Prolog Lists

- A special recursive data structure with special syntax to make dealing with lists simpler
- List is
  - Empty list: []
  - An element on top of a list: [Top | RestOfList]

- Rewriting our path structure in List notation:
  - [1 | [2 | [6 | [10 | []]]]]

- To get top and remainder, unify with [Top | Rest]

- Syntax shortform:
  - [1 | [2 | [6 | [10 | []]]]] can be written as [1, 2, 6, 10]
  - or as [1 | [2 | [6 | 10]]]
  - or as [1 | [2, 6, 10]]
  - or as ....

- How does [X, Y] unify with [a, b]?
- How does [X | Y] unify with [a, b]?

The 'Pro' Stands for Programming

- 'Pro' stands for Programming
- In which Programming language is Program written?
- and Prolog execution is Prolog execution, which is written in a Programming language.

- In which Programming language is Program written?
- Search
- Depth-first
- Depth-first search
- Breadth-first
- Search

The 'Pro' Stands for Programming

- Programming search in Prolog
- More on Programming in Prolog
- Prolog's Search Strategy

Overview
Overview

• The 'Pro' Part of Prolog
  ⇒ Search
  • Depth-first
  • Breadth-first
• Prolog’s Search Strategy
• More on Programming in Prolog
  ⇒ Programming Search in Prolog

Singleton Variables

• '\' special syntax for naming a variable
  - Used when don’t care about, used for singleton variables
  - Can be used multiple times in some clauses, but each use is really a different variable

length([|Rest],Len) ← length(Rest,L)
Len is L + 1

• Special syntax for naming a variable

Append and Member

• Rewrite concat in new list syntax
  concat(nil,Z,Z)
  concat(p(A,X),Y,p(A,Z)) ← concat(X,Y,Z).

• Rewrite member
  member(X, LIST) true if X is an element in LIST

Rewrite member
concat([X],Z) ←
concat([X|Y],Z) ←
concat([X],Z).
concat([Z],Z).

Rewrite concat in new list syntax

Append and Member
Examples

- Maze
- Office example
- Electrical circuit
- Top-down theorem proving

Search

A graph consists of:

- A set of nodes, \( N \)
- A set of ordered pairs of nodes, called arcs, \( A \)

A path is a sequence of nodes, \( n_0, n_1, \ldots , n_k \) such that:

- \( \forall i \in \{0, \ldots , k\} \), \( n_i \in N \)
- \( \forall i \in \{1, \ldots , k\} \), \( \langle n_{i-1}, n_i \rangle \in A \)

Given a set of start nodes and goal nodes, a solution is a path from a start node to a goal node.

A problem is a pair, \( (\text{query}, \text{domain}) \), where the goal is a sequence of nodes, \( \text{goal} \)

To convert a proof procedure into a reasoning procedure, need to replace non-determinism with a backtracking search.
Illustration of Graph Searching

- The way in which the frontier is expanded defines the search strategy.
  - As the search proceeds, the frontier expands into the unexplored nodes until a goal node is encountered.
  - Maintaining a frontier of paths from the start node that have been explored incrementally expands the frontier from the start nodes and goal nodes.
  - Given a graph, start nodes, and goal nodes, a generic search algorithm searches the graph.

Search Graph for Resolution (Top-Down) Proof

- From Official Lecture slides.
Overview

The 'Pro' Part of Prolog
Search
• Depth-first
• Breadth-first

More on Programming in Prolog
Programming Search in Prolog

Summary of Generic Search Algorithm

Generic Graph Search Algorithm
Complexity of Depth-first Search

- Depth-first search is not guaranteed to halt on infinite graphs or graphs with cycles.
- The space complexity is linear in the size of the path being explored.
- Depth-first search is unconstrained by the goal until it happens to stumble on it.

Illustration of Depth-first Search

Depth-first Search

- Depth-first search treats the frontier as a stack: it always selects the last element added to the frontier.
- The last element added to the frontier is always selected.

Depth-first Search

- The last element added to the frontier is always selected.
- Depth-first search treats the frontier as a stack: it always selects
- the last element added to the frontier.
Breadth-first Search

- Breadth-first search treats the frontier as a queue: it always selects the earliest element added to the end of the queue.

```prolog
select(Node, [Node | Frontier], Frontier).
add_to_frontier(Neighbors, Frontier1, Frontier2).% Frontier: [e1, e2, ...]
```

- E1 is selected next. All of its neighbors are added to the end of the queue.
- E2 is selected next. Its neighbors are added to the end of the queue.

Breadth-first search needs the frontier as a queue: it always selects the earliest element added to the frontier.
Top-Down Resolution

- Need search strategy for doing top-down resolution
  - Always resolve first atom of answer clause first
  - Knowledge engineer can order atoms to help constrain search
    - Example:
      - \( a \leftarrow b \land c \land d \) Order by which ones will be bound
  - But many rules/facts from KB might unify with first atom
  - Can search through these using depth-first search
    - Search is ordered by order of clauses in KB
    - Knowledge engineer can order clauses to ensure solution is found quickly
  - Search over space of answer clauses, not over space of proofs
    - Make the node be an answer clause, not the entire derivation

Prolog

Overview

- The 'Pro' Part of Prolog
- Search
  - Depth-first
  - Breadth-first
- Prolog’s Search Strategy
- More on Programming in Prolog
- Programming Search in Prolog
- The Pro Part of Prolog

Complexity of Breadth-first Search

- The space complexity is exponential in path length
- The branching factor is exponential in the path length, \( b^n \), where \( b \) is the branching factor in Prolog and \( n \) is the path length
- Search is unconstrained by the goal
- Search is unconstrained by the goal
'Not' in Prolog

- The operator `not(X)` means that `-X` is not derivable in Prolog given the current instantiation of `X`.
- Following two definitions are not equivalent:
  
  - `brother(X,Y) ← brother(X,Y)`
  - `brother(X,Y) ← mother(X,Z), mother(X,Z), not(X = Y)`

  - Semantics are not very clean:
    - Truth depends on where it is in the body of a clause.
    - Truth does not correlate with semantics of model.

  - In Prolog, the `not(X)` operator is computed by searching for a proof of `X` and failing. If a proof is found, it is not derivable.

Cycles in Prolog

- What about cycles (since Prolog is depth-first search):

  - Could check for cycles: list of atoms in answer clause identical to list earlier up in proof.

  - Need to keep answer clauses along the current path we are exploring.

  - Cycles part of larger problem of endless loops.

  - Knowledge engineer's job to be careful in defining clauses.

  - Cannot detect all endless loops (halting problem).

  - Prolog doesn't do any cycle checking.
Overview

The 'Pro' Part of Prolog

• Search
• Depth-first
• Breadth-first

Prolog's Search Strategy

More on Programming in Prolog

Programmable Search in Prolog

Prolog's Findall

- findall(X, connected(1,X), L)
  - Returns a list of all X in which connected(1,X) is true
- findall can return a list of any arbitrary structures
  - findall(cell(X), connected(Y,X), L)
  - findall(connected(X,Y), connected(X,Y), L)
  - findall(X+Y, connected(X,Y), L)

- findall(connected(X,Y), connected(X,Y), L)

+ is just an infix operator that we chose to use
  + Could have used any valid Prolog term, e.g. a(X,Y), [X, Y]

- Semantics of findall are messy

- Requires universal quantification, which is not part of Datalog
- Universal quantification on things that can be named
- Universal quantification on things that can be un-named

Maze Example

- Example: path through a maze

- connected(1,2).
- connected(2,3).
- connected(2,6).
- connected(4,8).
- ...
  connectedto(X,Y) :- connected(X,Y).
  connectedto(X,Y) :- connected(Y,X).

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
</tbody>
</table>

- Book defined breadth-first and depth-first search in Datalog
- No way to do this in Datalog:
- On define neighbors(X,Y) in terms of connected(X,Y)
- Could define neighbors as primitive
- But how can we define the neighbors clause then is needed?
Breadth-First Search

Define breadth-first search using KB

\[ \text{search(Path | Path)} \leftarrow \text{Path = [1 | Path]} \]

\[ \text{search(Frontier0, Answer)} \leftarrow \text{Frontier0 = [Path | Frontier1], Path = [X | Rest], findall([Z,X | Rest], connected(Z,X), NN), concat(Frontier1, NN, Frontier2)} \]

\[ \text{search(Frontier2, Answer)} \]

?- search([1], Answer)

Depth-First Search

Depth-First Search on Nodes

Define path using KB and use Prolog's depth-first search

\[ \text{search(16).} \]

\[ \text{search(X)} \leftarrow \text{connected(X,Z), search(Z).} \]

?- search(1)

- Prolog keeps track of backtracking alternatives automatically

Define depth-first search using KB

\[ \text{search([16 | Path])} \leftarrow \text{Path = [16 | Path]} \]

\[ \text{search(Frontier0)} \leftarrow \text{Frontier0 = [X | Frontier1], findall(Z, connected(Z,X), NN), concat(NN, Frontier1, Frontier2), search(Frontier2).} \]

?- search([1])

- Explicitly keep backtracking alternatives, and don't use Prolog's deterministic code: little backtracking in Prolog code

Depth-First Search Saving Paths

Define depth-first search using KB

\[ \text{search([Path])} \leftarrow \text{Path = [16 | Path]} \]

\[ \text{search(Frontier0, Answer)} \leftarrow \text{Frontier0 = [Path | Frontier1], Path = [X | Rest], findall([Z,X | Rest], connected(Z,X), NN)} \]

\[ \text{append(NN, Frontier1, Frontier2), search(Frontier2, Answer).} \]

?- search([1])

+ Determine order for breadth-first search

- Explicitly keep backtracking alternatives and don't use Prolog's deterministic code

Depth-First Search on Nodes

- Prolog keeps track of backtracking alternatives automatically

Define depth-first search using KB

\[ \text{search([Path])} \leftarrow \text{Path = [16 | Path]} \]

\[ \text{search(Frontier0)} \leftarrow \text{Frontier0 = [X | Frontier1], findall(Z, connected(Z,X), NN), concat(NN, Frontier1, Frontier2), search(Frontier2).} \]

?- search([1])

- Prolog's depth-first search

Depth-First Search Saving Paths

- Depth-first search can backtrack

Depth-First Search on Nodes

- Prolog keeps track of backtracking alternatives automatically

Depth-First Search Saving Paths

- Determine order for breadth-first search

Depth-First Search on Nodes

- Prolog keeps track of backtracking alternatives automatically

Depth-First Search Saving Paths

- Directly implement code to reflect backtracking in Prolog code

Depth-First Search Saving Paths

- Explicitly keep backtracking alternatives and don't use Prolog's deterministic code

Depth-First Search Saving Paths

- Determine order for breadth-first search

Depth-First Search Saving Paths

- Prolog keeps track of backtracking alternatives automatically

Depth-First Search Saving Paths

- Directly implement code to reflect backtracking in Prolog code

Depth-First Search Saving Paths

- Explicitly keep backtracking alternatives and don't use Prolog's deterministic code
Adding in Cycle Checking

- Ensure new cells not already in path

```
search(Path,Path) ← Path = [16|].
search([X|Rest],Path) ← connected(X,Z), not(member(Z,[X|Rest])), search([Z,X|Rest],Path).
?search([1],Answer)
```

Cycle Checking

- Depth-First search of Maze can easily get stuck in cycle - i.e. 1 - 2 - 1 - 2 - 1
- Approach:
  - Use version where we keep the paths and build the path on the way in

```
search(Frontier0,Answer) ← Frontier0 = [Path|Frontier1], Path = [X|Rest], findall([Z,X|Rest],neighbor(Z,[X|Rest]),NN), concat(Frontier1,NN,Frontier2), search(Frontier2).
neighbor(Z,[X|Rest]) ← connected(Z,X), not(member(Z,[X|Rest]))
?search([[1]],Rest)
```

Comments

- Should we use Prolog to implement search?
  - Where we use prolog to implement search rather than use prolog's built in backtracking
- We can:
  - Prolog is indeed a full programming language
  - But can implement it in many different languages
  - Problem is that we lose the declarative nature
  - Should we use Prolog to implement search?
Another Cycle Checker

Ensure that any cell is just visited once

- Can not do this in version that uses Prolog's backtracking
- For simplicity, done in this version where front is list of cells

```
search(Frontier0, Seen0) ← Frontier0 = [X | Frontier1]                     
findall(Z, neighbor(Z, X, Seen0), NN)                                     
append(Frontier1, NN, Frontier2)                                          
append(Seen0, NN, Seen1)                                                  
search(Frontier2, Seen1)                                                 
neighbor(Z, X, Seen) ← connected(Z, X)                                   
not(member(Z, Seen))                                                     
?search([1], [])
```