University of Malta
BSc(Hons.) IT Year II

Problem Solving Techniques in Artificial Intelligence

Lecture 6

Agenda Based Search

- Data Abstraction for Search Nodes
- Recovering the Solution Path
- Loop Detection
- Informed Search

Department of Computer Science and AI
1998/9

Behaviour of Agenda Search

```
?- solve(bs,ls,N,Final).
bs
gp
oc
bs
cc
pc
bs
pc
tc
gp
oc
gp
ls

N = 13
Final = ls;
No
```

Data Abstraction

Basic idea: separation between

- Data Structure **Operations**
- Data Structure **Implementation**

Current data structure for search nodes is simply the name of the state.

This suffices as long as we don't want to associate other information with search nodes.

Other information might include

- Node Count
- Parent Information (to recover path)
- Costs associated with reaching this node
- Anything else we might need

Node Data Abstraction: Operations

- **Selectors**
  - **node_state_of(Node,S): the state name of N is S.**
  - **node_parent_of(Node,P): the parent of N is P.**
  - node_count_of(Node,C): the count of Node is C.
  - node_gcost_of(Node,G): the path_cost of Node is G.
  - node_hcost_of(Node,H): the hcost of Node is H.
  - node_fcost_of(Node,F): the fcost of Node is F.
- **Constructors**
  - **node_new(N): create a new node**
  - node_make(S,P,Node): create a new node with state S and parent P.

Abstract Predicate Examples

```
node_new_state(S,N) :-
 node_new(N),
 node_state_of(N,S).

node_new_parent(P,N) :-
 node_new(N),
 node_parent_of(N,P).

node_new_state_parent(S,P,N) :-
 node_new(N),
 node_parent_of(N,P),
 node_state_of(N,S).
```

Node Data Abstraction: Implementations

- **Implementation 1**

```
node_state_of( n(X,_),  X).
node_parent_of( n(_,X),  X).

node_new( n(_,_) ).
```

- **Implementation 2**

```
node_state_of(   [node,X,_],  X).
node_parent_of(  [node,_,X],  X).

node_new( [_,_,_] ).
```

Changes

```
node_display(N) :-
 node_state_of(N,S), write(S), nl.

start_init(S) :-
 abolish(start_state,1),
 asserta(start_state(S)).

goal_init(S) :-
 abolish(goal_state,1),
 asserta(goal_state(S)).

solve(Start,Goal,N,Ans) :-
 start_init(Start),
 goal_init(Goal),
 agenda_init(A),
 agenda_search(0,N,A,Ans).
```

Modifying node_expand

Currently, we have

```
node_expand(N,NL) :- stn(N,NL).
```

which worked fine when nodes looked like states, since

```
stn(bs,[gp,oc]).
stn(gp,[bs,cc,pc]).
```

etc.

Now we must modify node_expand so that it produces a *list* of nodes in the new format.

Modified Version of node_expand

```
node_expand(N,NL) :-
 node_state_of(N,S),
 stn(S,SL),
 sl2nl(SL,NL).

% sl2nl(SL, NL)
% list of states SL => list of nodes NL

sl2nl([],[]).

sl2nl([S|Y],[N|NL]) :-
 node_new(N),
 node_state_of(N,S),
 sl2nl(Y,NL).
```

Behaviour After Modifying Node Data Structure

```
?- solve(bs,ls,X,Y).
bs
gp
oc
bs
cc
pc
bs
pc
tc
gp
oc
gp
ls

X = 13
Y = n(ls,_G331);
No
```

Recovering the Solution Path

The new node data structure has a slot for the node's parent. We must make sure that it is correctly initialised when the node is created. Currently, this happens in two places.

- When the initial node is created.

```
% start node
node_initial(N) :-
 start_state(S),
 node_new(N),
 node_state_of(N,S).
```

- Whenever a node is expanded

```
sl2nl([S|Y],[N|NL]) :-
 node_new(N),
 node_state_of(N,S),
 sl2nl(Y,NL).
```

Modifications Necessary

- When the initial node is created.

```
% start node
node_initial(N) :-
 start_state(S),
 node_new(N),
 node_state_of(N,S),
 node_parent_of(N,-1).    <---- parent of root node=-1
```

- Whenever a node is expanded: an extra paramter must be provided.

```
sl2nl(P,[S|Y],[N|NL]) :-
 node_new(N),
 node_state_of(N,S),
 node_parent_of(N,P),     <----- P is parent
 sl2nl(P,Y,NL).                  of all
```

Effect of Adding Parent Info

```
solve(bs,ls,X,LastNode).
bs
gp
oc
bs
cc
pc
bs
pc
tc
gp
oc
gp
ls

X = 13
LastNode = n(ls, n(cc, n(gp, n(bs,-1))))
```

Extracting the Path from the Node

In the last example, the node

n(ls, n(cc, n(gp, n(bs,-1))))

clearly contains the information we need to recover the path. The following algorithm converts a node into a (reversed) path. Observe how its definition is almost completely independent of the manner in which nodes have been implemented.

```
node2path(N,[SN]) :-
 node_state_of(N,SN),
 node_parent_of(N,-1),
 !.

node2path(N,[SN|Y]) :-
 node_state_of(N,SN),
 node_parent_of(N,P),
 node2path(P,Y).
```

node2path in Action

?- node2path(n(ls, n(cc, n(gp, n(bs, -1)))),Z).

Z = [ls, cc, gp, bs]

Note that the path is reversed. To reverse it we can use the builtin reverse/2.

Exercise. Give a definition of reverse/2.

Looping Behaviour

First, observe that to simulate depth first search we simply modify the definition of agenda_putl/3 (this puts a list of expanded items onto the agenda) from

```
agenda_putl(A,L,Anew) :- append(A,L,Anew).
```

to

```
agenda_putl(A,L,Anew) :- append(L,A,Anew).
```

The algorithm loops

- When DFS is used.
- It also loops with BFS if asked to find a path that does not exist.

Checkpoint. Verify both of these assertions.

Adding Loop Detection

In order to detect loops we can

- Maintain an "exclusion list" of states which have already been processed.
- Never process the state if it is already on the processed list.
- **N.B.** We maintain a list of *states* rather than nodes because, being less complex structures, they are faster to compare.

There are several ways of doing this, depending on *where* we do the check.

Different Ways of Adding Loop Detection

Here are 3 possibilities:

- When a node is selected from the agenda, do not process if its state has already been processed. **Warning** Do not add the state to the exclusion list until you are sure it has been processed.
- When a node is expanded: do not expand the same state twice.
- When a node is put on the agenda: when adding nodes to the agenda (putl/3), check each to make sure that there are no other nodes with the same state.

Below we explore the second of these solutions in detail.

Maintaining EXCL

The list of states already expanded (called EXCL for "exclude") is maintained as an extra argument to agenda_search and initialised to [ ]. We then update EXCL as follows.

The single last clause of agenda_search_1 is replaced with a pair of clauses:

- The first handles the case where the state S of the current node N has already been visited. In such a case, don't expand (and don't backtrack to the next clause, hence the cut).
- The second expands the node, and puts its state S on the exclusion list.

Old/New Code

Old Code

```
agenda_search_1(C,Cfinal,N,A,Ans) :-
   node_expand(N,NodeList),
   agenda_putl(A,NodeList,NewA),
   agenda_search(C,Cfinal,NewA,Ans).
```

New Code

```
agenda_search_1(EXCL,C,Cfinal,N,A,Ans) :-
   node_state_of(N,S),
   member(S,EXCL),
   !,
   agenda_search(EXCL,C,Cfinal,A,Ans).

agenda_search_1(EXCL,C,Cfinal,N,A,Ans) :-
   node_state_of(N,S),
   node_expand(N,NodeList),
   agenda_putl(A,NodeList,NewA),
   agenda_search([S|EXCL],C,Cfinal,NewA,Ans).
```

Behaviour with Exclusion List

Doesn't loop with an impossible goal under BFS:

```
?- solve(bs,foo,X,Y,Z).
..
no
```

Doesn't loop under DFS:

```
?- solve(bs,ls,X,Z).
bs
gp
bs
cc
gp
ls

X = 6,
Z = [ls,cc,gp,bs] ? ;
```

Putting More Detail into Nodes

Now we have the mechanism, we can let nodes carry further information, e.g.

- Node Depth: (depth limited search; iterative deepening strategy).
- G-cost (path cost - for uniform cost search).
- H-cost (heuristic cost - for greedy search)
- F-cost (= f(H,G) - for A* and other search strategies)

```
% node data structure

node_state_of(  n(X,_,_,_),  X).
node_parent_of( n(_,X,_,_),  X).
node_depth_of(  n(_,_,X,_),  X).
node_gcost_of(  n(_,_,_,X),  X).

% make node

node_new(n(_,_,_,_)).
```

Next Week

- Informed Search
- Coursework Description