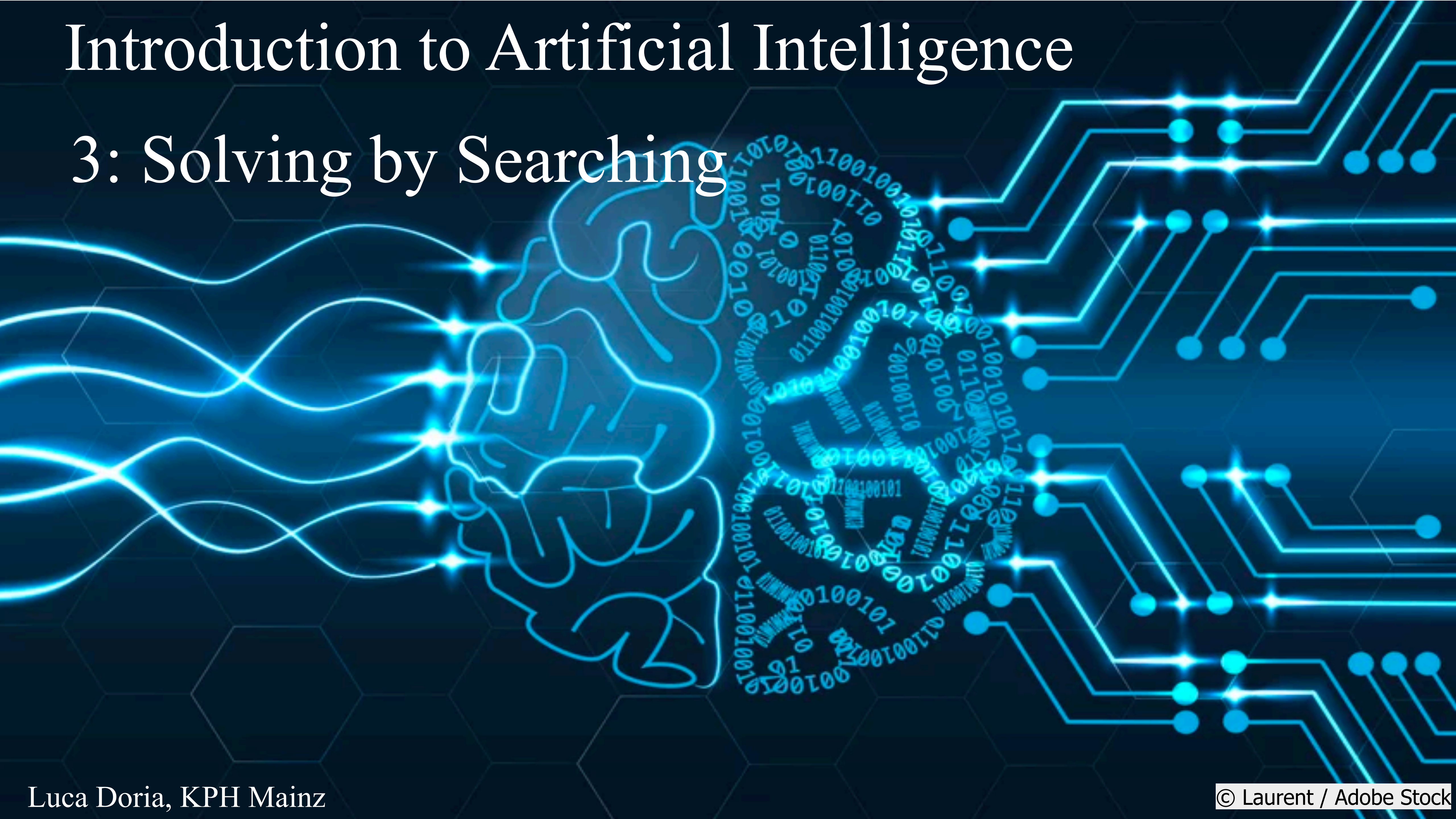


# Introduction to Artificial Intelligence

## 3: Solving by Searching





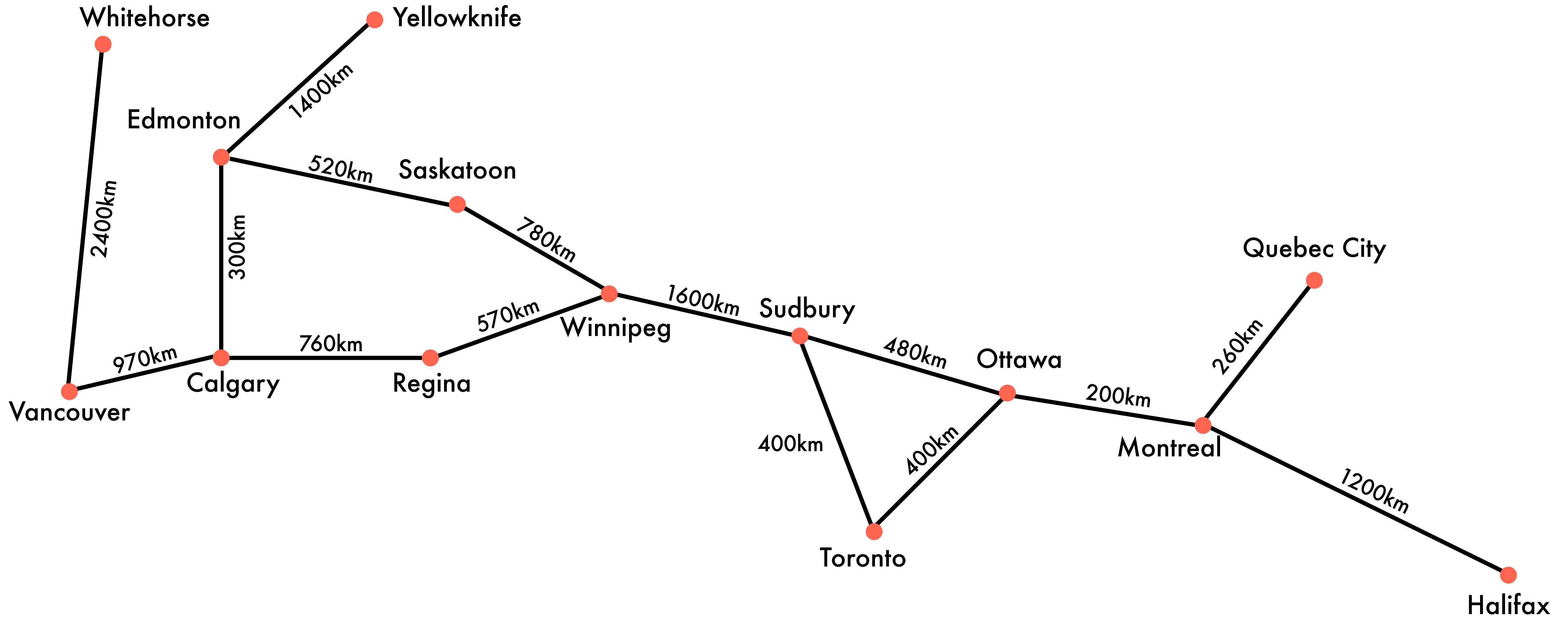
# Problem-Solving Agents

- Agents can better solve problems by planning ahead.
- Planning might imply looking for possible solutions —> **Search Problem**.
- A search can be informed or uninformed.

## Goal-based agents:

- problem described as a state-space and goal(s).
- Goal can be reaching a specific state through appropriate actions
- Search a specific action sequence and execute the actions.

# A Search Problem



# Problem Specification

## Specifications for the Canada Trip case:

A **set of states** (the cities and their distances)

The **initial state** (initial city)

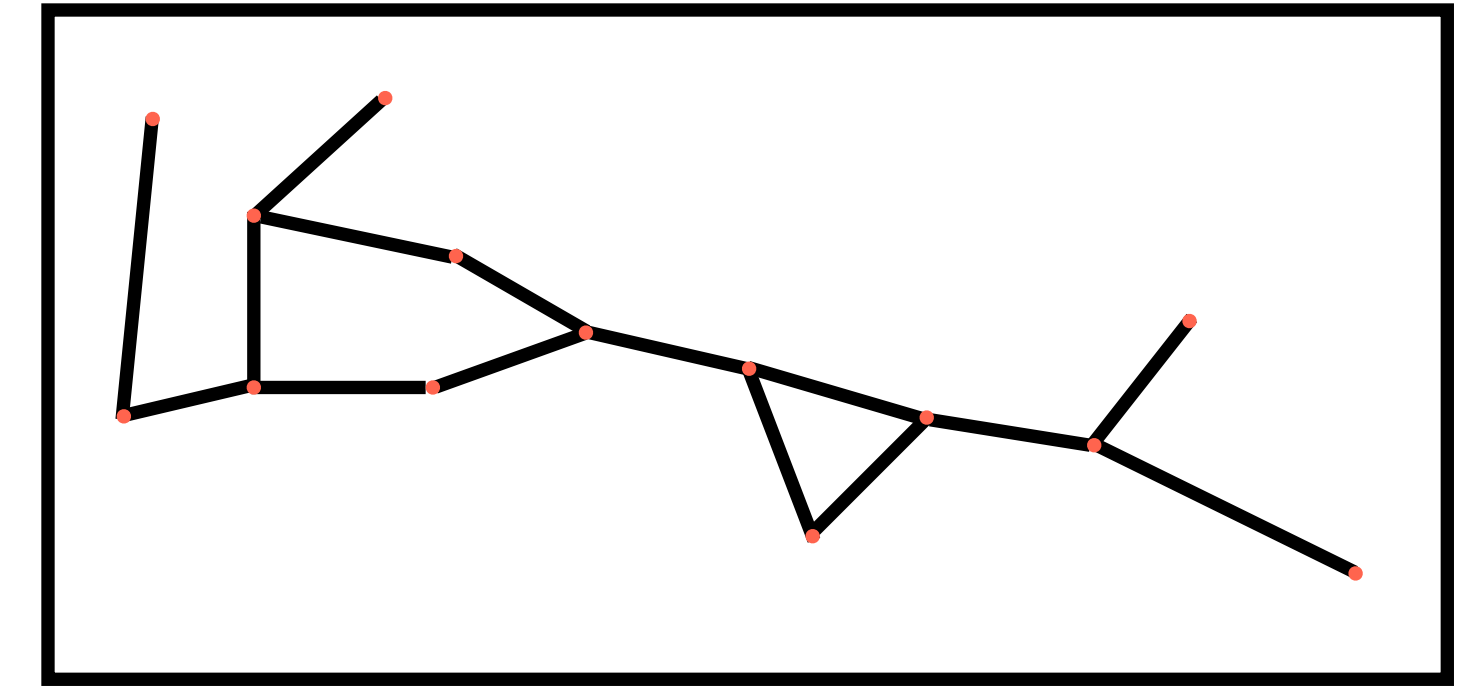
A **set of goal states** (final city, final city with shorter trip, ...)

**Actions:** ACTION(Ottawa) = {toSudbury, toToronto, toMontreal}

A **transition model:** RESULT(state,action)=state  $\longrightarrow$  e.g.: RESULT(Ottawa,toMontreal)=Montreal

An **action-cost function** ACTION-COST(state, action,state) $\longrightarrow$

e.g. ACTION-COST(Ottawa,toMontreal,Montreal)=200



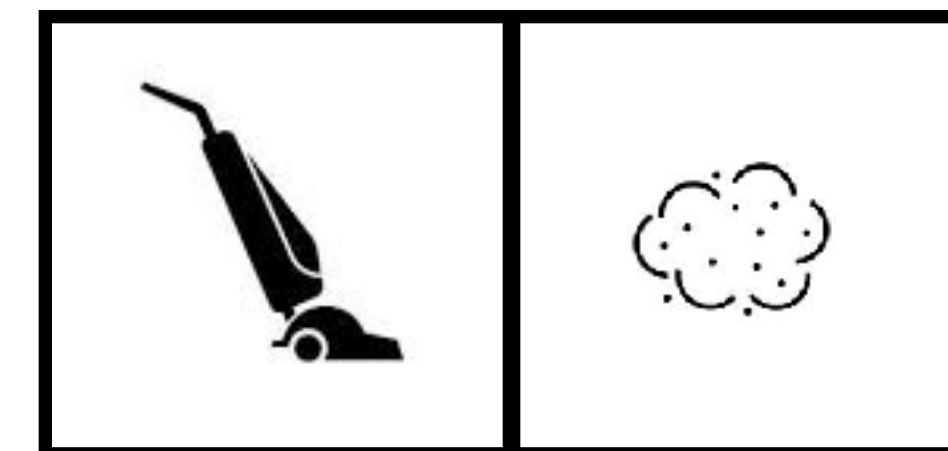
## Specifications for the “vacuum-cleaner problem”:

Set of states: Where is the VC, is(are) the square(s) dirty?

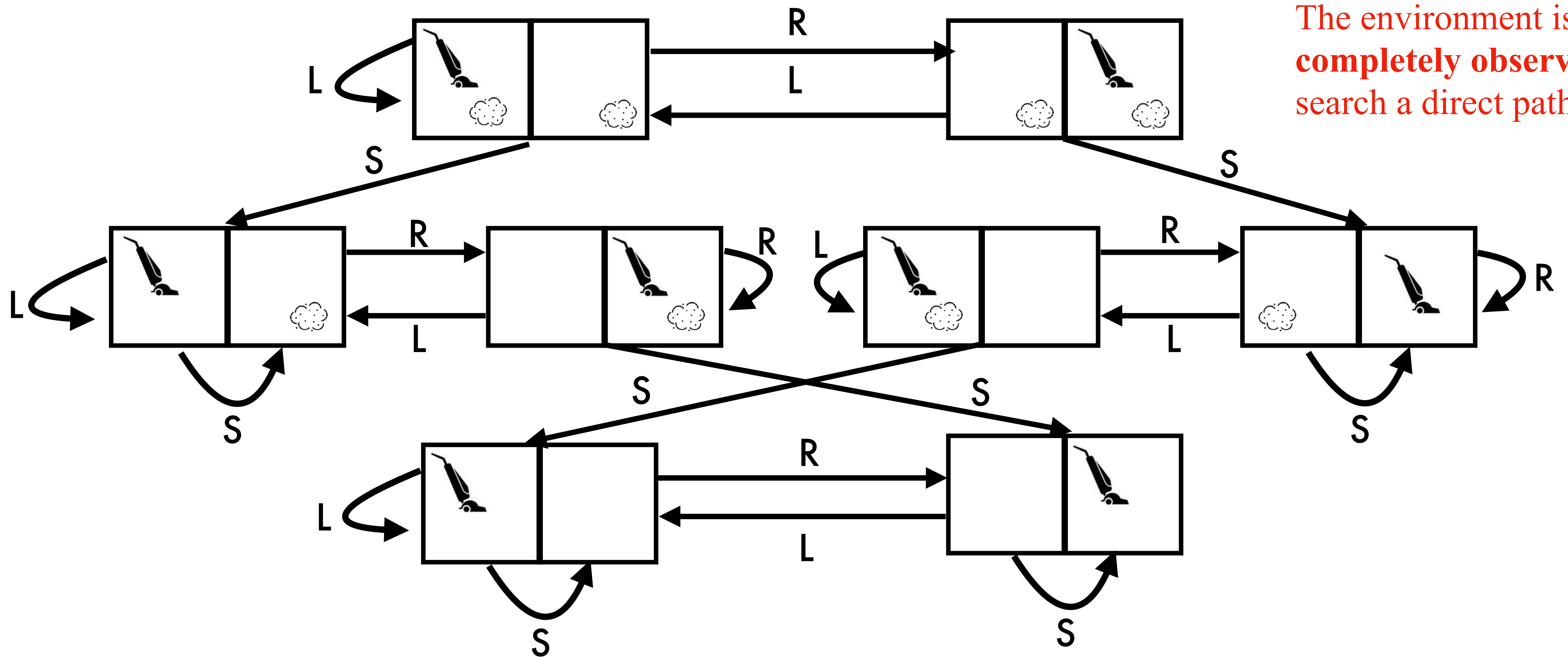
Initial state, Set of goal states: all the squares must be clean.

Actions: MoveDX, MoveDS, on, off

Action cost: e.g. 1.



# The State-Space Graph



The environment is completely observable: search a direct path.

8 states = (2 cells)x(L dirty?)x(R dirty?) = 2x2x2. For an n-cells problem:  $n \times 2^n$  states.  
The state space can become quickly very large!

# Another Example

Given by D. Knuth (1964)

The state-space can be even infinite!

Problem: starting from the integer number 4, reach a certain integer  $x$  using the factorial, floor, and square\_root operations only.

Initial state: 4.

State space: all the positive real numbers.

Actions: Apply sqrt(), floor(), factorial().

Transition model: as defined by the allowed mathematical operations.

Goal:  $x$ .

Action cost: 1.

**Knuth's example**: starting from 4, the shortest path to 5 is:  $\text{floor} \left( \sqrt{\sqrt{\sqrt{\sqrt{\sqrt{(4!)!}}}}} \right) = 5$  (!)

# Another Example: the 8-Puzzle

7	2	4
5		6
8	3	1

Start State

	1	2
3	4	5
6	7	8

Goal State

- States: 8 tiles + blank square
- Initial state: initial configuration of the puzzle
- Actions: moving the blank N,S,E,W
- Goal test: does the present configuration match the “Goal State”?
- Cost function: 1 unit (equal to the move length)



# Notation

- Node expansion**: generating all the successor nodes considering the available actions.
- Frontier**: set of all nodes available for expansion.
- Search Strategy**: defines which node is expanded next.
- Tree-based Search**: some states can repeat, leading to infinite loops. To avoid this, graph searches keeps a set of already visited nodes (the “explored set”).



# Searches on trees and graphs

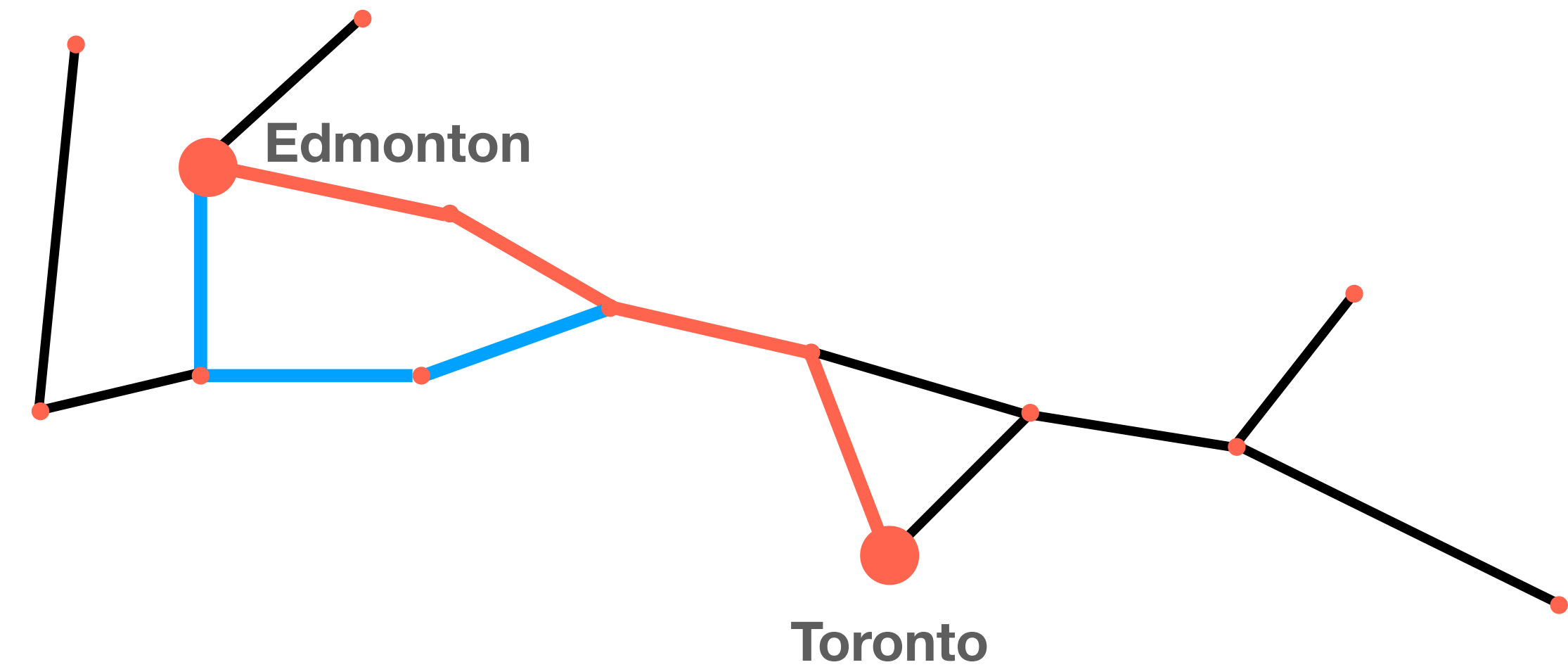
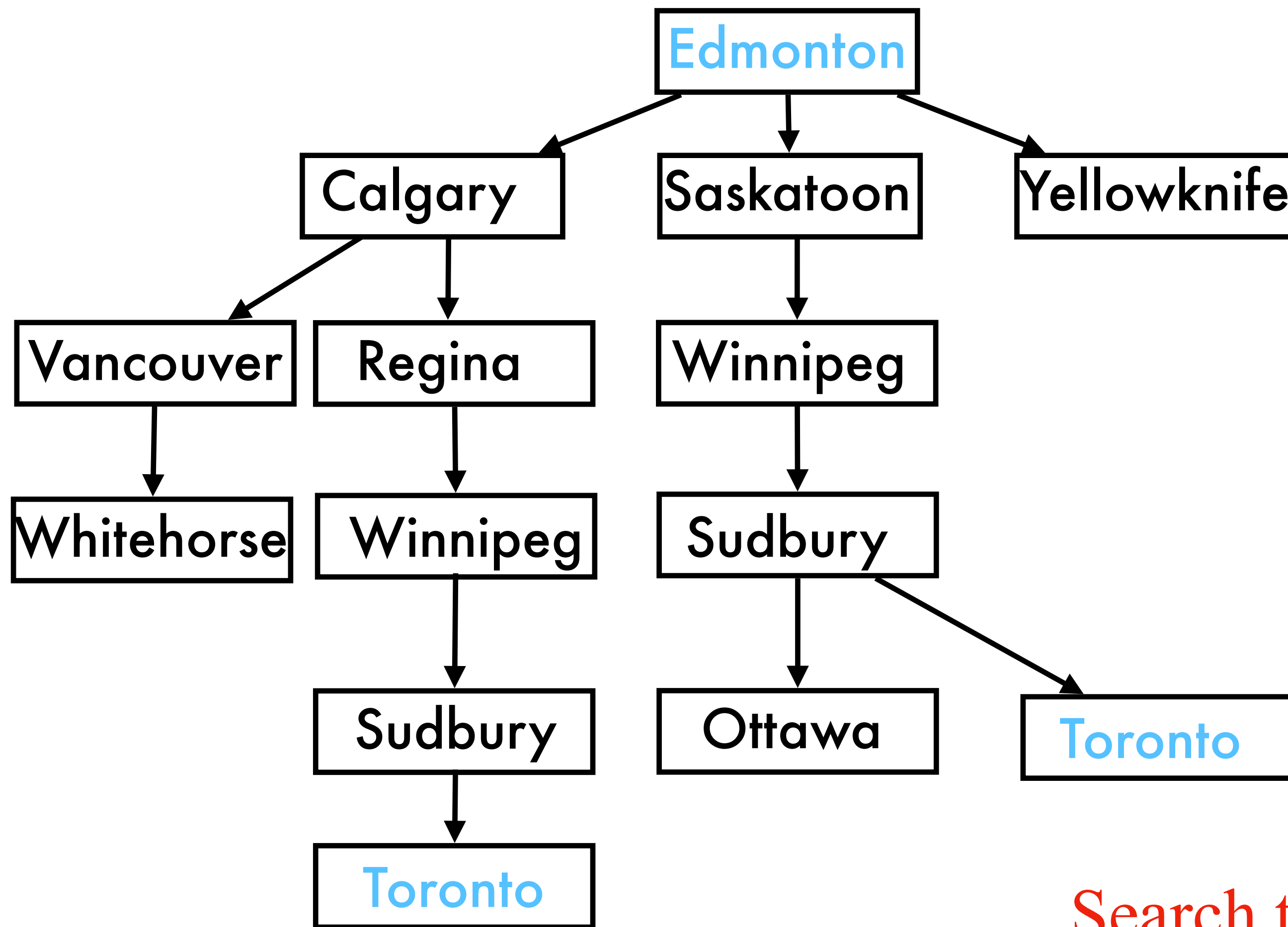
```
function TREE-SEARCH(problem) returns a solution or failure
  initialise the frontier using the initial state of the problem
  loop do:
    if the frontier is empty, then failure
      choose a leaf node and remove it from the frontier
    if the node contains a goal state, then return corresponding solution.
      expand the chosen node, adding the resulting nodes to the frontier.
```

```
function GRAPH-SEARCH(problem) returns a solution or failure
  initialise the frontier using the initial state of the problem
  initialise the explored set to be empty
  loop do:
    if the frontier is empty, then failure
      choose a leaf node and remove it from the frontier
    if the node contains a goal state, then return corresponding solution.
      add the node to the explored set
      expand the chosen node, adding the resulting nodes to the
      frontier only if not in the frontier or explored set.
```

# Searches on trees and graphs

A search tree can be superimposed to the state space.

Example from the Canada trip: a journey from Edmonton to Toronto:



Search tree and state space are two different things!

# Characterization of Search Strategies

- **Completeness**: Is it guaranteed that the strategy will find a solution (if it exists)?
  - **Time Complexity**: How long does it take for finding the solution?
  - **Space Complexity**: How much memory does the search strategy require?
  - **Optimality**: Does the strategy find the “best” solution (with lowest path cost)?
- Problem describing quantities:
- **b**: branching factor
  - **d**: depth of the shallowest goal node
  - **m**: maximum length of any path in the state space.



# Uninformed Search Strategies

How to navigate the state space and find the appropriate action?

The simplest strategy is a through search.

Uninformed (or blind) search strategies have no idea about where the target might be and their only chance of finding it is to perform an exhaustive search.

Examples: breadth-first, depth-first, inform cost search, bidirectional search, iterative-deepening search,...

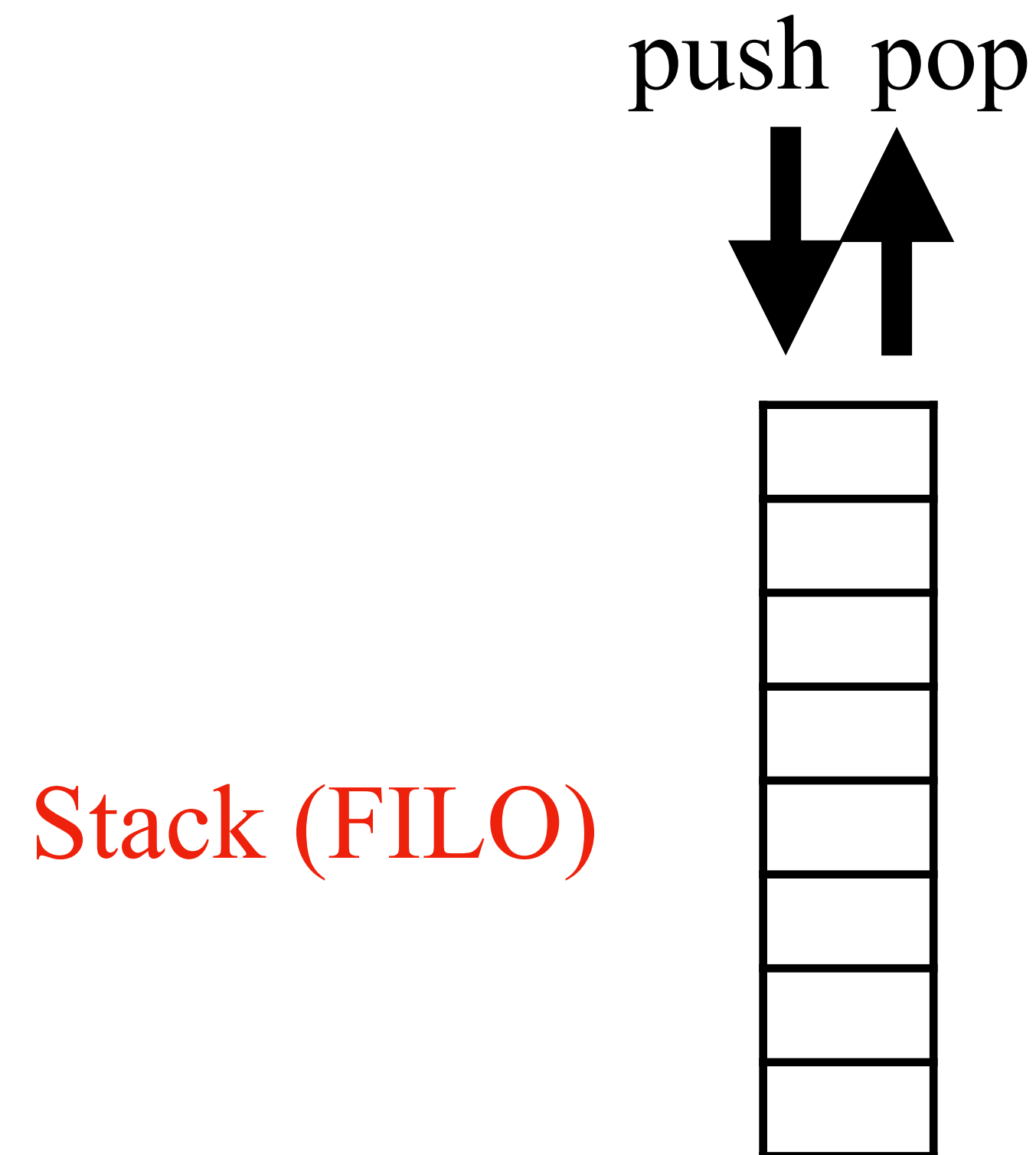
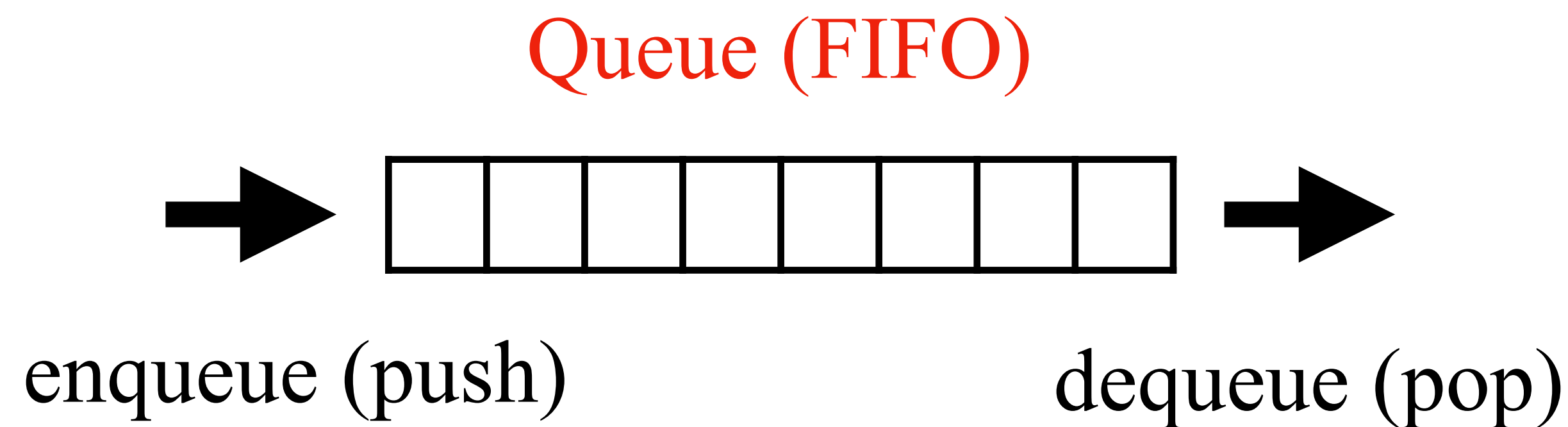
In contrast with informed or heuristic approaches.

# Uninformed Search Strategies

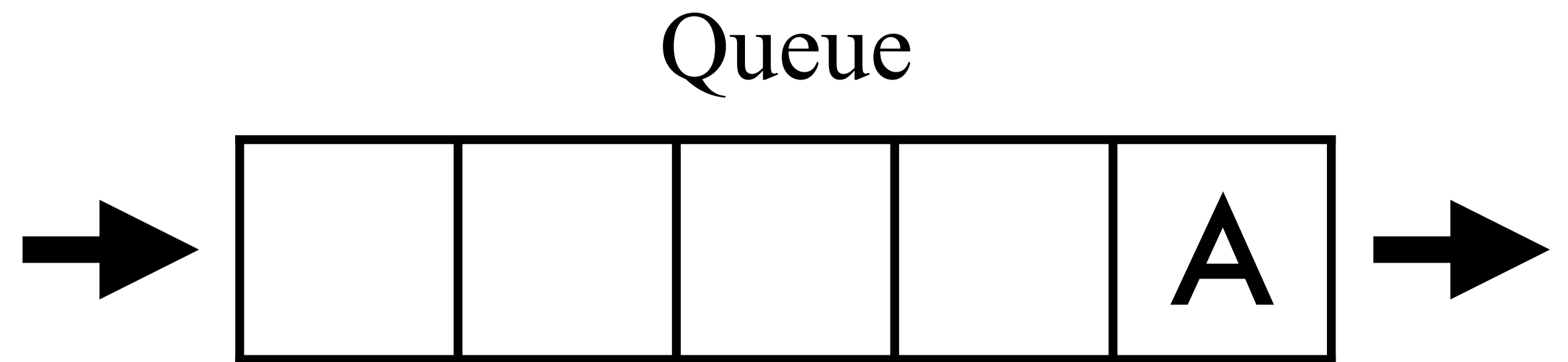
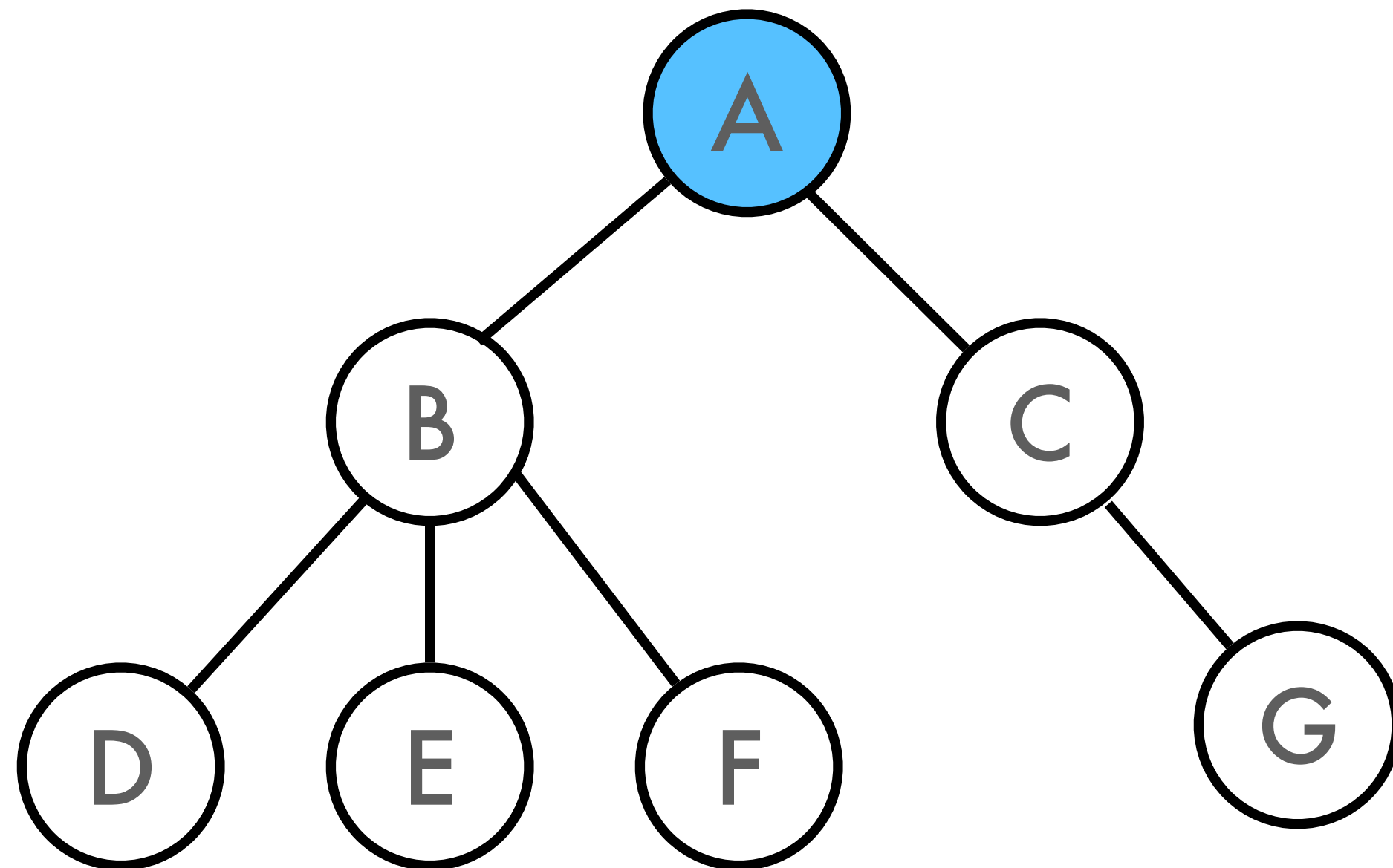
Two basic uninformed algorithms for traversing (searching) trees and graphs:

- **Breadth-first search:** In BFS, start from a node and explore all of its neighbours at the current depth level before moving on to the nodes at the next depth level.
- **Depth-first search:** DFS explores the tree/graph as deep as possible before backtracking.

Based on two basic data structures:



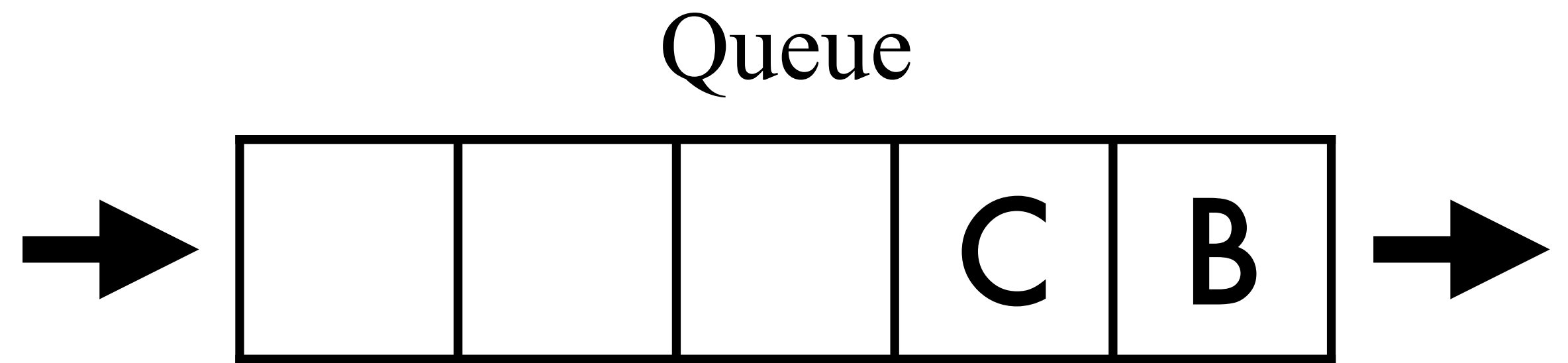
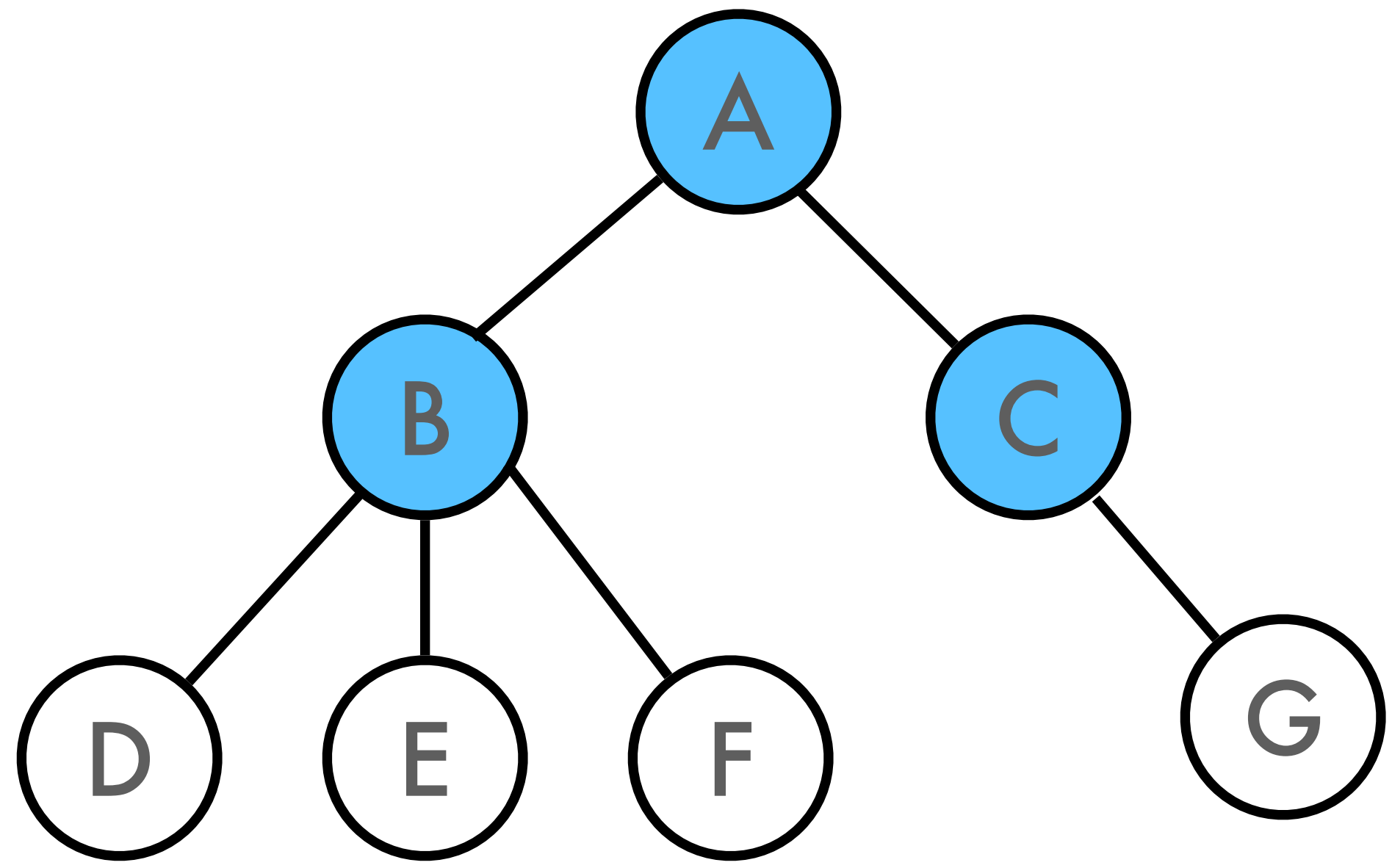
# Breadth-First Search



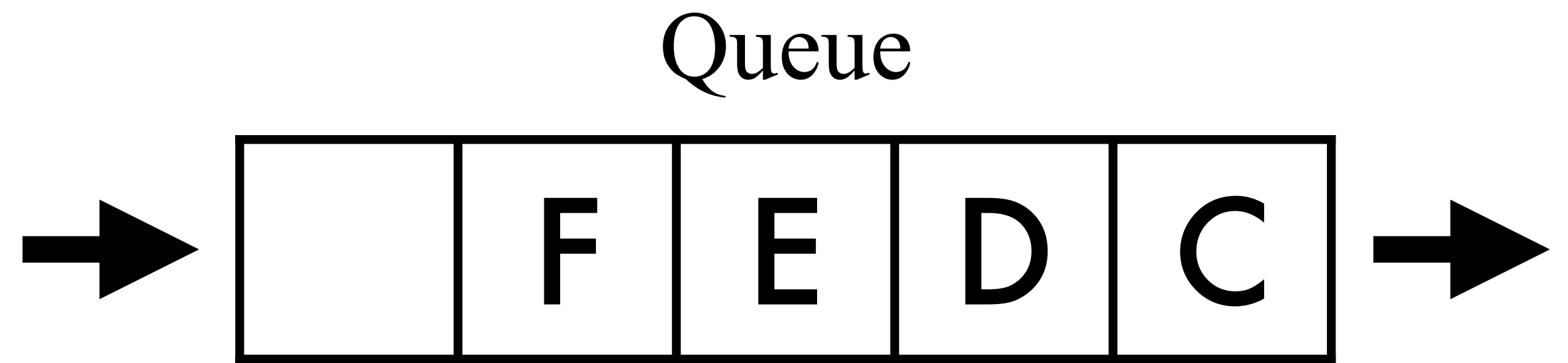
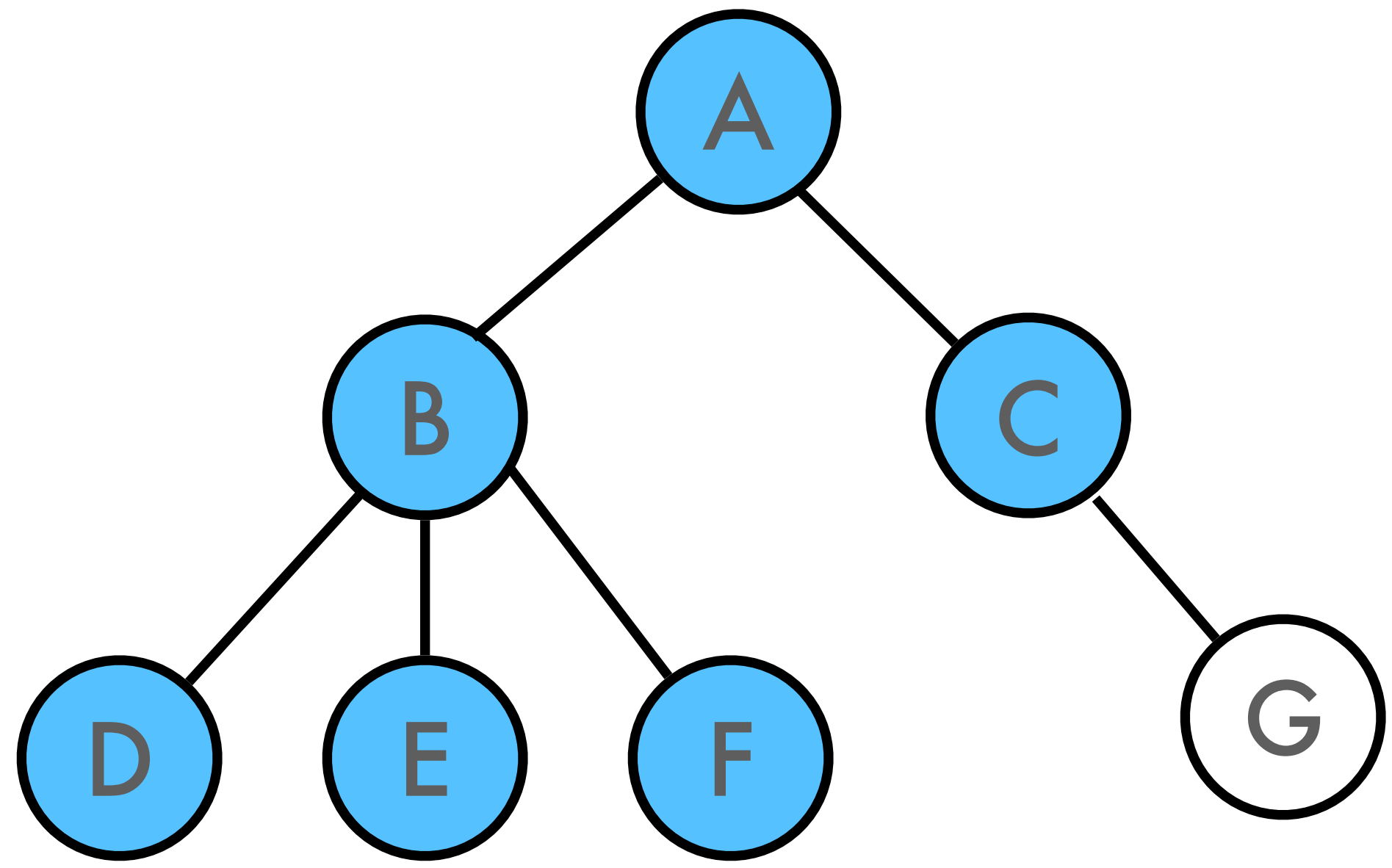
- A **queue** contains the list of the nodes to visit.
- The root node is initially introduced in a queue.
- The children nodes are introduced in the queue.
- The visited nodes are popped from the queue.



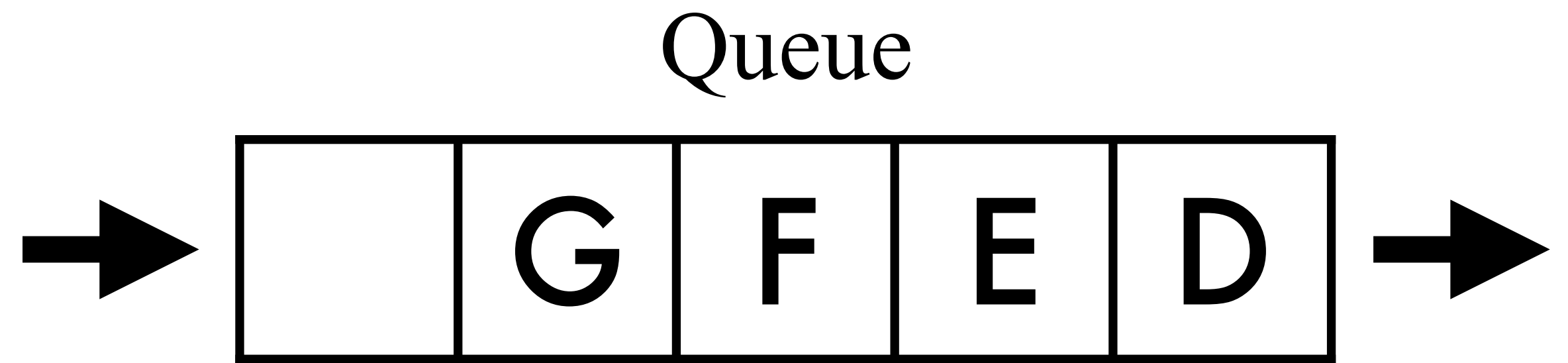
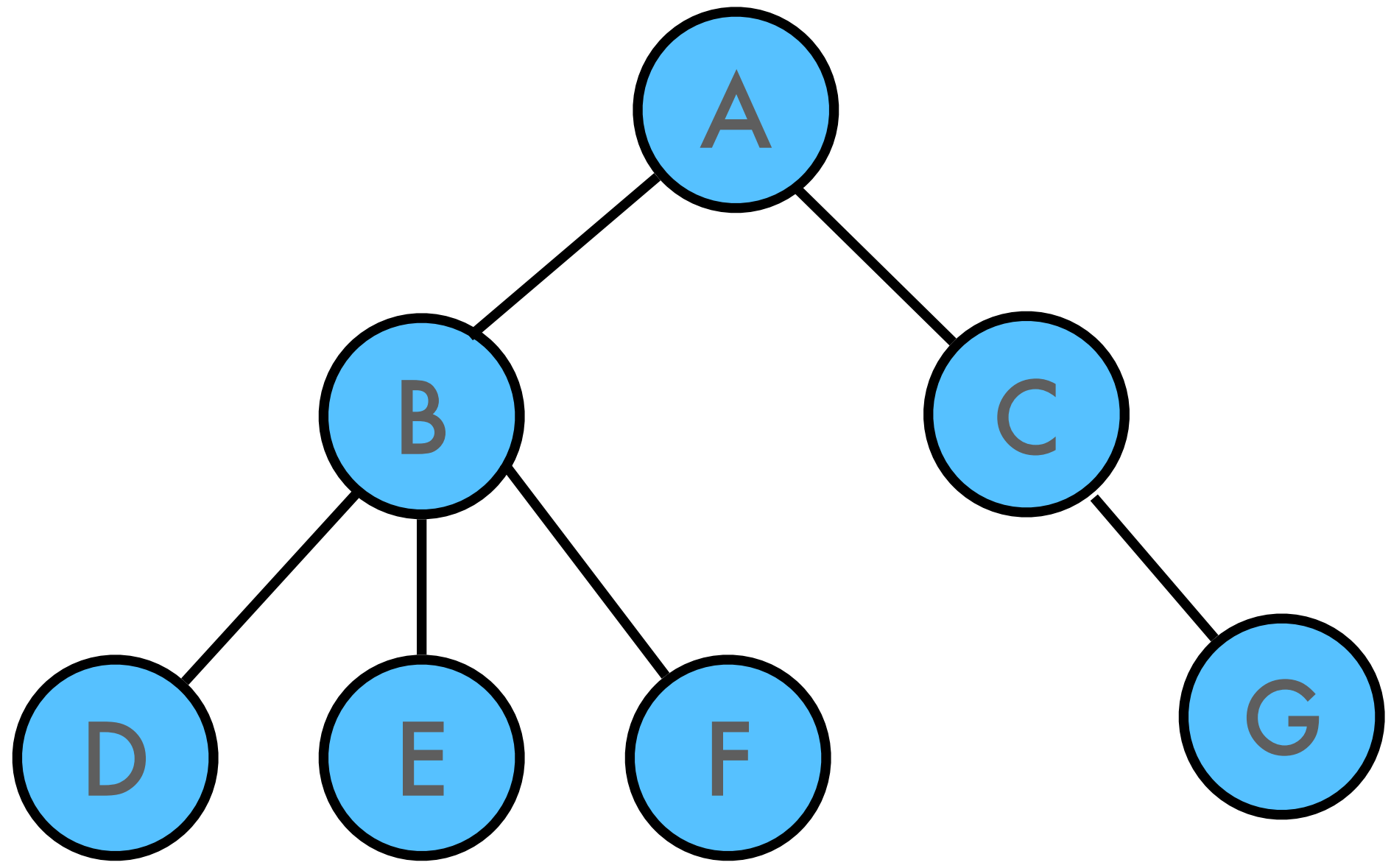
# Breadth-First Search



# Breadth-First Search



# Breadth-First Search





# Breadth-First Search

- The search is **complete**
- The search is **optimal** if the cost is constant and positive
- Time complexity:**
  - $b$  is the maximal branching factor
  - maximal # of expanded nodes:  $T \sim b + b^2 + b^3 + \dots + b^d \sim O(b^d)$
- Space complexity:**
  - every generated node is kept in memory
  - Space needed for the frontier:  $O(b^d)$ .

# Example

- Suppose we have a branching factor of 10.
- We can visit  $10^6$  nodes/second.
- Every node occupies 1kB of memory.

Depth	Nodes	Time	Memory
2	110	11ms	107kB
4	11110	11ms	10.6MB
6	$10^6$	1.1s	1GB
8	$10^8$	2m	103GB
10	$10^{10}$	3h	10TB
12	$10^{12}$	13d	1PB
14	$10^{14}$	3.5y	99PB
21	$10^{21}$	>age universe	$10^6$ EB

# Uniform-Cost Search(\*) (Dijkstra's Algorithm)

If the cost for the possible actions are equal, breadth-first search finds the path with the optimal costs.

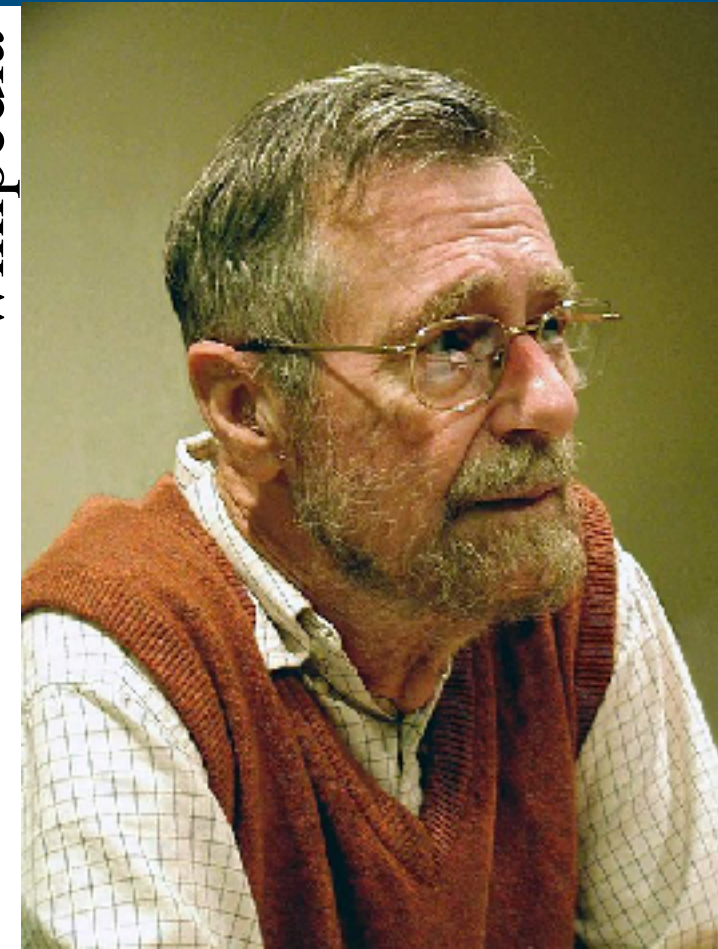
If action costs are different, then **uniform-cost** can find the optimal solution.

Uniform-cost search expands the node with the lowest path cost  $g(n)$ .

The data structure used for realising this search is the **priority queue (heap)**.

Another way to describe the algorithm is as “finding the shortest path between two points”.  
In the graph terminology, we are looking for the shortest path on a **weighted graph**.

wikipedia



Edsger W. Dijkstra  
(1930-2002)

(\*) “Uniform cost search” is the name used in the AI community for the general Dijkstra's Algorithm

# Uniform-Cost Search (Dijkstra's Algorithm)

Problem: shortest path from Regina to Yellowknife

Initialization:

$$g(\text{Regina}) = 0$$

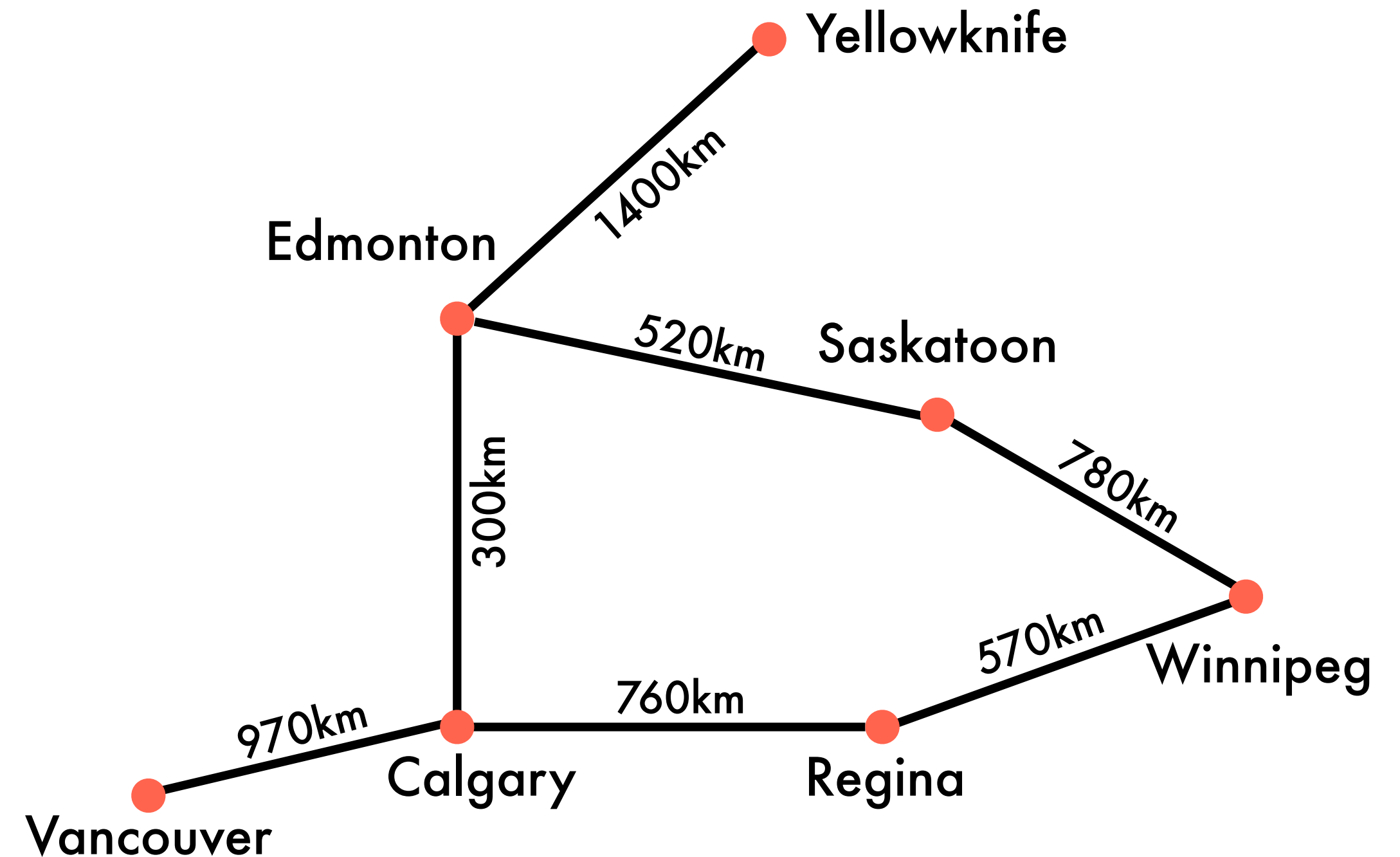
$$g(\text{other cities}) = +\infty$$

Expand “Regina” and put in the priority queue:

$$g(\text{Winnipeg}) = 570$$

$$g(\text{Calgary}) = 760$$

Go for the minimum cost: “Winnipeg”





# Uniform-Cost Search (Dijkstra's Algorithm)

Problem: shortest path from Regina to Yellowknife

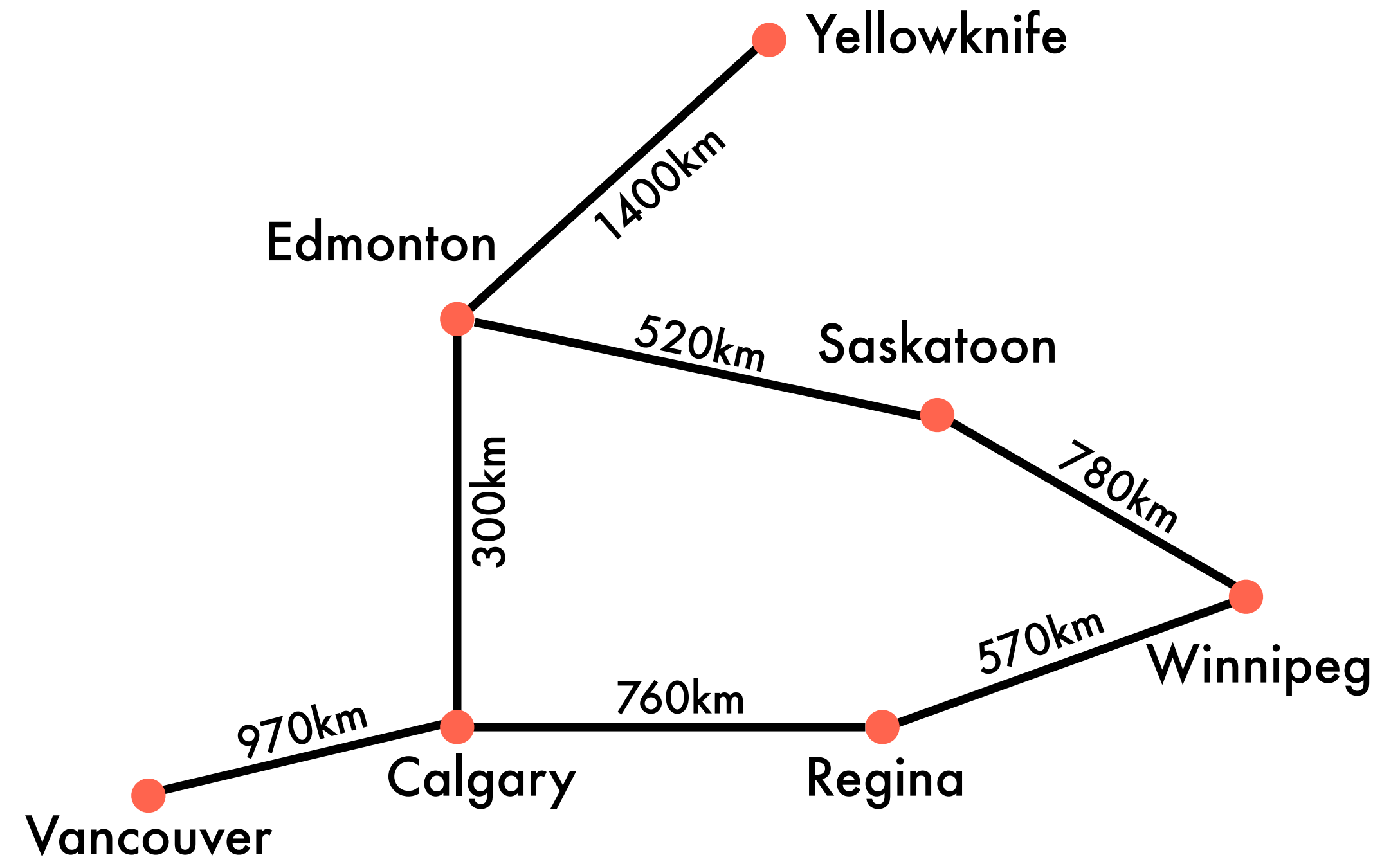
Expand Winnipeg:

$$g(\text{Saskatoon}) = 570 + 780 = 1350$$

Priority queue:

$$g(\text{Calgary}) = 760$$

$$g(\text{Saskatoon}) = 1350 \text{ (from Winnipeg)}$$



# Uniform-Cost Search (Dijkstra's Algorithm)

**Problem:** shortest path from Regina to Yellowknife

Expand Calgary:

$$g(\text{Vancouver}) = 760 + 970 = 1730$$

$$g(\text{Edmonton}) = 760 + 300 = 1060$$

Priority queue:

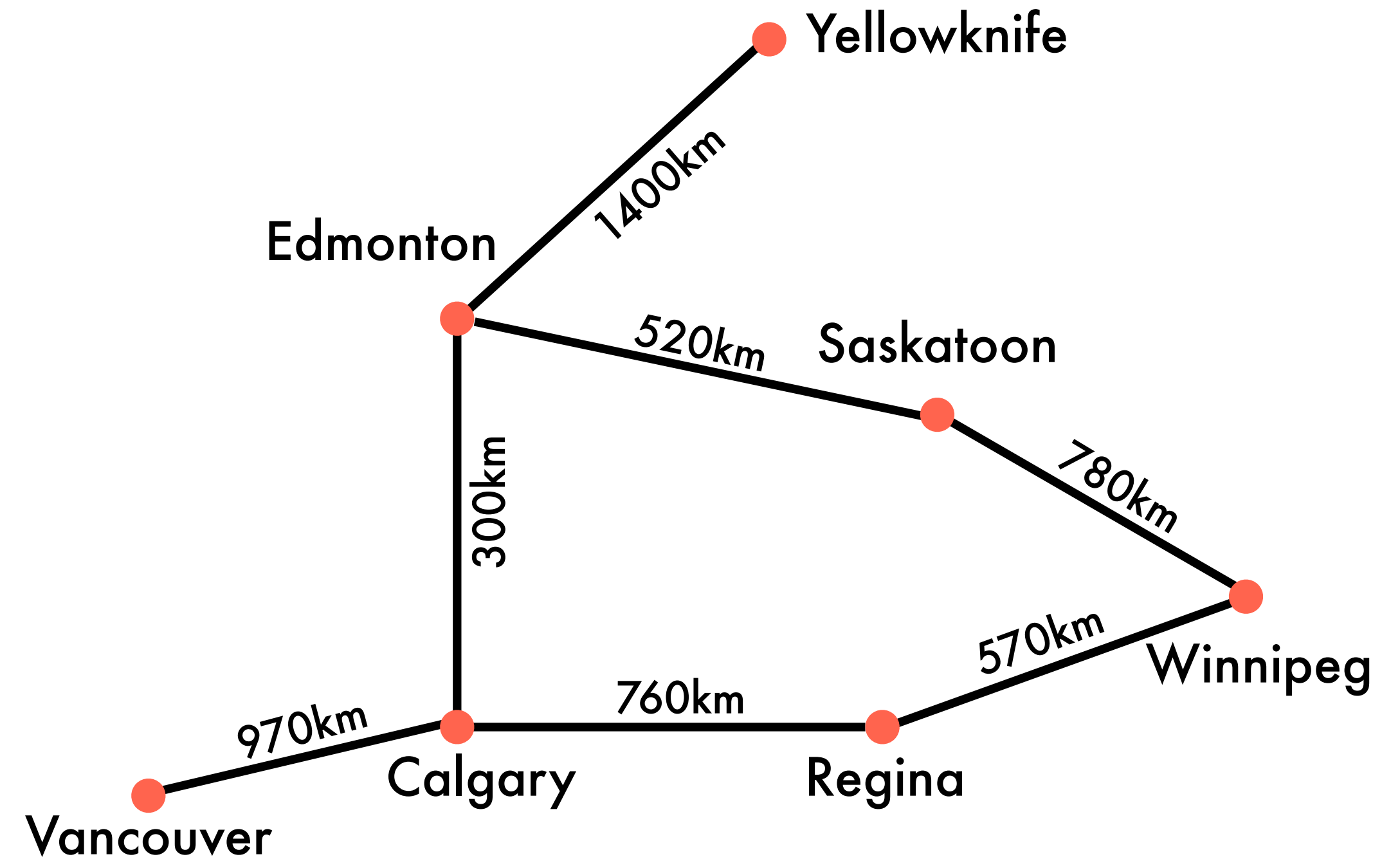
$$g(\text{Edmonton}) = 1060$$

$$g(\text{Saskatoon}) = 1350 \text{ (from Winnipeg)}$$

....and so on (try to complete the search).

**Important note:** the algorithm checks for the goal only when it **expands** a node.

This means that when you reach Yellowknife from Edmonton the first time, you have still to check you queue.



# Uniform-Cost Search (Dijkstra's Algorithm)

**function** Dijkstra(graph, start):

Initialize distances to all nodes as infinity

Distance from start node to itself is 0

priority\_queue = [(0, start)]

**while** priority\_queue is not empty:

current\_distance, current\_node = **pop**(priority\_queue) ← Pop node with smaller distance

**if** current\_distance > distances[current\_node]: ← If current distance is larger, ignore  
**continue**

**for** neighbour, weight **in** graph.current\_node: ← Check all neighbours of the current node

distance = current\_distance + weight

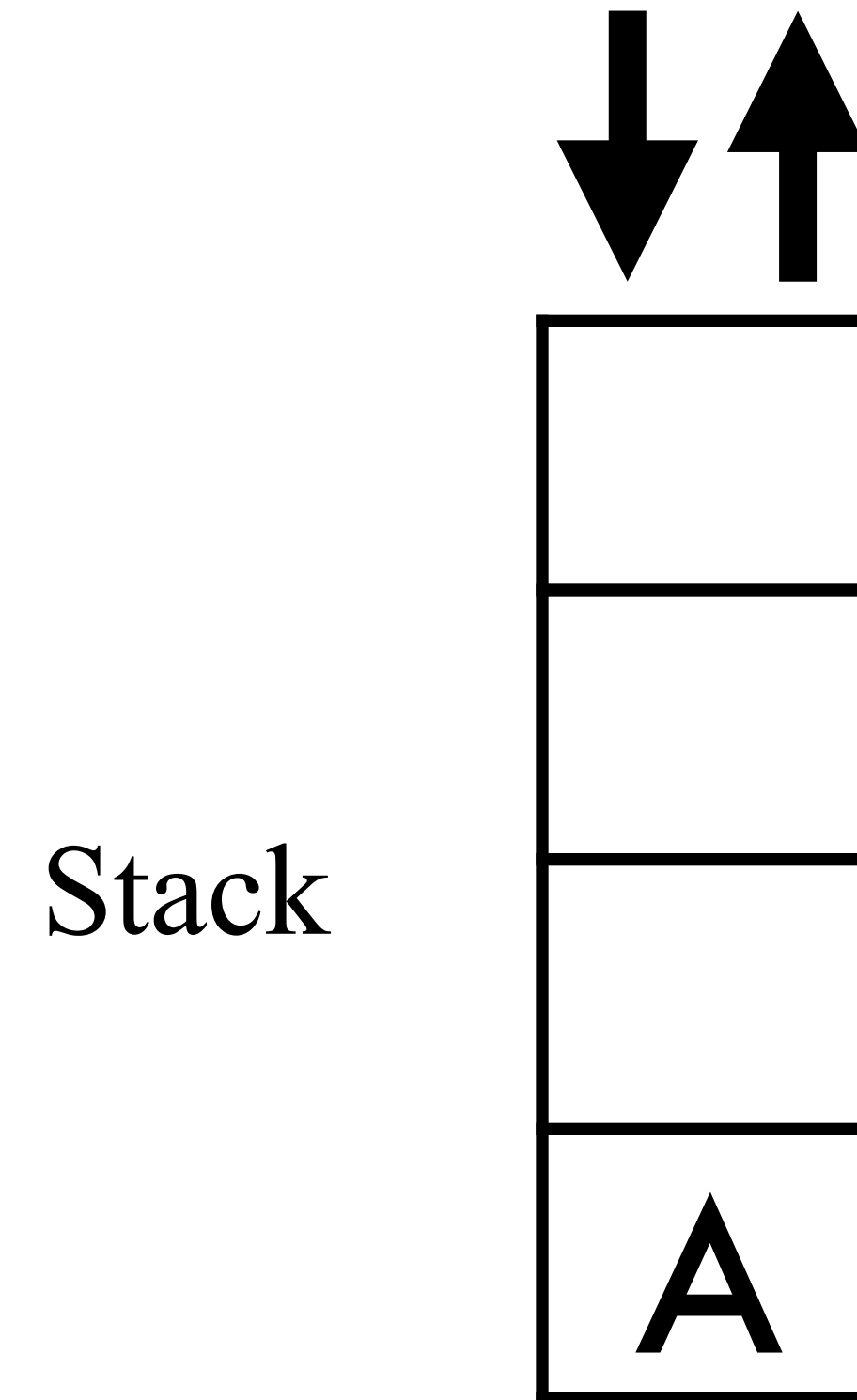
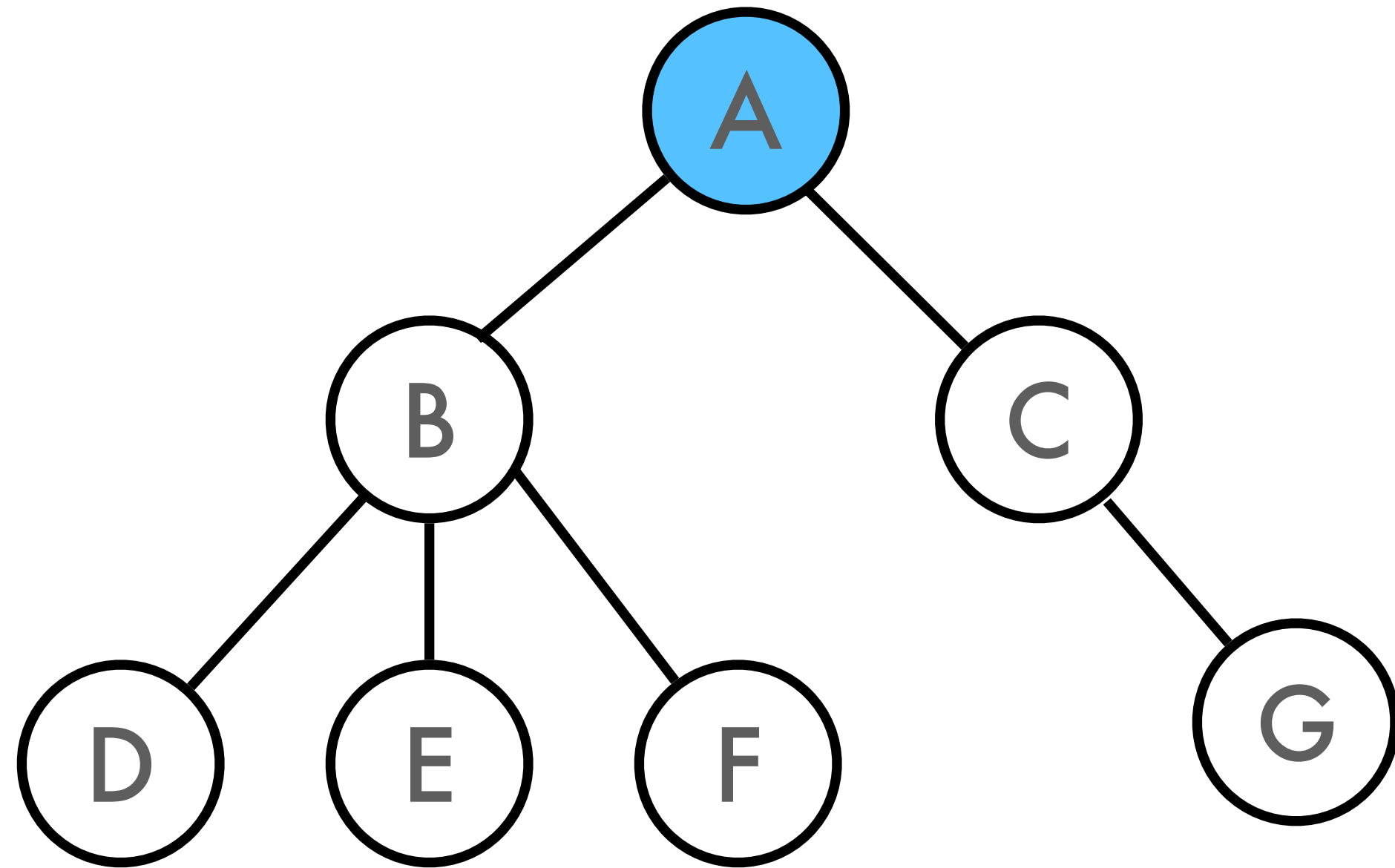
**if** distance < distances[neighbour]: ← If shorter path found, update distance

distances[neighbour] = distance

**push**(priority\_queue, (distance, neighbour))

**return** distances

# Depth-First Search

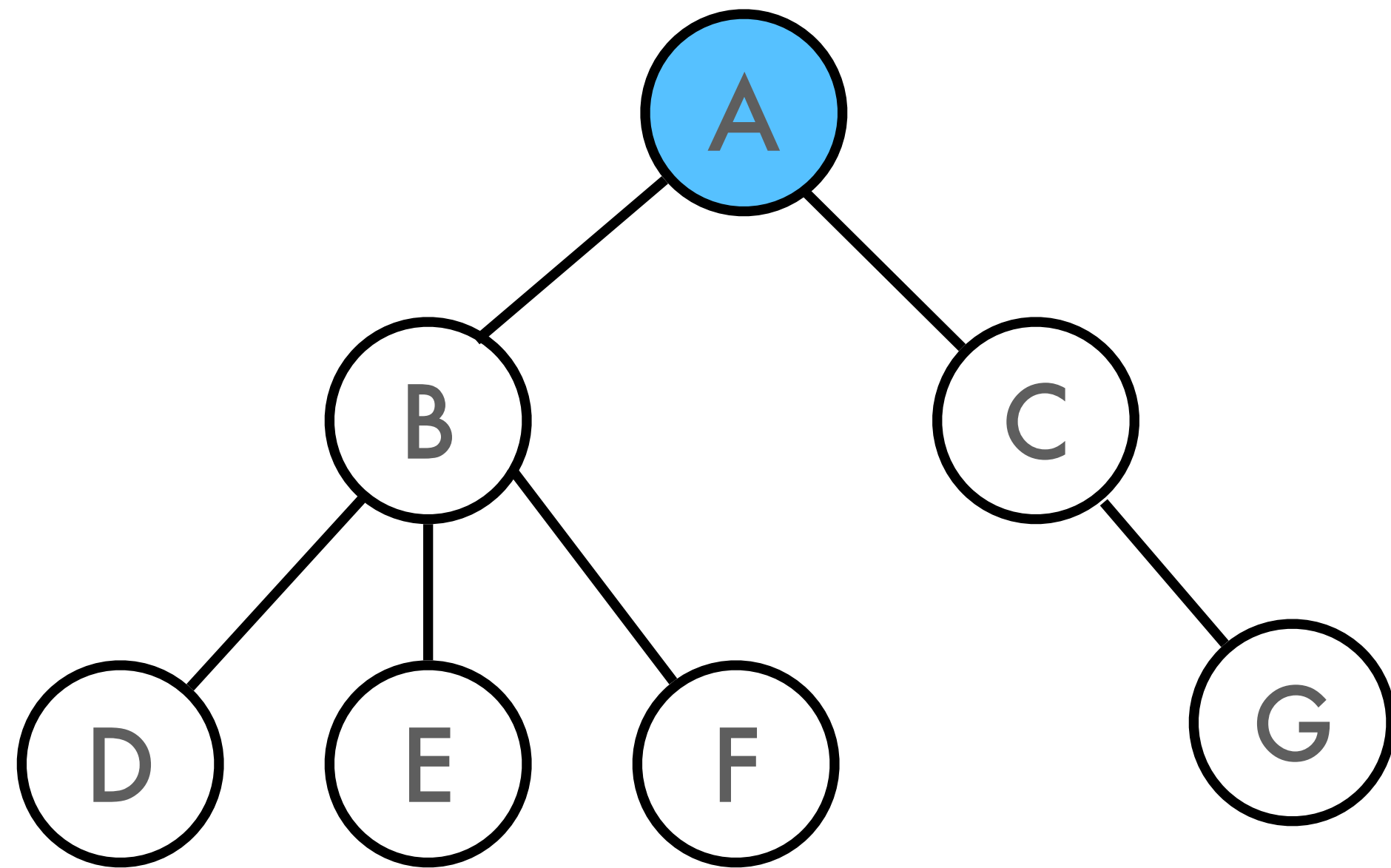


- A **stack**(\*) contains the list of the nodes to visit.
- The root node is initially introduced in a stack.
- The children nodes are introduced in the stack.
- The visited nodes are popped from the stack.

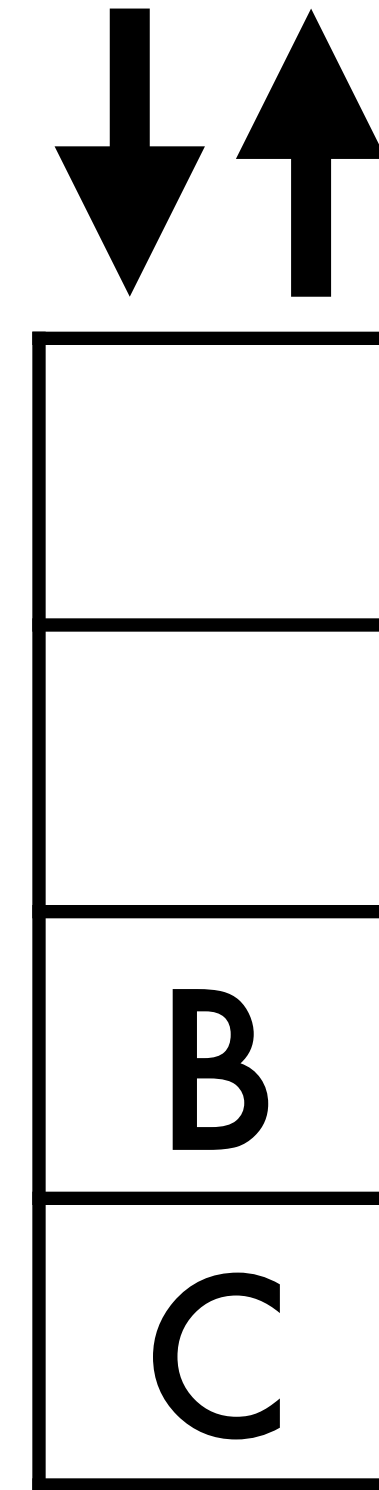
(\*) Note: given the use of a stack, it is common to realise this search with recursion.



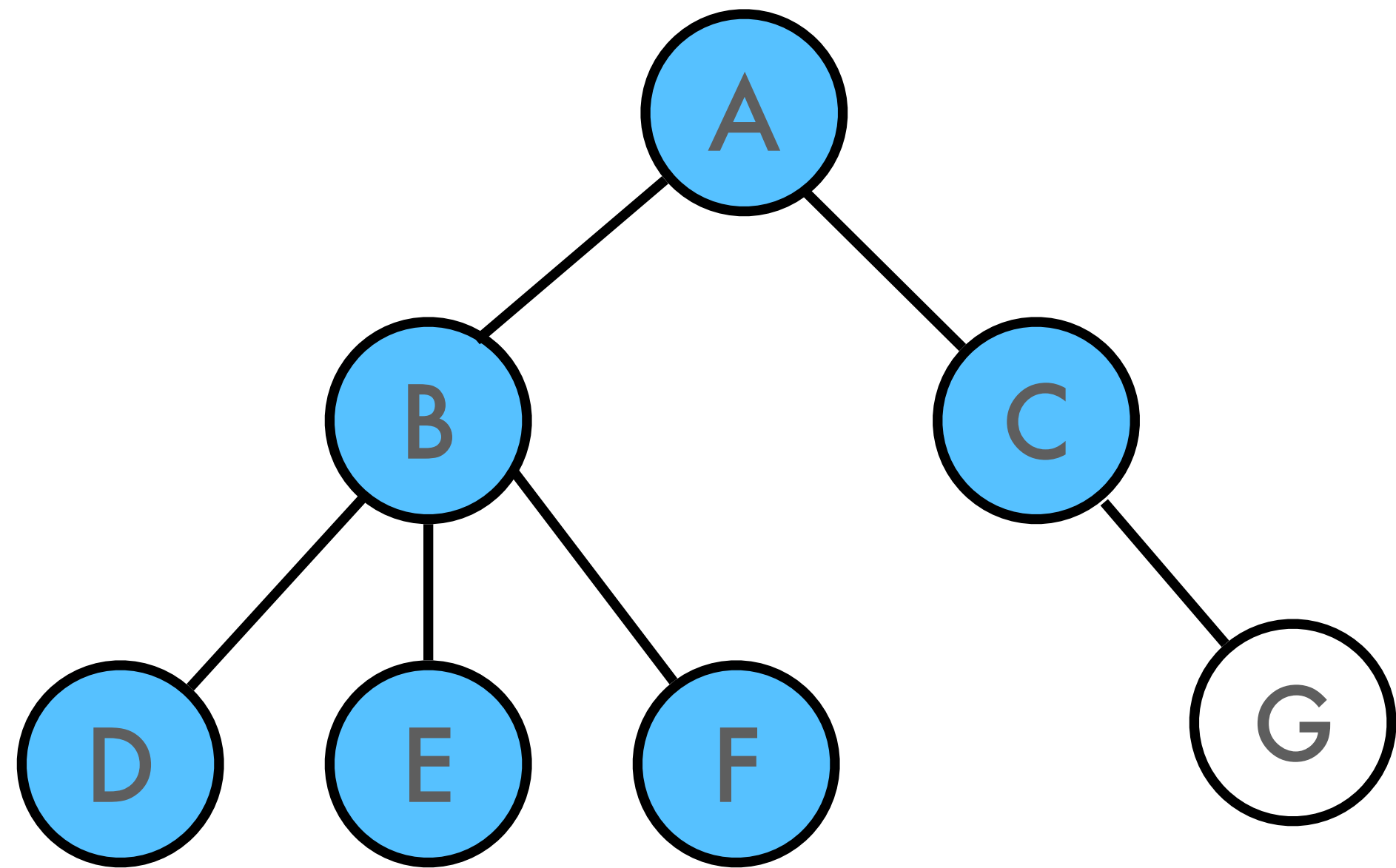
# Depth-First Search



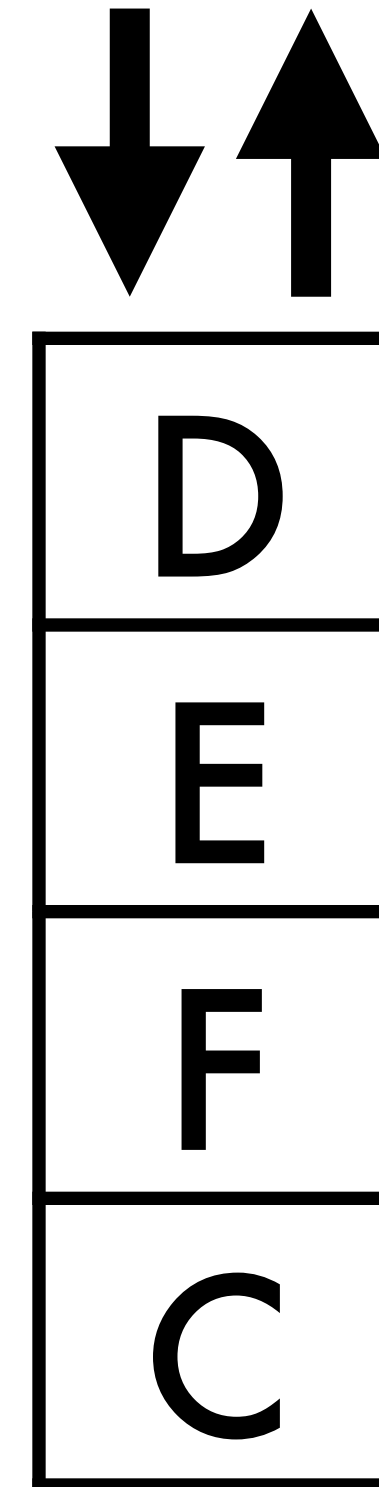
Stack



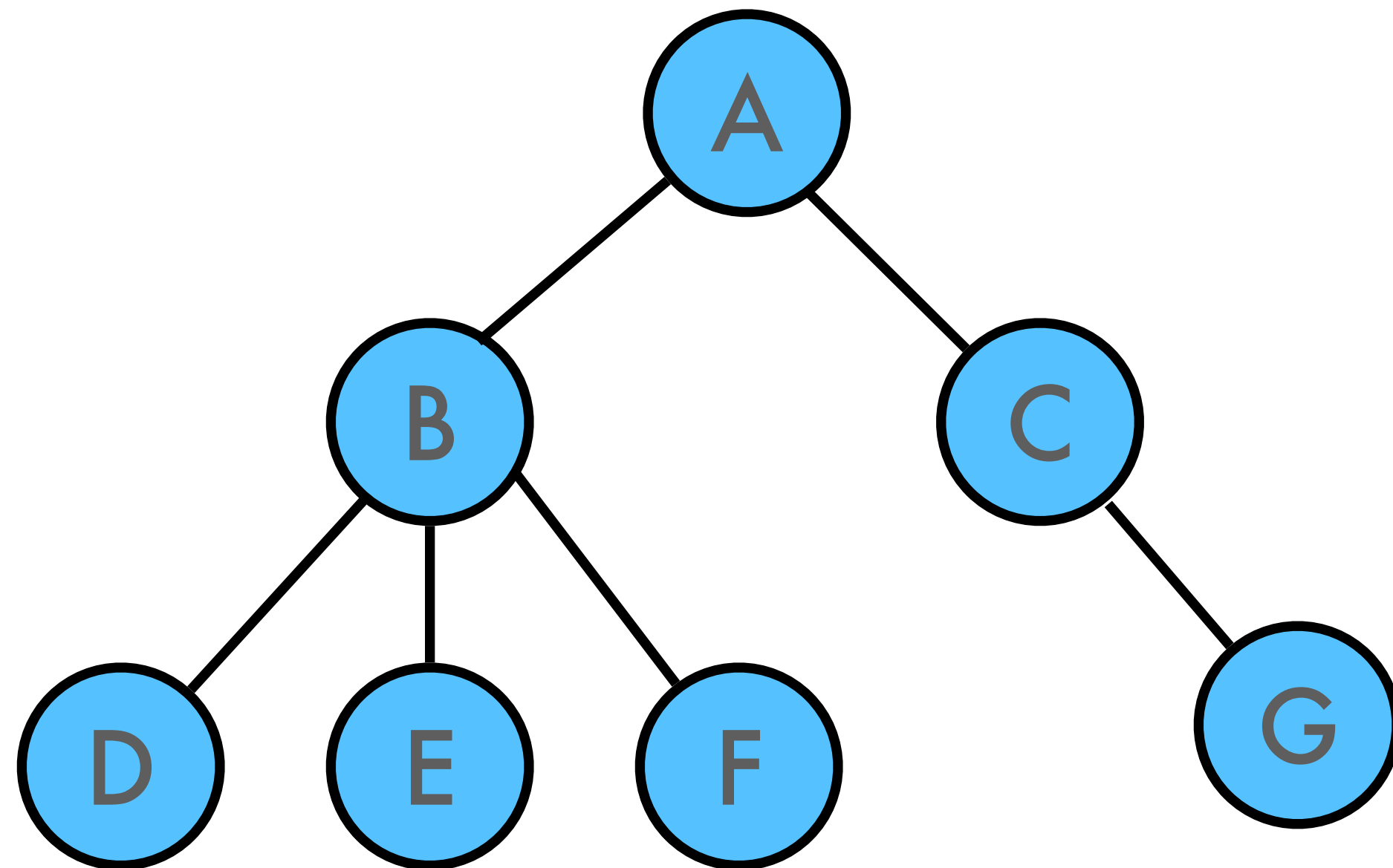
# Depth-First Search



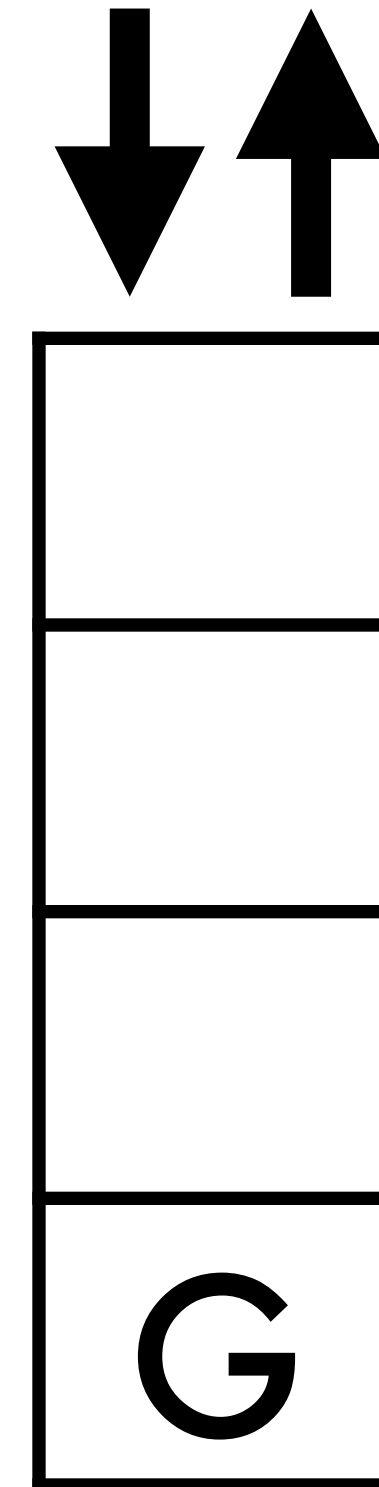
Stack



# Depth-First Search



Stack



# Depth-First Search

- The solution is not, in general, optimal.
- Completeness is guaranteed only for graph-based searches and a finite state space.
- **Time complexity:**
  - In a **graph-based** search, bounded by the size of the state space (can be infinite).
  - In **tree-based search**, the algorithm might generate  $O(b^m)$  nodes in the search tree which can be larger than the state space size ( $m$  is the max. length of a path in the state space).
- **Space complexity:**
  - **Tree-based search:** needs to store only the nodes along the path from the root to the leaf node. Once a node has been expanded, it can be removed from the memory as soon as all its descendants have been fully explored. Therefore, the memory requirement is only  $O(bm)$ . This is the reason, why it is practically so relevant despite all the other shortcomings.
  - **Graph-based search:** in the worst case, all states need to be stored in the explored set (no advantage over breadth-first).



# Depth-Limited Search

Cut-off on the maximum reachable depth.

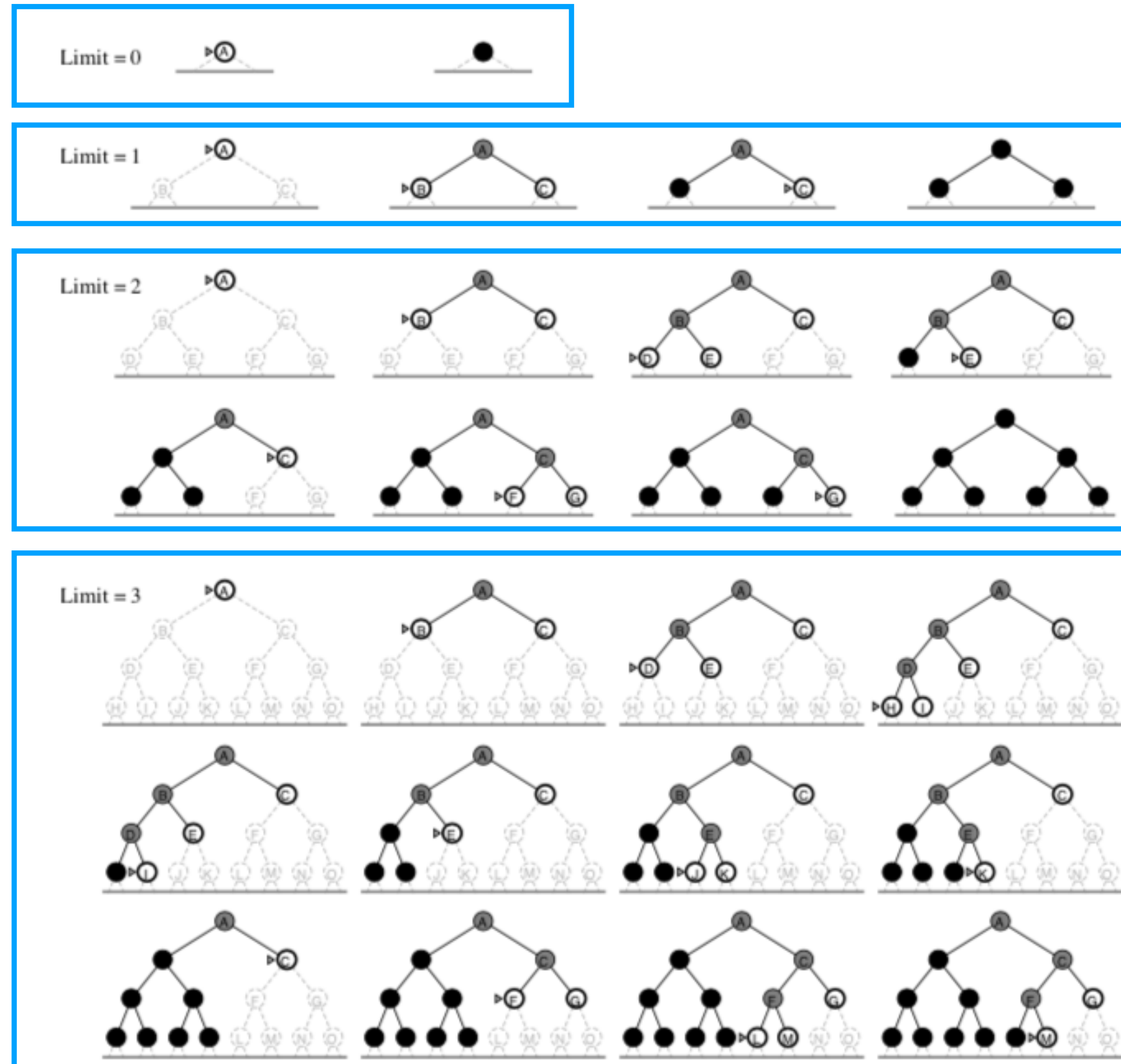
How to choose the cut-off:

- Choose maximum depth  $L = \#nodes - 1$
- Choose the diameter  $d$  of the graph: maximum number of steps for reaching any node from any other one (in the “Canada graph”,  $d=9$ ).

# Iterative-deepening Search

- Increase  $L$  by 1 starting from  $L=0$  until a solution is found or  $L_{\max}$  is reached.
- Combines depth- and breadth-first searches benefits.
- Optimal and complete like BFS but less memory requirement  $O(bd)$ .
- Time complexity slightly worse than BFS.

# Iterative-deepening Search



# Iterative-deepening Search

Like BFS, IDS is optimal for problems where with same-cost actions and is complete for acyclical state spaces or finite state spaces with checks for cycles.

IDS seems expensive, but repeating the expansion is cheap initially and grows with time. So the first node is repeated  $d$  times, the second nodes  $(d-1)$  times and so on:

Number of node expansions:

**Iterative Deepening Search:**  $(d)b + (d-1)b^2 + (d-2)b^3 + \dots + b^d \sim O(b^d)$

**Breadth-First Search:**  $b + b^2 + \dots + b^{d-1} + b^d \sim O(b^d)$

Example ( $b=10$ ,  $d=5$ ):

$$N(\text{BFS}) = 10 + 100 + 1000 + 10000 + 100000 = 111110$$

$$N(\text{IDS}) = 50 + 400 + 3000 + 20000 + 100000 = 123450 \sim 11\% \text{ higher}$$

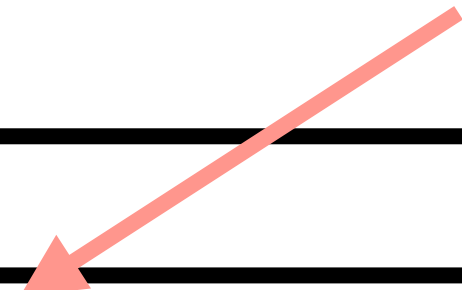


# Iterative-deepening Search

- IDS is the preferred uninformed search strategy when the entire search space cannot fit into the memory.
- If memory is enough, BFS can be used.
- Another option: hybrid approach:
  - Use BFS until memory allows.
  - Switch to IDS from the last frontier found by BFS.

# IDS and DLS Algorithms

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution or failure
  for depth=0 to infinity do
    result ←- DEPTH-LIMITED-SEARCH(problem, depth)
    if result != cutoff then return result
```



```
function DEPTH-LIMITED-SEARCH(problem, L) returns a node or failure or cutoff
  frontier ←- a stack with NODE(problem, INITIAL) as element
  result ←- failure
  while not IS-EMPTY(frontier) do
    node ←- POP(frontier)
    if problem.IS-GOAL(node, state) then return node
    if DEPTH(node) > L then
      result ←- cutoff
    else if not IS-CYCLE(node) do
      for each child in EXPAND(problem, node) do
        add child to frontier
  return result
```

# Bidirectional Search

- Simultaneously searches forward from the starting node and backwards from the goal node.
- Assume that the two paths will meet.
- Motivation:  $b^{d/2} + b^{d/2} \ll b^d$

**Example:**  $b=10, d=10$ . Gain factor:  $\frac{10^{10}}{2 \cdot 10^5} = 50.000$

The algorithm needs to keep track of two frontiers and two lists of reached nodes.  
The two paths can be followed by different search strategies.

# Comparison of the Search Strategies

For tree structures without repetition check

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes <sup>a</sup>	Yes <sup>a,b</sup>	No	No	Yes <sup>a</sup>	Yes <sup>a,d</sup>
Time	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^l)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(bl)$	$O(bd)$	$O(b^{d/2})$
Optimal?	Yes <sup>c</sup>	Yes	No	No	Yes <sup>c</sup>	Yes <sup>c,d</sup>

$b$  branching factor  
 $d$  depth of solution  
 $m$  maximum depth of the search tree  
 $l$  depth limit  
 $C^*$  cost of the optimal solution  
 $\epsilon$  minimal cost of an action

Superscripts:

<sup>a</sup>  $b$  is finite

<sup>b</sup> if step costs not less than  $\epsilon$

<sup>c</sup> if step costs are all identical

<sup>d</sup> if both directions use breadth-first search

# Summary

- Before an agent can start the search for a solution, a goal must be defined. The goal must be used for formulating the problem.
- A problem consists in 5 parts: The state space, initial condition, actions, goal test and, path costs.
- A search algorithm can in principle be used for solving any problem.
- Search algorithms are described on the basis of completeness, optimality, time complexity, and space complexity.