Prolog Lists

• A special recursive data structure
  - with special syntax to make dealing with lists simpler

- 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]

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

• Progammable Search in Prolog

• More on Progammable in Prolog

• Prolog’s Search Strategy
  • Depth-first
  • Breadth-first

• The Pro Part of Prolog

Singleton Variables

• ‘ʼ special syntax for naming a variable
  - Used when don’t care about, used for singleton variables

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

  Can be used multiple times in same clause, but each use is really a
different variable.

Append and Member

• Rewrite
  concat(nil,Z,Z)
  concat(p(A,X),Y ,p(A,Z)) ← concat(X,Y ,Z).

• Rewrite

  member(X,List) → member(X,[X|List])

  Rewrite in new list syntax

  member(X,X)
  member(X,[Y|List]) ← member(X,List)
Examples

1. What are nodes and arcs for:
   - Maze example
   - Electrical circuit
   - Office example
   - Feature

Search Graphs

A graph consists of:
- A set of nodes, called the node set: $N$
- A set of ordered pairs of nodes, called the arc set: $A$

A graph consists of:
- A set $N$ of nodes (called vertices)
- A set $A$ of arcs (called edges)

A path is a sequence of nodes $n_1, n_2, ..., n_k$ such that:
- $\langle n_i, n_{i+1} \rangle \in A$

A path is a sequence of nodes $n_1, n_2, ..., n_k$ such that:
- $\langle n_i, n_{i+1} \rangle \in A$

A path is a sequence of nodes $n_1, n_2, ..., n_k$ such that:
- $\langle n_i, n_{i+1} \rangle \in A$

A path is a sequence of nodes $n_1, n_2, ..., n_k$ such that:
- $\langle n_i, n_{i+1} \rangle \in A$

A path is a sequence of nodes $n_1, n_2, ..., n_k$ such that:
- $\langle n_i, n_{i+1} \rangle \in A$

A path is a sequence of nodes $n_1, n_2, ..., n_k$ such that:
- $\langle n_i, n_{i+1} \rangle \in A$

A path is a sequence of nodes $n_1, n_2, ..., n_k$ such that:
- $\langle n_i, n_{i+1} \rangle \in A$

A path is a sequence of nodes $n_1, n_2, ..., n_k$ such that:
- $\langle n_i, n_{i+1} \rangle \in A$

A path is a sequence of nodes $n_1, n_2, ..., n_k$ such that:
- $\langle n_i, n_{i+1} \rangle \in A$

Search

To convert proof procedure into a reasoning procedure, need to:
- Resolve non-determinism
- Search

Search

To convert proof procedure into a reasoning procedure, need to:
- Resolve non-determinism
- Search

Search

To convert proof procedure into a reasoning procedure, need to:
- Resolve non-determinism
- Search
Graph Searching

- Generic search algorithm: given a graph, start nodes, and go al
- As search proceeds, the frontier expands into the unexplored nodes. Maintain a frontier of paths from the start node that have been explored. Explore new nodes incrementally explore paths from the start nodes and goal nodes.
- Graph Search Algorithm: Given a graph, start nodes, and goal nodes.

Search Graph for Resolution (Top-Down) Proof

\( p \lor q \rightarrow r \)
\( p \rightarrow m \rightarrow q \)
\( p \lor r \rightarrow m \lor q \)
\( p \lor q \rightarrow m \lor r \)
\( p \rightarrow q \lor p \lor r \)
\( p \rightarrow q \lor r \)
\( p \rightarrow q \lor p \lor r \)

(dummy clinical knowledge sign)

Illustration of Graph Searching
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

\[
\text{search}(F_0) \leftarrow \text{select}(\text{Node}, F_0, F_1) \land \text{is goal}(\text{Node}).
\]

\[
\text{search}(F_0) \leftarrow \text{select}(\text{Node}, F_0, F_1) \land \text{neighbors}(\text{Node}, NN) \land \text{add to frontier}(NN, F_1, F_2) \land \text{search}(F_2).
\]

- **Definition of predicates:**
  - `select(N,F0,F1)` means `N` ∈ `F0` and `F1 = F0 - {N}`. Fails if `F0` is empty.
  - `add to frontier(NN,F1,F2)` means `F2 = F1 ∪ NN`.
  - `is goal(N)` defines what is a solution.
  - `neighbors(N,NN)` defines the graph.
  - `select(Node,F0,F1)` defines the search strategy.
  - Whether `add to frontier` puts new elements on top or bottom of list.

As this can be easily converted to procedural code, could also be easily done within a functional procedure. Which is in practice, we will be using this to implement a database.

We wrote this in datalog, which is a bit perverse, as we will be using this to implement a database. Could just as easily have written it in Tcl or Python as this can be easily converted to procedural code.

We wrote this in datalog, which is a bit perverse, as we will be using this to implement a database.
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 until it happens to stumble on the goal. Depth-first search treats the frontier as a stack: it always selects the last element added to the frontier. Its neighbors are added to the front of the stack. The last element added to the frontier is always selected. Depth-first search is only succeeds when all paths from the root have been explored. Depth-first search is not constrained by the goal until it happens to stumble on the goal.
Breadth-first search treats the frontier as a queue: it always selects the earliest element added to the frontier.

```prolog
select(Node, [Node | Frontier], Frontier).
add_to_frontier(Neighbors, Frontier1, Frontier)
← concat(Frontier1, Neighbors, Frontier2).
```

- The earliest element added to the frontier is always selected next.
- Selects the earliest element added to the end of the queue.

Breadth-first search always adds 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
  - Can search through KB without unification
  - Knowledge engineer can order KB to help constrained search
  - Knowledge engineer can order KB to help constrained search
  - Always resolve first atom of answer clause first

Prolog

Overview

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

Complexity of Breadth-first Search

- Search is unconstrained by the goal
- The space complexity is exponential in path length
- The branching factor is polynomial
- The space complexity is exponential in the path length, \( b^n \), where \( b \) is the branching factor to find a solution if one exists
- If the branching factor for all nodes is finite, breadth-first search is guaranteed to find a solution if one exists
- The branching factor of a node is the number of its neighbors

Prolog

Overview

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

Complexity of Breadth-first Search

- Search is unconstrained by the goal
- The space complexity is exponential in path length
- The branching factor is polynomial
- The space complexity is exponential in the path length, \( b^n \), where \( b \) is the branching factor to find a solution if one exists
- If the branching factor for all nodes is finite, breadth-first search is guaranteed to find a solution if one exists
- The branching factor of a node is the number of its neighbors
'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)
mother(Y,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
  - Depends on the context and the order of clauses

- Truth does not correlate with semantics of models

X is derivable in Prolog given the current instantiation of X.

The operator `not(X)` means that `-X` is not derivable in Prolog given the current instantiation of `X`.

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

Programming Search in Prolog

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

Overview

- Know what meta-programming, logic programming, and functional programming are
- Know what data structures are important for cycle detection
- Cycles part of larger problem of endless loops
  - Would have to keep answer clauses along the current path we are exploring
  - Could track for cycles: list of atoms in answer clause identical to list earlier up in proof
  - Cycles part of larger problem of endless loops

- Why should Prolog (since Prolog is depth-first search)?
Overview

- More on Programming in Prolog
- Prolog's Search Strategy
  - Breadth-first
  - Depth-first
  - Search

The "Pro" Part of Prolog

Prolog's Findall

\[ \text{findall}(X, \text{connected}(1, X), L) \]

- Returns a list of all \( X \) in which \( \text{connected}(1, X) \) is true

\[ \text{findall}(\text{cell}(X), \text{connected}(Y, X), L) \]

- \text{findall} can return a list of any arbitrary structures

\[ \text{findall}(\text{connected}(X, Y), \text{connected}(X, Y), L) \]

\[ \text{findall}(X + Y, \text{connected}(X, Y), L) \]

+ is just an infix operator that we chose to use

\[ \text{findall}(\text{connected}(X), L) \]

- Requires a list of all \( X \) in which \( \text{connected}(X) \) is true

Maze Example

Example: path through a maze
Breadth-First Search

• Define breadth-first search using KB
  search([Path|Path|Path],Path)
  ← Path = [16|Path].
  search(Frontier0,Answer)
  ← Frontier0 = [Path|Frontier1], Path = [X|Rest],
  findall([Z,X|Rest],connected(Z,X),NN)
  append(Frontier1,NN,Frontier2)
  search(Frontier2,Answer)

?search([1],Answer)

• Should we use Prolog to implement search?
  - We can, since Prolog is intended as a full programming language
  - But, can implement it in software or other languages
  + Might make it more clear what logic programming is, if we don't try to do
    everything in Prolog

Depth-First Search

• Define depth-first search using KB
  search([Path|Path|Path],Path)
  ← Path = [16|Path].
  search(Frontier0,Answer)
  ← Frontier0 = [Path|Frontier1], Path = [X|Rest],
  findall([Z,X|Rest],connected(Z,X),NN)
  append(NN,Frontier1,Frontier2)
  search(Frontier2,Answer)

?search([1],Answer)

• Define depth-first search saving Paths
  search(16).
  search(X) ← connected(X,Z), search(Z).

?search(1)

- Prolog keeps track of backtracking alternatives automatically

• Define depth-first search on Nodes
  search(Frontier0)
  ← Frontier0 = [X|Frontier1], findall(Z,connected(Z,X),NN),
  calls Prolog to find all solutions
  append(NN,Frontier1,Frontier2), search(Frontier2).

?search([1])

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

- Ensure new cells not already in path

Cycle Checking

- Depth-first search of Maze can easily get stuck in cycle
  - If we keep the paths
  - Change our definition of path

Another Cycle Checker

- Ensure that any cell is just visited once
  - For simplicity, done this with version where frontier is list of cells
  - If frontier is a 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)

- Ensure new cells not already in path