Outline of this lecture

• recap: depth-first search

• variants of depth-first search for infinite search spaces
  – depth limited search
  – iterative deepening search

• Prolog implementations
Depth-first search

• proceeds down a single branch of the tree at a time

• expands the root node, then the leftmost child of the root node, then the leftmost child of that node etc.

• always expands a node at the deepest level of the tree

• only when the search hits a dead end (a partial solution which can’t be extended) does the search backtrack and expand nodes at higher levels
Example: depth-first search
Properties of depth-first (tree) search

- space complexity is $O(bm)$ where $m$ is the maximum depth of the tree

- time complexity is $O(b^m)$

- not complete (unless the state space is finite and contains no loops)—we may get stuck going down an infinite branch that doesn’t lead to a solution

- even if the state space is finite and contains no loops, the first solution found by depth-first search may not be the shortest
Checking for cycles

• to extend simple depth first search to general graphs, we need to add loop detection (i.e., a closed list)

• we can then keep track of which states (cities) we have visited before, and backtrack if expanding the current node with the operator would result in a loop
Example: route planning

route(X,Y,R) :-
    route(X,Y,[X],R).

route(X,Y,V,[drive(X,Y)]) :-
    travel(X,Y).
route(X,Y,V,[drive(X,Z)|R]) :-
    travel(X,Z),
    \+ member(Z,V),
    route(Z,Y,[Z|V],R),
    Z \= Y.

travel(X,Y) :- road(X,Y).
travel(X,Y) :- road(Y,X).
Depth-first search with cycle detection

- space complexity is $O(s)$ where $s$ is the number of states

- time complexity is $O(b^m \times s)$ where $b$ is the branching factor, $m$ is the maximum depth of the tree

- complete (unless the state space is infinite)

- even if the state space is finite, the first solution found by depth-first search with cycle detection may not be the shortest
Incompleteness of DFS

• depth first search is incomplete if there is an infinite branch in the search tree

• infinite branches can happen if:
  – paths contain loops
  – infinite number of states and/or operators

• we can check for paths containing loops but if the number of states/operators is potentially infinite, we can still have infinite branches
Variants of depth first search

• for problems with infinite (or just very large) state spaces, several variants of depth-first search have been developed:

  – depth limited search

  – iterative deepening search

• other variants possible – e.g., iterative deepening with a depth bound
Depth limited search

- depth limited search (DLS) is a form of depth-first search
- expands the search tree depth-first up to a maximum depth $l$
- nodes at depth $l$ are treated as if they had no successors
- if the search reaches a node at depth $l$ where the path is not a solution, we backtrack to the next choice point at depth $< l$
- depth-first search can be viewed as a special case of DLS with $l = \infty$
Properties of depth limited search

• complete if there is a solution within the depth bound $d \leq l$

• if $l < d$, i.e., the shallowest solution is deeper than the bound, DLS will return ‘no’ even though there is a solution

• space complexity is $O(bl)$ where $b$ is the branching factor and $l$ is the depth bound

• time complexity is $O(b^l)$

• if $l >> d$ this can significantly increase the cost of the search compared to, e.g., breadth-first search

• always terminates
Choosing a depth bound

• the depth bound can sometimes be chosen based on knowledge of the problem

• e.g., in the route planning problem, the longest route has length $s - 1$, where $s$ is the number of cities (states), so we can set $l = s - 1$

• however even when $s$ is finite, setting $l = s - 1$ is inefficient if $d << l$

• for most problems, $d$ is unknown

• tendency to set the depth bound ‘high’ to ensure that any solutions that do exist are found
Cycle detection in DLS

• we can omit cycle detection in DLS and still retain completeness, as any path containing a cycle will eventually reach the depth bound

• however cycle detection may still be useful:
  
  – paths containing cycles are technically ‘solutions’ but contain redundant steps that arguably should be eliminated
  
  – allows us to fail a branch and backtrack before we reach the depth bound, saving computation
  
  – particularly important if the search space contains many loops
Depth limited search

% dls(?initial_state, ?goal_state, ?solution, +limit)
% Does not include cycle detection

dls(X,Y,[A],D) :-
    D > 0, s(X,Y,A).
dls(X,Y,[A|P],D) :-
    D > 0, s(X,Z,A), D1 is D - 1, dls(Z,Y,P,D1).

% s(?state, ?next_state, ?operator).
s(a,b,go(a,b)).
s(a,c,go(a,c)).

etc...
Aside: usage comments

- predicates often assume a particular pattern of usage – which variables must be (non)ground when the predicate is evaluated as a goal

- intended usage is often indicated by a comment \( \% \text{name}(\text{spec}, \ldots, \text{spec}) \) where each \text{spec} denotes how that argument \text{should be instantiated in goals}, and has one of the following forms:

  +\text{ArgName}: argument should be instantiated to a non-variable term

  -\text{ArgName}: argument should be uninstantiated

  ?\text{ArgName}: argument may or may not be instantiated

- other information may be necessary, e.g., which combinations of inputs result in (non)termination
Representation of states & operators

• **states** are represented by atoms in arguments to the $s/3$ successor relation and $dls/4$

• **operators** are represented by complex terms ($go/2$ terms) that represent an action that transforms one state into another, e.g., $go(a, b)$

• **applicability of operators** is determined by the (implicit) current state and the existence of an appropriate successor of the current state

• **goal** is explicit and is an argument to $dls/4$
Representation of nodes & paths

• **paths** (and solutions) are represented by lists of operator terms, e.g.,
  \[ go(a,b), go(b,d) \ldots, \]

• **nodes** are implicit – there is a single current node (corresponding to the current path), and its parents are represented by backtrack points in the Prolog interpreter

• the **open list** is similarly implicit – represented by backtrack points

• the **closed list** is semi-implicit – Prolog remembers branches of the search tree that resulted in failure and won’t try them again, but there is no list of previously visited states, for example
Iterative deepening search

- iterative deepening (depth-first) search (IDS) is a form of depth limited search which progressively increases the bound

- it first tries $l = 1$, then $l = 2$, then $l = 3$, etc. until a solution is found

- solution will be found when $l = d$

- don’t need to worry about how to set the depth bound
Properties of iterative deepening search

- complete if the branching factor is finite and there is a solution at some finite depth

- optimal in that it will find the shortest solution first (unlike depth limited search)

- space complexity is $O(bd)$ where $b$ is the branching factor and $d$ is the depth of the shallowest solution

- time complexity is $O(b^d)$

- may not terminate
Overhead of repeated computation in IDS

• iterative deepening search potentially explores each (non-solution) branch of the search tree many (approximately $d$) times

• this often isn’t a problem in practice:
  – in a search tree with an approximately constant branching factor, most of the nodes are in the bottom layer
  – e.g., in a binary tree, there are 512 nodes at level 9 and 1024 nodes at level 10
  – for $b > 2$, the bottom layer forms an even larger proportion of the total number of nodes
  – nodes at level $d$ are generated only once
Cycle detection in IDS

- as with DLS we can omit cycle detection and still retain completeness:
  - paths containing cycles are only expanded as far as the current depth bound – search can’t run off down an infinite branch
  - first solution found must contain no cycles

- however cycle detection can still be useful if there are many cyclic paths of length $< d$
Iterative deepening

% depth_first_iterative_deepening(?state,-solution)
depth_first_iterative_deepening(Node, Solution) :-
    path(Node, GoalNode, Solution),
    goal(GoalNode).

path(Node, Node, [Node]).
path(FirstNode, LastNode, [LastNode | Path]) :-
    path(FirstNode, OneButLast, Path),
    s(OneButLast, LastNode),
    	+ member(LastNode, Path).

– see Bratko (2001), chapter 11.2
Aside: terminology

- the variable naming conventions in Bratko’s implementation are rather confusing in mixing up states and nodes

- the variables Node, GoalNode, FirstNode, LastNode, and OneButLast actually refer to states rather than nodes in the search tree

- there is a single node representing the current path from the initial state

- path consists of a sequence of states implicitly connected by a (single) unnamed operator
Limitations of this approach

- Bratko’s implementation is quite elegant, and it will return a solution if one exists

- returns all alternative solutions on backtracking

- however if there are no solutions (or no more solutions on backtracking), search does not terminate even if the state space is finite

- can’t be used to exhaustively enumerate all solutions (e.g, with all solutions predicates)
Revised iterative deepening

% ids(\texttt{?initial\_state,\texttt{?goal\_state,\texttt{?solution}})
% Note: paths returned in reverse order; does not include cycle detection
ids(X,Y,[A]) :-
    s(X,Y,A).
ids(X,Y,[A|P]) :-
    ids(X,Z,P), s(Z,Y,A).

% s(\texttt{?state, \texttt{?next\_state, \texttt{?operator}}).
 s(a,b,go(a,b)).
 s(a,c,go(a,c)).

etc...
Implementation

- representations of states, operators, nodes and paths is as for \( dls/4 \)

- note that paths are returned in \textit{reverse} order

- cycle detection can be added as in the route planning example

- termination is possible, but requires either additional computation, or use of extra-logical features of Prolog

- alternatively, if the maximum path length is known, we can add a depth bound to \( ids/3 \)
Terminating iterative deepening (sketch)

% tids(?initial_state,?goal_state,?solution)
% Does not include cycle detection

\[
tids(X,Y,P) :-
  tids(X,Y,P,1).
\]

\[
tids(X,Y,P,D) :-
  eds(X,Y,P,D).
\]

\[
tids(X,Y,P,D) :-
  D1 is D + 1, eds(X,_,_,D1), !, tids(X,Y,P,D1).
\]

% Find a path from X to Y of length exactly D

\[
eds(X,Y,[A],D) :-
  D = 1, s(X,Y,A).
\]

\[
eds(X,Y,[A|P],D) :-
  D >= 1, s(X,Z,A), D1 is D - 1, eds(Z,Y,P,D1).
\]
Good (search) implementations …

• if there is a solution, a (search) procedure should return it

• if the state space is finite, a (search) procedure should return all solutions on backtracking

• if the state space is finite and there are no solutions, a (search) procedure should return ‘no’

• for some problems, a ‘good’ implementation may be difficult

• as a rule of thumb:
  – ‘utility’ predicates should always behave properly on backtracking
  – a Prolog program to solve a particular problem should ideally behave properly on backtracking – if this is not practicable it’s behaviour should be clearly documented
The next lecture

*Breadth-first search in Prolog*

Suggested reading:

- Bratko (2001) chapter 11.3