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.
- A typical problem is when the agent is in one state, it has a set of deterministic actions it can carry out, and wants to get to a goal state.
- Many AI problems can be abstracted into the problem of finding a path in a directed graph.
- Often there is more than one way to represent a problem as a graph.

The idea of search is straightforward

- The agent constructs a set of potential partial solutions to a problem;
- these partial solutions can be checked to see if they truly are solutions or if they could lead to solutions;
- search proceeds by repeatedly selecting a partial solution, stopping if it is a path to a goal, and otherwise extending it by one more arc in all possible ways.

- Search underlies much of artificial intelligence:
  - When an agent is given a problem, it is usually given only a description that lets it recognize a solution, not an algorithm to solve it.
  - The agent has to search for a solution.

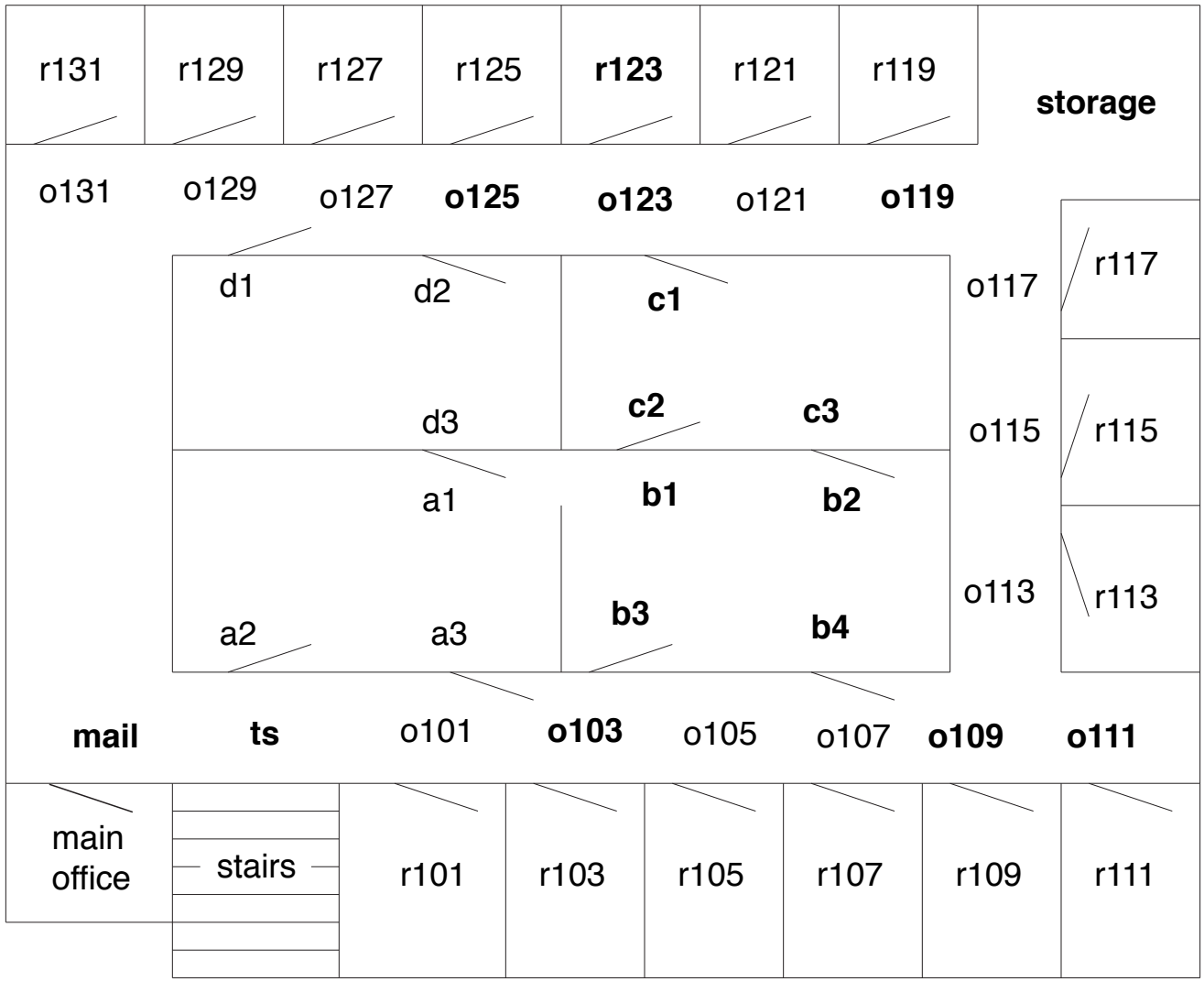
State Spaces

One general formulation of intelligent action is in terms of state space.

- A state contains all of the information necessary to predict the effects of an action and to determine if it is a goal state.
- State-space searching assumes that:
  - the agent has perfect knowledge of the state space and can observe what state it is in (i.e., there is full observability);
  - the agent has a set of actions that have known deterministic effects;
  - some states are goal states, the agent wants to reach one of these goal states, and the agent can recognize a goal state;
  - a solution is a sequence of actions that will get the agent from its current state to a goal state.

Example Problem for Delivery Robot

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



A State for the Robot

- The state consists of the location of the robot.
- The state consists of the location of the robot, the parcels the robot is carrying, and the locations of the other parcels.

A state-space problem

A state-space problem consists of:

- a set of states;
- a distinguished set of states called the *start states*;
- a set of actions available to the agent in each state;
- an *action function* that, given a state and an action, returns a new state;
- a set of goal states, often specified as a Boolean function, goal(s), that is true when s is a goal state; and
- 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. This is called an *optimal* solution. Alternatively, it may be satisfied with any solution that is within 10% of optimal.

Extensions of the state-space problem

This framework will be extended to include cases:

- where an agent can exploit the internal features of the states,
- where the state is not fully observable (e.g., the robot does not know where the parcels are, or the teacher does not know the aptitude of the student),
- where the actions are stochastic (e.g., the robot may overshoot, or the student perhaps does not learn a topic that is taught), and
- where complex preferences exist in terms of rewards and punishments, not just goal states.

Graph Searching

- We abstract the general mechanism of searching and present it in terms of 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 can be transformed into the problem of finding a path in a graph.
- Searching in graphs provides an appropriate level of abstraction within which to study simple 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 a start node to a goal node.

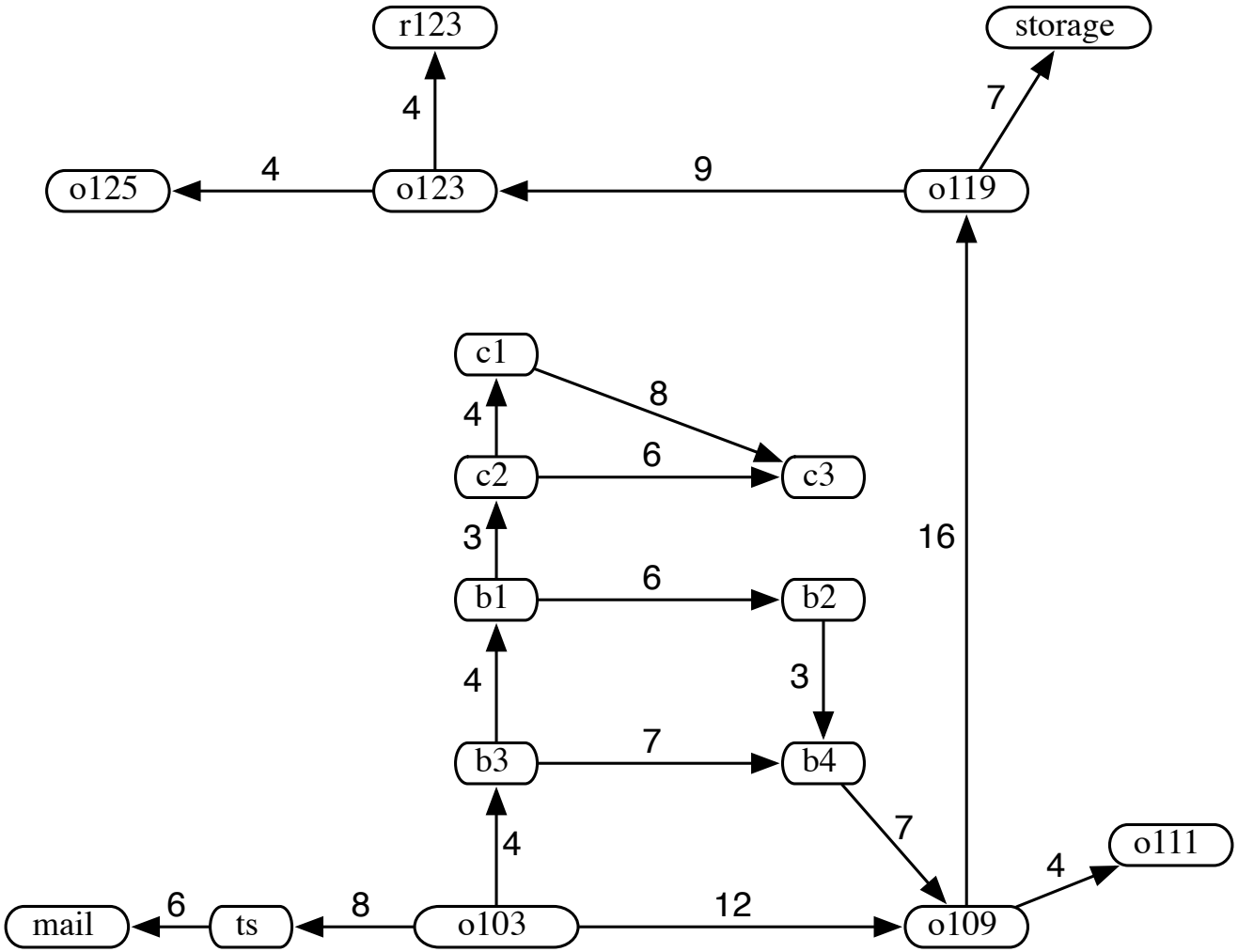
Directed Graphs

- 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 $\langle n_1, n_2 \rangle \in A$. *Neighborhood* is not symmetric.
- A path is a sequence of nodes $\langle n_0, n_1, \ldots, n_k \rangle$ such that $\langle n_{i-1}, n_i \rangle \in A$.
- A cycle is a nonempty path such that the end node is the same as the start node – that is, a cycle is a path $\langle n_0, n_1, ..., n_k \rangle$ such that $n_0 = n_k$ and $k \neq 0$.
- A directed graph without any cycles is called a directed acyclic graph (DAG).
- A tree is a DAG where there is one node with no incoming arcs and every other node has exactly one incoming arc. The node with no incoming arcs is called the root of the tree and nodes with no outgoing arcs are called leaves.

Directed Graphs for Searching

- Arcs may be labeled, for example, with the action that will take the agent from one state to another.
- Given a set of start nodes and goal nodes, a solution is a path from a start node to a goal node.
- Often there is a cost associated with arcs and the cost of a path is the sum of the costs of the arcs in the path.
- An optimal solution is one of the least-cost solutions; that is, it is a path p from a start node to a goal node such that there is no path p' from a start node to a goal node where $cost(p') < cost(p)$.

Graph for the Delivery Robot



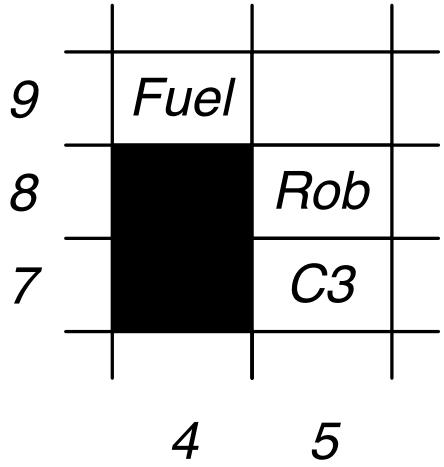
There are three paths from o103 to r123.

Branching

- The forward branching factor of a node is the number of arcs leaving the node.
- The backward branching factor of a node is the number of arcs entering the node.
- These factors provide measures of the complexity of graphs. When we discuss the time and space complexity of the search algorithms, we assume that the branching factors are bounded from above by a constant.
- The branching factor is important because it is a key component in the size of the graph. If the forward branching factor for each node is b, and the graph is a tree, there are b^n nodes that are n arcs away from any node.

Partial Search Space for a Video Game

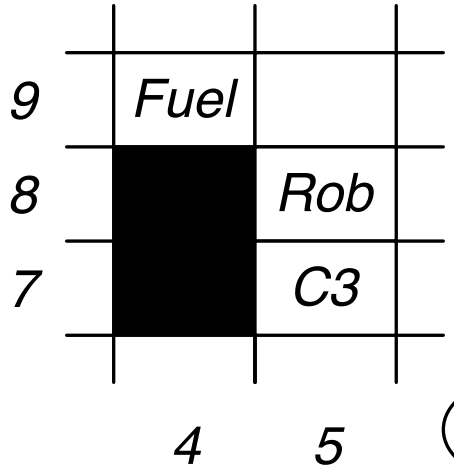
Grid game: collect coins C_1, C_2, C_3, C_4, don't run out of fuel, and end up at location $(1, 1)$:

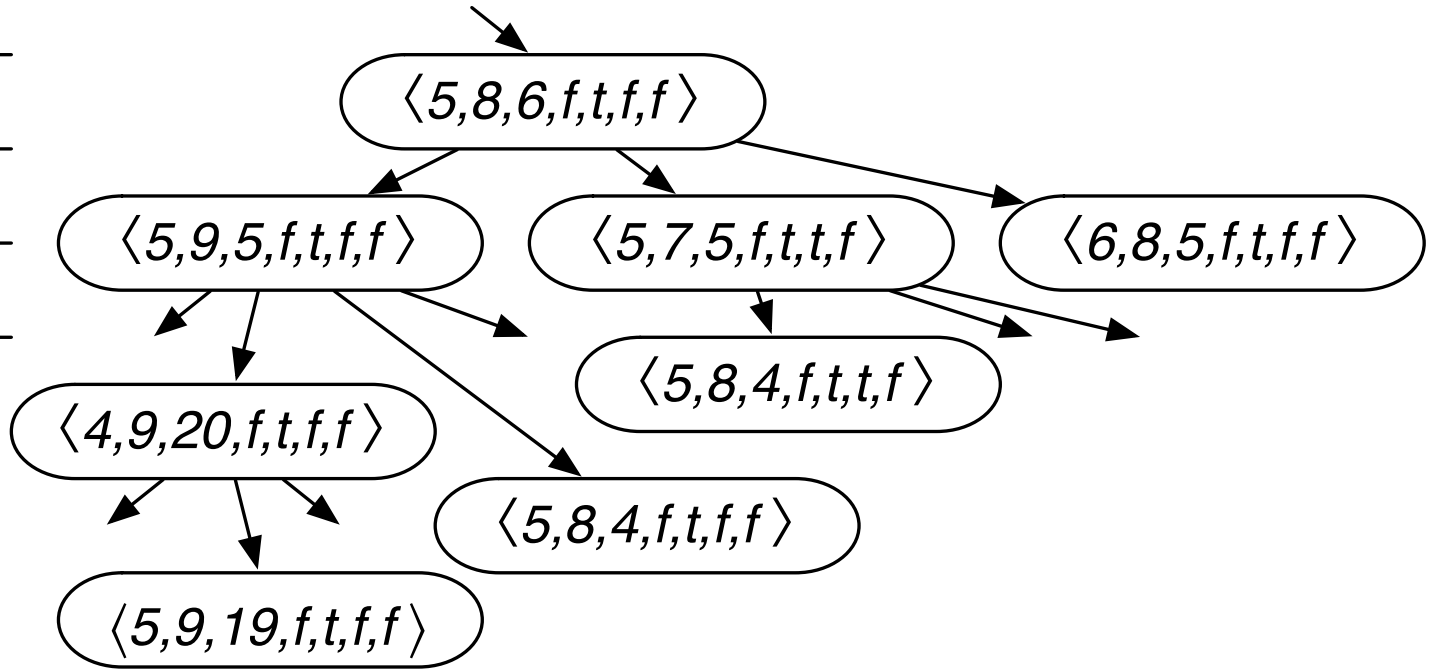


.

Partial Search Space for a Video Game

Grid game: collect coins C_1, C_2, C_3, C_4, don't run out of fuel, and end up at location $(1, 1)$:





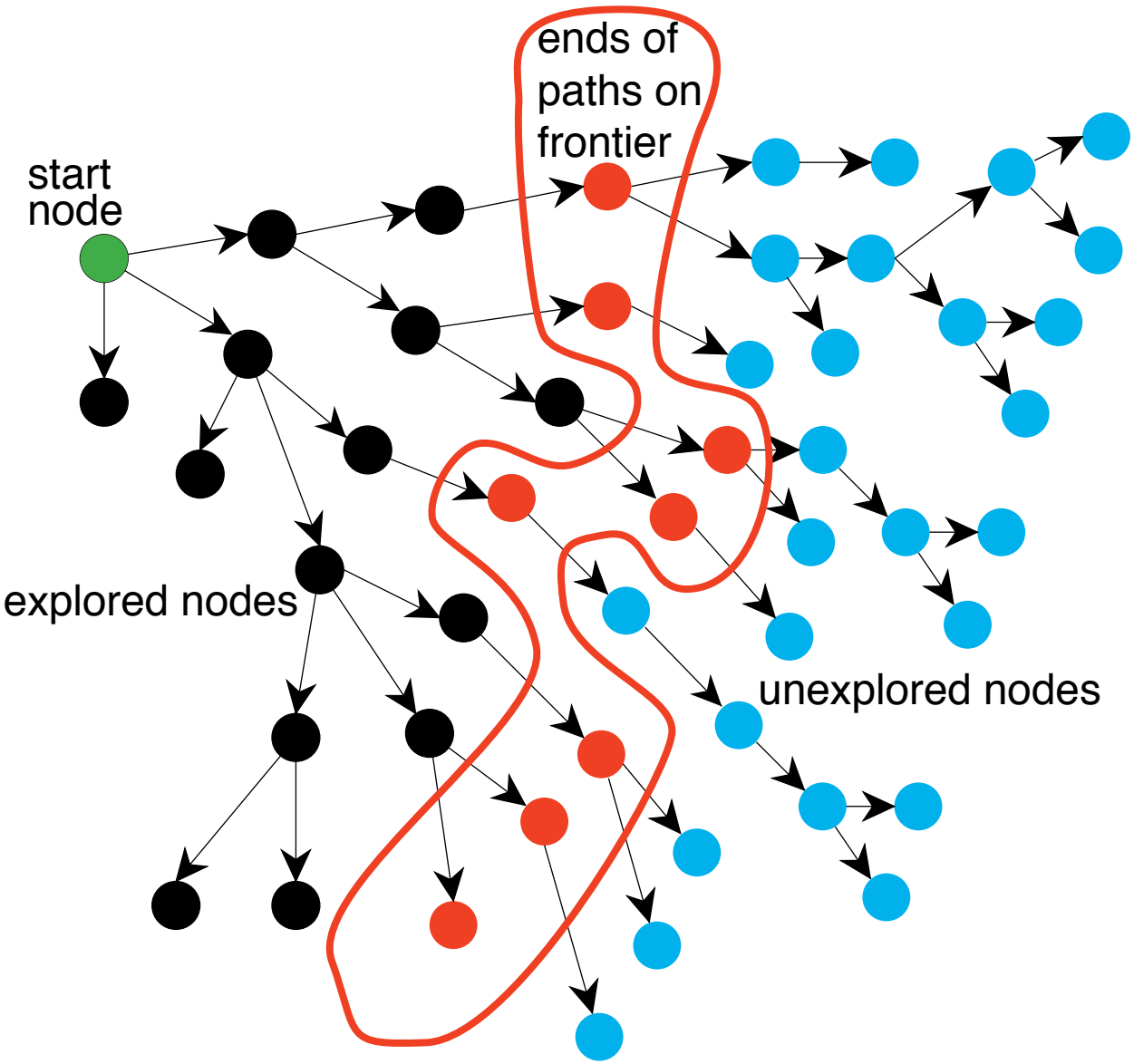
State:
⟨*X-pos,Y-pos,Fuel,C1,C2,C3,C4*⟩

Goal:
⟨*1,1,?,t,t,t,t*⟩

Graph Searching

- Generic search algorithm: given a graph, start nodes, and goal nodes, incrementally explore paths from the start nodes.
- Maintain a frontier (or *fringe*) of paths from the start node that have been explored.
- Initially, the frontier contains trivial paths containing no arcs from the start nodes.
- As search proceeds, the frontier expands into the unexplored nodes until a goal node is encountered.
- To expand the frontier, the searcher selects and removes a path from the frontier, extends the path with each arc leaving the last node, and adds these new paths to the frontier.
- The way in which the frontier is expanded defines the search strategy.

Problem Solving by Graph Searching



Graph Search Algorithm

Input: a graph,
a set of start nodes,
Boolean procedure $goal(n)$ that tests if n is a goal node.

$frontier := \{\langle s \rangle : s \text{ is a start node}\}$;
while *frontier* is not empty:
  select and **remove** path $\langle n_0, \ldots, n_k \rangle$ from *frontier*;
  if $goal(n_k)$
    return $\langle n_0, \ldots, n_k \rangle$;
  for every neighbor n of n_k
    add $\langle n_0, \ldots, n_k, n \rangle$ to *frontier*;
end while
return $\perp$

Features of the algorithm

- The neighbors define the graph.
- *goal* defines what is a solution.
- The selection of a path is non-deterministic.
  - The choice of path that is selected can affect the efficiency.
  - A particular *search strategy* will determine which path is selected.
- It is useful to think of the return of the goal path as a temporary return; another path to a goal can be searched for by continuing after the return.
- If the procedure returns $\perp$, no solutions exist (or there are no remaining solutions if the proof has been retried).

Testing for goals

- The algorithm only tests if a path ends in a goal node after the path has been selected from the frontier, not when it is added to the frontier.
- There are two main reasons for this.
  - Sometimes a very costly arc exists from a node on the frontier to a goal node. The search should not always return the path with this arc, because a lower-cost solution may exist. This is crucial when the least-cost path is required.
  - The second reason is that it may be expensive to determine whether a node is a goal node.
- If the path chosen does not end at a goal node and the node at the end has no neighbors, extending the path means removing the path. This outcome is reasonable because this path could not be part of a path from a start node to a goal node.