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 \( n_2 \)
  - \( \langle n_1, n_2 \rangle \in A \)

• A path is a sequence of nodes \( n_0, n_1, ..., n_k \) such that
  - \( \langle n_i, n_{i+1} \rangle \in A \)

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

Search

• To convert proof procedure into a reasoning procedure, need to
  - resolve non-determinism

Search is a way to implement non-determinism in

To convert proof procedure into a reasoning procedure, need to

Programming Search in Prolog

- The proof part of Prolog
- Prolog's search strategy
- Backtracking
- Depth-first
**Generic search algorithm:**

- Given a graph, start nodes, and go to 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. Incrementally explore paths from the start nodes.
- The way in which the frontier is expanded defines the **search strategy**.

---

**Search Graph for Resolution (Top-Down) Proof**

(from Official Lecture slides)

- $a \leftarrow b \land c$
- $a \leftarrow g$
- $a \leftarrow h$
- $b \leftarrow j$
- $b \leftarrow k$
- $d \leftarrow m$
- $d \leftarrow p$
- $f \leftarrow m$
- $f \leftarrow p$
- $g \leftarrow m$
- $g \leftarrow f$
- $k \leftarrow m$
- $h \leftarrow m$
- $p$

**Examples**

- Maze
- Office example
- Top-down theorem proving
Summary of Generic Search Algorithm

- select a new element to add to the frontier
  - Define the search strategy
    - Whether to add to the top or bottom of the list matters
  - neighbors defines the graph
  - is goal defines what is a solution

We wrote this in Datalog—Which is a bit perverse because we will be using this to implement a top-down reasoning procedure.

Could just as easily do it in Tcl or Python as this can be easily converted to procedural code.

Generic Graph Search Algorithm

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

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

- Definition of predicates:
  - \( \text{search}(F) \) is true if path from element of Frontier to 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 \( N \)
  - \( \text{select}(N, F_0, F_1) \) means \( N \in F_0 \) and \( F_1 = F_0 - \{ N \} \). Fails if \( F_0 \) is empty.
  - \( \text{add to frontier}(NN, F_1, F_2) \) means that \( F_2 = F_1 \cup NN \).

Illustration of Graph Searching
Depth-first Search

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

select(Node, p(Node, Frontier), Frontier).

add to frontier(Neighbors, Frontier1, Frontier) ← append(Neighbors, Frontier1, Frontier2).

- Frontier: p(e1, p(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.

Illustration of Depth-first Search
Breadth-first Search

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

```prolog
select(Node, p(Node, Frontier), Frontier).
```

- Add to frontier(
  Neighbors, Frontier1, Frontier).

    ```prolog
    add to frontier(Neighbors, Frontier1, Frontier) ←
    append(Frontier1, Neighbors, Frontier2).
    ```

- Frontier: p(e1, p(e2, ...))
  - e1 is selected. Its neighbors are added to the end of the queue
  - e2 is selected next

Breadth-first search needs the frontier as a queue; it always selects the earliest element added to the frontier.

Overview

- Search
- Depth-first
  - Breadth-first

Complexity of Depth-first Search

- Depth-first search is unbounded by the goal until it happens to stumble on the goal.
- The space complexity is linear in the size of the path between the start node and the goal.
Overview

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

• The 'Pro' part of Prolog

Programming Search in Prolog

Complexity of Breadth-first Search

• The branching factor of a node is the number of its neighbors
• If the branching factor is exponential in path length, the time complexity is exponential in the path length, where b is the branching factor and n is the path length.
• The space complexity is exponential in the path length.

Illustration of Breadth-first Search
Overview

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

⇒ The 'Pro' part of Prolog

• Programming Search in Prolog

Cycles in Prolog

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

connected(X, Y) ← connected(Y, X)

- Could check for cycles: list of atoms in answer clause identical to list earlier up in proof
- Would need to keep answer clauses since the current path we are
currently up to proof
- Could check for cycles: the atoms in answer clause referenced to the
connected(X) ← connected(Y)

When should cycles (since Prolog is depth-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 ← b ∧ c ∧ d.

+ Knowledge engineer can order clauses to ensure solution is found quickly

- Search over space of answer clauses, not over space of proofs

+ Knowledge engineer can order clauses in KB

- Search is directed by order of clauses in KB

- Can search through facts using depth-first search
Append and Member

• Rewrite append in new list syntax
  \[ \text{append(nil, Z, Z)} \]
  \[ \text{append(p(A, X), Y, p(A, Z)) ↔ append(X, Y, Z).} \]

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

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 
  \[[1, 2, 6, 10]\]
  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

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

Maze Example

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
</tr>
</tbody>
</table>

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

- In this example, searching for end point
- But how can we define the neighbors clause that is needed?
- Could define neighbors as primitive
- Or, define neighbors(X,Set) in terms of connected(X,Y)

No way to do this in Datalog:
Set = all Y such that connected(X,Y)

Not in Prolog

- Truth does not correlate with semantics of models
- Inference depends on where it is in the body of a clause
- Semantics are not very clean
- Truth does not correlate with semantics of models
- The operator not(X) means that X is not derivable in Prolog from the current instantion of X
- Two definitions are not equivalent

Singleton Variables

'\_' 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
Depth-First Search on Nodes

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

```
search(X) ← connected(X,Z), search(Z).
```

?search(1)

Prolog keeps track of backtracking alternatives automatically.

• Define depth-first search using KB.

```
search([1|F]) ← Frontier0 = [X|Frontier1], findall(Z, connected(Z,X), NN), append(NN, Frontier1, Frontier2), search(Frontier2).
```

?search([1])

Explicitly keep backtracking alternatives, and don't use Prolog's depth-first search.

Overview

• Search
• Depth-first
• Breadth-first
• Prolog's Search Strategy
• The 'Pro' part of Prolog

Programming Search in Prolog

The Pro part of Prolog:

Prolog's Search Strategy:

• Depth-first

Semantics of `findall` are messy:

`findall(X, connected(1, X), L)` returns a list of all X in which `connected(1, X)` is true.

`findall(cell(X), connected(Y, X), L)` assumes 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. `a(X, Y)`.

Semantics of `findall` are messy:

- Requires universal quantification, which is not part of Prolog.
- Semantics are dependent on what can be named.
- Bounded universal quantification works, but only part of Prolog.

Structure of `findall` is messy:

- Requires a depth-first search.
- Version of Prolog used.
- Structure is dependent on what can be named.
- Requires a first list in which `connected(X, Y)` is true.
- Structure is dependent on what can be named.

Prolog's `findall`
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
  - Change our definition of path
    - A path is a list of nodes, each connected to the next one, but where any node only occurs once

Breadth-First Search

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

- Should we use Prolog to implement search?
  - We can, since Prolog is intended as a full programming language
  - But, can implement it in anything, including Tcl/Tk

Depth-First Search saving Paths

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

  `search([Path | ], Path) ← Path = X | Rest, search([X | Rest], Answer) ← connected(X, Z),
  search([Z, X | Rest], Answer).`
Another Cycle Checker

Ensure that any cell is just visited once

Adding in Cycle Checking

Ensure new cells not already in path

Chrome 06: 35-6 of 34

© P. Heeman, 2010

• Another Cycle Checker

• Adding in Cycle Checking

Chrome 06: 35-6 of 34

© P. Heeman, 2010