State Spaces

- A **state-space problem** consists of
  - a set of states
  - a distinguished state called the start state
  - for each state, a set of actions available to the agent in that state
  - an action function that, given a state and an action, returns a new state
  - a goal specified as a Boolean function, goal(s) that is true when state s satisfies the goal, in which case we say that s is a goal state
  - a criterion that specifies the quality of an acceptable solution.

    For example, any sequence of actions that gets the agent to the goal state may be acceptable, or there may be costs associated with actions and the agent may be required to find a sequence that has minimal total cost.
  - A solution that is best according to some criterion is called an optimal solution. We do not always need an optimal solution, for example, we may be satisfied with any solution that is within 10% of optimal.

Graph Searching

The problem of finding a sequence of actions to achieve a goal is abstracted as searching for paths in directed graphs.

To solve a problem,

first define the underlying search space and then

apply a search algorithm to that search space.

Many problem-solving tasks are transformable into the problem of finding a path in a graph.

Searching in graphs provides an appropriate abstract model of problem solving independent of a particular domain.

A directed graph consists of a set of nodes and a set of directed arcs between nodes. The idea is to find a path along these arcs from the start node to a goal node.

In representing a state-space problem, the states are represented as nodes, and the actions as arcs.

Graph Searching

- A **directed graph** consists of
  - a set N of nodes and
  - a set A of arcs, where an arc is an ordered pair of nodes.
- In this definition, a node could be anything. There may be infinitely many nodes and arcs. We do not assume that a graph is represented explicitly; we require only a procedure to generate nodes and arcs as needed.
- The arc ⟨n1,n2⟩ is an **outgoing arc** from n1 and an **incoming arc** to n2.
- A node n2 is a **neighbor** of n1n1 if there is an arc from n1 to n2; that is, if ⟨n1,n2⟩∈A.
- Arcs may be **labeled**, for example, with the action that will take the agent from one node to another or with the cost of an action or both
- A **path** from node s to node g is a sequence of nodes ⟨n0,n1,...,nk⟩ such that s=n0, g=nk, and ⟨ni-1,ni⟩∈A;

that is, there is an arc from ni−1 to ni for each i.

Sometimes it is useful to view a path as the sequence of arcs, ⟨n0,n1⟩, ⟨n1,n2⟩,..., ⟨nk-1,nk⟩, or

a sequence of labels of these arcs. Path ⟨n0,n1,...,ni⟩ is an **initial part** of ⟨n0,n1,...,nk⟩, when i≤ki≤k.

- A **goal** is a Boolean function on nodes. If goal(n) is true, we say that node n satisfies the goal, and n is a **goal node**

State Spaces

Example:

Consider the robot delivery domain and the task of finding a path from one location to another.

This can be modeled as a state space search problem, where the states are locations.

Assume that the agent can use a lower-level controller to carry out the high-level action of getting from one location to a neighboring location.

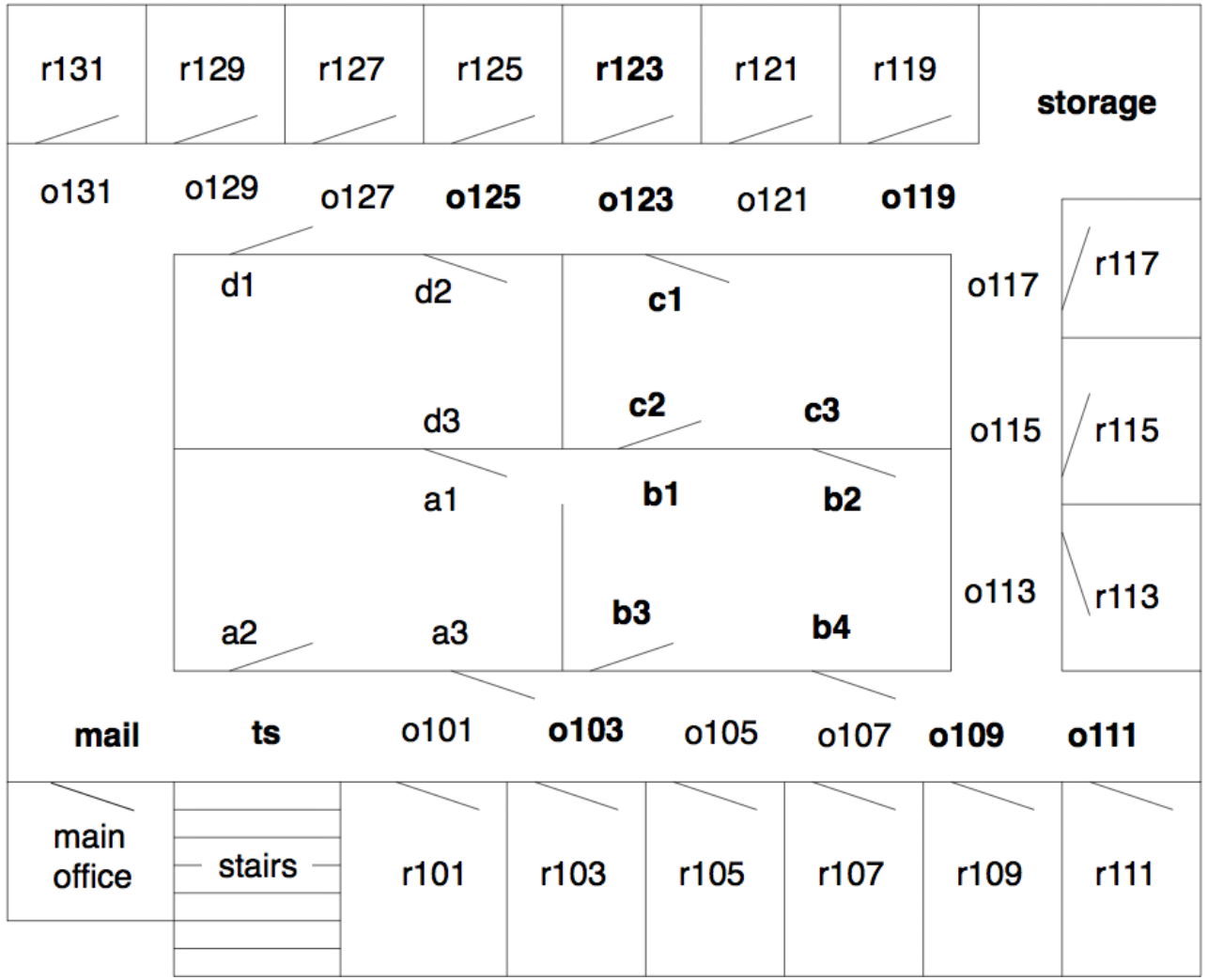
Thus, at this level of abstraction, the actions can involve deterministic traveling between neighboring locations.

An example problem is where the robot is outside room r103, at position o103,

and the goal is to get to room r123.

A solution is a sequence of actions that will get the robot to room r123.

delivery robot domain with interesting locations labeled



Graph Searching

- To encode problems as graphs, one node is identified as the **start node**. A **solution** is a path from the start node to a node that satisfies the goal.
- Sometimes there is a **cost** – a non-negative number – associated with arcs. We write the cost of arc $\langle ni,nj \rangle$ as $cost(\langle ni,nj \rangle)$.
- The costs of arcs induce a cost of paths. Given a path $p=\langle n0,n1,\ldots,nk \rangle$ the cost of path p is the sum of the costs of the arcs in the path:

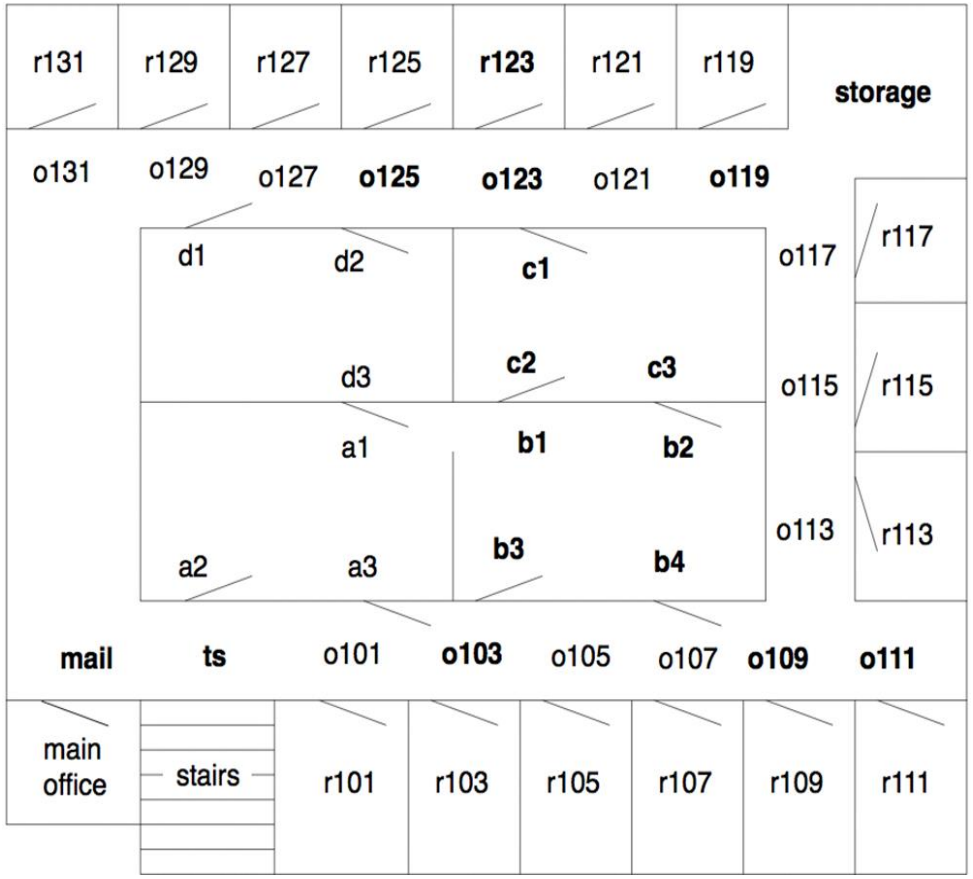
$$cost(p) = \sum_{i=1}^{k} cost(\langle n_{i-1}, n_i \rangle) = cost(\langle n_0, n_1 \rangle) + \ldots + cost(\langle n_{k-1}, n_k \rangle)$$

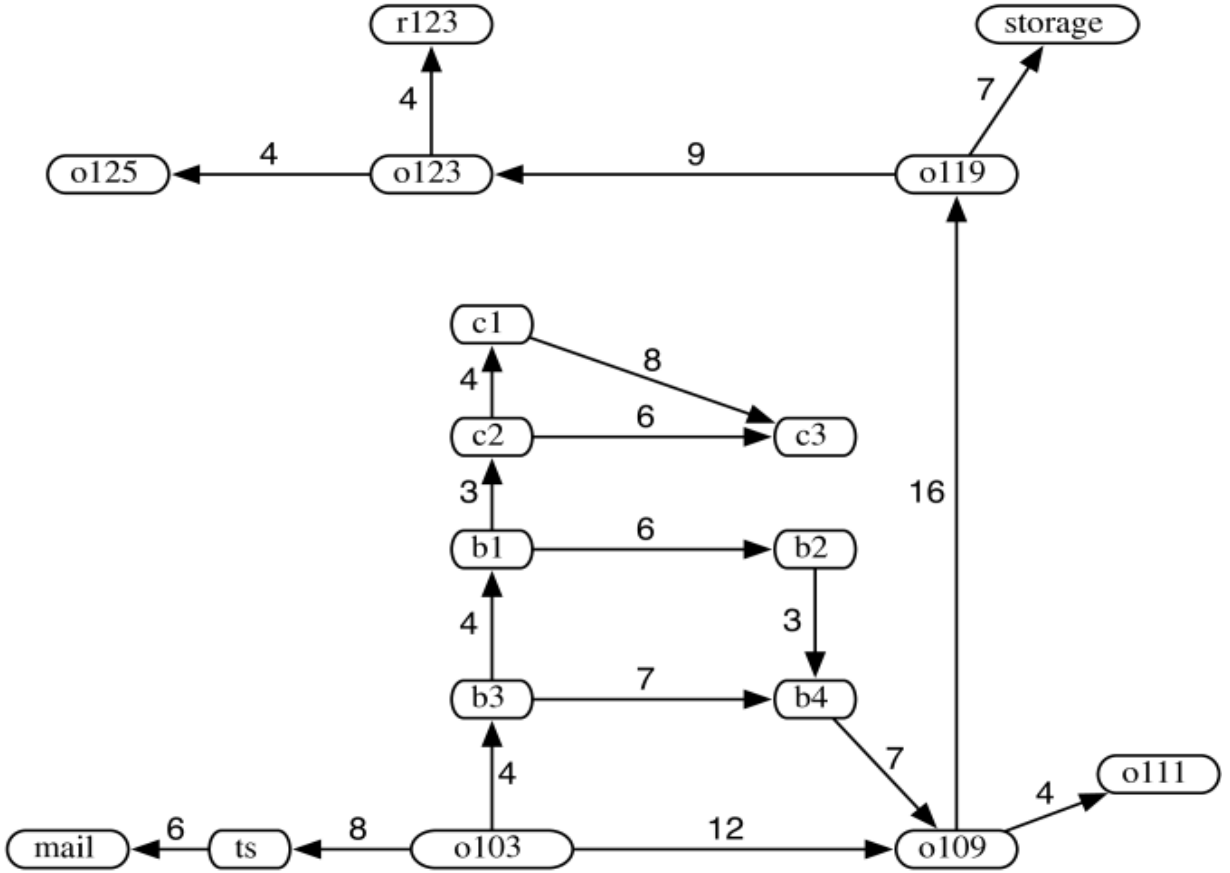
An **optimal solution** is one of the solutions that has the lowest cost. That is, an optimal solution is a path p from the start node to a goal node such that there is no path p' from the start node to a goal node where

$cost(p') < cost(p)$

A graph with arc costs for the delivery robot domain

Consider the problem of the delivery robot finding a path from location o103o103 to location r123





A Generic Searching Algorithm

The intuitive idea behind the generic search algorithm, given a graph, a start node, and a goal predicate, is to explore paths incrementally from the start node.

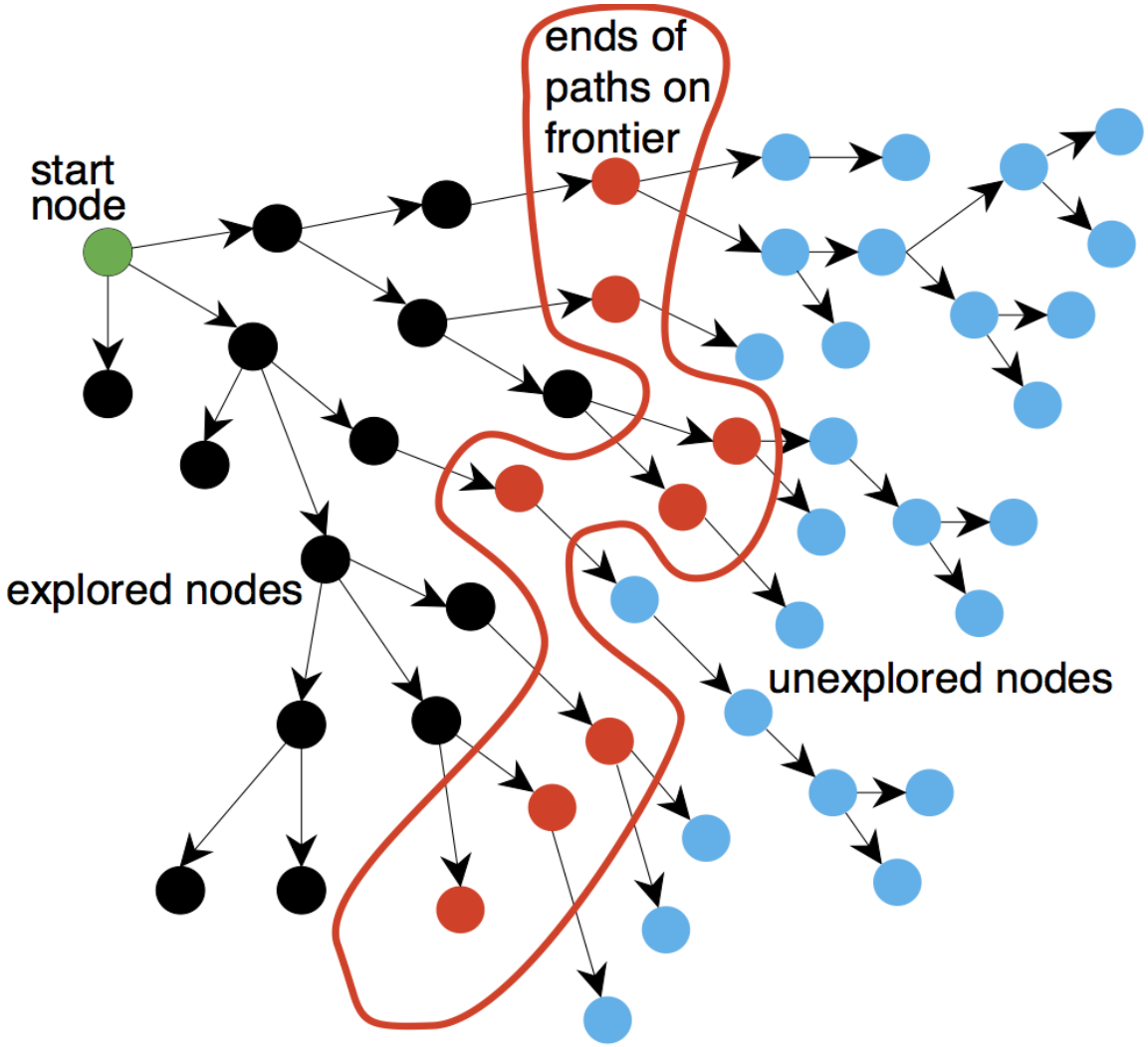
This is done by maintaining a **frontier** (or **fringe**) of paths from the start node. The frontier contains all of the paths that could form initial segments of paths from the start node to a goal node.

Initially, the frontier contains the trivial path containing just the start node, and no arcs.

As the search proceeds, the frontier expands into the unexplored nodes until a goal node is encountered.

Different search strategies are obtained by providing an appropriate implementation of the frontier.

A Generic Searching Algorithm



Search: generic graph searching algorithm

```
1: procedure Search(G, S, goal)
2:     Inputs
3:         G: graph with nodes N and arcs A
4:         s: start node
5:         goal: Boolean function of nodes
6:     Output
7:         path from s to a node for which goal is true
8:         or ⊥ if there are no solution paths
9:     Local
10:        Frontier: set of paths
11:    Frontier:= {⟨s⟩}
12:    while Frontier ≠ {}  do
13:        select and remove ⟨n0,...,nk⟩ from Frontier
14:        if  goal(nk) then
15:            return ⟨n0,...,nk⟩
16:        Frontier:= Frontier ∪ {⟨n0,...,nk, n⟩: ⟨nk, n⟩∈A}
17:    return ⊥
```

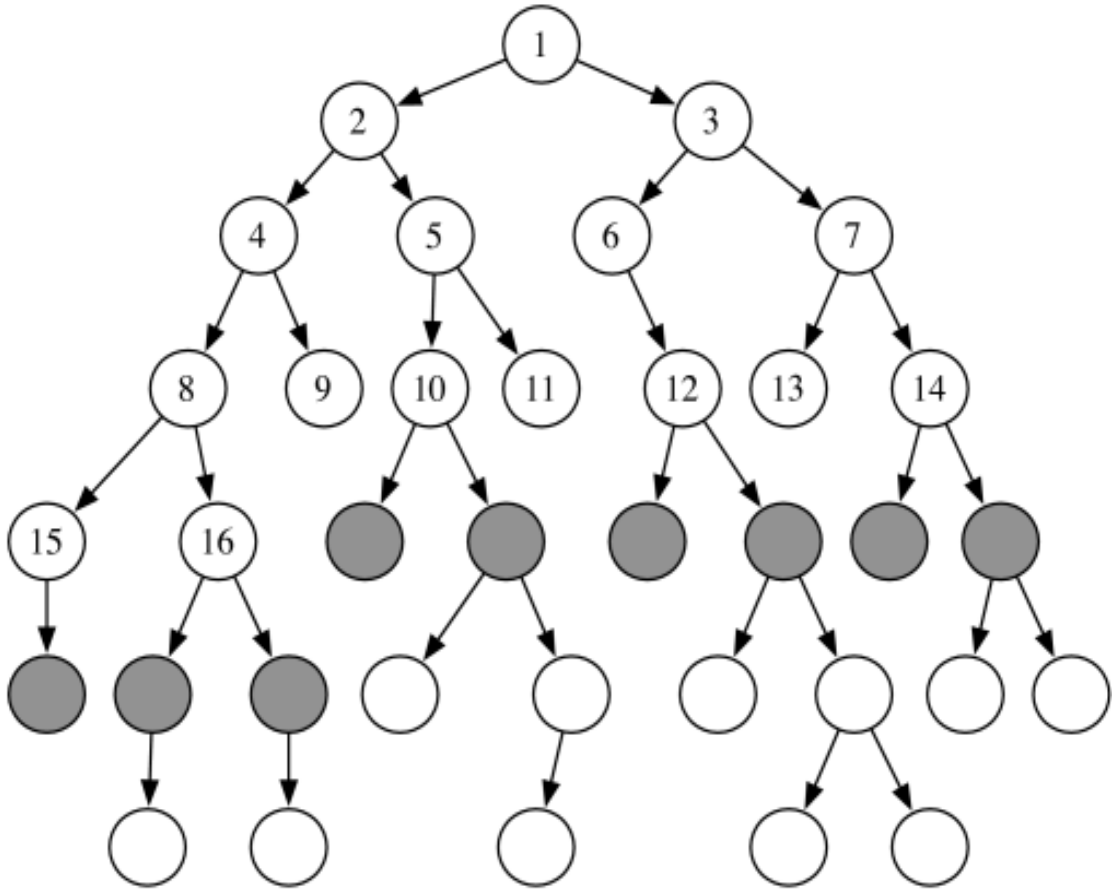
BFS Algorithm

Suppose the start node is the node at the top, and the children of a node are added in a left-to-right order.

In breadth-first search, the order in which the paths are expanded does not depend on the location of the goal.

The nodes at the end of the first sixteen paths expanded are numbered in order of expansion in the figure.

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



DFS Algorithm

In **depth-first search**, the frontier acts like a **LIFO** (last-in, first-out) **stack** of paths.

In a stack, elements are added and removed from the top of the stack.

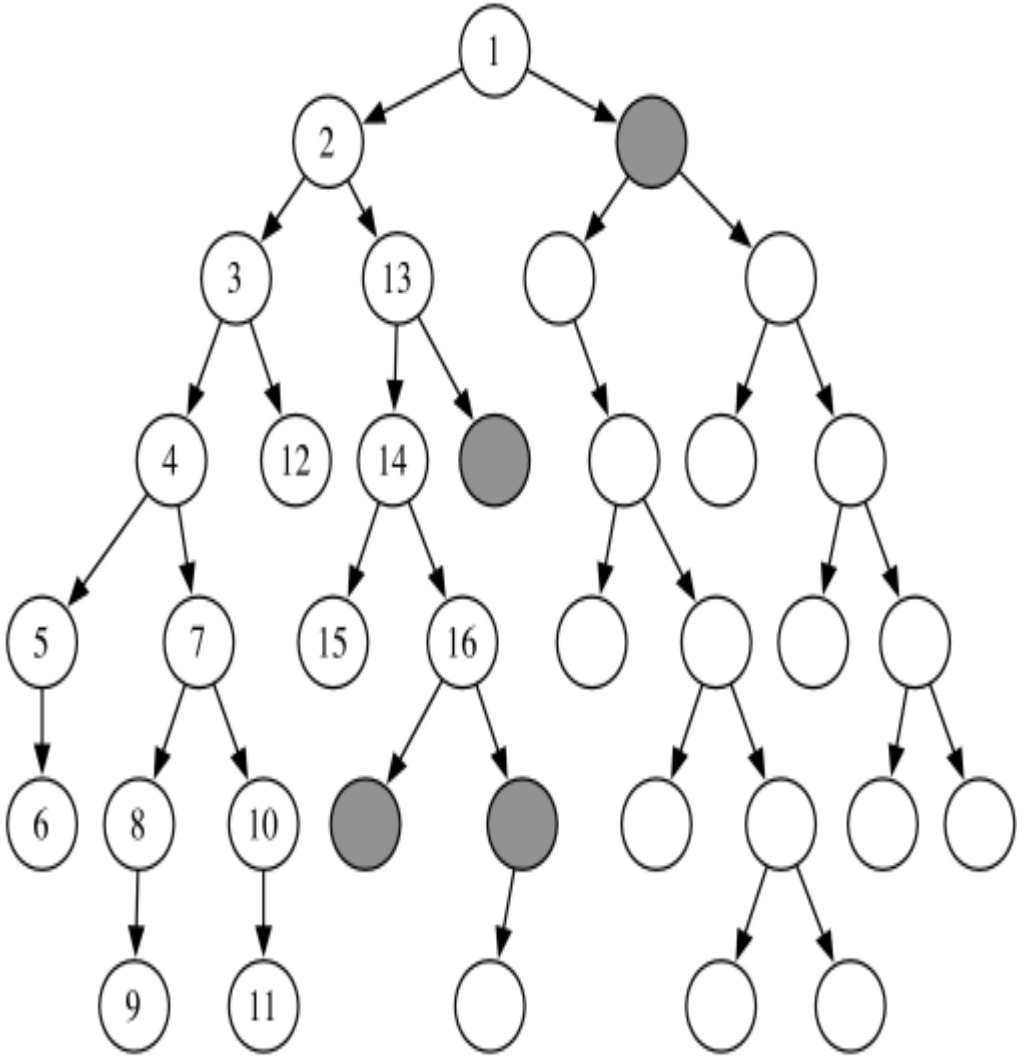
Using a stack means that the path selected and removed from the frontier at any time is the last path that was added

In depth-first search, like breadth-first search, the order in which the paths are expanded does not depend on the goal.

Implementing the frontier as a stack results in paths being pursued in a depth-first manner – searching one path to its completion before trying an alternative path.

This method is said to involve 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.



Algorithm

Algorithm

Algorithm

Algorithm

Algorithm

Algorithm