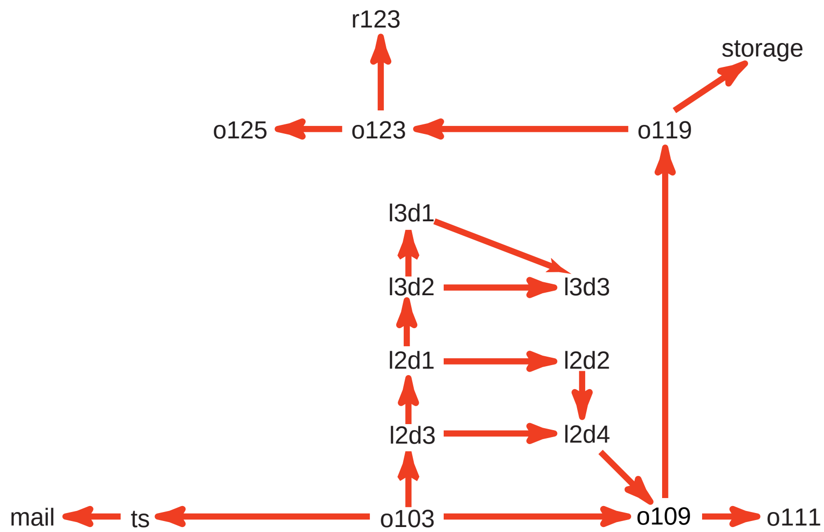
Searching

- 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.
- So far we have seen how to convert a semantic problem of finding logical consequence to a search problem of finding derivations.

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. That is, if $\langle n_1, n_2 \rangle \in A$.
- 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 Graph for the Delivery Robot



Search Graph for SLD Resolution

$a \leftarrow b \wedge c.$

$a \leftarrow h.$

$b \leftarrow k.$

$d \leftarrow p.$

$f \leftarrow p.$

$g \leftarrow f.$

$h \leftarrow m.$

$?a \wedge d$

$a \leftarrow g.$

$b \leftarrow j.$

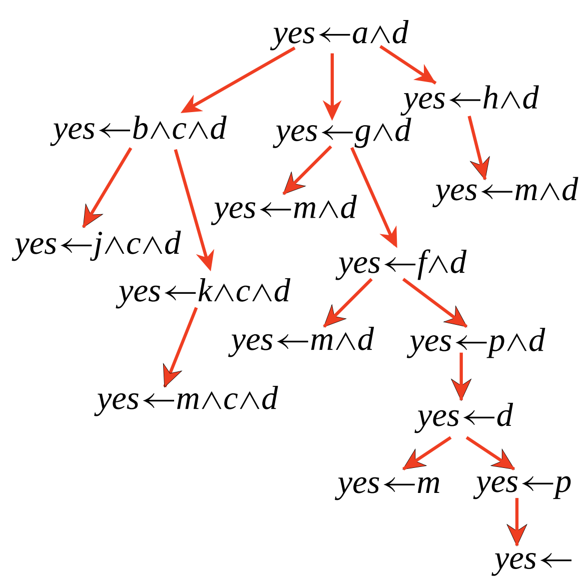
$d \leftarrow m.$

$f \leftarrow m.$

$g \leftarrow m.$

$k \leftarrow m.$

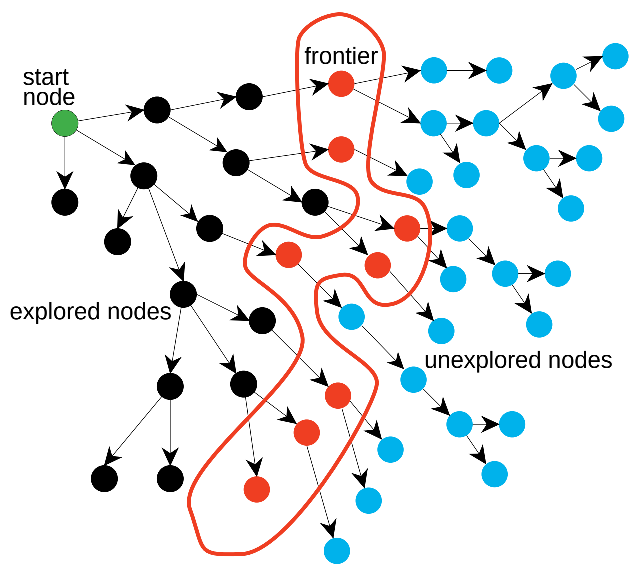
$p.$



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.

Problem Solving by Graph Searching



Generic Graph Search Algorithm

$search(F_0) \leftarrow$
$\quad select(Node, F_0, F_1) \wedge$
$\quad is\_goal(Node).$

$search(F_0) \leftarrow$
$\quad select(Node, F_0, F_1) \wedge$
$\quad neighbors(Node, NN) \wedge$
$\quad add\_to\_frontier(NN, F_1, F_2) \wedge$
$\quad search(F_2).$

- *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, F_1, F_2) 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.

Depth-first Search

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

$$select(Node, [Node|Frontier], Frontier).$$

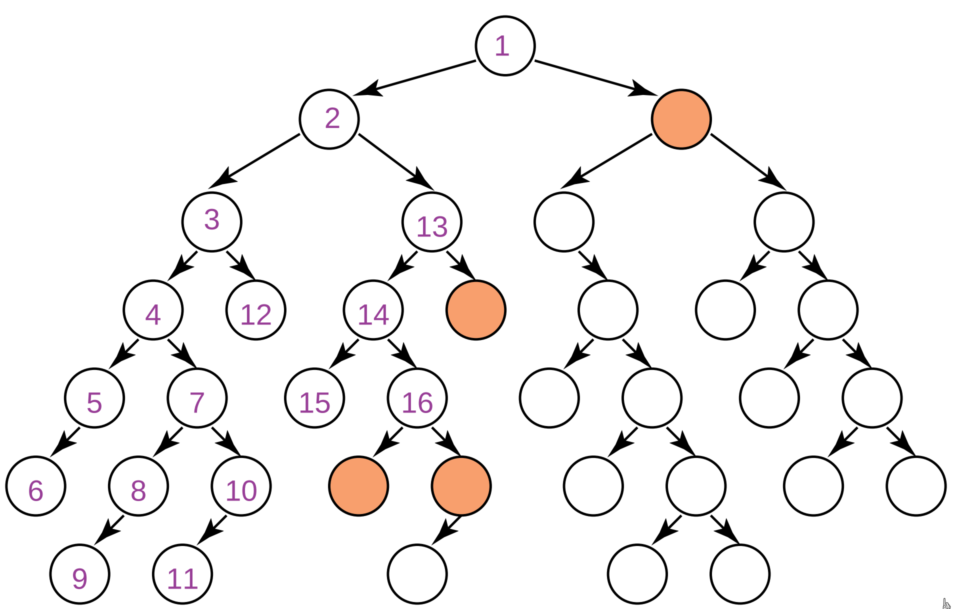
$$add\_to\_frontier(Neighbors, Frontier_1, Frontier_2) \leftarrow$$
$$append(Neighbors, Frontier_1, Frontier_2).$$

Frontier: $[e_1, e_2, \ldots]$

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.

Illustrative Graph — Depth-first Search



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.

Breadth-first Search

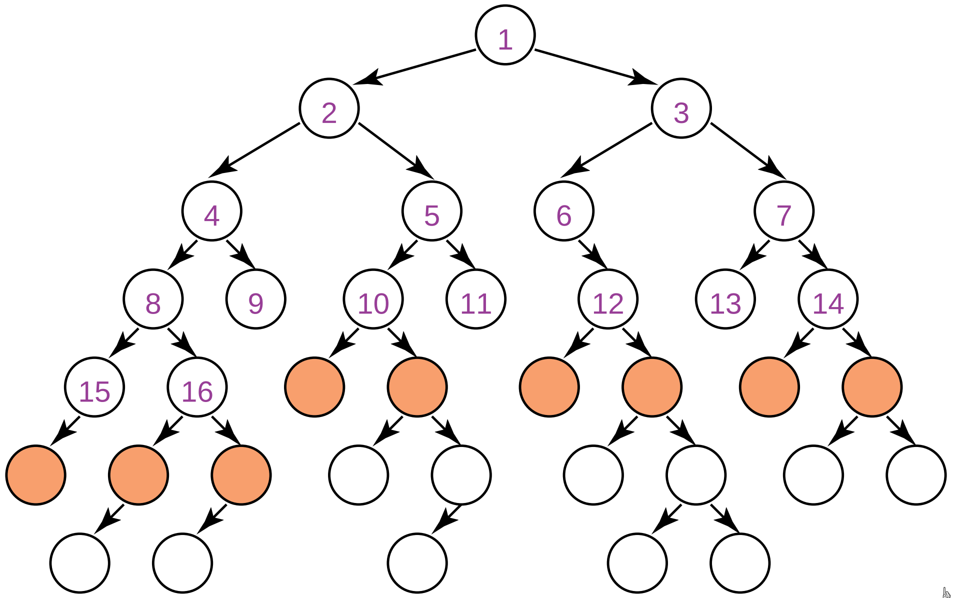
Breadth-first search treats the frontier as a queue: it always selects the earliest element added to the frontier.

$$select(Node, [Node|Frontier], Frontier).$$

$$add\_to\_frontier(Neighbors, Frontier_1, Frontier_2) \leftarrow$$
$$append(Frontier_1, Neighbors, Frontier_2).$$

Frontier: $[e_1, e_2, \ldots]$
e_1 is selected. Its neighbors are added to the end of the queue.
e_2 is selected next.

Illustrative Graph — Breadth-first Search



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.

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.
- When arc costs are equal $\Longrightarrow$ breadth-first search.

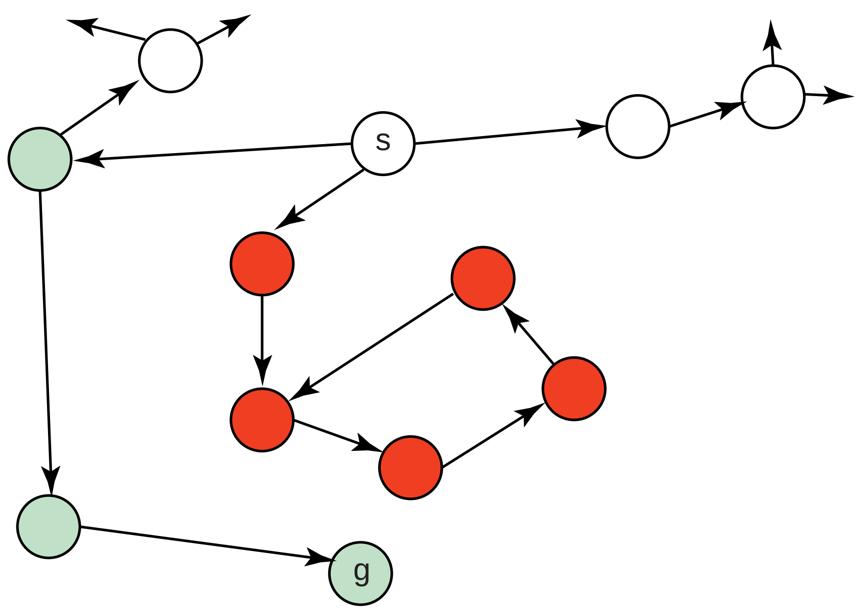
Heuristic Search

- Previous methods do not take into account the goal until they are at a goal node.
- Often there is extra knowledge that can be used to guide the search: heuristics.
- We use $h(n)$ as an estimate of the distance from node n to a goal node.
- $h(n)$ is an underestimate if it is less than or equal to the actual cost of the shortest path from node n to a goal.
- $h(n)$ uses only readily obtainable information about a node.

Best-first Search

- Idea: always choose the node on the frontier with the smallest *h*-value.
- It treats the frontier as a priority queue ordered by *h*.
- It uses space exponential in path length.
- It isn't guaranteed to find a solution, even of one exists. It doesn't always find the shortest path.

Illustrative Graph — Best-first Search



Heuristic Depth-first Search

- It's a way to use heuristic knowledge in depth-first search.
- Idea: order the neighbors of a node (by h) before adding them to the front of the frontier.
- Locally chooses which subtree to develop, but still does depth-first search. It explores all paths from the node at the head of the frontier before exploring paths from the next node.
- Space is linear in path length. It isn't guaranteed to find a solution. It can get led up the garden path.

A^* Search

- A^* search takes the path to a node and heuristic value into account.
- Let $g(n)$ be the cost of the path found to node n.
- Let $h(n)$ be the estimate of the cost from n to a goal.
- Let $f(n) = g(n) + h(n)$. It is an estimate of a path from the start to a goal via n.

$$\underbrace{\underbrace{start \xrightarrow{\text{actual}} n}_{g(n)} \underbrace{\xrightarrow{\text{estimate}} goal}_{h(n)}}_{f(n)}$$

A^* Search Algorithm

- A^* is a mix of lowest-cost-first and best-first search.
- It treats the frontier as a priority queue ordered by $f(n)$.
- It always chooses the node on the frontier with the lowest estimated distance from the start to a goal node constrained to go via that node.

Admissibility of A^*

If there is a solution, A^* always finds an optimal solution —the first path to a goal selected— if

- the branching factor is finite
- arc costs are bounded above zero (there is some $\epsilon > 0$ such that all of the arc costs are greater than ϵ), and
- $h(n)$ is an underestimate of the length of the shortest path from n to a goal node.

Why is A^* admissible?

- The f-value for any node on an optimal solution path is less than or equal to the f-value of an optimal solution. (As h is an underestimate).
- The search never selects a node with a higher f-value than the f-value of an optimal solution. A non-optimal solution has a higher f value — so it will never be selected.
- It halts, as the minimum g-value on the frontier keeps increasing, and will eventually exceed any finite number.