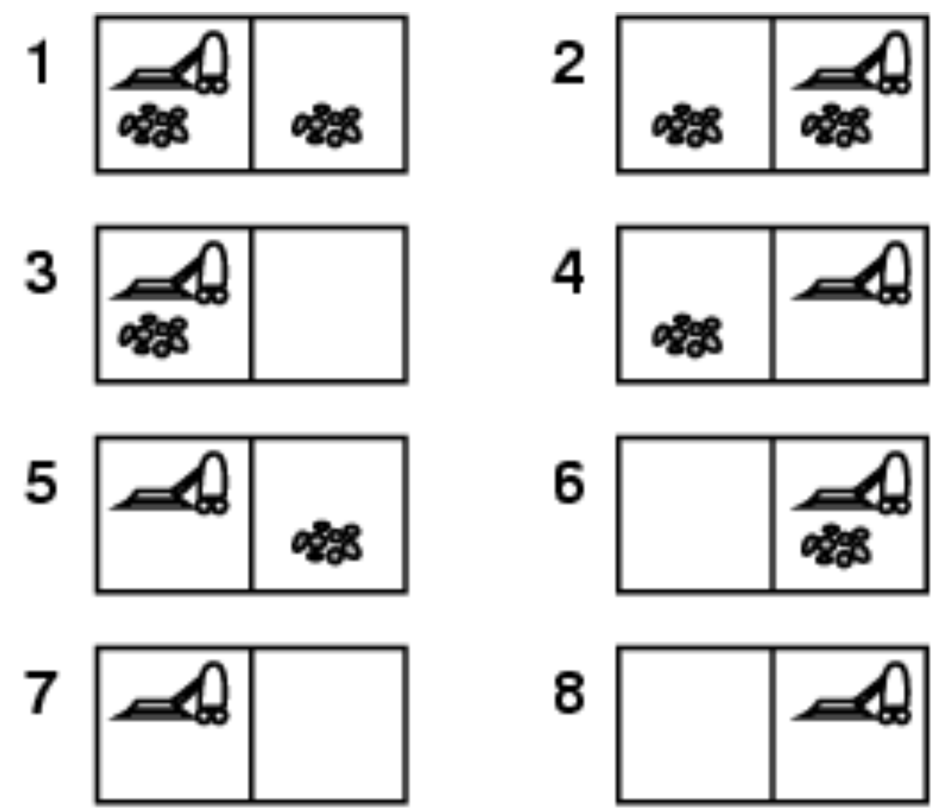
Searching the State Space

Part 1

State space

- A **state** is a data structure (object) that represents a possible configuration of the world (agent and environment).
- The **state space** of a problem is the set of all possible states for that problem.
- Example: A vacuum cleaner agent in two adjacent rooms which can be either clean or dirty.



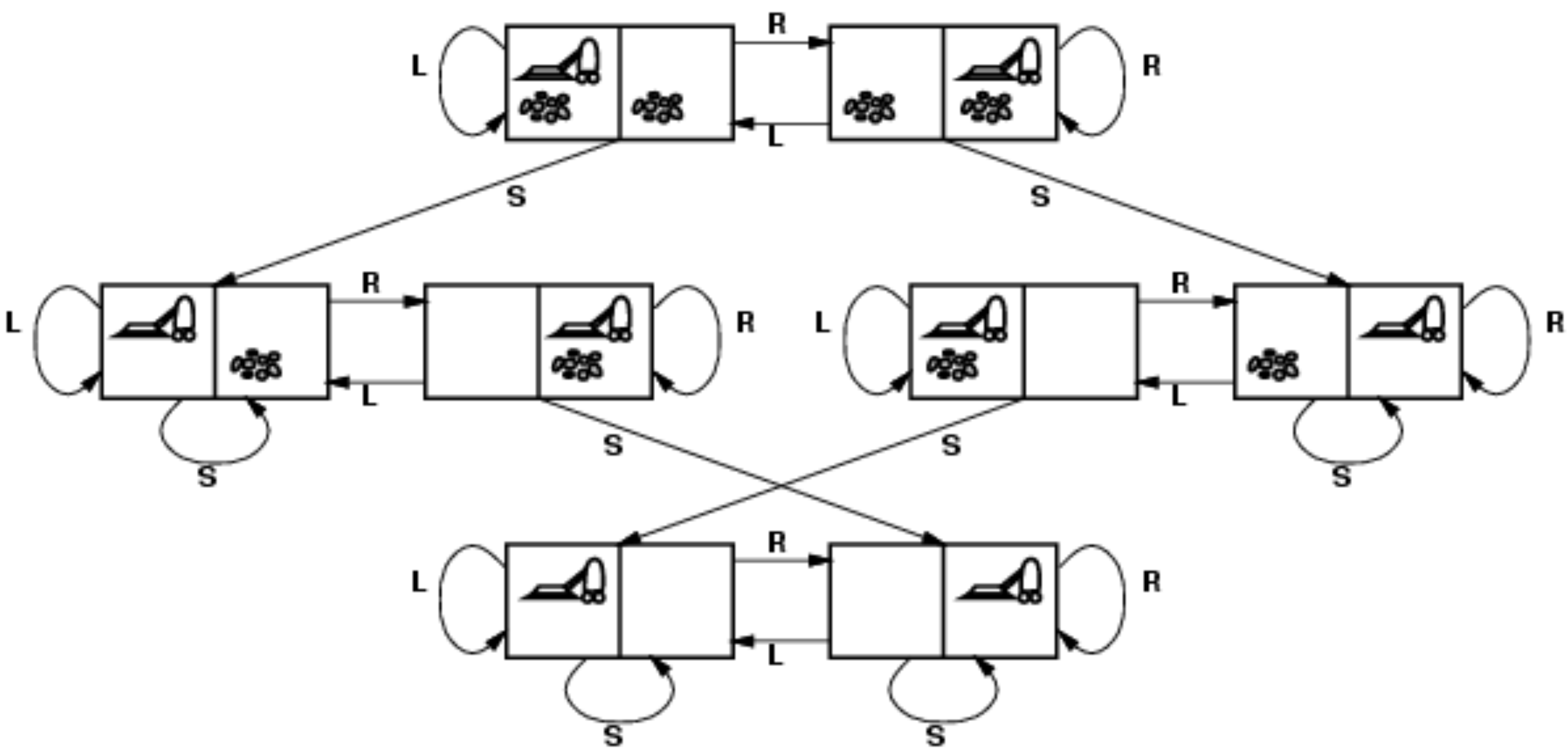
- Location = {left, right}
- Left-room-condition = {dirty, clean}
- Right-room-condition = {dirty, clean}
- State-space = Location
  × Left-room-condition
  × Right-room-condition

In this example, each state is represented by a triple (3-tuple).

State space graphs

- Actions change the state of the world.

- Example: Suppose the vacuum cleaner agent can take the following actions: L (go left), R (go right), S (suck).



Directed graphs

- Many AI problems can be abstracted into the problem of finding a path in a directed graph.

- A graph consists of a set of **nodes** and a set of **arcs**.
  - nodes are usually used for states.
  - arcs are used for actions.

- In theory, a **path** is sequence of nodes $\langle n_0, n_1, \ldots, n_k \rangle$ where there is an arc from n_i to n_{i+1}. In practice, a path is sequence of arcs (see the provided Python module in the quiz.)

- The **length** of path $\langle n_0, n_1, \ldots, n_k \rangle$ is k.

- An arc can have an associated cost (which could mean distance, time, energy, etc). The **cost** of a path is the sum of the cost of the arcs along the path.

- Given a set of **starting nodes** and a set of **goal nodes** (desired states), a **solution** is a path from a start node to a goal node.

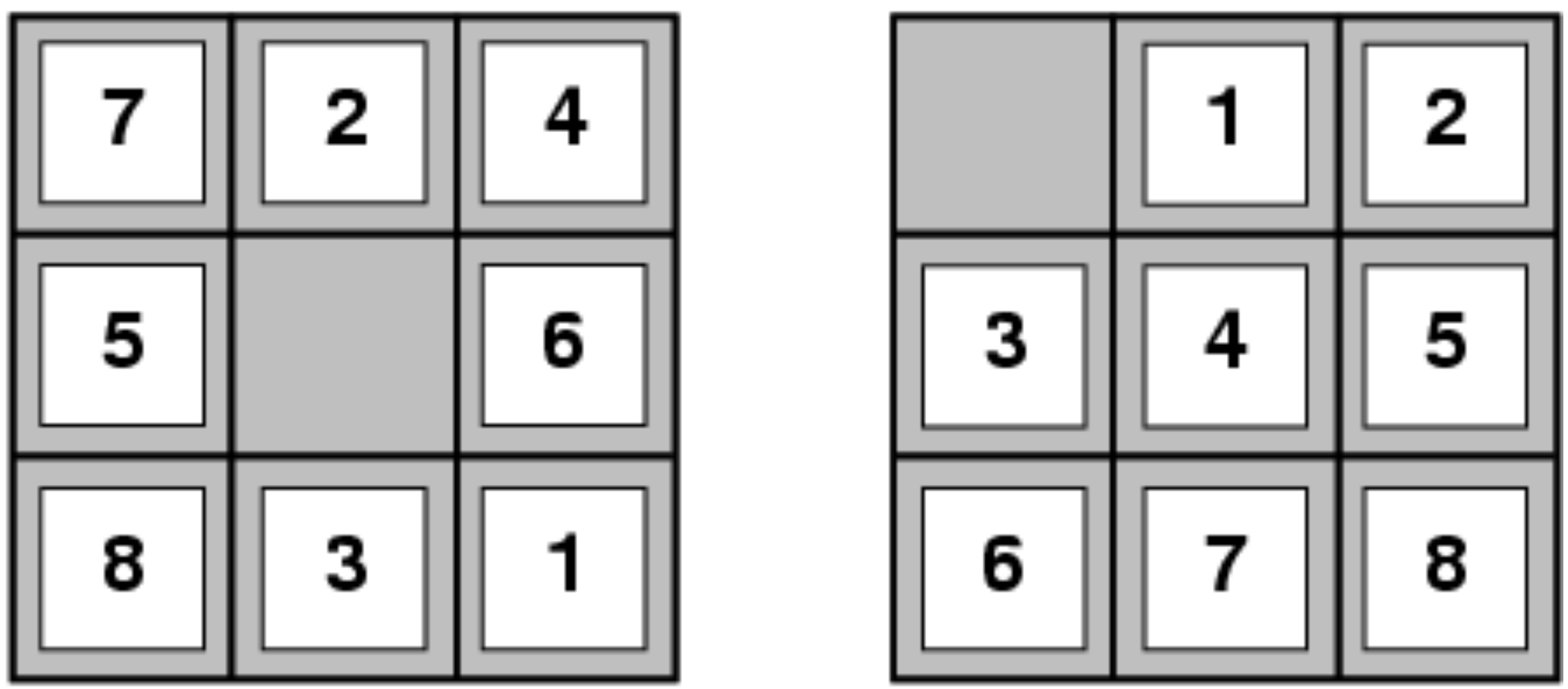
Explicit vs Implicit graphs

- In **explicit graphs** nodes (vertices) and arcs (edges) are readily available. They are read from the input and stored in a data structure such as an adjacency list or an adjacency matrix.
  - The entire graph is in the memory
  - The complexity of algorithms are measured in terms of the number of vertices and/or edges.

- In **implicit graphs** a procedure `outgoing_arcs` (or *neighbours*) is defined that given a node, returns the set of directed arcs that connect the node to other nodes.
  - The graph is generated on the fly.
  - The complexity of algorithms are measured in terms of the *depth* of a goal node and the *average branching factor* (average number of outgoing arcs).

Example: the 8-puzzle

- States: location of tiles
- Actions: Move tiles that can be moved (or move the blank space) left, right, up, down.

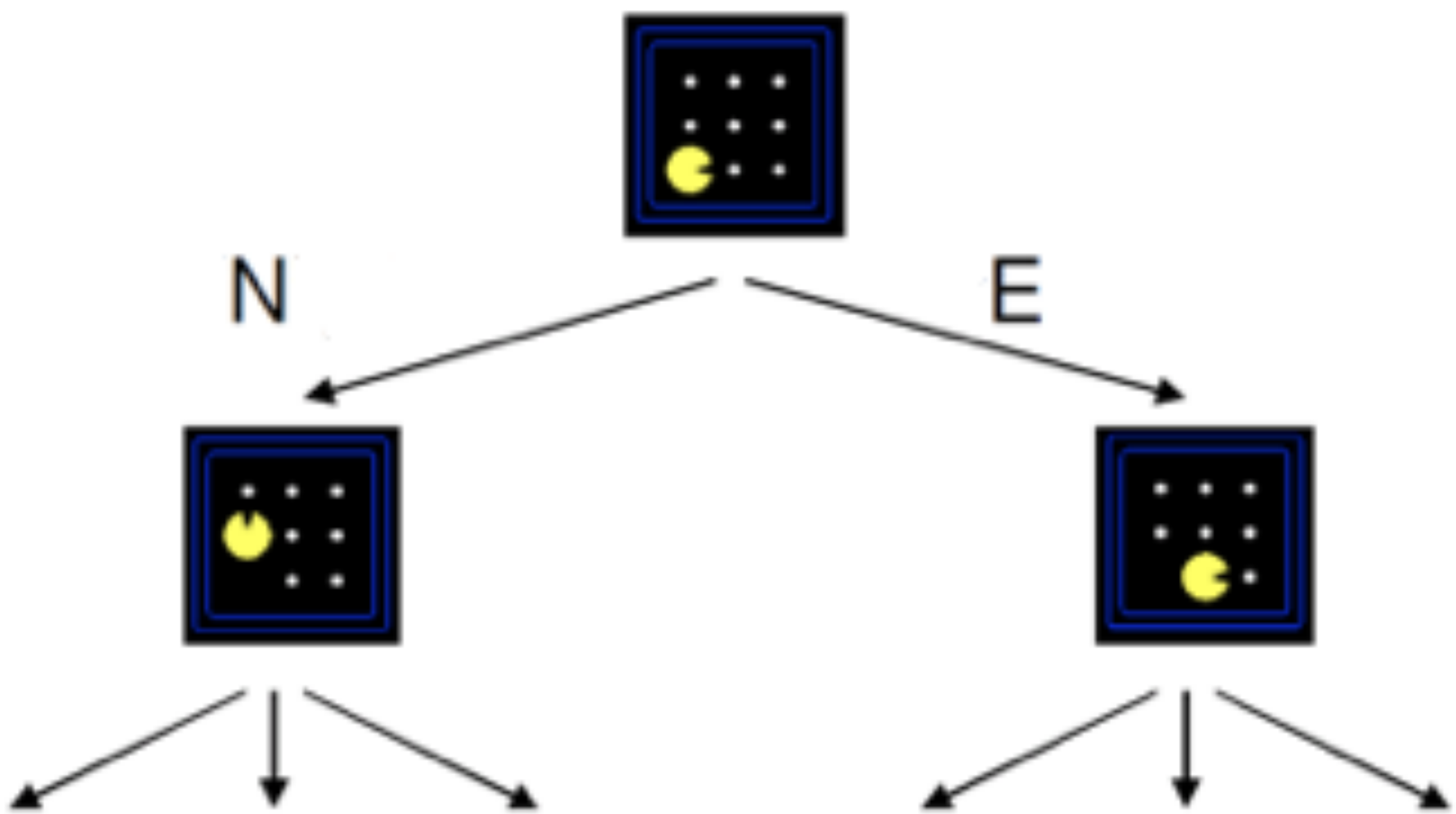


Start State

Goal State

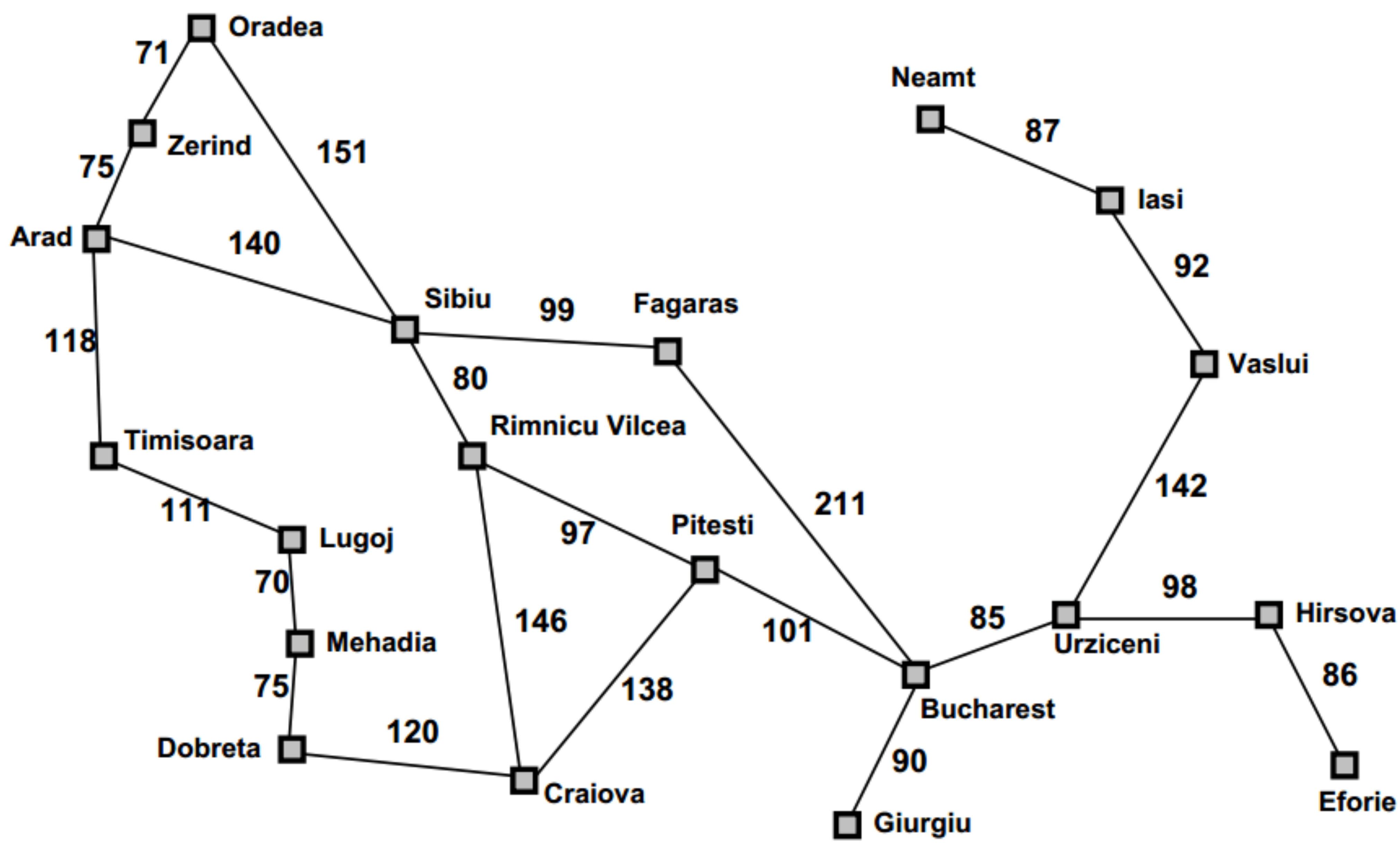
Example: Pac-Man

- States: permutation of food and Pacman in its world.
- Actions: N, E, S, W (when possible)
- Goal states: eaten all the food



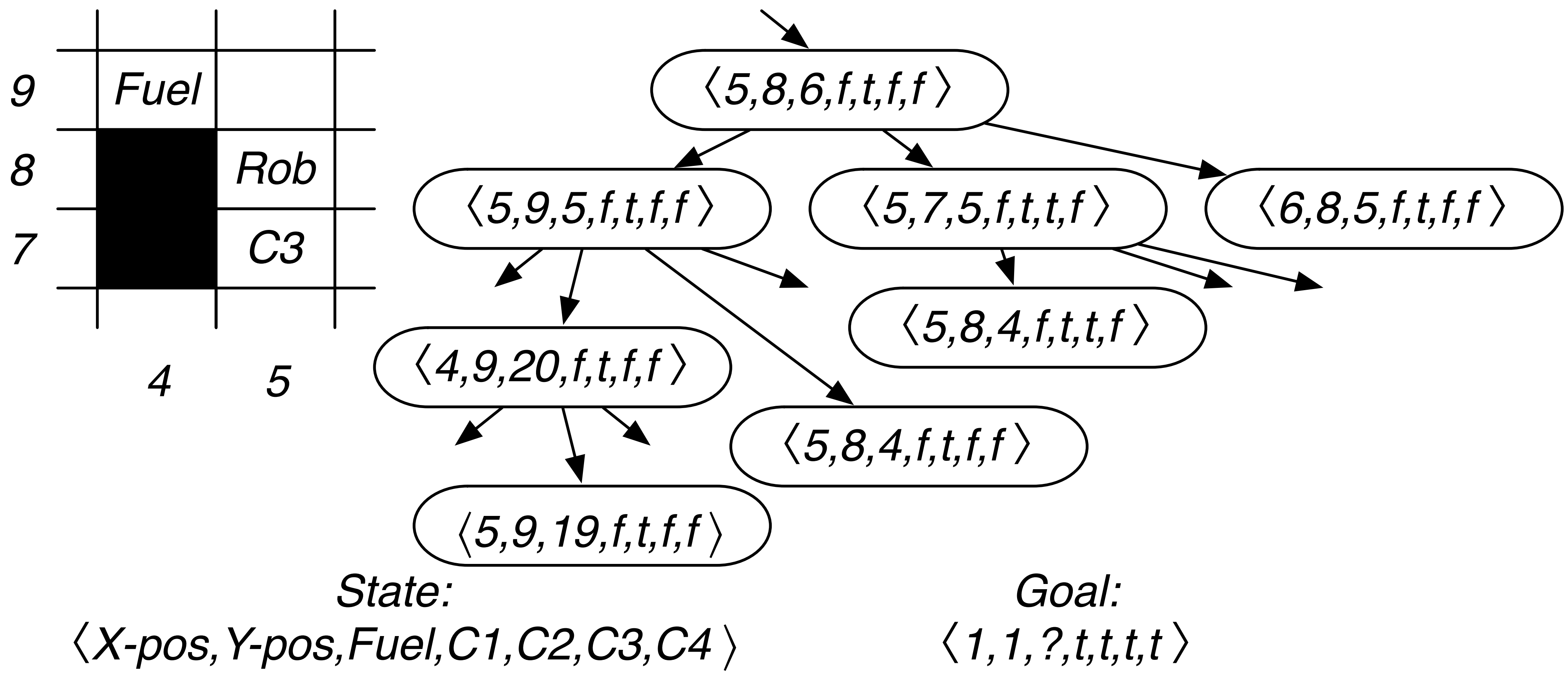
Example: holiday in Romania

- We are on holiday in Romania; currently in Arad. Flight leaves tomorrow from Bucharest. We need to get there.



Example: a grid game

- Grid game: Rob needs to collect coins C1, C2, C3, C4, without running out of fuel, and end up at location (1, 1)



Searching graphs

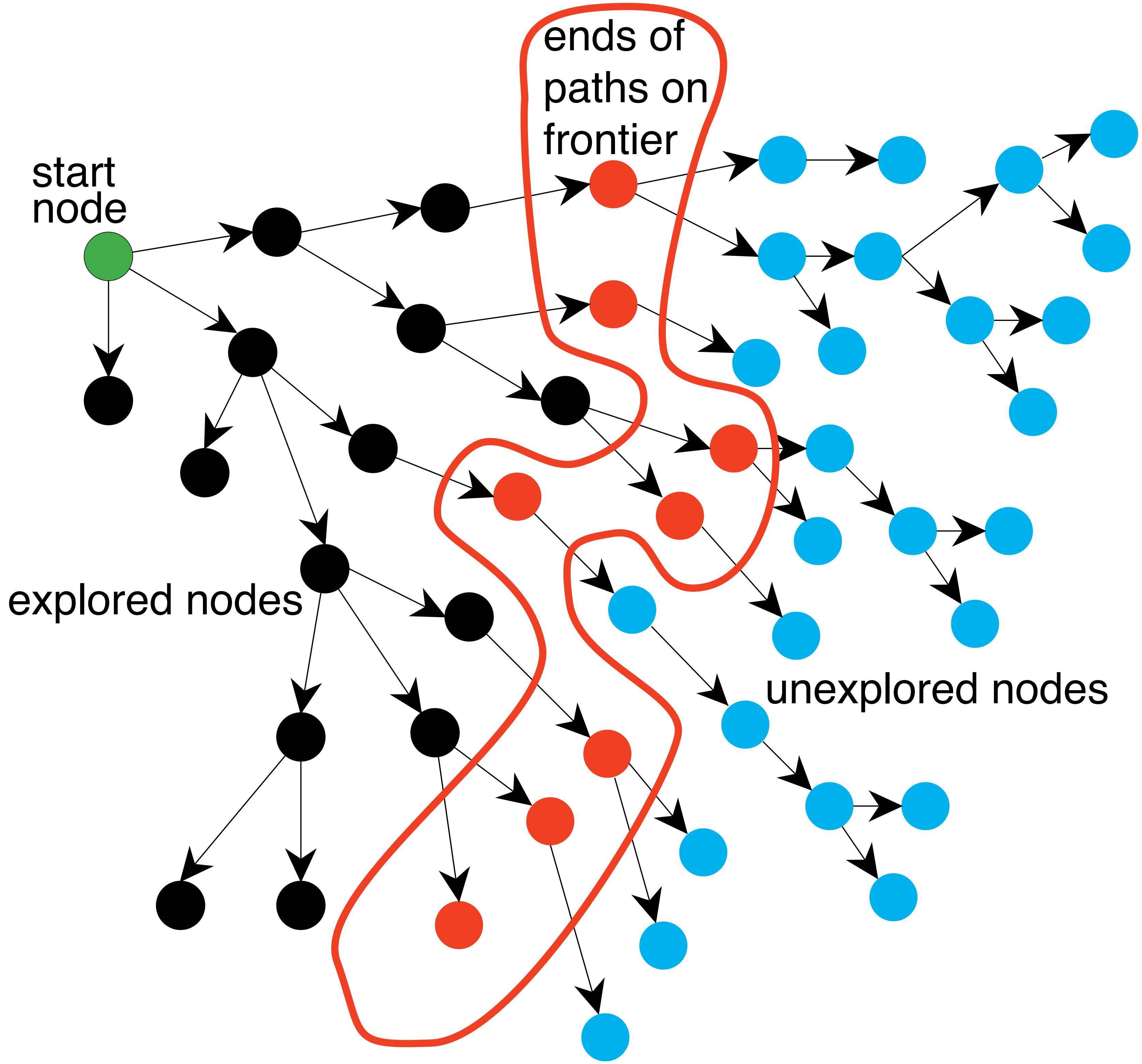
- Generic search algorithm: given a graph, start nodes, and goal nodes, incrementally explore paths from the start nodes.

- Maintain a **frontier** of paths that have been explored. [All the paths are from the starting node(s).]

- As search proceeds, the frontier is updated and the graph is explored until a goal node is encountered.

- The way (order) in which paths are removed from (and added to) the frontier defines the **search strategy**.

Search tree


ends of
paths on
frontier
start
node
explored nodes
unexplored nodes

Generic 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

Remarks

- Which value is selected from the frontier at each stage defines the search strategy. In our implementation we pass a frontier object to the search procedure. The frontier object is in charge of selecting the next path.

- The function *neighbors* defines the graph. In our implementation we use the method `outgoing_arcs` for this purpose.

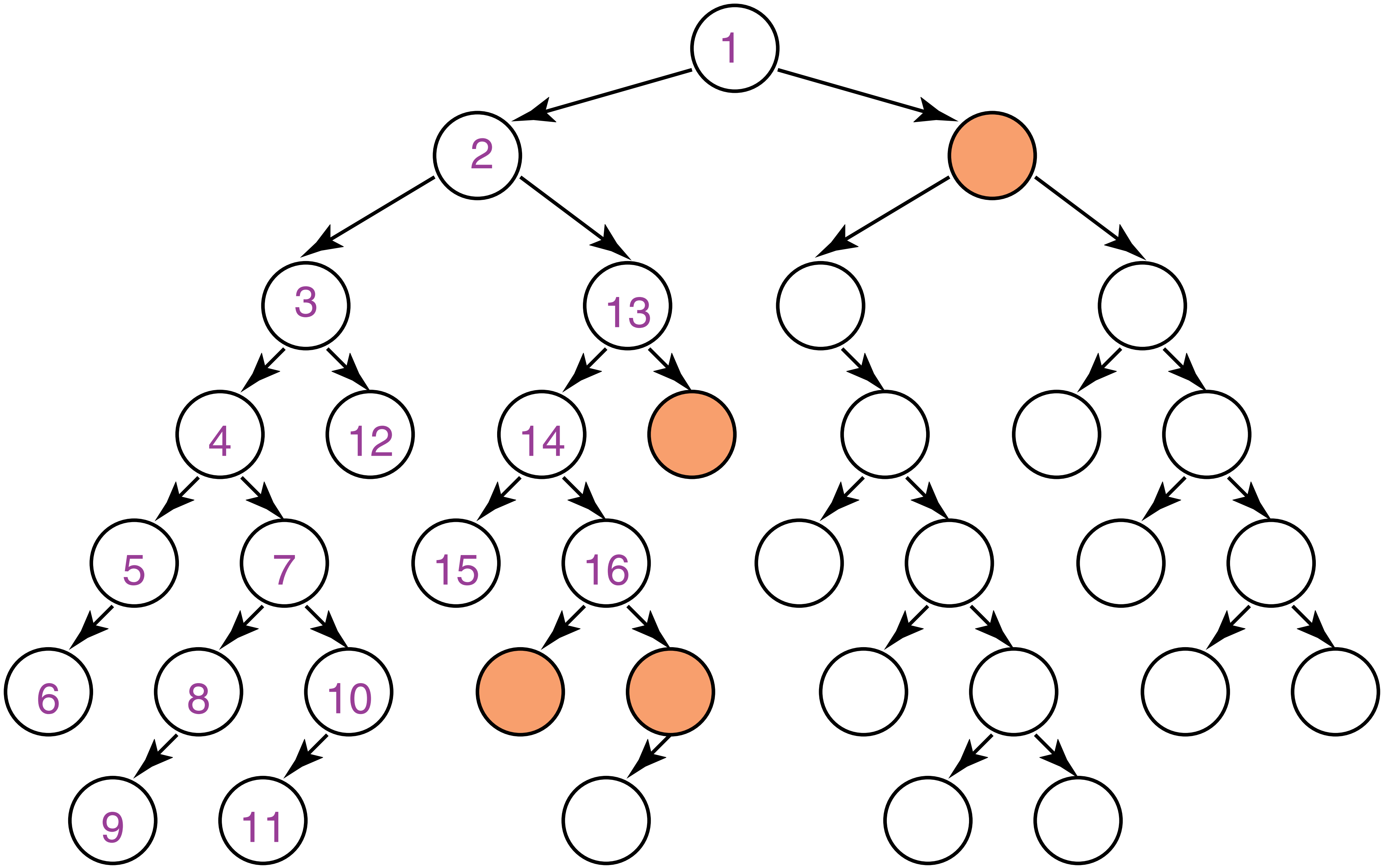
- The function *goal* defines what is a solution. In our implementation we use the method `is_goal` for this purpose.

- If more than one answer is required, the search can continue. For this reason, in our implementation we use `yield` instead of return.

Depth-first search

- In order to perform DFS, the generic graph search must be used with a stack frontier (last in, first out).
- If the stack is a Python list of the form [*..., p, q*] where each element is a path, then
    - *q* is selected (popped).
    - if the algorithm continues, then paths that extend *q* are pushed (appended) to the stack.
    - *p* is only selected (popped) when all paths from *q* have been explored.
- As a result, at each stage, the algorithm expands the deepest path.

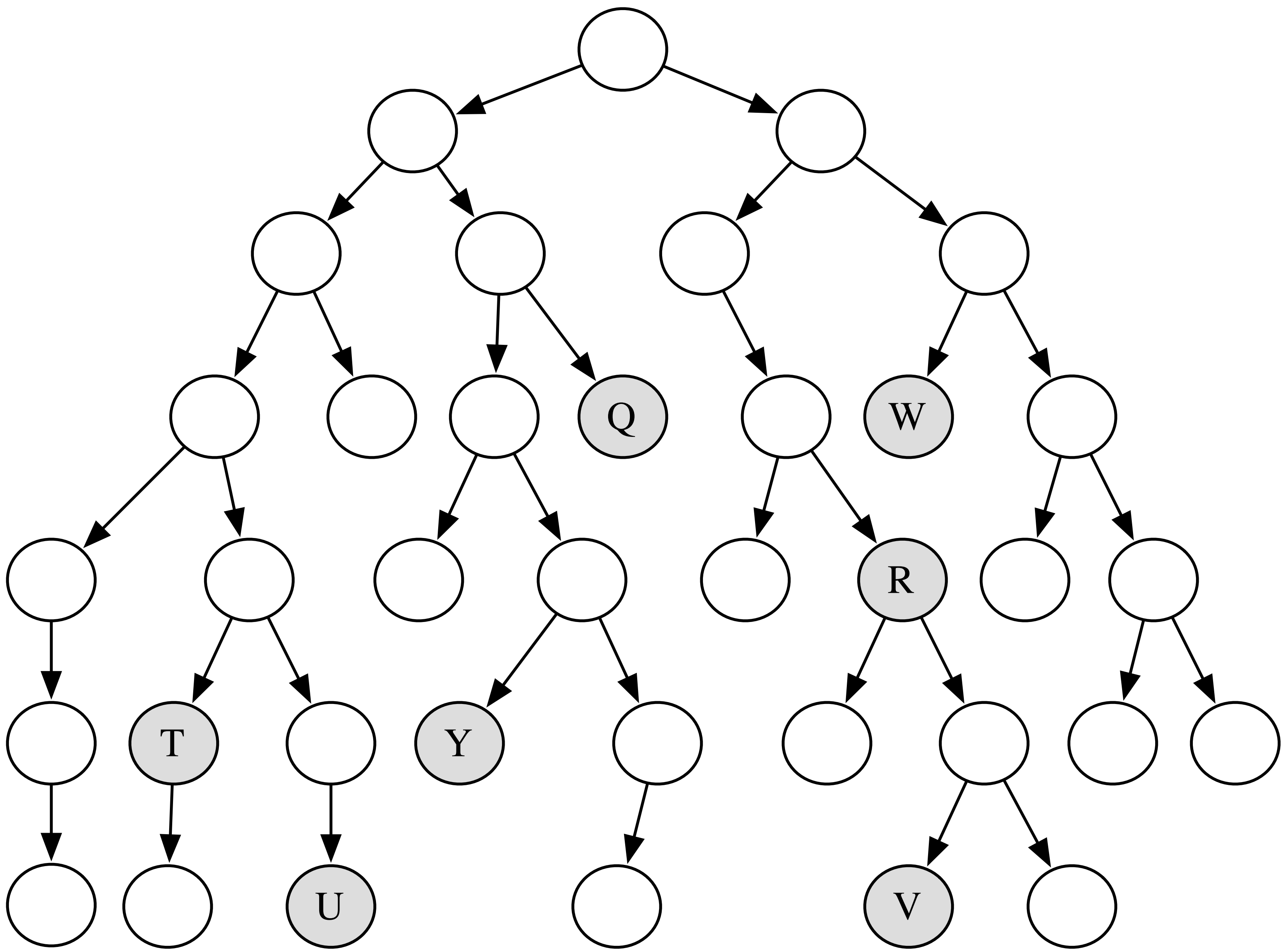
DFS: illustration



- [Assuming that in the sequence of Arcs returned by the function *neighbors* (or the method `outgoing_arcs` in the implementation), <u>the right arc comes before the left arc</u>.]

DFS: behaviour

- Which shaded goal will a depth-first search find first?



DFS: behaviour and complexity

- Does DFS guarantee to find a solution with the fewest arcs (if there is a solution)?
- Is it complete? (i.e. does it guarantee to find a solution if there is one?)
- Does it halt on every graph?
- Does the goal state affect the search behaviour?

Assume a finite search tree of depth ***d*** and branching factor ***b***:

- What is the time complexity?
- What is the space complexity?

Explicit graphs in quizzes

- In some exercises we use small explicit graphs to study the behaviour of various search frontiers.

- Nodes are specified in a set. The use of set should remind you that <u>the order of the nodes does not matter</u>. The elements of the sets are strings (node names).

```
nodes={‘a’, ‘b’, ‘c’, ‘d’}
```

- Edges are specified in a list. The use of list should remind you that <u>the order of the edges matters</u>. The elements are either:
  - pairs of nodes (tail, head); or
  - triples of nodes (tail, head, cost).

```
edge_list = [(‘a’,’b’,5), (‘a’, ’c’, 3)]
```

How do we trace the frontier

- Starting with an empty frontier we record all the calls to the frontier: to add or to get a path. We dedicate one line per call.
- When we ask the frontier to add a path, we start the line with a **+** followed by the path that is being added.
- When we ask for a path from the frontier, we start the line with a **-** followed by the path that is being removed.
- When using a priority queue, the path is followed by a comma and then the key (e.g. cost, heuristic, f-value, …).
- The lines of the trace should match the following regular expression (case and space insensitive): `^[+-][a-z]+(,\d+)?!?$`

Example: tracing frontier in DFS

Given the following graph

nodes={a, b, c, d},

edge_list=[(a,b), (a,d), (a, c), (c, d)],

starting_nodes = [a],

goal_nodes = {d}

trace the frontier in depth-first search (DFS).

Answer:

```
+ a
- a
+ ab
+ ad
+ ac
- ac
+ acd
- acd
```

Breadth-first search

- In order to perform BFS, the generic graph search must be used with a queue frontier (first in, first out).

- If the queue is a Python deque of the form [*p, q, ..., r*], then

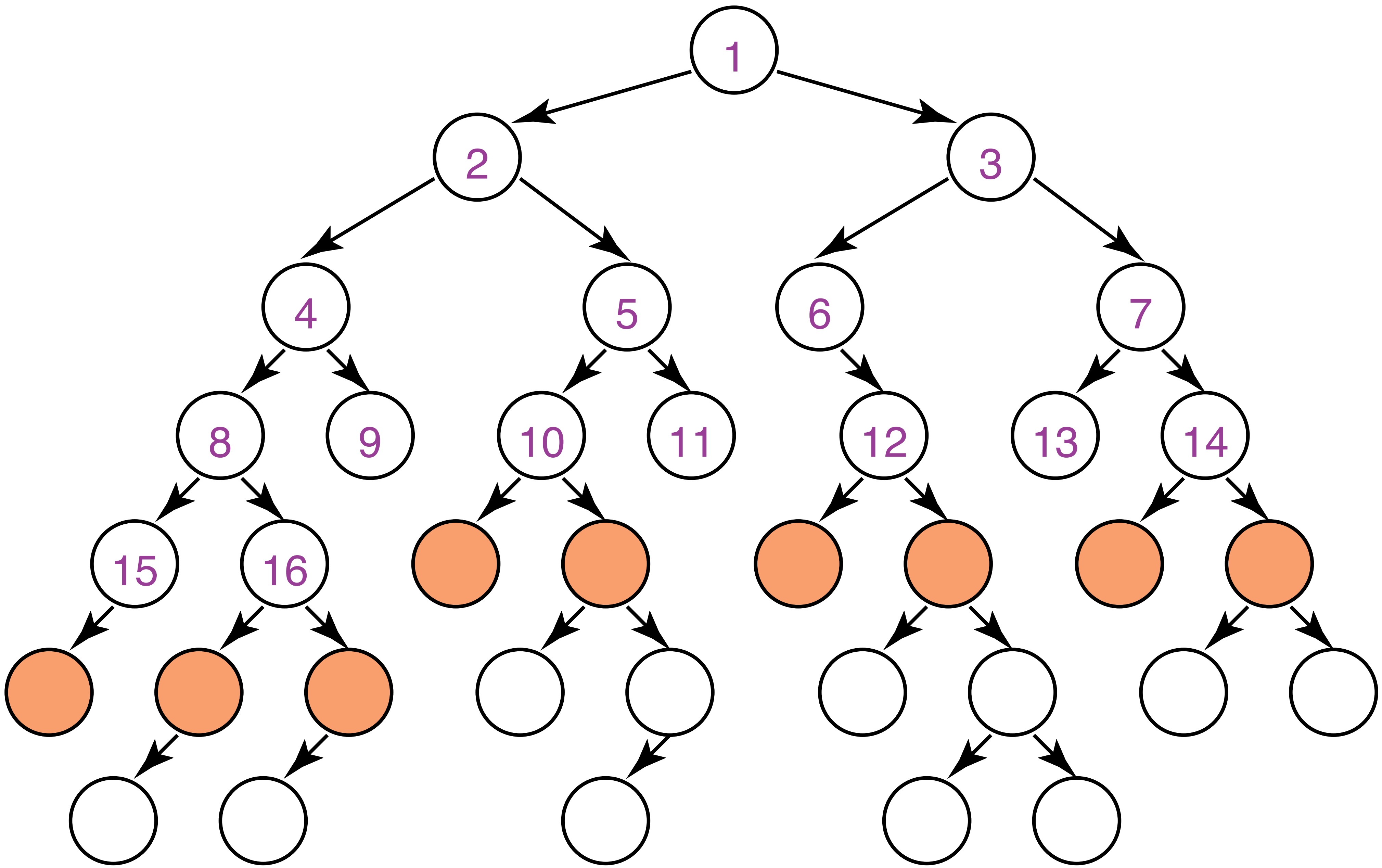
  - *p* is selected (dequeued from left).

  - if the algorithm continues then paths that extend *p* are enqueued (appended) to the queue after *r*.

  - in the next iteration *q* is selected (dequeued) from the frontier.

- As a result, at each stage, the algorithm expands the shallowest path.

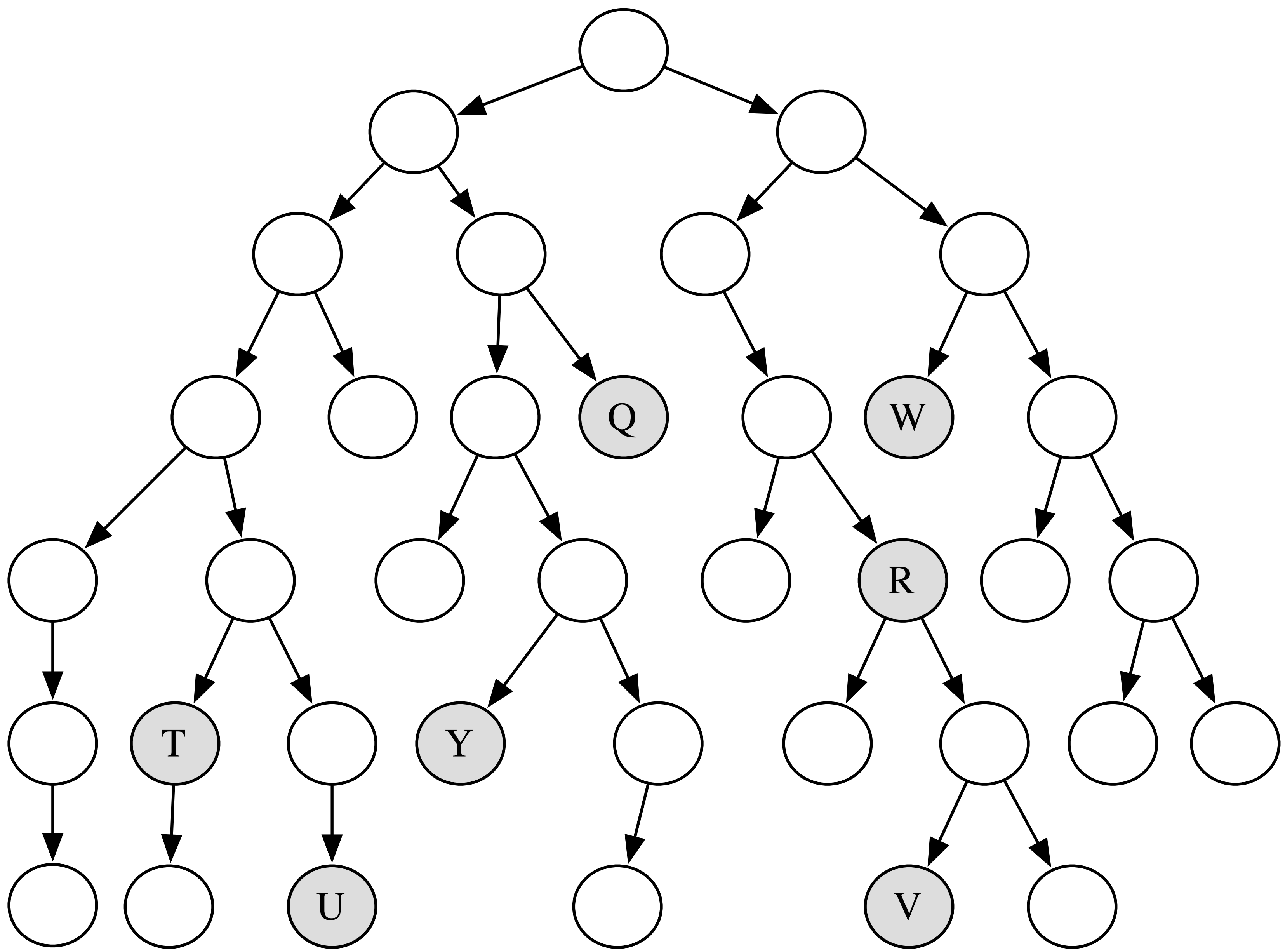
BFS: illustration of search tree



- [Assuming that in the sequence of Arcs returned by the function *neighbors* (or the method `outgoing_arcs` in the implementation), <u>the left arc comes before the right arc</u>.]

BFS: behaviour

- Which shaded goal will a breadth-first search find first?



BFS: behaviour and complexity

- Does BFS guarantee to find a solution with the fewest arcs (if there is a solution)?

- Is it complete? (i.e. does it guarantee to find a solution if there is one?)

- Does it halt on every graph?

- Does the goal state affect the search behaviour?

Assume a finite search tree of depth ***d*** and branching factor ***b***:

- What is the time complexity?

- What is the space complexity?

Example: tracing frontier in BFS

Given the following graph

nodes={a, b, c, d},

edge_list=[(a,b), (a,d), (a, c), (c, d)],

starting_nodes = [a],

goal_nodes = {d}

trace the frontier in breadth-first search (BFS).

Answer:

```
+ a
- a
+ ab
+ ad
+ ac
- ab
- ad
```

Lowest-cost-first search (LCFS)

- The cost of a path is the sum of the costs of its arcs.
- LCFS selects a path on the frontier with the lowest cost.
- The frontier is a priority queue ordered by path cost.
- LCFS finds an optimal solution: a least-cost path to a goal node.
- Another name for this algorithm is *uniform-cost search* (which is somewhat misleading).

Priority queue refresher

1. A container in which each element has a priority (cost).

2. An element (path) with higher priority (lower cost) is always selected/removed/dequeued before an element with lower priority (higher cost).

3. We require the priority queue to be stable: if two or more elements have the same priority, elements that were enqueued earlier are dequeued earlier.

4. In Python you can use `heapq`. You need to store objects in a way that the above properties hold.

Priority queue example

On an empty frontier, after executing:

```
+ c, 5
+ b, 10
+ a, 5
+ d, 10
```

a sequence of selection/removals yields:

```
- c, 5
- a, 5
- b, 10
- d, 10
```

Example: tracing frontier in LCFS

Given the following graph

nodes={a, b, c, d, g},

edge_lists=[(a,b,4), (a,c,2), (a,d,1),
(b,g,4), (c,g,2), (d,g,4)],

starting_nodes = [a],

goal_nodes = {g}

trace the frontier in lowest-cost-first search (LCFS).

Answer:

```
+ a, 0
- a, 0
+ ab, 4
+ ac, 2
+ ad, 1
- ad, 1
+ adg, 5
- ac, 2
+ acg, 4
- ab, 4
+ abg, 8
- acg, 4
```