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’ Stands for Programming

• Prolog is intended as a programming language, in which programs are written as theorems, and program execution is theorem proving
• Tension in Prolog between its role as
  - A logic specification
  - A general purpose programming language
• Has the built-in \texttt{is(X,Expr)} command
  - \texttt{Expr} evaluated using normal rules for expressions
  - Semantics for \texttt{is} are messy, as it only has a value if \texttt{Expr} is ground
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]?

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`

* `member` is a built-in predicate. If you want to define it yourself, use another name
Singleton Variables

- `\_` special syntax for naming a variable
  - Used when don’t care about, used for singleton variables
    
    ```prolog
    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

Overview

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

- To convert proof procedure into a reasoning procedure, need to resolve non-determinism
- **Search** is a way to implement nondeterminism
- As Prolog is a programming language, can program search in it
  - Can even write a prolog interpreter in Prolog
  - Can even write an alternate interpreter as well
    + One that might use heuristics to pick which atom in the body of the answer clause to work on
    + One that uses breadth-first search rather than depth-first search
  - Or can do this in python

Search Graphs

- A graph consists of
  - a set $N$ of nodes
  - a set $A$ of ordered pairs of nodes, called arcs
- Node $n_2$ is a neighbor of $n_1$ if there is an arc from $n_1$ to $n_2$
  - if $<n_1, n_2> \in A$
- A path is a sequence of nodes $n_0, n_1, ..., n_k$ such that
  $<n_{i-1}, n_i> \in A$
- Given a set of start nodes and goal nodes, a solution is a path from a start node to a goal node
Examples

- What are nodes and arcs for
  - Maze
  - Office example
  - Electrical circuit
  - Top-Down Theorem Proving

Search Graph for Resolution (Top-Down) Proof

(From Official Lecture slides)

\[
\begin{align*}
\text{a} & \leftarrow \text{b} \land \text{c}. & \text{a} & \leftarrow \text{g}. & \text{yes} & \leftarrow \text{b} \land \text{c} \land \text{d}. & \text{yes} & \leftarrow \text{g} \land \text{d}. & \text{yes} & \leftarrow \text{h} \land \text{d}. \\
\text{a} & \leftarrow \text{h}. & \text{b} & \leftarrow \text{j}. & \text{yes} & \leftarrow \text{f} \land \text{c} \land \text{d}. & \text{yes} & \leftarrow \text{m} \land \text{d}. & \text{yes} & \leftarrow \text{m} \land \text{d}. \\
\text{b} & \leftarrow \text{k}. & \text{d} & \leftarrow \text{m}. & \text{yes} & \leftarrow \text{f} \land \text{c} \land \text{d}. & \text{yes} & \leftarrow \text{p} \land \text{d}. \\
\text{d} & \leftarrow \text{p}. & \text{f} & \leftarrow \text{m}. & \text{yes} & \leftarrow \text{m} \land \text{d}. & \text{yes} & \leftarrow \text{p} \land \text{d}. \\
\text{f} & \leftarrow \text{p}. & \text{g} & \leftarrow \text{m}. & \text{yes} & \leftarrow \text{m} \land \text{c} \land \text{d}. & \text{yes} & \leftarrow \text{d}. \\
\text{g} & \leftarrow \text{f}. & \text{k} & \leftarrow \text{m}. & \text{yes} & \leftarrow \text{m} \land \text{c} \land \text{d}. \\
\text{h} & \leftarrow \text{m}. & \text{p}. & \text{yes} & \leftarrow \text{m} \land \text{d}. & \text{yes} & \leftarrow \text{p}. \\
? & \text{a} \land \text{d}.
\end{align*}
\]
Graph Searching

• Generic search algorithm: given a graph, start nodes, and goal 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 until a goal node is encountered

• The way in which the frontier is expanded defines the search strategy
Generic Graph Search Algorithm

\[
\text{search}(F0) \leftarrow \\
\quad \text{select}(\text{Node}, F0, F1) \land \\
\quad \text{is\_goal}(\text{Node}). \\
\text{search}(F0) \leftarrow \\
\quad \text{select}(\text{Node}, F0, F1) \land \\
\quad \text{neighbors}(\text{Node}, NN) \land \\
\quad \text{add\_to\_frontier}(NN, F1, F2) \land \\
\quad \text{search}(F2). \\
\]

- Definition of predicates:
  + \text{search(Frontier)} is true if path from element of Frontier to a goal node
  + \text{is\_goal}(N) is true if \( N \) is a goal node
  + \text{neighbors}(N, NN) means \( NN \) is list of neighbors of \( N \)
  + \text{select}(N, F0, F1) means \( N \in F0 \) and \( F1 = F0 - \{N\} \). Fails if \( F0 \) is empty
  + \text{add\_to\_frontier}(NN, F1, F2) means that \( F2 = F1 \cup NN \)

Summary of Generic Search Algorithm

- \text{select} and \text{add\_to\_frontier} define the search strategy
  - Whether \text{add\_to\_frontier} puts new elements on top or bottom of list matters
- \text{neighbors} defines the graph
- \text{is\_goal} defines what is a solution

- We wrote this in datalog
  - Which is a bit perverse as we will be using this to implement a datalog top-down reasoning procedure
- Could just as easily do written it in Python
  - Code is deterministic, so can easily be converted to procedural code
Depth-first Search

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

\[
\text{select(Node,[Node|Frontier],Frontier).} \\
\text{add_to_frontier(Neighbors,Frontier1,Frontier) \leftarrow} \\
\text{concat(Neighbors,Frontier1,Frontier2).}
\]

- Frontier: \([e_1,e_2,...]\)
  - \(e_1\) is selected. Its neighbors are added to the front of the stack
  - \(e_2\) is only selected when all paths from \(e_1\) have been explored
• 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
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).
add_to_frontier(Neighbors,Frontier1,Frontier) ←
concat(Frontier1,Neighbors,Frontier2).

• Frontier: [e₁,e₂,...]
  - e₁ is selected. Its neighbors are added to the end of the queue
  - e₂ is selected next
Complexity of Breadth-first Search

- The branching factor of a node is the number of its neighbors.
- If the branching factor for all nodes is finite, breadth-first search is guaranteed to find a solution if one exists.
  - It is guaranteed to find the path with fewest arcs.
- Time complexity is exponential in the path length: $b^n$, where $b$ is branching factor, $n$ is path length.
- The space complexity is exponential in path length: $\delta^n$
- Search is unconstrained by the goal.
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
Cycles in Prolog

• What about cycles (since Prolog is depth-first search)?
  \[\text{connected}(X, Y) \leftarrow \text{connected}(Y, X)\]
  - 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

* There are other theorem provers, and some might use heuristics to pick which atom in answer clause to work on, and which rule from KB to pick, and might limit the depth of a search

Overview

• The ‘Pro’ Part of Prolog
• Search
• Depth-first
• Breadth-first
• Prolog’s Search Strategy
  ⇒ More on Programming in Prolog
• Programming Search in Prolog
‘Not’ in Prolog

- The operator \texttt{not(X)} means that
  - \texttt{X} is not derivable in Prolog given the current instantiation of \texttt{X}
  - Prolog does an embedded proof of \texttt{X}
    - If it fails on the embedded proof, \texttt{not(X)} succeeds
    - If it succeeds, \texttt{not(X)} fails
    - Backtracking over \texttt{not(X)} does not change its value

- 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
  - Following two definitions are not equivalent
    
    \begin{align*}
    \texttt{brother}(X,Y) & \leftarrow \\
    \texttt{mother}(X,Z) & \leftarrow \\
    \texttt{mother}(Y,Z) & \leftarrow \\
    \texttt{not}(X = Y) & \leftarrow
    \end{align*}


Maze Example

- Example: path through a maze
  
  \[
  \begin{array}{cccc}
  1 & 2 & 3 & 4 \\
  5 & 6 & 7 & 8 \\
  9 & 10 & 11 & 12 \\
  13 & 14 & 15 & 16 \\
  \end{array}
  \]

- Book defined breadth-first and depth-first search in Datalog
  - In this example, searching for end point

- But how can we define the \texttt{neighbors} clause that is needed?
  - Could define \texttt{neighbors} as primitive
  - Or, define \texttt{neighbors}(X,Set) in terms of \texttt{connected}(X,Y)
    - No way to do this in Datalog: \texttt{Set} = all \texttt{Y} s.t. \texttt{connected}(X,Y) is true
Prolog’s Findall

- `findall(X, connected(1,X), L)
  - L = [X s.t. connected(1,X)]`

- `findall` can return a list of any arbitrary structures
  - `findall(cell(X), connected(Y,X), L)`  
    - Assume Y is already instantiated
  - `findall(connected(X,Y), connected(X,Y), L)`
  - `findall(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. `X/Y`, `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
    - Rather than on objects in the domain, as first order predicate defines it

Overview

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

- Define path using KB and use Prolog’s depth-first search
  ```prolog
  search(16).
  search(X) ← connected(X, Z), search(Z).
  ?search(1)
  ```
- Prolog keeps track of backtracking alternatives automatically

- Define depth-first search using KB
  ```prolog
  search([1|Path], Path) ← Path = [1|Path].
  search(Frontier0) ← Frontier0 = [X|Frontier1],
  findall(Z, connected(Z, X), NN),
  search([Z|X|Rest], Path),
  search(Frontier2).
  ?search([1|Path], Path).
  ```
- Explicitly keep backtracking alternatives, and don’t use Prolog’s
  + Deterministic code: little backtracking in prolog code

Depth-First Search saving Paths

- Define path using KB and use Prolog’s depth-first search
  ```prolog
  search(Path, Path) ← Path = [1|Path].
  search([X|Rest], Path) ← connected(X, Z), search([Z|X|Rest], Path).  
  ?search([1], Answer)
  ```

- Define depth-first search using KB
  ```prolog
  search(Path, Path) ← Path = [1|Path].
  search(Frontier0, Answer) ← Frontier0 = [Path|Frontier1],
  Path = [X|Rest],
  findall([Z|X|Rest], connected(Z, X), NN),
  search([Z|X|Rest], Path),
  findall([Z|X|Rest], connected(Z, X), NN),
  append(=[NN|Frontier1, Frontier2),
  search(Frontier2, Answer).
  ?search([1], Answer)
  ```
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 intended as a full programming language
- But, can implement it in anything, including Python
  - Might make it more clear what logic programming is, if we don’t try to do everything in it
  - `findall` could still be call to a theorem prover

Breadth-First Search

- Define breadth-first search using KB
  ```prolog
  search([Path|_], Path) ←
  Path = [1|_].
  search(Frontier0, Answer) ←
      Frontier0 = [Path|Frontier1],
      Path = [X|Rest],
      findall([Z,X|Rest], connected(Z,X,NN)
      concat(Frontier1, NN, Frontier2) \= changed order for breadth-first search
      search(Frontier2, Answer)
  ?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
  - Change our definition of path
    - Path is a list of nodes, each connected to the next one, but where any node only occurs once

Adding in Cycle Checking

- Ensure new cells not already in path

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

search([Path], Path) ←
Path = [1|].
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)
```
Another Cycle Checker

- Ensure that any cell is just visited once
  - For simplicity, doing this with version where frontier is list of cells
  - Can not do this in version that uses Prolog’s backtracking

```prolog
search([],_).
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],[])```