Programming in Logic: Prolog

More Built-in Predicates

Readings: 7.1-4
Review

- Planning problems involve sequences of actions/state: initial state => goal state.
- Naïve approach to generation too expensive.
- Incorporate legal trans and init state testers.
- Leaves goal state tester.
- Traverse search space using iterative deepening.
- Use heuristics (e.g., loop ident) to prune search.
/* Iterative Deepening Search : 
  ids(+StartState,+GoalState,?Solution) */

ids(StartNode, GoalNode, Solution) :-
    path(StartNode, EndNode, Solution),
    EndNode = GoalNode,
    !.

/* path(+StartNode, ?EndNode, ?Path) */
path(StartNode, StartNode, [StartNode]).
path(StartNode, EndNode, [EndNode|Path]) :-
    path(StartNode, PenUltimateNode, Path),
    newState(PenUltimateNode, EndNode),
    not(member(EndNode,Path)).
Declarative Programming Revisited

• The goal of declarative programming is to enable one to easily understand a program simply by reading it without any need to figure out how Prolog actually executes it.

• Current definition of *ids* fails in this respect.

• Need to reimplement *ids* so that it is easy to understand and is obviously correct.
Reimplementing *ids*

- Instead of relying on instantiating Path to increasing number of unbound variables, we will reply on explicit depth bound term stating the maximum depth allowed on this iteration.
- Split search into lower level bounded-depth-first search relation and an upper level *ids* which manages the explicit depth.
- First look at bounded-depth-first search.
Bounded Depth First Search

/* Bounded Depth First Search bdfs(+Start, +Goal, ?Solution, +Bound)*/

bdfs(Goal,Goal,[Goal],_).
bdfs(Start, Goal, [Start|Path], Bound) :-
    Bound > 1,
    newState(Start, Next),
    NewBound is Bound - 1,
    bdfs(Next, Goal, Path, NewBound).
New *ids* Structure

- Our new version of *ids* will have 3 cases:
  - Initialization, setting depth bound to 1.
  - When *bdfs* succeeds within bound.
  - When *bdfs* fails and need to increase bound.

- In the latter 2 cases, *ids* needs to know the current bound, while in the first one it doesn’t.

- Make 2 relations.
**ids(+Start,+Goal,?Solution) revisited**

/\* Iterative Deepening Search (usig bdfs): ids2(+Start, +Goal, ?Solution)*/

ids2(Start, Goal, Solution) :-
   ids2(Start, Goal, Solution, 1).

ids2(Start, Goal, Solution, N) :-
   bdfs(Start, Goal, Solution, N).
ids2(Start, Goal, Solution, N) :-
   N1 is N + 1,
   ids2(Start, Goal, Solution, N1).
Displaying Messages to the User

• `write(\texttt{?Term})` displays \texttt{Term} to the user.
• `write(\texttt{`Hi there'})`. Displays “Hi there”.
• \texttt{X = 4}, `write(X)`. Displays “4”
• `nl` displays a new line.
• `tab(N)`. Display \texttt{N} spaces.
Consulting Programs

• Typing in *simple* causes the file simple.pl to be read into Prolog’s KB as a Prolog program.
• This notation is shorthand for consult(simple).
• You can consult many files during one session.
• Can consult many files at one time:
  – consult(ListofFiles)
• Consults can be nested inside of files, a file being consulted, can itself consult another file.
Compiling Prolog Programs

• Often Prolog programs run slower than desired.
• Since they’re normally interpreted, one quick fix is to compile it.
• `compile(FileName)`. Causes the programs to be compiled and then loaded into memory.
• `compile/1` similar to `consult/1`.
Term Types

- term
  - variable
    - bound
      - constant
    - unbound
  - constant
  - simple
    - atom
    - number
      - integer
      - float
  - structure
Term Type Tests

```
term
  └── variable
      └── bound
          └── constant
  └── constant: nonvar
      └── unbound: var
          └── simple: atomic
              └── structure: compound
                  └── atom: atom
                      └── number: number
                          └── number: number
                              └── float: float
```
Making and Breaking Atoms

- `name(Atom, CharList)`: relation between Prolog atom & ascii codes for characters making it up. One/both arguments needs to be instantiated.

- `name(zx232, [122, 120, 50, 51, 50]).`

- `name(zx232, L).`

- `name(A, [122, 120, 50, 51, 50]).`

- Often want to make up new atoms from old (sort of like concat, except for atoms): e.g., `gensym(cut, 45, Atom).` \(\Rightarrow\) \textbf{Atom} = \textbf{cut}45
gensym(+Prefix, +Number, ?Atom)

gensym(Prefix, Number, Atom) :-
    name(Prefix, PList),
    name(Number, NList),
    concat(PList, NList, ALList),
    name(Atom, ALList).
Making & Breaking Terms

- **Term = \( L \)**
  - \( f(x, 2) = L \) \( \Rightarrow \) \( L = [f, x, 2] \)
  - \( T = [f,x,2] \) \( \Rightarrow \) \( T = f(x,2) \)

- **functor(Term, F, A):** Term has functor \( F \) with arity \( A \).
  - \( \text{functor}(f(x,2), F, A) \) \( \Rightarrow \) \( F = f, A = 2 \)

- **arg(N, Term, Argument):** \( N \)th arg of \( Term \)
  - \( \text{arg}(1, f(x,2), A) \) \( \Rightarrow \) \( A = x \)
Example: \textit{ground1(+Term)}

- \textit{ground1/1} tests whether \textit{Term} has any unbound variables.

- \textit{ground1(X) => no} \newline
  \hspace*{1cm} \textit{ground1(f(2,g(3,h(1,2))))} => \textit{yes} \newline
  \hspace*{1cm} \textit{ground1(f(2,g(3,h(1,X))))} => \textit{no} \newline
  \hspace*{1cm} X=1, \textit{ground1(f(2,g(3,h(1,X))))} => \textit{X = 1}
ground1(Term) :-
    atomic(Term).

ground1(Term) :-
    compound(Term),
    Term =.. L,
    ground2(L).

ground2([]).

ground2([X|Rest]) :-
    ground1(X),
    ground2(Rest).
Using Constructed Terms: \textit{call/1}

- Not only can we dynamically construct terms on the fly, we can then call them as goals.
- For example, might define relation \textit{binOpEval}$(+Op, +X, +Y, ?Z)$ as:
  
  \begin{verbatim}
  binOpEval(Op,X,Y,Z) :-
      BinOp =.. [Op, X,Y],
      T =.. [is, Z, BinOp],
      call(T).
  \end{verbatim}
not/1 definition revisited

• \( \text{not}(P) :- \text{call}(P), !, \text{fail}. \)

\( \text{not}(_) \).

• \( P \) is being called as a goal, most Prologs allow a syntactic shorthand, called a MetaVariable facility, which allows us to abbreviate above to:

\( \text{not}(P) :- P, !, \text{fail}. \)

\( \text{not}(_) \).
Generic Puzzle Solver

solvePuzzle(puzzle(\text{Clues}, \text{Queries}, \text{Solution}), \text{Solution}) :-
    solve(\text{Clues}),
    solve(\text{Queries}).

solve([\text{Clue}|\text{Clues}]) :-
    \text{Clue},
    solve(\text{Clues}).

solve([ ]).
Example Puzzle

• Three friends came first, second, and third in a programming contest. Each had a different first name, liked a different sport, and had a different nationality. Michael likes basketball, and did better than the American. Simon, the Israeli, did better than the tennis player. The cricket player came first. Who is the Australian? What sport does Richard play?
testPuzzle(puzzle(Clues,Queries,Solution) :-
    structure(Structure),
    clues(Structure,Clues),
    queries(Structure, Queries,Solution) .

structure([[friend(N1,C1,S1), friend(N2,C2,S2), friend(N3,C3,S3)]]).

clues(Friends, [(didBetter(M1C1,M2C1,Friends),
                 name(M1C1,michael),  sport(M1C1,basketball),
                 nationality(M2C1,american)),
                 (didBetter(M1C2,M2C2,Friends), name(M1C2, simon)
                   ,
                 nationality(M1C2,israeli), sport(M2C2,tennis)),
                 (first(Friends,MC3), sport(MC3, cricket))]).

queries(Friends, [member(Q1,Friends), name(Q1, Name),
                  nationality(Q1, australian), member(Q2,Friends),
                  name(Q2,richard), sport(Q2,Sport)],
[[‘Aussie is’, Name], [‘Richard plays ‘, Sport]]).
Types of Equality/Comparison

- \( X = Y \): Does \( X \) match \( Y \)?
- \( X \) is \( E \): Does \( X = \) arithmetic evaluation of \( E \)?
- \( E1 := E2 \): \( E1 \)’s arith eval = \( E2 \)’s arith eval?
- \( E1 \neq E2 \): \( E1 \)’s arith eval \( \neq \) \( E2 \)’s arith eval?
- \( T1 == T2 \): \( T1 \)’s structure & content = \( T2 \)’s including variable naming.
- \( T1 \neq T2 \): not(\( T1 == T2 \))
Examples

- $f(a, X) == f(a, b) \Rightarrow \text{no}$

- $f(a, X) == f(a, X) \Rightarrow \text{yes}$

- $f(a, X) == f(a, Y) \Rightarrow \text{no}$

- Could use to implement occurs check.
Dynamically Modifying Prolog’s KB

- Prolog allows adding and retracting clauses from its KB.
- Since clauses in the KB define the program, changing the KB changes program being run.
- This enables Prolog programs to “learn” from their experience.
- Dynamically modifying KB means programs can be much harder to understand & debug!
Adding Clauses to KB

- `assert/1` adds clause somewhere in KB, `asserta/1` adds clause at beginning of KB, `assertz/1` adds clause at end of KB.
- Calling `assert` always succeeds.
- Redoing `assert` always fails.
- Backtracking through `assert` doesn’t retract clause from KB, need to use `retract/1`.
- `assert( :- (x(Y,Z), is(Z, +(Y,1))))` adds the clause `x(Y,Z) :- Z is Y + 1` to KB.
Retracting Clauses from KB

• `retract(Clause)` causes the first clause matching `Clause` to be retracted from KB.
• Calling `retract(Clause)` only succeeds if some clause is retracted.
• Redoing `retract(Clause)` succeeds if retracts something else.
• `retract(\(\neg\)(x(Y,Z), Body), fail)` causes all clauses whose heads match `x(Y,Z)` to be retracted.
Technicalities of Dynamic Predicates

• `dynamic(+predspec)` is a load command, tells Prolog that if there are no clauses for `predspec` simply fail, otherwise many Prologs raise runtime exception.

• There are rules governing assertion/retraction of relation’s clauses defined in consulted/compiled files versus ones that are not, will ignore them.
gensym revisited

• Defined `gensym(+Prefix, +Number, ?Atom)` because had no way of keeping track of number between calls.

• With `assert & retract` can let Prolog keep track.

• Use them to store number between calls:
  - `gensym(cut, A) => cut1`
  - `gensym(up,B) => up2`
  - `gensym(up,B) => up3`
gensym(+Prefix, ?Atom)

definition:
:- dynamic(gensym/1).
gensym(Prefix, Atom) :-
   name(Prefix, Plist),
   (gensym(Number) ->
      retract(gensym(Number))
   ;
   Number = 1),
NewNumber is Number + 1,
assert(gensym(NewNumber)),
name(Number, NList),
concat(PList, NList, AList),
name(Atom, AList).
gensym/2 Behavior

• What do you think will happen if we try to redo gensym/2?
• What part of gensym’s definition causes that behavior?
• What would happen if it didn’t behave this way?
• What other way could it behave and does it have problems?
Rote Learning

• The simplest type of learning is rote learning.

• For example, if define factorial relation & factorial of 9 is computed, next time it is asked for simply recall it rather than recomputing it.

• This is also called memoization.
Fibonacci & Rote Learning

fib(1,1).
fib(2,1).

fib(N,F) :-
  N > 2,
  N1 is N - 1,
  fib(N1, F1),
  N2 is N1 - 1, fib(N2, F2),
  F is F1 + F2,
  assert(fib(N,F)).
Recap of What We Can Now Do

• With `name/2` can now create new atoms, which we could use to name new concepts, etc.
• With `=../2` can now create new terms (e.g., new goals, etc.)
• With `call/1` can actually “invoke” those constructed goals.
• With `assert/retract` can actually add/delete clauses/relations from Prolog’s KB, e.g., add to or rewrite the program.
Summary

- Ease of understanding and obviousness of correctness are very important.
- `write(Term), nl, tab(N)`: communication with user
- `consult/1 & compile/1`: ways of loading programs into KB
- Term type tests: `var/1, nonvar/1, …`
Summary cont’d

• *name/2*: making and breaking atoms
• =../2, *functor/2, arg/3*: making & breaking clauses
• Using constructed clauses: *call/1* & meta-variable facility => data <-> code
• Modifying KB: *assert/retract*
  – Keeping track of values across calls
  – Learning from experience