Search and Planning

COMP3431 Robot Software Architectures
So far ...

- Simple behaviour-based robots with no world model
- Robots that build models of space around them and remember events
- Robots that can use abstract models to answer questions and derive relations
This time ...

- Using abstract representations to plan and solve problems
Search
Search

• Search is fundamental to AI

• We usually deal with problems that have no direct solution

• Must choose between alternatives
State Transition Graph
Problem Formulation

- **Sets of states**: description of one instant in time
- **Initial state**: where are we starting from?
- **Actions**: set of available actions
- **Transition model**: effect of each action
- **Goal test**: have we finished?
- **Path cost**: transitions may have different cost
Search for Solutions

• Expand a node generating parent and child nodes

• Create *frontier* list (or *open* list)

• Search strategy: order in which frontier is expanded
Example

Initial state

After expanding Arad

After expanding Sibiu
Uninformed Search

- Have no additional information beyond problem definition
  - Depth-first
  - Breadth-First
Depth-First Search
Breadth-First Search
def dfs(start):
    frontier = [start]
    while frontier != []:
        first, rest = frontier[0], frontier[1:]
        print("Node ", first.id)
        frontier = first.neighbours + rest
Breadth-First Search

def dfs(start):
    frontier = [start]
    while frontier != []:
        first, rest = frontier[0], frontier[1:]
        print("Node ", first.id)
        frontier = rest + first.neighbours
Iterative Deepening

- Depth-first to a limited depth
- Increase limit if goal not reached
Informed (Heuristic) Search

• Use a heuristic (informed guess) to estimate cost to goal

• Greedy search expands nodes with lowest cost first
Greedy Search
A* Search

- Minimises total cost

\[ f(n) = g(n) + h(n) \]

- \( g \) is the cost to reach the node \( n \), known exactly

- \( h \) heuristic that guesses the cost to go from \( n \) to the goal

- \( h \) is *admissible* if it never overestimates the cost
A* Example

```
  Arad
   Sibiu
    Arad  Fagaras  Cradea
    648=280+368  671=291+380
    Sibiu  Bucharest  Craiova
    591=338+253  450=450+0  526=366+160
    Timisoara  Pitesti  Sibiu
    447=118+329  417=317+100  553=300+253
    Zerind
   449=75+374
```

History of A*

• Invented for Shakey:


• Shakey also gave rise to a planning framework that is still used
Acknowledgments

• Examples from:

Graph Search in Prolog
Graph Representation

A graph may be represented by a set of edge predicates and a list of vertices.

edge(1, 5).
edge(1, 7).
edge(2, 1).
edge(2, 7).
edge(3, 1).
edge(3, 6).
edge(4, 3).
edge(4, 5).
edge(5, 8).
edge(6, 4).
edge(6, 5).
edge(7, 5).
edge(8, 6).
edge(8, 7).

vertices([1, 2, 3, 4, 5, 6, 7, 8]).
Finding a path

- Write a program to find a path from one node to another.
- Must avoid cycles (i.e. going around in circle).
- A template for the clause is:

  
  path(Start, Finish, Visited, Path).

  Start is the name of the starting node
  Finish is the name of the finishing node
  Visited is the list of nodes already visited.
  Path is the list of nodes on the path, including Start and Finish.
The path program

• The search for a path terminates when we have nowhere to go.
  
  path(Node, Node, _, [Node]).

• A path from Start to Finish starts with a node, X, connected to Start followed by a path from X to Finish.
  
  path(Start, Finish, Visited, [Start | Path]) :-
    edge(Start, X),
    not(member(X, Visited)),
    path(X, Finish, [X | Visited], Path).

  member(X, [X|_]).
  member(X, [_|Y]) :- member(X, Y).
Hamiltonian Paths

A Hamiltonian path is a path which spans the entire graph without any repetition of nodes in the path.

\[
\text{hamiltonian}(P) :- \\
\text{vertices}(V), \\
\text{member}(S, V), \\
\text{path}(S, \_, [S], P), \\
\text{subset}(V, P).
\]

:- \text{hamiltonian}(P).

\[
P = [2, 1, 7, 5, 8, 6, 4, 3] \\
P = [2, 7, 5, 8, 6, 4, 3, 1]
\]
Missionaries and Cannibals

• There are three missionaries and three cannibals on the left bank of a river.

• They wish to cross over to the right bank using a boat that can only carry two at a time.

• The number of cannibals on either bank must never exceed the number of missionaries on the same bank, otherwise the missionaries will become the cannibals' dinner!

• Plan a sequence of crossings that will take everyone safely across.
Representing the state

- A state is one "snapshot" in time.
- For this problem, the only information we need to fully characterise the state is:
  - the number of missionaries on the left bank,
  - the number of cannibals on the left bank,
  - the side the boat is on.
- All other information can be deduced from these three items.
- In Prolog, the state can be represented by a 3-arity term,
  - state(Missionaries, Cannibals, Side)
Representing the Solution

• The solution consists of a list of moves, e.g.

\[ \text{move}(1, 1, \text{right}), \text{move}(2, 0, \text{left}) \]

which we will take to mean that 1 missionary and 1 cannibal moved to the right bank, then 2 missionaries moved to the left bank.

• Like the graph search problem, we must avoid returning to a state we have visited before.

• The visited list will have the form:

\[ \text{[MostRecent State} | \text{ListOfPreviousStates]} \]
Overview of Solution

- We follow a simple graph search procedure:
  - Start from an initial state
  - Find a neighbouring state
  - Check that the new state has not been visited before
  - Find a path from the neighbour to the goal.
- The search terminates when we have found the state:
  
  \text{state}(0, 0, \text{right}).
Top-level Prolog Code

mandc(CurrentState, Visited, Path)

mandc(state(0, 0, right), _, []).  
mandc(CurrentState, Visited, [Move | RestOfMoves]) :-  
    newstate(CurrentState, NextState),  
    not(member(NextState, Visited)),  
    make_move(CurrentState, NextState, Move),  
    mandc(NextState, [NextState | Visited], RestOfMoves).

make_move(state(M1, C1, left), state(M2, C2, right), move(M, C, right)) :-  
    M is M1 - M2,  
    C is C1 - C2.
make_move(state(M1, C1, right), state(M2, C2, left), move(M, C, left)) :-  
    M is M2 - M1,  
    C is C2 - C1.
Possible Moves

- A move is characterised by the number of missionaries and the number of cannibals taken in the boat at one time.

- Since the boat can carry no more than two people at once, the only possible combinations are:
  
  - carry(2, 0).
  - carry(1, 0).
  - carry(1, 1).
  - carry(0, 1).
  - carry(0, 2).

Where carry(M, C) means the boat will carry M, missionaries and C, cannibals on one trip.
Feasible Moves

• Once we have found a possible move, we have to confirm that it is feasible.

• I.e. it is not feasible to move more missionaries or more cannibals than are present on one bank.

• When the state is state(M1, C1, left) and we try carry(M, C) then

\[ M \leq M1 \text{ and } C \leq C1 \]

must be true.

• When the state is state(M1, C1, right) and we try carry(M, C) then

\[ M + M1 \leq 3 \text{ and } C + C1 \leq 3 \]

must be true.
Legal Moves

- Once we have found a feasible move, we must check that is legal.

- I.e. no missionaries must be eaten.

  ```prolog
  legal(X, X) :- !.
  legal(3, X) :- !.
  legal(0, X).
  ```

- The only safe combinations are when there are equal numbers of missionaries and cannibals or all the missionaries are on one side.
Generating the next state

\[
\text{newstate}(\text{state}(M_1, C_1, \text{left}), \text{state}(M_2, C_2, \text{right})) :-
\begin{align*}
\text{carry}(M, C), \\
M & \leq M_1, \\
C & \leq C_1, \\
M_2 & \text{ is } M_1 - M, \\
C_2 & \text{ is } C_1 - C, \\
\text{legal}(M_2, C_2).
\end{align*}
\]

\[
\text{newstate}(\text{state}(M_1, C_1, \text{right}), \text{state}(M_2, C_2, \text{left})) :-
\begin{align*}
\text{carry}(M, C), \\
M_2 & \text{ is } M_1 + M, \\
C_2 & \text{ is } C_1 + C, \\
M_2 & \leq 3, \\
C_2 & \leq 3, \\
\text{legal}(M_2, C_2).
\end{align*}
\]
Exercise - Constraint Satisfaction

In five houses, each with a different colour, live five people of different nationalities, each of whom prefers a different brand of chocolates, a different drink, and a different pet. Given the following facts, answer the questions: “Where does the zebra live, and in which house do they drink water?”

- The Englishman lives in the red house.
- The Spaniard owns the dog.
- The Norwegian lives in the first house on the left.
- The green house is immediately to the right of the ivory house.
- The man who eats Cadburys lives in the house next to the man with the fox.
- Kit Kats are eaten in the yellow house.
- The Norwegian lives next to the blue house.
- The Smarties eater owns snails.
- The Snickers eater drinks orange juice.
- The Ukrainian drinks tea.
- The Japanese eats Milky Ways.
- Kit Kats are eaten in a house next to the house where the horse is kept.
- Coffee is drunk in the green house.
- Milk is drunk in the middle house.