Prolog Lists

A special recursive data structure

- 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]?

Prolog

The 'Pro' Stands for Programming

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

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
  - Length of List = L + 1
  - length([], 0)
  - length([X|List], Len) ← length(List, Len1), Len is Len1 + 1
- Can be used multiple times in same clause, but each use is really a
  - special syntax for naming a variable

Append and Member

- member(X, [X|List]) /
  - Rewrite member
    - member(X, [X|List]) /
      - Rewrite member
        - member(X, (List))
Examples

- Maze
- Office example
- Maze
- What are nodes and arcs for

Search

- To convert proof procedure into a reasoning procedure, need to
  - Top-down theorem proving
  - Exhausted search
  - Pruning
  - Program search

Search Graphs

- A graph consists of a set of nodes and a set of arcs. A solution is a path.
  - A path is a sequence of nodes, $n_1, n_2, \ldots, n_k$, such that $<n_1, n_2> \in A$.
  - A set of ordered pairs of nodes, called arcs.
  - A graph consists of a set of start nodes and goal nodes. A solution is a path from a start node to a goal node.

Search

- To convert proof procedure into a reasoning procedure, need to
  - Top-down theorem proving
  - Exhausted search
  - Pruning
  - Program search

Search Graphs

- A graph consists of a set of nodes and a set of arcs. A solution is a path.
  - A path is a sequence of nodes, $n_1, n_2, \ldots, n_k$, such that $<n_1, n_2> \in A$.
  - A set of ordered pairs of nodes, called arcs.
  - A graph consists of a set of start nodes and goal nodes. A solution is a path from a start node to a goal node.

Search

- To convert proof procedure into a reasoning procedure, need to
  - Top-down theorem proving
  - Exhausted search
  - Pruning
  - Program search
Graph Searching

• Generic search algorithm: given a graph, start nodes, and go al
nodes, incrementally explore paths from the start nodes
• Maintain a frontier of paths from the start node that have been
explored
• As search proceeds, the frontier expands into the unexplored
nodes
• The way in which the frontier is expanded defines the search
strategy

Search Graph for Resolution (Top-Down) Proof

(from official lecture slides)
Overview

The 'Pro' Part of Prolog

Search

• Depth-first

• Breadth-first

Prolog's Search Strategy

• More on Programming in Prolog

• Programming Search in Prolog

Summary of Generic Search Algorithm

1. Define the search function: search(F0)
2. Select a node from the frontier: select(Node,F0,F1)
3. Is the node a goal? is goal(Node).
4. If not, add its neighbors to the frontier: neighbors(Node,NN) → add to frontier(NN,F1,F2)
5. Recursively search the neighbors: search(F2).

Definition of predicates:

+ search(Frontier) is true if path from element of Frontier to a goal node
+ is goal(N) is true if node N is a goal node
+ neighbors(N,NN) means NN is list of neighbors of N
+ select(Node,F0,F1) means Node ∈ F0 and F1 = F0 - {Node}. Fails if F0 is empty
+ add to frontier(NN,F1,F2) means F2 = F1 ∪ NN

Generic Graph Search Algorithm
Complexity of Depth-first Search

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

Illustration of Depth-first Search

- Depth-first search treats the frontier as a stack; it always selects the last element added to the frontier.
- Frontier: [e1, e2, ...]
  - e1 is selected. Its neighbors are added to the front of the stack.
  - e2 is only selected when all paths from e1 have been explored.

Depth-first Search

- Depth-first search is a search algorithm for finding a path in a graph or a tree. It starts at the root node and explores as far as possible along each branch before backtracking.
- It uses a stack to keep track of the nodes to be visited.
- It visits the nodes in a depth-first manner, meaning it goes as deep as possible on a given branch before backtracking.
- It is useful for finding solutions to problems where a path can be extended indefinitely, such as in game trees or exploring all possible moves in a game.
Breadth-first Search

- Breadth-first search treats the frontier as a queue: it always selects the earliest element added to the frontier.
- `select(Node, [Node | Frontier], Frontier)`: selects the earliest element added to the frontier.
- `add(Neighbors, Frontier1, Frontier2)`: adds the neighbors of the selected element to the frontier.
- The empty element is always added to the frontier.

Illustration of Breadth-first Search
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
  + Exampl: resolution with first atom which are will be bound
  + Knowledge engineer can order atoms to help constrain search
  + Always resolve first atom of answer clause first

⇒ Prolog

Overview

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

Complexity of Breadth-first Search

• Search is unconstrained by the goal
• Time complexity is exponential in path length
  • More branching cycles in path length of \( b \), where \( b \) is path length
  • Space complexity is exponential in path length of \( b \), where \( b \) is path length
  • It is guaranteed to find the path with least depth
  • If the branching factor for all nodes is finite (breadth-first search)
  • The branching factor of a node is the number of its neighbors

⇒ Prolog
'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) ← mother(X,Z) mother(X,Z) not(X = Y) not(X = Y) mother(Y,Z)
```

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

X is not derivable in Prolog given the current instantiation of `X`.

Not in 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
  - rope on Programming in Prolog
  - Prolog's Search Strategy
  - Breadth-first
  - Depth-first
  - Search
• The rope on Prolog

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

• Cycles part of larger problem of endless loops.
  - Cannot detect all endless loops (halting problem).
  - Prolog also doesn't do any cycle checking.
  - Knowledge engineer's job to be careful in defining clauses.

• Would have to keep answer clauses along the current path we are exploring.
  - Could check for cycles: list of atoms in answer clause identical to list earlier up in proof.

When should cycles (since Prolog is depth-first search)?

Cycles in Prolog
Overview

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

Programming Search in Prolog

Prolog's Findall

- \texttt{findall(X, connected(1,X), L)}
  - Returns a list of all X in which \texttt{connected(1,X)} is true
- \texttt{findall} can return a list of any arbitrary structures
  - \texttt{findall(cell(X), connected(Y, X), L)}
    - Assumes \texttt{Y} is already instantiated
  - \texttt{findall(connected(X,Y), connected(X,Y), L)}
- \texttt{findall(X+Y, connected(X,Y), L)}
  + \texttt{+} is just an infix operator that we chose to use
  - Could have used any valid Prolog term, e.g. \texttt{a(X, Y)}

- Semantics of \texttt{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 cannot be named

Maze Example

- Example: path through a maze

- \texttt{connected(1,2)}.
- \texttt{connected(2,3)}.
- \texttt{connected(2,6)}.
- \texttt{connected(4,8)}.
  - \ldots
- \texttt{connectedto(X,Y) :- connected(X,Y).}
- \texttt{connectedto(X,Y) :- connected(Y, X).}

- Book defined breadth-first and depth-first search in Datalog

- Maze Example: path through a maze
Breadth-First Search

• Define breadth-first search using KB

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

\[
\text{search}([\text{Frontier0}|\text{Answer}], \text{Answer}) \leftarrow \text{Frontier0} = [\text{Path}|\text{Frontier1}], \text{Path} = [X|\text{Rest}], \text{findall}([Z,X|\text{Rest}], \text{connected}(Z,X), \text{NN}) \leftarrow \text{changed order for breadth-first search}.
\]

?\text{search}([1], \text{Answer})

Depth-First Search saving Paths

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

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

\[
\text{search}([X|\text{Rest}], \text{Path}) \leftarrow \text{connected}(X,Z), \text{search}([Z,X|\text{Rest}], \text{Path}). \leftarrow \text{building paths on way into recursion}
\]

?\text{search}([1], \text{Answer})

• Define depth-first search using KB

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

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

\[
\text{append}([\text{NN}], \text{Frontier1}, \text{Frontier2}) \leftarrow \text{changed order for depth-first search}.
\]

?\text{search}([1], \text{Answer})

Depth-First Search on Nodes

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

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

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

?\text{search}(1)

- Prolog keeps track of backtracking alternatives automatically and don’t use Prolog’s + Deterministic code: little backtracking in Prolog code

• Define depth-first search using KB

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

\[
\text{search}([\text{Frontier0}], \text{Answer}) \leftarrow \text{Frontier0} = [X|\text{Frontier1}], \text{findall}(Z, \text{connected}(Z,X), \text{NN}) \leftarrow \text{embedded call to Prolog to find all solutions}.
\]

\[
\text{concat}([\text{NN}], \text{Frontier1}, \text{Frontier2}) \leftarrow \text{building paths in findall}.
\]

?\text{search}([1])

- Explicitly keep backtracking alternatives, and don’t use Prolog’s + Deterministic code: little backtracking in Prolog code

Depth-First Search on Nodes
Adding in Cycle Checking

Ensure new cells not already in path

search(Path, Path) ← Path = [1 | Rest].

search([X | Rest], Path) ← connected(X, Z), not(member(Z, [X | Rest])), search([Z | X | Rest], Path).

?search([1], Answer)

search([Path | Rest], Path) ← search(Frontier0, Answer)

search(Frontier2)

neighbor(Z, [X | Rest]) ← connected(Z, X), not(member(Z, [X | Rest]))

?search([[1]], Rest)

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

Cycle Checking

Comments

Should we use Prolog to implement search?
- Where we use Prolog to implement search
- Why we use Prolog to implement search
- Should we use Prolog to implement search

We can
- Prolog is intended as a full programming language
- But can implemented in Mathematica/Maple
- But can implemented in Mathematica/Maple, including pattern matching
- What we end up with is nice and pretty
- But can implemented in Mathematica/Maple
- Prolog is intended as a full programming language
- 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 this with version where frontier is list of cells.

\[
\text{search}([1], [], \text{frontier}).
\]

\[
\text{search}([\text{frontier}0], [\text{seen}0]) \leftarrow \text{frontier0} = [X | \text{frontier1}]
\]

\[
\text{findall}(Z, \text{neighbor}(Z, X, \text{seen}), \text{NN})
\]

\[
\text{append}(\text{frontier1}, \text{NN}, \text{frontier2})
\]

\[
\text{append}([\text{seen}0], \text{NN}, [\text{seen}1])
\]

\[
\text{search}([\text{frontier2}], [\text{seen}1])
\]

\[
\text{neighbor}(Z, X, \text{seen}) \leftarrow \text{connected}(Z, X)
\]

\[
\text{not}((\text{member}(Z, \text{seen})))
\]