Search

CS 475

March 10, 2003

Search is an important part of our problem solving process. Practically, we search for a solution every time we try to solve a problem. Search is often needed when we do not have a step-by-step algorithm but we know what is a solution. Examples:

- *Travesing salesman* Given n cities, distance between every pair of two cities. A salesman needs to visit these cities, each at least one. Find for him a shortest route through the cities.
- *Knap-sack problem* Given n items, the weight and value of each item, and a knap-sack and its capacity. Find the most valuable way to pack the items into the knap-sack.
- *Navigation path* Find a path connecting the two points on a map for a robot.
- *SLD derivation* Given a goal $?g$, find a SLD derivation for g.

Definition 0.1 *Search is an enumeration of a set of potential partial solutions to a problem so that they can be checked to see uf they truly are solutions, or could lead to a solutions.*

To carry out a search, we need:

- A definition of a potential solution.
- A method of generating the potential solutions (hopefully in a clever way).
- A way to check whether a potential solution is a solution.

1 Graph Searching

Use to present general mechanism of searching. To solve a problem using search, we translate it into a graph searching problem and use the graph searching algorithms to solve it.

Definition 1.1 *A graph consists of a set N of nodes and a set A of ordered pairs of nodes, called arcs.*

Two possible ways to represent a problem as a graph:

- *State-space graph*: each node represents a state of the world and an acr represents changing from one state to another.
- *Problem-space graph*: each node represents a problem to be solved and an arc represents alternate decomposition of the problems.

Example:

- *State-space graph*: finding path for robot – each node is a location. The state of the world is the location of the robot.
- *Problem-space graph*: SLD resolution – each node is a goal. Connection from one node to the other represents that the second one is obtained from the other through a SLD resolution.

Node n_2 is a **neighbor** of n_1 if there is an arc from n_1 to n_2. That is, if $\langle n_1, n_2 \rangle \in A$. An arc may be labeled.

A **path** is a sequence of nodes $\langle n_0, n_1, ..., n_k \rangle$ such that $\langle n_{i+1}, n_i \rangle \in A$.

A **cycle** is a nonempty path such that the end node is the same as the start node. A graph with out cycle is called **directed acyclic graph** or DAG.

Given a set of start nodes and goal nodes, a solution is a path from a start node to a goal node. (See example in the book: Fig. 4.2)

The *forward branching factor* of a node is the number of arcs going out from the node, and he *backward branching factor* of a node is the number of arcs going into form the node.

2 A Generic Searching Algorithm

Given a graph, the set of start nodes, and the set of goal nodes. A path between a start node and a goal node is a solution. Searching algorithms provide us a way to find a solution.

Idea: Incrementally explore paths from start nodes. Maintaining a **frontier** or **fringe** of paths from the start nodes that have been explored.

The algorithm:

```
Input: a graph,
       a set of start nodes,
       Boolean procedure goal(n) that tests if n is a goal node.

frontier := {<s> : s is a start node};

while frontier is not empty:
       select and remove path <n0, . . . , nk> from frontier;
       if goal(nk)
            return <n0, . . . , nk>;
       for every neighbor n of nk
            add <n0, . . . , nk, n> to frontier;
end while
```

2.1 Implementation

The algorithm returns an answer. We can implement it in such a way that when more answers are needed, the implementation will continue.

- Which value is selected from the frontier at each stage defines the search strategy.

- The neighbors defines the graph.
- *is_goal* defines what is a solution.

The implementation: we need following predicates:

- $is\_goal(N)$ – N is a goal
- $neighbours(X, L)$ – the set of neighbors of X is L
- $add\_to\_frontier(LN, F, NF)$ – add the list of nodes LN to the frontier F and create a new frontier NF
- $select(N, F, NF)$ – select a node N from the frontier F and the remaining nodes of the frontier is NF
- $search(F)$ – there exists a path from one element of the frontier to a goal node.

Example 1: The delivery robot.

The graph:

```
neighbours(o103,[ts,l2d3,o109]).
neighbours(ts,[mail]).
neighbours(mail,[]).
neighbours(o109,[o111,o119]).
neighbours(o111,[]).
neighbours(o119,[storage,o123]).
neighbours(storage,[]).
neighbours(o123,[r123,o125]).
neighbours(o125,[]).
neighbours(l2d1,[l3d2,l2d2]).
neighbours(l2d2,[l2d4]).
neighbours(l2d3,[l2d1,l2d4]).
neighbours(l2d4,[o109]).
neighbours(l3d2,[l3d3,l3d1]).
neighbours(l3d1,[l3d3]).
neighbours(l3d3,[]).
neighbours(r123,[]).
```

The goal: $is\_goal(r123)$.

```
search(F) :- select(N, F, F1), is_goal(N).
search(F) :- select(N, F, F1), neighbours(N, NF),
             add_to_frontier(NF, F1, F2),
             search(F2).

select(N,[N|F1],F1).

add_to_frontier(NF, F1, F2):- append(NF, F1, F2).
```

This implementation ignores the paths that lead to the goal. It only says that there exists a path from the frontier to a goal. To find the paths, we need to change the following:

- For each node N in the frontier we must store the path that leads to N.
- When we add a node $N1$ to the frontier because it is a neighbor of $N2$, we add the path lead to $N2$ by extending the path that leads to $N1$ with $N2$. This could be done by representing a path as a list and use the function 'append'.
- We need a second parameter that allows us to store the paths that lead to the goal.
- We need to change the checking of goal in the first clause because element of the frontier is now a path rather than a node. We should check for the last node of being a goal.
- We need to change the way we expand the frontier.

```
search(F,[N]) :- select(N, F, _), is_good_path(N).

search(F,[N|P]) :- select(N, F, F1),
   last_of(N, Node), neighbours(Node, NF), create_path(N, NF, NN),
   add_to_frontier(NN, F1, F2),
   search(F2,P).

select(N,[N|F1],F1).

create_path(N, [], []).
create_path(N, [H|T], [H1|T1]):-
   append(N, [H], H1), create_path(N, T, T1).

is_good_path(P):- last_of(P, N), is_goal(N).

last_of([L], L).
last_of([H|T], L):- length(T) >= 1, last_of(T, L).
```

2.2 Cost

Associated cost to each arc by using atoms of the form

```
cost(o103,ts,8).
cost(o103,o109,12).
....
```

add_to_frontier can be modified so that the cost of each path in the frontier is attached to it as well.

3 Blind Search Strategies

So far, we do not pay attention to the detail of how to select the next node when expand the frontier (the predicates *select* and *add_to_frontier*). The algorithm does not specify how they should be implemented. In our implementation, we use a list to store the frontier and always select the first element of the list. When we add new elements to the frontier, we put it to the end.

Definition. A *search strategy* specifies how *select* and *add_to_frontier* should be implemented.

Definition. A *blind search strategy* is a search strategy that does not take into account where the goal is.