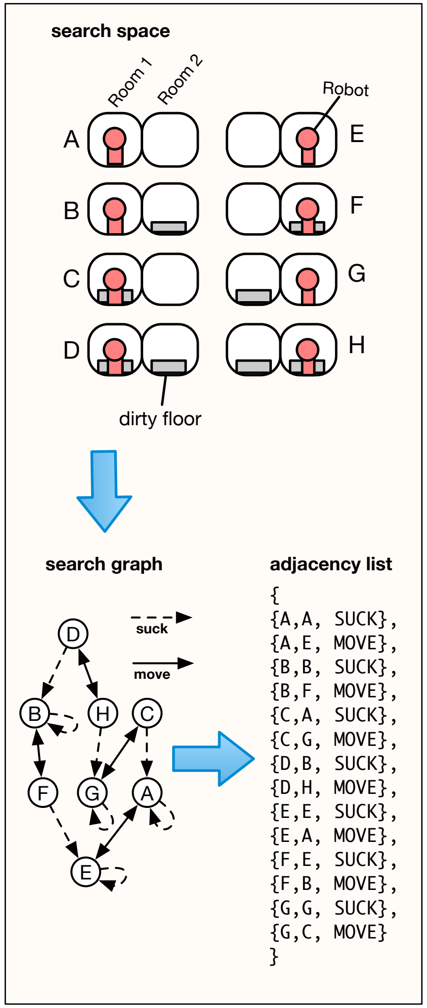
The states in a SS all have the same form but vary in terms of content. For example, the game of Chess can be represented with a SS where each state is the game board with the arrangement of the pieces each player has at a given moment. When one player takes a turn the game changes to another state but the new state is still a board and a configuration of pieces.

The states in a SS capture **information** that is **relevant** about our domain. If we have a robot that keeps a set of rooms clean, it does not need to know the colour of the chairs in the rooms. It needs information about the space where it will move, and to determine whether a room is dirty.

On a first round **we must define a SS completely** and justify all design choices that are made. The idea is to keep the SS representation minimal while maximising its information content. Later we prepare the SS to be used in search algorithms by representing it as a search graph, in **adjacency list** representation



The automatic robot cleaner



adjacency list

```
{
{A,A, SUCK},
{A,E, MOVE},
{B,B, SUCK},
{B,F, MOVE},
{C,A, SUCK},
{C,G, MOVE},
{D,B, SUCK},
{D,H, MOVE},
{E,E, SUCK},
{E,A, MOVE},
{F,E, SUCK},
{F,B, MOVE},
{G,G, SUCK},
{G,C, MOVE}
}
```

Search (Backbone)

Any search algorithm in AI needs some raw material to get started. Universally, **every form of search needs a representation of the search space**. Otherwise there is nowhere to search for anything at all. For us here that means we need to have the adjacency list that captures de SS. In the example on the left, we have that the adjacency list contains items (tuples) where each describes a starting state, and end state, and what action takes you there.

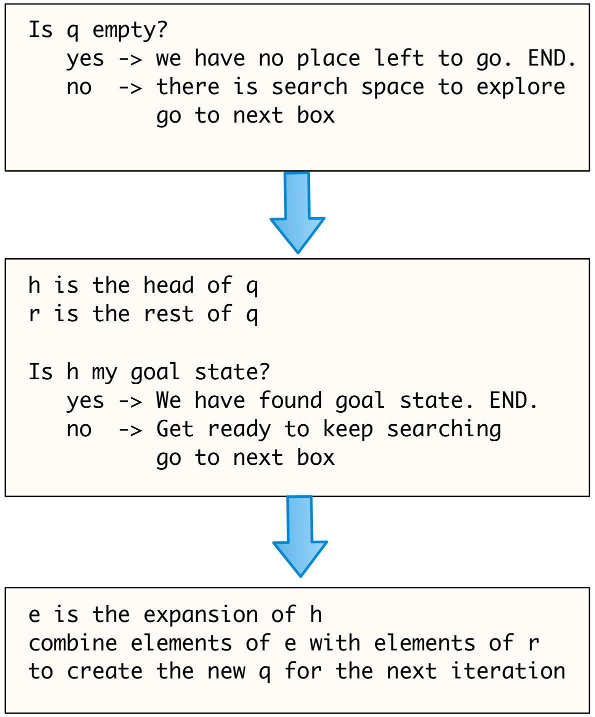
Next we initialise a variable < q > which is the central variable of the algorithm. It is a **list**, that initially contains the initial state from which we start searching. This variable q tells the algorithm where to search.

Next we obviously need to tell the algorithm what we are looking for, the goal state g.

We discussed in class an add-on module for the algorithm that checks if a node has been visited and if it has, it will not appear in the updated q. This is important to avoid loops in some types of search.

For **informed search** you need to add to your adjacency list the cost for every action, and must also supply a **heuristic** table (or function) that estimates cost from any node to the goal state. The heuristic must be admissible if we want to guarantee A* finds shortest path. This means Heuristic must never overestimate real cost. A dominant heuristic is closest to real costs and ensures A* works faster.

Backbone Search Algorithm



```
Is q empty?
   yes -> we have no place left to go. END.
   no  -> there is search space to explore
          go to next box
```

```
h is the head of q
r is the rest of q

Is h my goal state?
   yes -> We have found goal state. END.
   no  -> Get ready to keep searching
          go to next box
```

```
e is the expansion of h
combine elements of e with elements of r
to create the new q for the next iteration
```

Uninformed Search

```
Depth First Search:   q = [e]+[r]
Breadth First Search: q = [r]+[q]
```

Informed Search (A*)

```
Pick the next node for path with
lowest value of

f(node) = cost so far + H(node)

where H is Heuristic
```