TDDD08 — Tutorial 4

Who? Victor Lagerkvist

From? Theoretical Computer Science Laboratory, Linköpings Universitet, Sweden

When? 28 september 2015

Definite Clause Grammars and Search

- Definite clause grammars. Used for e.g. parsing (lab 3).
- Search (lab 4). Searching through a state space (lab 4).

Definite Clause Grammars

- *Definite clause grammars* (DCG) provides a declarative and efficient way to write parsers in Prolog. The name stems from the fact that we essentially take a set of context-free grammar rules and encodes it as a set of definite clauses.
- Examples on white board.

Search

- Search problems in Prolog.
- Depth-first search.
- Breadth-first search.

Search problems

Many problems are encodable as search problems. Examples:

- Path finding: Can we get from a to f in a graph, given `edge(X,Y)`?
- Planning: How can we get the goat, wolf and cabbage across the river without anyone getting eaten?
- Logic programming: Is there a successful SLD-derivation from our query, using the available clauses.

All these share some basic structure. Graph example on white board.

Search problems in logic programming

Given a start state, a transition relation, and a goal state (or states), can we reach a goal state from the start state?

Components:

- A state space (vertices, states of the world, logic programming goals, etc), represented as Prolog objects.
- A transition relation, e.g. `edge(X,Y)` or `move(state(A1,B1),state(A2,B2))`.
- Start state e.g. `start(State)`
- A goal condition (e.g. `goal(State)` predicate).
- A search strategy, there are several.

General depth-first search

Suppose that the following predicates are provided:

- `init(State)`: Gives the initial state.
- `goal(State)`: State is a goal state.
- `action(State1,State2)`: There exists some action (move, edge, etc) taking us from `State1` to `State2`.

This is enough to write a general problem solver (for problems with a finite number of states).

General depth-first search

```
%% df_search(Path): True if Path is a path taken
%% from S0 to a goal state
df_search(Path) :-
    init(S0),
    df_search([S0], Path).

%% df_search(Partial,Path): True if Partial can
%% be extended into a path Path to a goal state
df_search([S|Visited], [S|Visited]) :-
    goal(S).

df_search([S1|Visited], Path) :-
    action(S1, S2),
    nonmember(S2, [S1|Visited]),
    df_search([S2,S1|Visited], Path).
```

Example, encoding of the previous problem:

```
init(a).
goal(c).
action(X, Y) :- edge(X, Y).
```

Maze example

Larger example: A small game, a maze with locked doors and coloured keys.

- **State**: Current room (a–e), carried keys (blue, green, etc). Start with state(a,[no key]).
- **Actions**:Pick up a key (if present), walk through a door (if carrying the right key).

We use predicates `has_key(Room, Key)` and `door(R1, R2, Key)` to encode the actual maze.

Maze example

Possible description:

```
has_key(a,blue_key).
has_key(d,green_key).
has_key(c,red_key).

door(a,b,no_key).
door(b,d,blue_key).
door(a,c,green_key).
door(a,e,red_key).

init(state(a,[no_key])).

goal(state(e,_)).

%% Action: walk through door
action(state(X,Keys), state(Y,Keys)) :-
    (door(X,Y,Key) ; door(Y,X,Key)),
    member(Key, Keys).

%% Action: pick up key
action(state(X,Keys), state(X,[Key|Keys])) :-
    has_key(X, Key),
    nonmember(Key, Keys).
```

Maze example

Solution:

```
| ?- df_search(P).
P = [state(e,[red_key,green_key,blue_key,no_key]),
       state(a,[red_key,green_key,blue_key,no_key]),
       state(c,[red_key,green_key,blue_key,no_key]),
       state(c,[green_key,blue_key,no_key]),
       state(a,[green_key,blue_key,no_key]),
       state(b,[green_key,blue_key,no_key]),
       state(d,[green_key,blue_key,no_key]),
       state(d,[blue_key,no_key]),
       state(b,[blue_key,no_key]),
       state(a,[blue_key,no_key]),
       state(a,[no_key])] ?
```

Remember that it returns the path backwards.

Other search orders

With depth-first search Prolog handles the search order for us. With other search strategies we need to keep track of this ourselves: maintain a set of all generated candidates (partial paths), and expand them one at a time.
Pseudocode:

```
search([Solution|OtherPaths], Solution) :-
    contains_goal(Solution).

search([FirstPath|OtherPaths], Solution) :-
    find_all_moves(FirstPath, NewStates),
    expand(FirstPath, NewStates, NewPaths),
    insert_paths(NewPaths, OtherPaths, Paths),
    search(Paths, Solution).
```

The order of insertion controls the search order (e.g., sort by heuristic for A*).

Breadth-first search in the maze

```
% bf_maze(Path): Path is a path from start to goal
bf_maze(Path) :-
     init(S0),
     bf_maze([[S0]], Path).

% bf_maze(Paths, FinalPath): Paths is a list of
% candidate branches, FinalPath is the solution
bf_maze([[S|Path]|_], [S|Path]) :-
    goal(S).
bf_maze([[S1|Path]|Partials], FinalPath) :-
    findall(S2, action(S1,S2), NewStates),
    expand([S1|Path], NewStates, NewPaths),
    append(Partials, NewPaths, NewPartials),
    bf_maze(NewPartials, FinalPath).

expand(L1, L2, L3) :-
    findall([X|L1], member(X,L2), L3).
```

Breadth-first search in the maze

Breadth-first with loop detection is left as an exercise (lab 4). This version will go in circles a lot, but it will still find the shortest solution.