

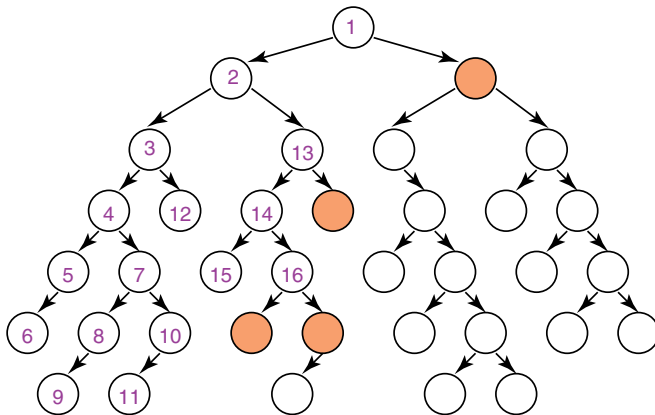
# Search Strategies

- A problem determines the graph and the goal but not which path to select from the frontier. This is the job of a search strategy.
- A search strategy specifies which paths are selected from the frontier.
- Different strategies are obtained by modifying how the selection of paths in the frontier is implemented.
- Three uninformed search strategies that do not take into account the location of the goal.
- Intuitively, these algorithms ignore where they are going until they find a goal and report success.
  - ▶ Depth-First Search
  - ▶ Breadth-First Search
  - ▶ Lowest-Cost-First Search

# Depth-first Search

- **Depth-first search** treats the frontier as a stack.
- It always selects one of the last elements added to the frontier.
- If the list of paths on the frontier is  $[p_1, p_2, \dots]$ 
  - ▶  $p_1$  is selected. Paths that extend  $p_1$  are added to the front of the stack (in front of  $p_2$ ).
  - ▶  $p_2$  is only selected when all paths from  $p_1$  have been explored.

## Illustrative Graph — Depth-first Search



The shaded nodes are the nodes at the ends of the paths on the frontier after the first sixteen steps.

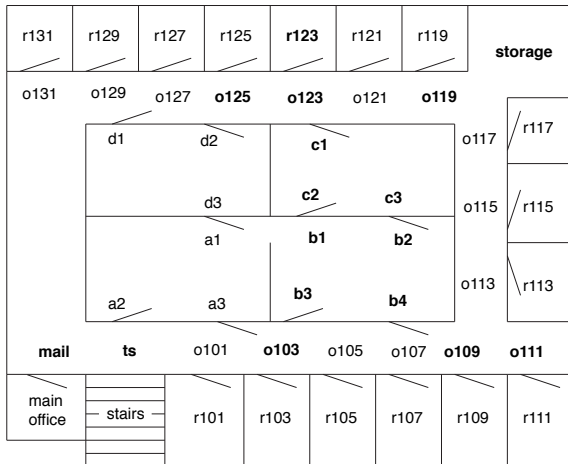
# Backtracking

- Searching one path to its completion before trying an alternative path: **backtracking**.
- The algorithm selects a first alternative at each node, and it backtracks to the next alternative when it has pursued all of the paths from the first selection.
- Some paths may be infinite when the graph has cycles or infinitely many nodes, in which case a depth-first search may never stop.
- This algorithm does not specify the order in which the neighbors are added to the stack that represents the frontier. The efficiency of the algorithm is sensitive to this ordering.

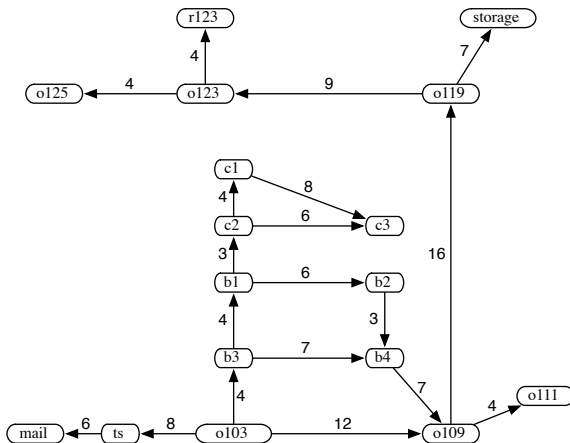


# Depth-first for the Delivery Robot

The robot wants to get from outside room 103 to the inside of room 123.



# Depth-first for the Delivery Robot



There are three paths from o103 to r123.

# Complexity of Depth-first Search

- If there is a solution on the first branch searched, then the time complexity is linear in the length of the path; it considers only those elements on the path, along with their siblings.
- If the graph is a finite tree, with the forward branching factor bounded by  $b$  and depth  $n$ , the worst-case complexity is  $O(b^n)$ .
- The worst-case complexity is infinite. Depth-first search can get trapped on infinite branches and never find a solution, even if one exists, for infinite graphs or for graphs with loops.
- Imagine the example of the robot in the case the robot can go back.
- An infinite path leads from *ts* to *mail*, back to *ts*, back to *mail*, and so forth. As presented, depth-first search follows this path forever, never considering alternative paths from *b3* or *o109*.





Depth-first search is appropriate when either

- space is restricted;
- many solutions exist, perhaps with long path lengths, particularly for the case where nearly all paths lead to a solution;
- or the order of the neighbors of a node are added to the stack can be tuned so that solutions are found on the first try.

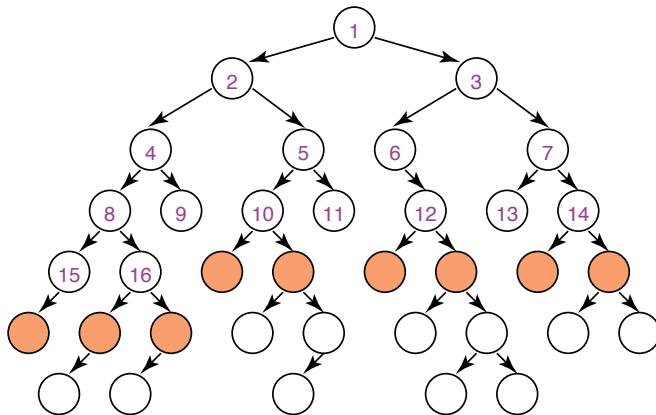
It is a poor method when

- it is possible to get caught in infinite paths; this occurs when the graph is infinite or when there are cycles in the graph; or
- solutions exist at shallow depth, because in this case the search may look at many long paths before finding the short solutions.

# Breadth-first Search

- **Breadth-first search** treats the frontier as a FIFO (first-in, first-out) queue.
- It always selects one of the earliest elements added to the frontier.
- If the list of paths on the frontier is  $[p_1, p_2, \dots, p_r]$ :
  - ▶  $p_1$  is selected. Its neighbors are added to the end of the queue, after  $p_r$ .
  - ▶  $p_2$  is selected next.
- This approach implies that the paths from the start node are generated in order of the number of arcs in the path.
- One of the paths with the fewest arcs is selected at each stage.

# Illustrative Graph — Breadth-first Search



The shaded nodes are the nodes at the ends of the paths of the frontier after the first sixteen steps.

- 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:  $b^n$ .
- Breadth-first search finds a solution with the fewest arcs first.

Breadth-first search is useful when

- space is not a problem;
- you want to find the solution containing the fewest arcs;
- few solutions may exist, and at least one has a short path length; and
- infinite paths may exist, because it explores all of the search space, even with infinite paths.

It is a poor method when

- all solutions have a long path length or
- there is some heuristic knowledge available.

It is not used very often because of its space complexity.



# Lowest-cost-first Search

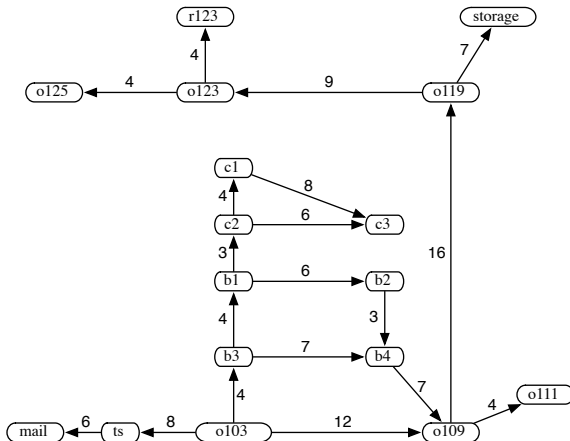
- Sometimes there are **costs** associated with arcs. The cost of a path is the sum of the costs of its arcs.

$$\text{cost}(\langle n_0, \dots, n_k \rangle) = \sum_{i=1}^k |\langle n_{i-1}, n_i \rangle|$$

- For example, for a delivery robot, costs may be distances and we may want a solution that gives the minimum total distance.
- At each stage, lowest-cost-first search selects a path on the frontier with lowest cost.
- The frontier is a priority queue ordered by path cost.
- It finds a least-cost path to a goal node.
- When arc costs are equal  $\Rightarrow$  breadth-first search.



# Cost for the Delivery Robot





# Features of Lowest-cost-first Search

- If the costs of the arcs are bounded below by a positive constant and the branching factor is finite, the lowest-cost-first search is guaranteed to find an optimal solution - a solution with lowest path cost - if a solution exists.
- Moreover, the first path to a goal that is found is a path with least cost.
- Such a solution is optimal, because the algorithm generates paths from the start in order of path cost.
- If a better path existed than the first solution found, it would have been selected from the frontier earlier.

# Complexity of Lowest-cost-first Search

- Like breadth-first search, lowest-cost-first search is typically exponential in both space and time.
- It generates all paths from the start that have a cost less than the cost of the solution.

# The bounded arc cost assumption

- The bounded arc cost is used to guarantee the lowest-cost search will find an optimal solution.
- Without such a bound there can be infinite paths with a finite cost.
- For example, there could be nodes  $n_0, n_1, \dots$  with an arc  $\langle n_{i-1}, n_i \rangle$  for each  $i > 0$  with cost  $(1/2)^i$ . Infinitely many paths of the form  $\langle n_0, n_1, \dots, n_k \rangle$  exist, all of which have a cost of less than 1. If there is an arc from  $n_0$  to a goal node with a cost greater than or equal to 1, it will never be selected. This is the basis of Zeno's paradoxes that Aristotle wrote about more than 2,300 years ago.