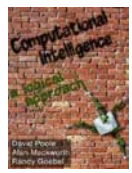
Chapter 4 (part 1): Searching

- Why Search? (4.1)
- Graph Searching (4.2)
- Generic Graph Search Algorithm (4.3)
- Blind Search Strategies (4.4)
  - Depth-First Search
  - Breadth-First Search
  - Lowest-Cost-First Search
- Summary
- Applications



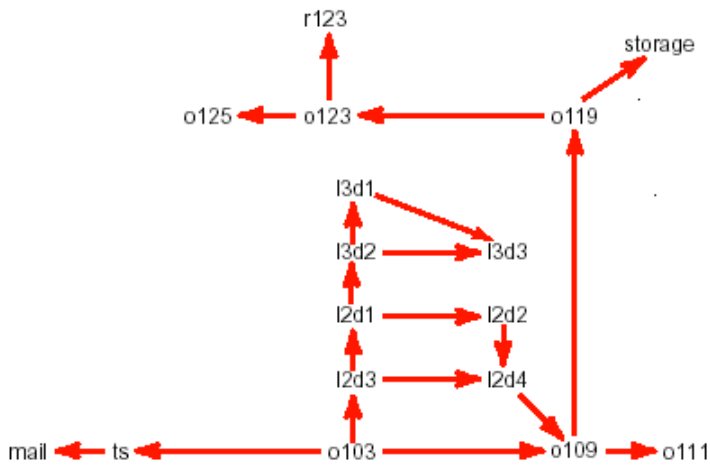
D. Poole, A. Mackworth, and R. Goebel, *Computational Intelligence: A Logical Approach*, Oxford University Press, January 1998

Why search (4.1)

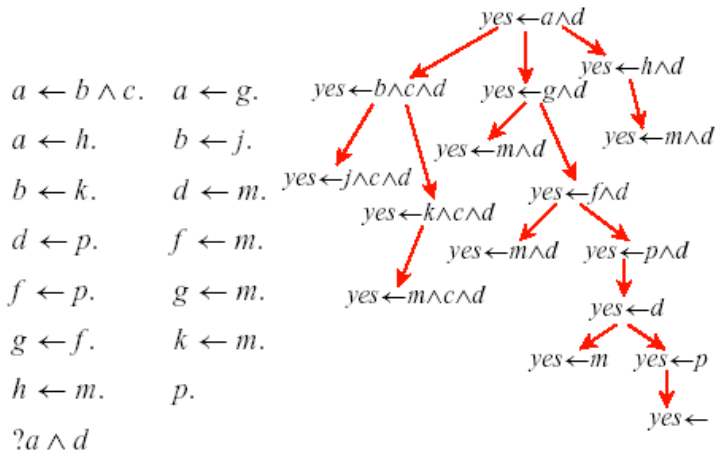
- Often we are not given an algorithm to solve a problem, but only a specification of what is a solution —we have to search for a solution.
- Search is a way to implement “don’t know nondeterminism” (choose all alternative paths…).
- So far we have seen how to convert a semantic problem of finding logical consequence to a search problem of finding derivations (syntactic problem).

Why search (4.1) (cont.)

- Search Graphs
  - A graph consists of a set N of nodes and 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. This is, $\langle n_1, n_2 \rangle \in A$.
  - A path is a sequence of nodes $n_0, n_1, \ldots, n_k$ such that $\langle n_{i-1}, n_i \rangle \in A$.
  - Given a set of start nodes and goal nodes, a solution is a path from a start node to a goal node.



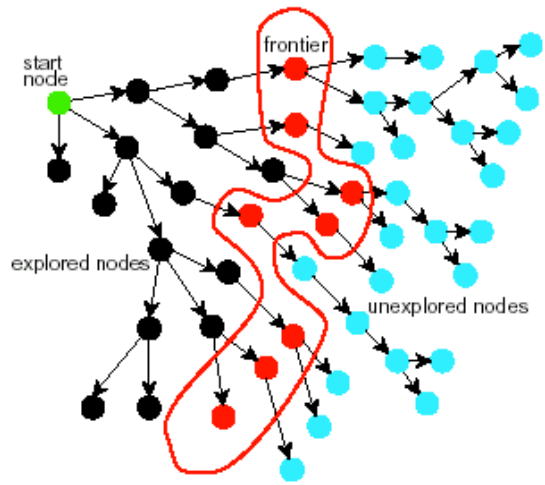
Example of Graph for the Delivery Robot



Search Graph for SLD Resolution

Graph Searching (4.2)

- 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.



Problem Solving by Graph Searching

Generic Graph Search Algorithm (4.3)

Search(F_0) ←
  Select(Node, F_0, F_1) ∧
  is_goal(Node).
Search(F_0) ←
  Select(Node, F_0, F_1) ∧
  Neighbors(Node, NN) ∧
  add_to_frontier(NN, F_1, F_2) ∧
  Search(F_2).

Generic Graph Search Algorithm (4.3) (cont.)

- *Search(Frontier)* is true if there is a path from one element of the 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, F_0, F_1)* means $N \in F_0$ and $F_1 = F_0 - \{N\}$. fails if F_0 is empty.
- *add_to_frontier(NN, F1, F2)* means that $F_2 = F_1 \cup NN$.

select and add_to_frontier define the search strategy.

neighbors defines the graph

is_goal defines what is a solution.

They do not take into account where the goal is: when they stumble on a goal, they report success.

Blind Search Strategies (4.4)

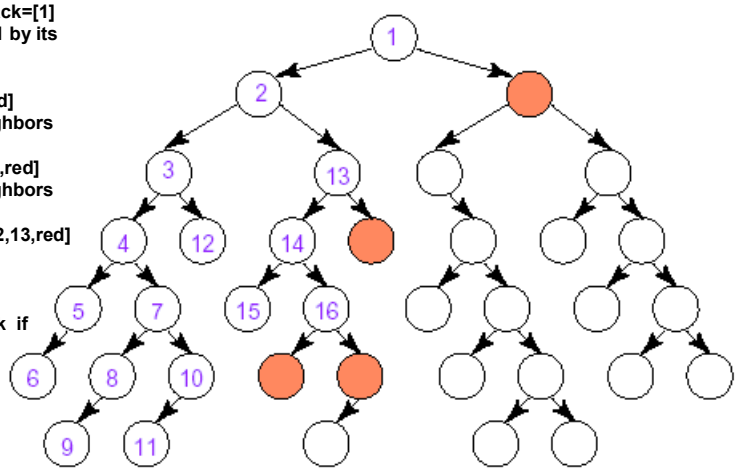
Depth-first Search

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

  Select(Node, [Node|Frontier], Frontier).

  add_to_frontier(Neighbors, $Frontier_1$, $Frontier_2$) ←
  Append(Neighbors, $Frontier_1$, $Frontier_2$).
- 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.

Initially: Frontier_stack=[1]
Next stage: replace 1 by its neighbors
(1 is removed)
Frontier_stack=[2,red]
Replace 2 by its neighbors
(2 is removed)
Frontier_stack=[3,13,red]
Replace 3 by its neighbors
(3 is removed)
Frontier_stack= [4,12,13,red]
.
.
.
Backtrack from stack if failure….



Illustrative Graph — Depth-first Search

Blind Search Strategies (4.4) (cont.)

- 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.

Blind Search Strategies (4.4) (cont.)

Breadth-first Search

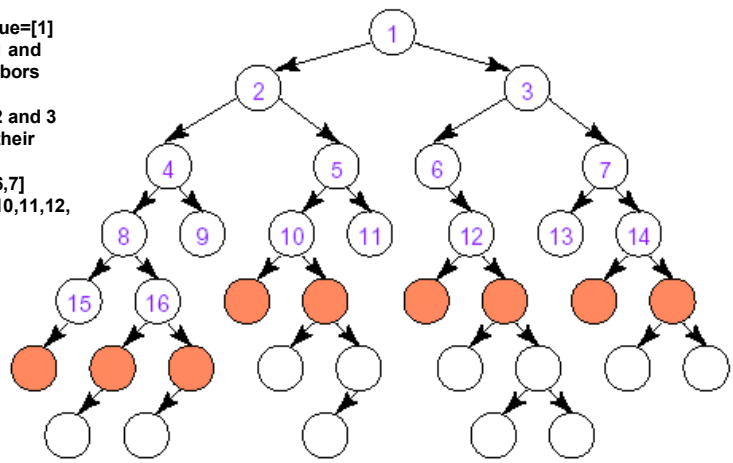
- Breadth-first search treats the frontier as a queue: it always selects the earliest element added to the frontier.

  Select(Node, [Node|Frontier], Frontier).

  add_to_frontier(Neighbors, $Frontier_1$, $Frontier_2$)

  ← Append($Frontier_1$, Neighbors, $Frontier_2$)
- Frontier: [e_1, e_2, …]
- e_1 is selected. Its neighbors are added to the end of the queue.
- e_2 is selected next.

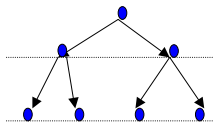
Initially: Frontier_queue=[1]
Next stage: Remove 1 and replace it by its neighbors
Frontier_queue=[2,3]
Next stage: Remove 2 and 3 and replace them by their neighbors
Frontier_queue=[4,5,6,7]
Frontier_queue=[8,9,10,11,12, 13,14]
.
.
.



Illustrative Graph — Breadth-first Search

Blind Search Strategies (4.4) (cont.)

- 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 (not necessarily an optimal policy !!!).
  - Time complexity is exponential in the path length: b^n, where b is forward branching factor of each node, n is path length.
  - The space complexity is exponential in path length: b^n.
  - Search is unconstrained by the goal.



d=4 nodes two-steps away!
d: depth of the solution

Blind Search Strategies (4.4) (cont.)

- Lowest-cost-first Search
  - Sometimes there are costs associated with arcs. The cost of a path is the sum of the costs of its arcs.
  - Lowest-cost-first search finds the shortest path to a goal node.
  - At each stage, it selects the shortest path on the frontier.
  - The frontier is implemented as a priority queue ordered by path length (or cost function).
  - When arc costs are equal ⇒ breadth-first search.
  - See example 4.13 p. 131 (textbook)

Summary

- **Data Structures for Search Tree:**

  A node is a data structure with five components:

  - The state in the state space to which the node corresponds
  - The node in the search tree that generated this node (parent node)
  - The operator applied to generate the node
  - The number of nodes on the path from the root to this node (the depth of the node)
  - The path cost of the path from the initial state to the node.

Summary (cont.)

- Strategies are evaluated in terms of four criteria:

  - **Completeness** [does the strategy guaranteed to find a solution when there is one ?]
  - **Time Complexity** [how long does it take to find a solution ?]
  - **Space Complexity** [How much memory does it need to perform the search ?]
  - **Optimality** [does the strategy find the highest-quality solution when there are several different solutions ?

Summary (cont.)

- **Uninformed Searches (or Blind)**

  "they have no information about the number of steps or the path cost from the current state to the goal- all they can do is distinguish a goal state from a nongoal state."

- **Informed Search (or Heuristic)**

" they are more clever, since they use external information during the search."

Summary (Cont.)

- Comparing Search Strategies

  b: branching factor

  d: the depth of solution (m: max depth)

Criterion	BF	DF	LCF
Time	b^d	b^m	b^d
Space	b^d	bm	b^d
Optimal	yes	no	yes
Complete	yes	no	yes

Our goal is to improve BF and LCF through Heuristic!

Applications

- The 8-puzzle problem:
  3x3 board with eight numbered tiles and a blank space.
- The 8-queens problem:
  Place eight queens on a chessboard such that no queen attacks any other.
- Cryptarithmetic (Forty + Ten + Ten = Sixty)
  Find a substitution of digits for letters such that the resulting sum is arithmetically correct. (Send + More = Money)
- Missionaries and cannibals:
  Find a way to get 3 missionaries and 3 cannibals to the other side of a river using a boat that holds 2 people maximum without ever leaving missionaries in one place outnumbered by the cannibals in that place.