Overview

⇒ Search
- Depth-first
- Breadth-first
- Prolog's Search Strategy
- The 'Pro' part of Prolog
- Programming Search in Prolog

Search

- To convert proof procedure into a reasoning procedure, need to resolve non-determism
- **Search** is a way to implement nondeterminism

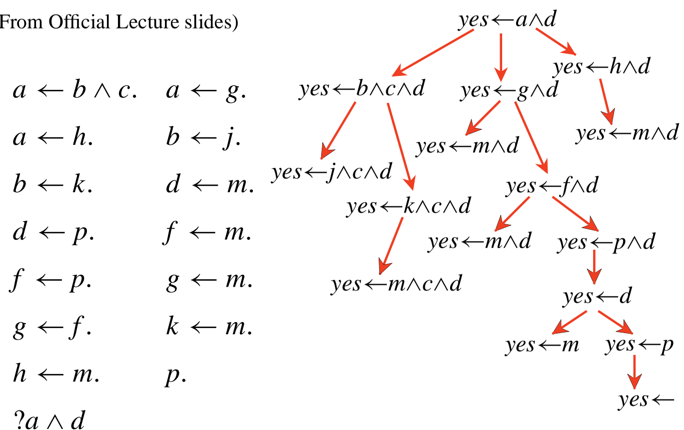
Search Graphs

- 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

Search Graph for Resolution (Top-Down) Proof

(From Official Lecture slides)

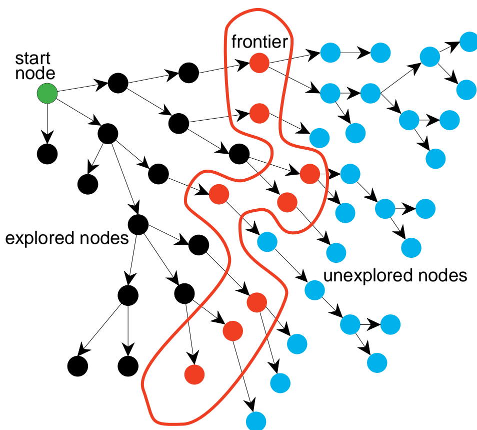
$a \leftarrow b \wedge c.$	$a \leftarrow g.$
$a \leftarrow h.$	$b \leftarrow j.$
$b \leftarrow k.$	$d \leftarrow m.$
$d \leftarrow p.$	$f \leftarrow m.$
$f \leftarrow p.$	$g \leftarrow m.$
$g \leftarrow f.$	$k \leftarrow m.$
$h \leftarrow m.$	$p.$
$?a \wedge d$	



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*

Illustration of Graph Searching



Generic Graph Search Algorithm

search(F0) ←
  select(Node,F0,F1) ∧
  is_goal(Node).
search(F0) ←
  select(Node,F0,F1) ∧
  neighbors(Node,NN) ∧
  add_to_frontier(NN,F1,F2) ∧
  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 N is a goal node
  + *neighbors(N,NN)* means *NN* is list of neighbors of N
  + *select(N,F0,F1)* means $N \in F0$ and $F1 = F0 - \{N\}$. Fails if $F0$ is empty
  + *add_to_frontier(NN,F1,F2)* means that $F2 = F1 \cup NN$

Summary of Generic Search Algorithm

- *select* and *add_to_frontier* define the search strategy
  - Whether *add_to_frontier* puts new elements on top or bottom of list matters
- *neighbors* defines the graph
- *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 Tcl as this can be easily converted to procedural code

Overview

- Search

⇒ Depth-first
- Breadth-first
- Prolog's Search Strategy
- The 'Pro' part of Prolog
- Programming Search in Prolog

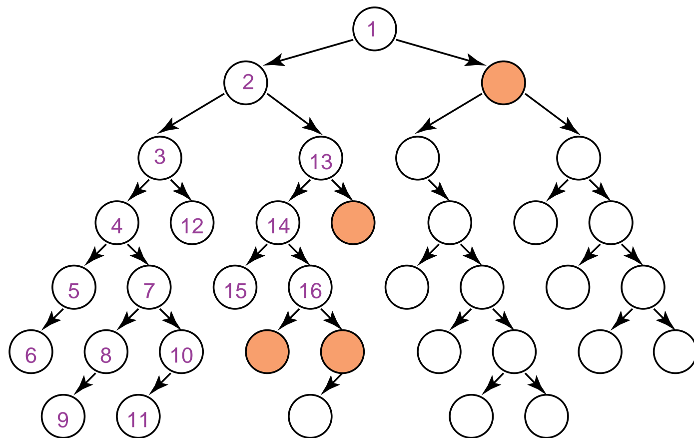
Depth-first Search

- 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



Complexity of Depth-first Search

- Depth-first search isnt 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

 © P. Heeman, 2010

Overview

- Search
- Depth-first

⇒ Breadth-first

- Prolog's Search Strategy
- The 'Pro' part of Prolog
- Programming Search in Prolog

 © P. Heeman, 2010

Breadth-first Search

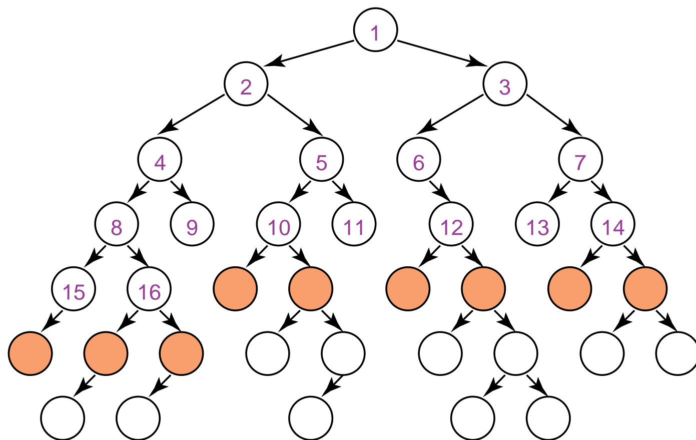
- Breadth-first search treats the frontier as a queue: it always selects the earliest element added to the frontier

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

 © P. Heeman, 2010

Illustration of Breadth-first Search



 © P. Heeman, 2010

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: b^n
- Search is unconstrained by the goal

 © P. Heeman, 2010

Overview

- Search
- Depth-first
- Breadth-first

⇒ Prolog's Search Strategy

- The 'Pro' part of Prolog
- Programming Search in Prolog

 © P. Heeman, 2010

Top-Down Resolution

- Need search strategy for doing top-down resolution
  - Always resolve first atom of answer clause first
  - 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

⇒ 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 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

Overview

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

⇒ The 'Pro' part of 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
- 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

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 $[\mathit{Top}|\mathit{Rest}]$
- 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 *append* in new list syntax

```
append(nil,Z,Z)
append(p(A,X),Y,p(A,Z)) ←
    append(X,Y,Z).
```

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

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 varaible

‘Not’ in Prolog

- The operator *not(X)* means that
  - X is not derivable in Prolog given the current instantion 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 order that clauses are evaluated
  - Truth does not correlate with semantics of models

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).
```

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

- 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 s.t. *connected(X,Y)* is true

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) 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. $a(X, Y)$, $[X, Y]$
- 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

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

⇒ Programming Search in Prolog

Depth-First Search on Nodes

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

```
search(16).
search(X) ←
        connected(X,Z),
        search(Z).
?search(1)
```

  - Prolog keeps track of backtracking alternatives automatically

- Define depth-first search using KB

```
search([16|_]).
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

Depth-First Search saving Paths

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

```
search(Path,Path) ←
        Path = [16|_].
search([X|Rest],Answer) ←
        connected(X,Z),
        search([Z,X|Rest],Answer).
?search([1],Answer)
```

- Define depth-first search using KB

```
search([Path|_],Path) ←
        Path = [16|_].
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)
```

 © P. Heeman, 2010

Breadth-First Search

- Define breadth-first search using KB

```
search([Path|_],Path) ←
        Path = [16|_].
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 anything, including Tcl/Tk
    - + Might make it more clear what logic programming is, if we don't try to do everything in it
    - + findall would be call to a theorem prover

 © P. Heeman, 2010

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

Adding in Cycle Checking

- Ensure new cells not already in path

```
search([Path|_],Path) ←
        Path = [16|_].
search(Frontier0,Answer) ←
        Frontier0 = [Path|Frontier1],
        Path = [X|Rest],
        findall([Z,X|Rest],neighbor(Z,[X|Rest]),NN)
        append(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

```
search([16|_],_).
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],[])
```