Lecture 03

Problem Solving Search

Search, Planning and Graphs

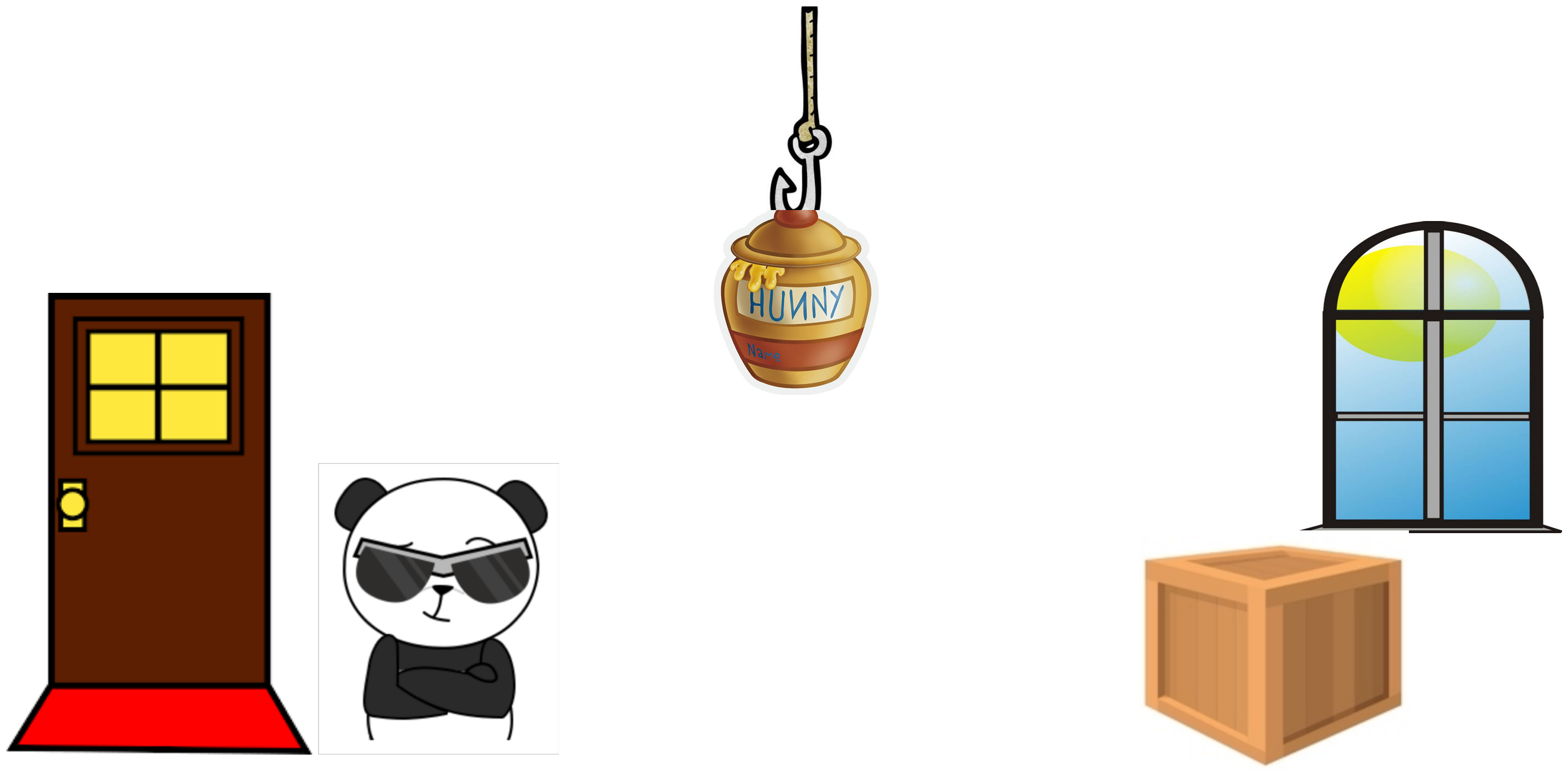
Jeremy Pitt
Department of Electrical & Electronic Engineering
Imperial College London, UK
Email: j.pitt _at_ imperial _dot_ ac _dot_ uk
Phone: (int) 46318
Office: 1010 EEE

Aims and Objectives

- Aims
  - To consider how to formulate problem-solving as a state space, and how to represent the state space as a graph
  - To introduce the General Graph Search Engine, a Prolog program for searching a graph to find a solution
  - To provide a basic introduction to graph theory

- Objectives
  - To understand AI problem-solving (and planning) as searching for paths in a graph
  - To understand the GGSE as a platform for different graph search algorithms

Example: Bear-Box-Honey Problem



Example: Bear-Box-Honey Problem

- Problem description
  - There is a room with a door and window. Both are locked.
  - In the middle of the room there is a hook. From the hook hangs a pot of honey.
  - By the window there is a box. It looks sturdy enough to take a bear's weight, but not so heavy that it cannot be moved by a sufficiently hefty beast, say a bear.
  - By the door there is a bear. The bear is hungry. The bear would not be hungry if it could eat the honey, but the honey is out of reach. If only the bear had something to stand on ...
- Problem statement
  - How does the bear get the honey?

Formulating a Solution (1)

- Question 1: What is the **state**? What information needs to be represented
  - Where's the bear?
    - A location in the room: by the window, in the middle, by the door
  - Where's the box?
    - A location in the room: by the window, in the middle, by the door
  - Is the bear on the box?
    - boolean
  - Has the bear got he honey?
    - boolean

- How do we represent this state? In Prolog? As a 4-tuple
  - (location-of-bear, location-of-box, bear-on-box, bear-has-honey)
  - For example, start state (**initial state**): (door, window, false, false)
  - End state (**goal state**): (_, _, _, true)

Formulating a Solution (2)

- Question 2: What **actions** can be done (and by whom)? The bear can
  - move from one location to another (provided . . .)
  - push the box from one location to another (provided . . .)
  - climb on the box (provided . . .)
  - grab the honey (provided . . .)

- Question 3: what **effects** do actions have on the state?
  - (Bear, Box, OnBox, Has) $\overset{move}{\Rightarrow}$ (Bear', Box, OnBox, Has), if connected(Bear, Bear')
  - (Bear, Box, OnBox, Has) $\overset{push}{\Rightarrow}$ (Bear', Box', OnBox, Has), if Bear$=$ Box $\wedge$ connected(Bear, Bear') $\wedge$ Bear'$=$ Box'
  - (Bear, Box, false, Has) $\overset{climb}{\Rightarrow}$ (Bear, Box, true, Has), if Bear$=$ Box
  - (middle, middle, true, false) $\overset{grab}{\Rightarrow}$ (middle, middle, true, true)

Formulating a Solution (3)

- Question4: How do we get from the start state to the goal state?

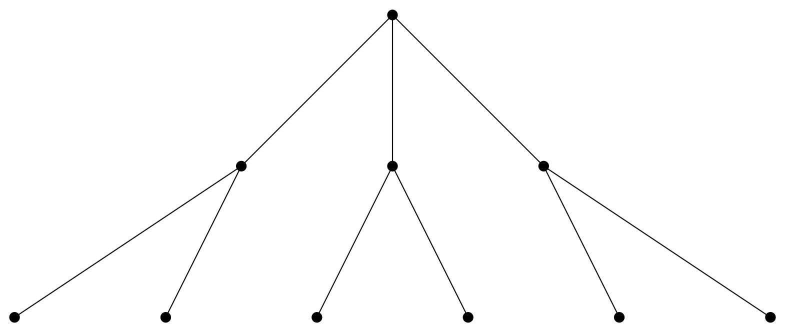
$$\begin{tikzcd}
\boxed{\text{(door, window, false, false)}} \arrow[d, "move"] \\
\boxed{\text{(middle, window, false, false)}} \arrow[d, "move"] \\
\boxed{\text{(window, window, false, false)}} \arrow[d, "push"] \\
\boxed{\text{(middle, middle, false, false)}} \arrow[d, "climb"] \\
\boxed{\text{(middle, middle, true, false)}} \arrow[d, "grab"] \\
\boxed{\text{(middle, middle, true, true)}}
\end{tikzcd}$$

Defining a State Space

- There are 3 x 3 x 2 x 2 = 36 different possible states: this defines a **state space** (or **search space**)
- Actions transform one state into another
- With 36 states, and 4 actions, you could draw it out
- By inspection, you could trace a path from any one state to any other
- If the "one state" is the initial state and the "other state" is a goal state, then the path is a solution, i.e. a sequence of actions which solves the problem
- But it's not quite as simple as that . . .

Search Spaces

- Some terminology
  - Search space, start state, goal state, state transformers, search path, search frontier



- Some things to know about search spaces
  - Get very large very quickly; have loops; have infinite paths; have inaccessible states, have constraints on state transition; have local minima; there may be multiple paths to a goal state; there may be multiple goal states

Search Space as a Graph

- A search space can be formally represented as a **graph** G
- A graph G defines a set of **paths** between nodes
- In theory, if we had an explicit representation of the graph, by inspection, we could 'look up' the path between the node representing the initial state and the node representing the final state
- In practice, we don't have such an explicit representation: all we 'know' is the start state, the goal state, and the state transformation

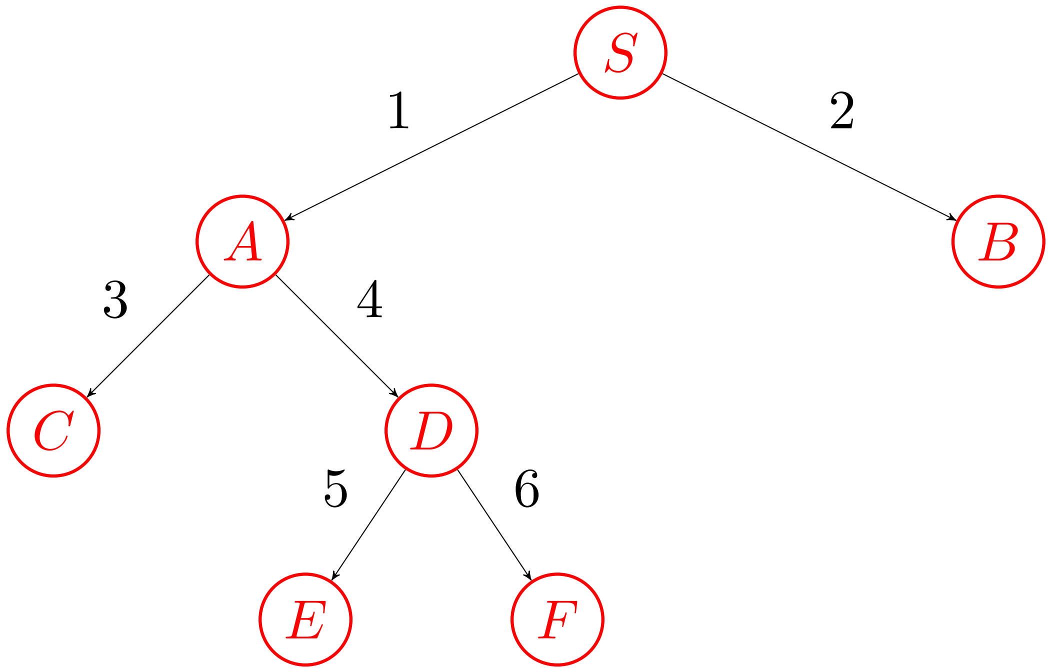
Two Issues

- Firstly
  - We need an implicit representation that is equivalent to the explicit representation
- Secondly
  - Given that implicit representation, we need a method for generating the paths emanating from the start state, and checking if a path so generated ends in a goal state

Issue (1): A Bit of Graph Theory

- An implicit representation that is equivalent to the explicit representation
- Agenda
  - Example
  - Explicit graph representation G
  - Paths
    - Path notation
    - Path definition (inductive)
  - Implicit graph representation G'
    - Operators
    - Inductive definitions
    - Equivalence of G and G'
  - Paths, Revisited, Inductively

Example



Explicit Graph Representation

- $G = \langle N, E, R \rangle$ where
  - N is the set of nodes
  - E is the set of edges
  - R is the **incidence relation** $R : N \times E \times N$

- In the example:
  - $N = \{S, A, B, C, D, E, F\}$
  - $E = \{1, 2, 3, 4, 5, 6\}$
  - $R = \{(S, 1, A), (S, 2, B), (A, 3, C), (A, 4, D), (D, 5, E), (D, 6, F)\}$

Paths

- Path Notation
  - A path is a sequence of one or more nodes enclosed in square brackets
  - E.g. $[S]$, $[A, S]$, $[D, A, S]$, $[E, D, A, S]$, $[F, A, S]$
  - Notes
    - Some sequences of nodes are *not* paths in the graph
    - A singleton node is a path. Otherwise, adjacent nodes in a path have to be connected by an edge in the incidence relation
    - We're going to read these 'back to front', i.e. right to left, so in the path $[D, A, S]$, S is the first node and D is the last node

- A couple of useful functions
  - $\mathit{frontier} : \mathit{path} \rightarrow N$ $\qquad\qquad |: N \times \mathit{path} \rightarrow \mathit{path}$
    - *frontier* returns the last node in a path
    - $|$ (pronounced "cons") prepends (prefixes) a path with a node
    - Instead of writing $|\ (n, p) = \ldots$, we will write $[n \mid p]$

Path Definition

- From the definition of G, we can give an inductive definition of the paths (and sub-paths) in G, starting from a specific node $S \in N$

$$P_G = \bigcup_{i=0}^{\infty} P_i$$

$$P_0 = \{[S]\}$$

$$P_{i+1} = \{[n \mid p] \mid \exists p \in P_i.(\mathit{frontier}(p), e, n) \in R\}$$

- Example

$$\begin{aligned}
P_0 &= \{[S]\} \\
P_1 &= \{[A, S], [B, S]\} \\
P_2 &= \{[C, A, S], [D, A, S], \} \\
P_3 &= \{[E, D, A, S], [F, D, A, S]\} \\
P_4 &= \emptyset \\
&\dots
\end{aligned}$$

- So: $P_G = \bigcup_{i=0}^{\infty} P_i = P_0 \cup P_1 \cup P_2 \cup P_3 \cup \dots = \dots$

Implicit Graph Representation

- Defines a set Op of operators (i.e. state transformers)
  - $Op = \{op_1, op_2, \ldots, op_n\}$
  - Each $op_i \in Op$ is a partial function: $op_i : N \mapsto (N \times E)$
- Define (implicitly) a graph G' by $G' = \langle S, Op \rangle$
- $G' = \langle S, Op \rangle$ defines the same graph as $G = \langle N, E, R \rangle$ provided

$$(n, e, n') \in R \leftrightarrow \exists op_i \in Op . op_i(n) = (n', e)$$

Reconstructing N and R (Inductively)

- The nodes (and edges) of the graph

$$N_G = \bigcup_{i=0}^{\infty} N_i$$
$$N_0 = \{S\}$$
$$N_{i+1} = \{n' \mid \exists n \in N_i . \exists op \in Op . op(n) = (n', e)\}$$

- The incidence relation

$$R_G = \bigcup_{i=0}^{\infty} R_i$$
$$R_0 = \{(S, e, n) \mid \exists op \in Op . op(S) = (n, e)\}$$
$$R_{i+1} = \{(n, e, n') \mid \exists(\_, \_, n) \in R_i . \exists op \in Op . op(n) = (n', e)\}$$

Paths, Revisited, Inductively

- Paths defined by G'

$$P_{G'} = \bigcup_{i=0}^{\infty} P'_i$$

$$P'_0 = \{[S]\}$$

$$P'_{i+1} = \{[n \mid p] \mid \exists p \in P'_i . \exists op \in Op . op(\mathit{frontier}(p)) = (n, e)\}$$

Example

- Suppose $Op = \{l, r\}$ and

$$\begin{array}{ll} l(S) = (A, 1) & r(S) = (B, 2) \\ l(A) = (C, 3) & r(A) = (D, 4) \\ l(D) = (E, 5) & r(D) = (F, 6) \end{array}$$

- No surprises

$$\begin{aligned} P'_0 &= \{[S]\} \\ P'_1 &= \{[A, S], [B, S]\} \\ P'_2 &= \{[C, A, S], [D, A, S]\} \\ P'_3 &= \{[E, D, A, S], [F, D, A, S]\} \\ P'_4 &= \emptyset \\ &\dots \end{aligned}$$

Issue (2): Search

- We need a method for generating the paths emanating from the start state, and checking if a path so generated ends in a goal state
- If we're going automate the process, in Prolog, first we need to formulate the search space of a problem
- Example: farmer-wolf-goat-cabbage problem
  - A farmer, wolf, goat and cabbage are on one side of a river with a boat, and they all want to get to the other side
  - The farmer can transport one of the wolf, goat or cabbage in the boat across the river, or he can go by himself
  - If he leaves the wolf with the goat, the wolf will eat the goat; if he leaves the goat with the cabbage, the goat will eat the cabbage
  - How do they all get across?

Formulating a Search Space

- If we are going to define it as graph, we need:
  - Define the state representation
  - Specify the initial state
  - Specify the goal state(s)
  - Specify the state transformers
- Note: some issues
  - Deciding what goes in the state description and what is left out
  - Deciding what effects of actions are significant and what are not
  - Abstraction: the process of removing unnecessary detail from the representation
  - A 'good' abstraction removes as much detail as possible, while retaining validity and ensuring that actions are as simple as possible to execute
  - Good abstraction and efficient data representation may be the key to the whole exercise

Formulation for FWGC

- State description
  - 4-tuple $= (F, W, G, C)$
    - Correspond to bank farmer/wolf/goat/cabbage are on, respectively
    - Can be instantiated to l (left) or r right
    - No representation of boat – assumed same bank as farmer
- Initial state
  - (l, l, l, l)
- Goal state
  - (r, r, r, r)
- State transformers – general format
  - `statechange(RuleName, OldState, NewState) :-`
    $computation, constraint\_checking.$

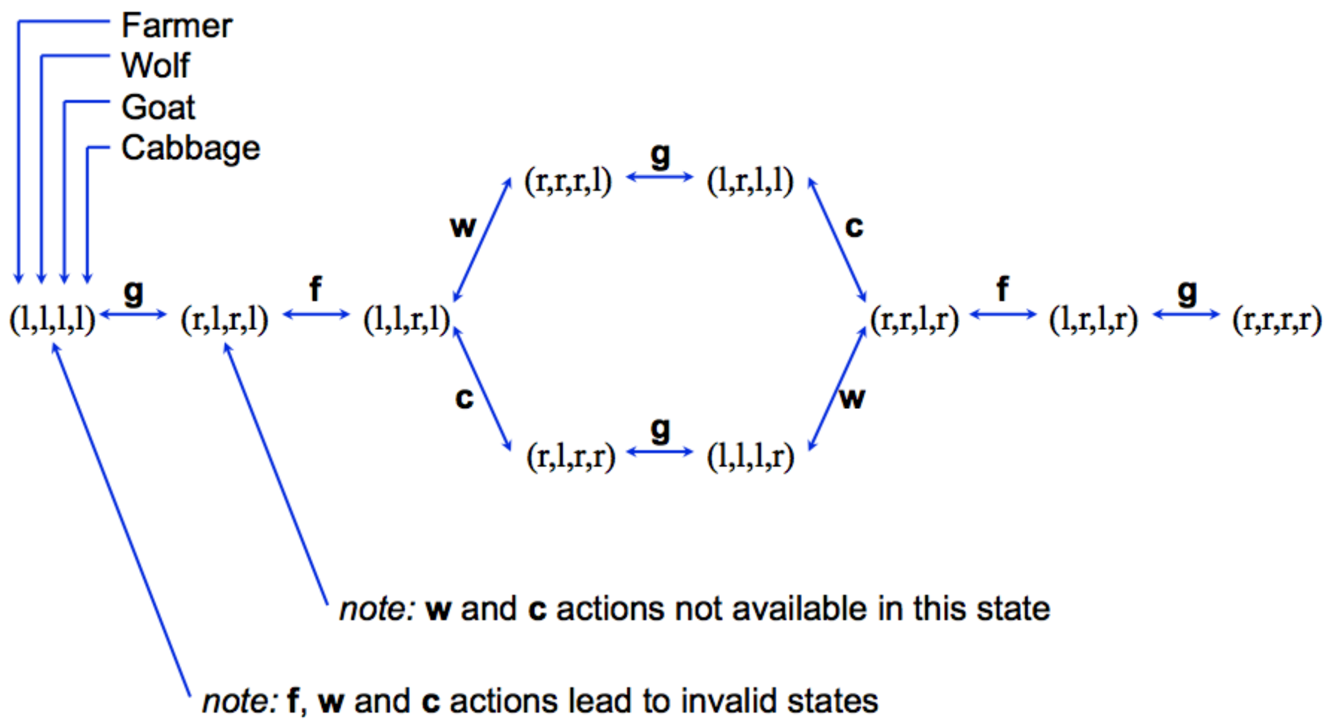
FWGC: State Transformers

- One state change (transformer) for each action

  -
    ```
    state_change( farmer, (B,W,G,C), (O,W,G,C) ) :-
       opposite(B,O), opposite(W,G), opposite(G,C).
    state_change( wolf, (B,B,G,C), (O,O,G,C) ) :-
      opposite(B,O), opposite(G,C).
    state_change( goat, (B,W,B,C), (O,W,O,C) ) :-
       opposite(B,O).
    state_change( cabbage, (B,W,G,B), (O,W,G,O) ) :-
       opposite(B,O), opposite(W,G).

    opposite( l, r ).
    opposite( r, l ).
    ```

FWGC: Search Space



- This is the explicit representation again
- How do we (get Prolog to) generate and search the graph, just from the initial state and the state transformers?

Some Definitions

- **State**: a data structure, representing the state of a problem
- **Node**: a data structure, including at least the state, but possibly other information as well
- **Path**: a sequence of nodes
- **Graph**: a set of paths

An (Informal) Declarative Specification

- A graph G can be searched for a Solution Path SP, if
  - Pick a path P in G, AND
  - Get frontier node N of path P, AND
  - Get problem-state S of node N, AND
  - S is a goal state. [So P *is* a Solution Path SP!]

- A graph G can be searched for a Solution Path SP, if
  - Pick a path P in G, AND
  - Set $OtherPaths \leftarrow G - \{P\}$, AND
  - Get frontier node N of path P, AND
  - Compute the set N' of all the new nodes reachable from N, AND
    - $*$ by applying all the state change rules to N
  - Set $NewPaths \leftarrow \{[n' \mid P] \mid \exists n' \in N'\}$, AND
  - Make a bigger graph G^+ from $NewPaths$ plus $OtherPaths$, AND
  - Graph G^+ can be searched for a Solution Path SP.

Prolog Representation

- A problem-state is represented by a **term**, depending on the formulation of the problem
- A node is represented by a **tuple**, including at least the state
- A path is represented by a **list** (of nodes)
- A graph is represented by a **list** (of paths, so a list of (lists of nodes))

The General Graph Search Engine

- GGSE

  -
    ```
    search( Graph, [Node|Path] ) :-
        choose( [Node|Path], Graph, _ ),
        state_of( Node, State ),
        goal_state( State ).
    search( Graph, SolnPath ) :-
        choose( Path, Graph, OtherPaths ),
        one_step_extensions( Path, NewPaths ),
        add_to_paths( NewPaths, OtherPaths, GraphPlus ),
        search( GraphPlus, SolnPath ).
    ```
- Notes
  - What about `choose` and `add_to_paths`?
  - Different behaviours are generated by different definitions
    $\Rightarrow$ **different algorithms**

Aside

- Now you know why, in the graph theory section:
  - Sequences of nodes were represented in reverse order
  - The | notation for prepending was introduced

- Prolog list representation makes it very easy to access the head of a list
  - When we wanted to get at the frontier node
  - When we wanted to compute the ‘one step extensions’ of a path (i.e. by prepending each of the nodes reachable from its frontier node by applying the state change rules to that node)

Summary

- Qualitative approach to problem-solving, based on a declarative specification of knowledge
  - Specified start state and state transformers
  - This defines a graph
  - A solution is a path in the graph from the start state to a goal state
- Specified a general engine for generating all the paths in a graph, so defined
- We now want specific algorithms for generating and searching through those paths