COMPSCI 761: ADVANCED TOPICS IN ARTIFICIAL INTELLIGENCE

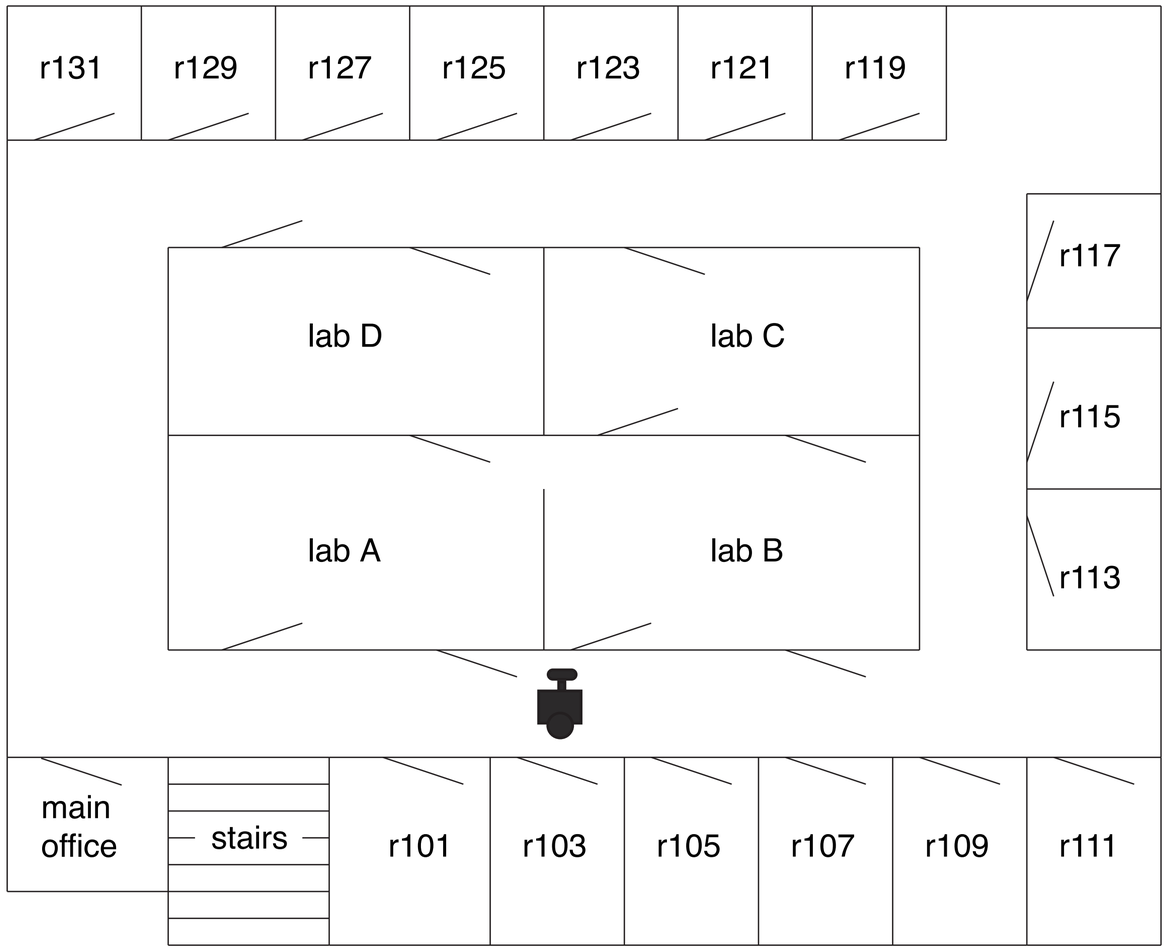
UNINFORMED SEARCH

Anna Trofimova, July 2022

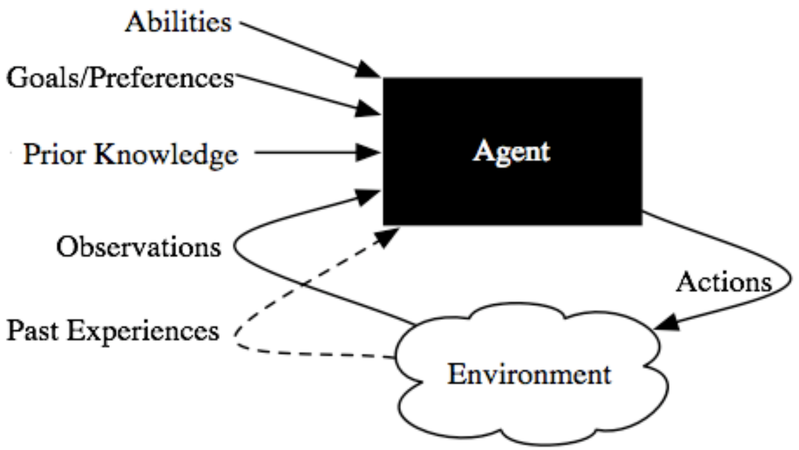
TODAY

- Problem solving as search
- Uninformed Search Algorithms

EXAMPLE: DELIVERY ROBOT



DELIVERY ROBOT MODEL



Abilities: movement, speech, pickup and place objects.

Prior knowledge: its capabilities, objects it may encounter, maps.

Past experience: which actions are useful and when, what objects are there, how its actions affect its position.

Goals: what it needs to deliver and when, tradeoffs between acting quickly and acting safely.

Observations: about its environment from cameras, sonar, sound, laser range finders, or keyboards.

DELIVERY ROBOT: TASKS

- Determine where Craig's office is. Where coffee is. . .
- Find a path between locations.
- Plan how to carry out multiple tasks.
- Make default assumptions about where Craig is.
- Make tradeoffs under uncertainty: should it go near the stairs?
- Learn from experience.
- Sense the world, avoid obstacles, pickup and put down coffee.

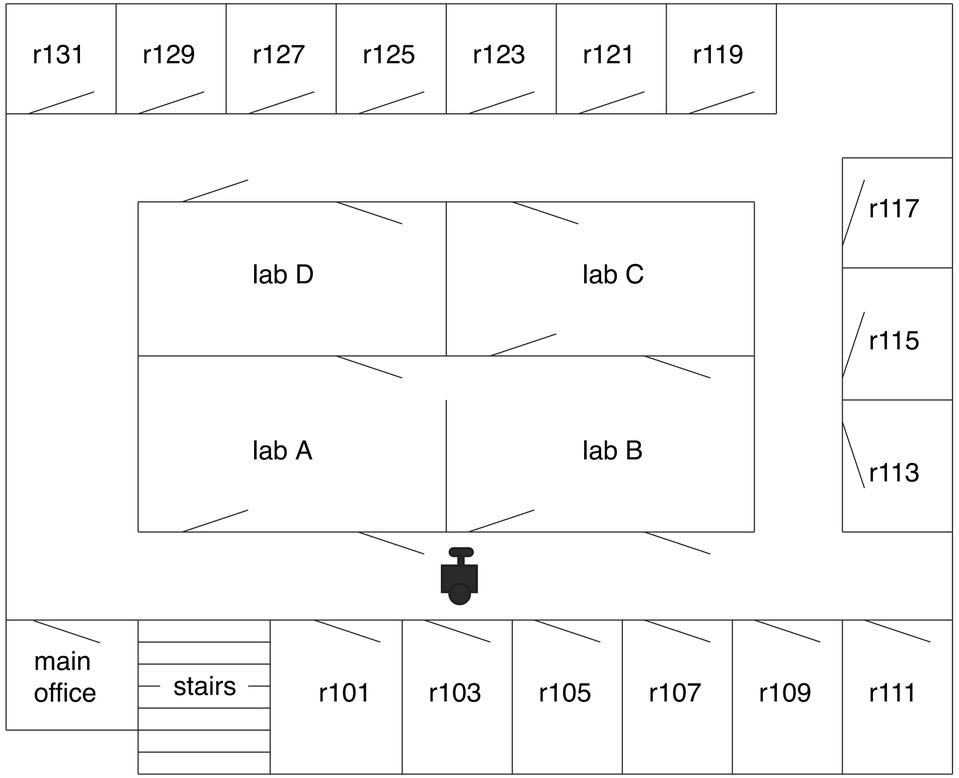
SEARCH

- 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.

MOTIVATION

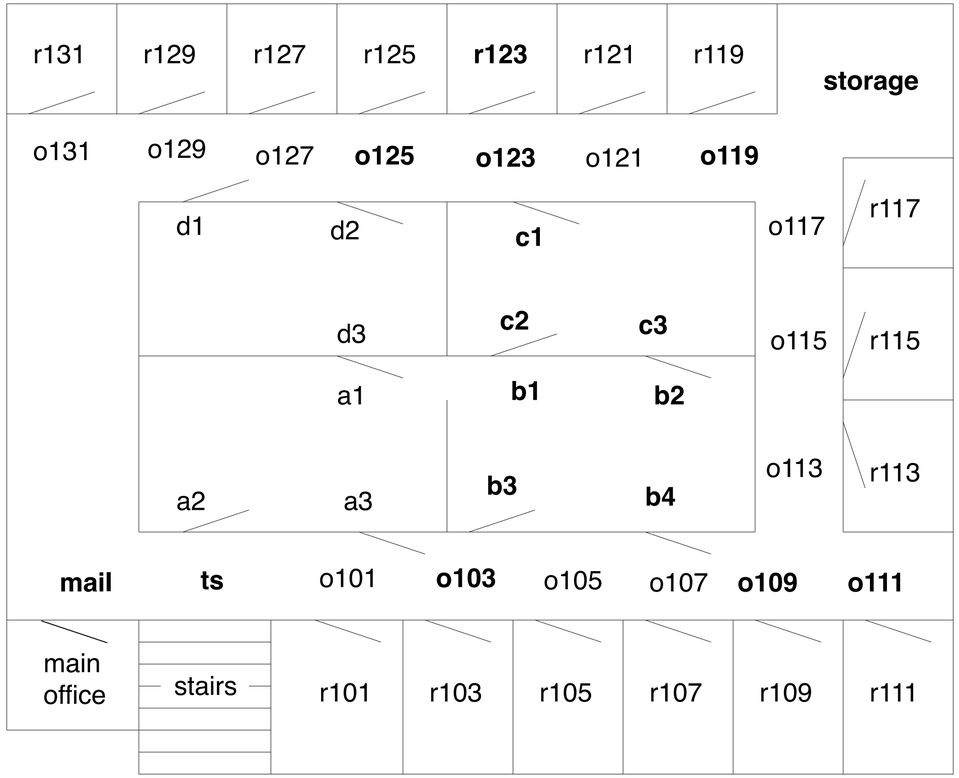
- Reactive and Model-Based Agents choose their actions based only on what they currently perceive, or have perceived in the past.
- A Planning Agent can use Search techniques to plan several steps ahead in order to achieve its goal(s).
- Two classes of search strategies:
  - Uninformed search strategies can only distinguish goal states from non-goal states
  - Informed search strategies use heuristics to try to get “closer” to the goal

EXAMPLE: DELIVERY ROBOT



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

EXAMPLE: DELIVERY ROBOT



The delivery robot domain with interesting locations labelled.

STATE SPACE

- One general formulation of intelligent action is in terms of a **state space**.
- A **state** contains all of the information necessary to predict the effects of an action and to determine whether a state satisfies the goal.
- State-space searching assumes:
  - The agent has perfect knowledge of the state space and is planning for the case where it observes what state it is in: there is full observability.
  - The agent has a set of actions that have known deterministic effects.
  - The agent can determine whether a state satisfies the goal.
- A **solution** is a sequence of actions that will get the agent from its current state to a state that satisfies the goal.

STATE SPACE PROBLEM

A state-space problem consists of:

- a set of states
- a subset of states called the **start states**
- a set of actions
- an **action function**: given a state and an action, returns a new state
- a (set of) goal state, specified as Boolean function, goal(s)
- a criterion that specifies the quality of an acceptable solution.

EXAMPLE: DELIVERY ROBOT

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

Step 1 Formulate goal:

- get from outside room 103 to the inside of room 123

Step 2 Formulate problem - Specify task

- states: various rooms
- actions (operators) (= transitions between states): moving between rooms

EXAMPLE: DELIVERY ROBOT

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

Step 1 Formulate goal:

- get from outside room 103 to the inside of room 123

Step 2 Formulate problem - Specify task

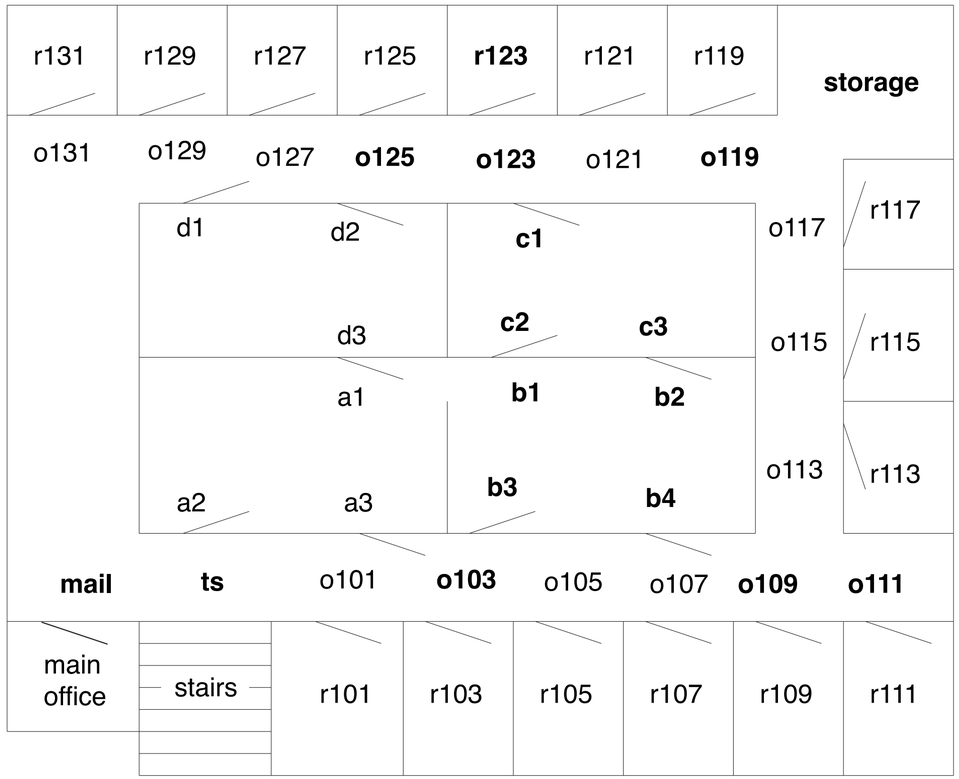
- states: position
- actions (operators) (= transitions between states): moving between rooms

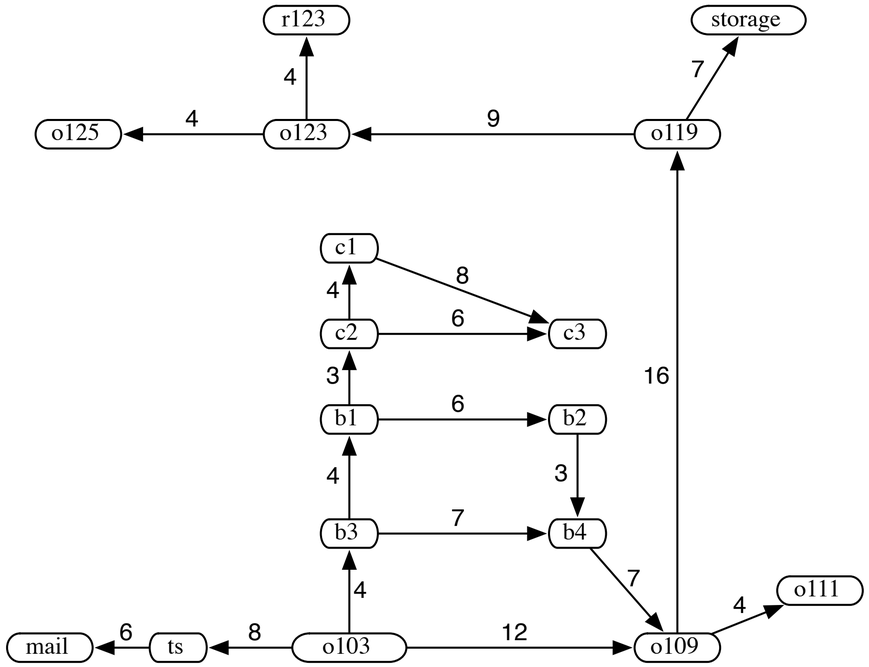
Step 3 Find solution (= action sequences): sequence of rooms

- A solution is a sequence of actions that moves the robot from o103 to room r123

Step 4 Execute: move through all locations given by the solution.

DELIVERY ROBOT: SPACE-SATE GRAPH





DIRECTED GRAPH

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

The arc <*n1, n2*> is an **outgoing arc** from *n1* and an **incoming arc** to *n2*
A node *n2* is a neighbour of *n1* if there is an arc from *n1* to *n2*.

A **path** from node *s* to node g is a sequence of nodes $\langle n_0, n_1, \ldots, n_k \rangle$ such that $s = n_0$, $g = n_k$, and $\langle n_{i-1}, n_i \rangle \in A$. The length of path $\langle n_0, n_1, \ldots, n_k \rangle$ is k.

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

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.

REAL WORLD PROBLEMS

- Route finding — robot navigation, airline travel planning, computer/phone networks
- Travelling salesman problem — planning movement of automatic circuit board drills
- LSI layout — design silicon chips
- Assembly sequencing — scheduling assembly of complex objects,
- Mixed/constrained problems — courier delivery, product distribution, fault service and repair
- Manufacturing process control

- These are optimisation problems but mathematical (operations research) techniques are not always effective.

SOLVING PROBLEMS BY SEARCHING

- Search as a “weak method” of problem solving with wide applicability
- Uninformed search methods (use no problem-specific information)
- Informed search methods (use heuristics to improve efficiency)

SOLVING PROBLEMS BY SEARCHING

- Search as a “weak method” of problem solving with wide applicability

- Uninformed search methods (use no problem-specific information)
  - Uninformed (or “blind”) search strategies use only the information available in the problem definition (can only distinguish a goal from a non-goal state)

- Informed search methods (use heuristics to improve efficiency)
  - Informed (or “heuristic”) search strategies use task-specific knowledge.

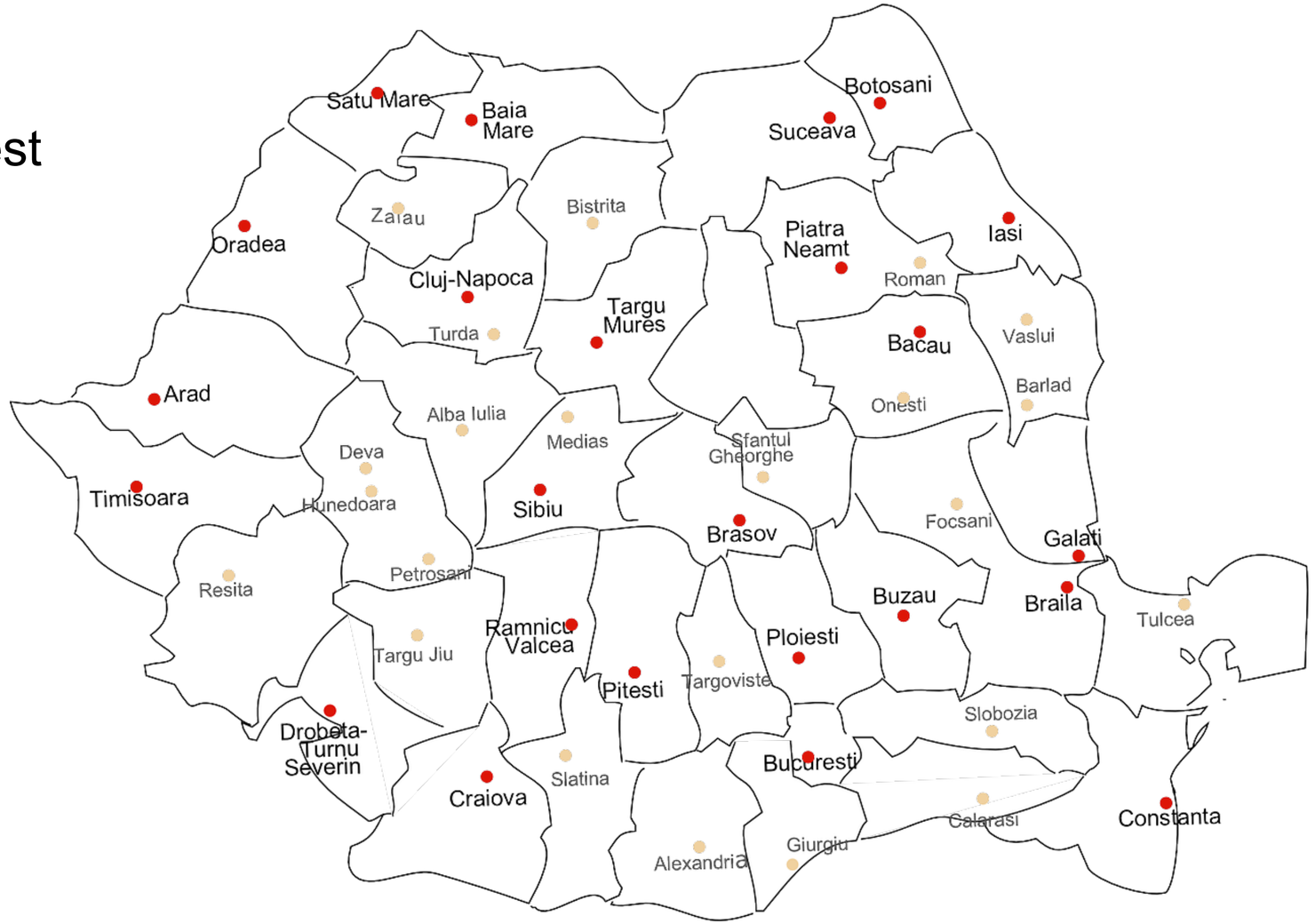
STATE SPACE SEARCH PROBLEMS

- State space — set of all states reachable from initial state(s) by any action sequence
- Initial state(s) — element(s) of the state space
- Transitions
  - Operators — set of possible actions at agent's disposal; describe state reached after performing action in current state, or
  - Successor function — s(x)= set of states reachable from state x by performing a single action
- Goal state(s) — element(s) of the state space
- Path cost — cost of a sequence of transitions used to evaluate solutions (apply to optimization problems)

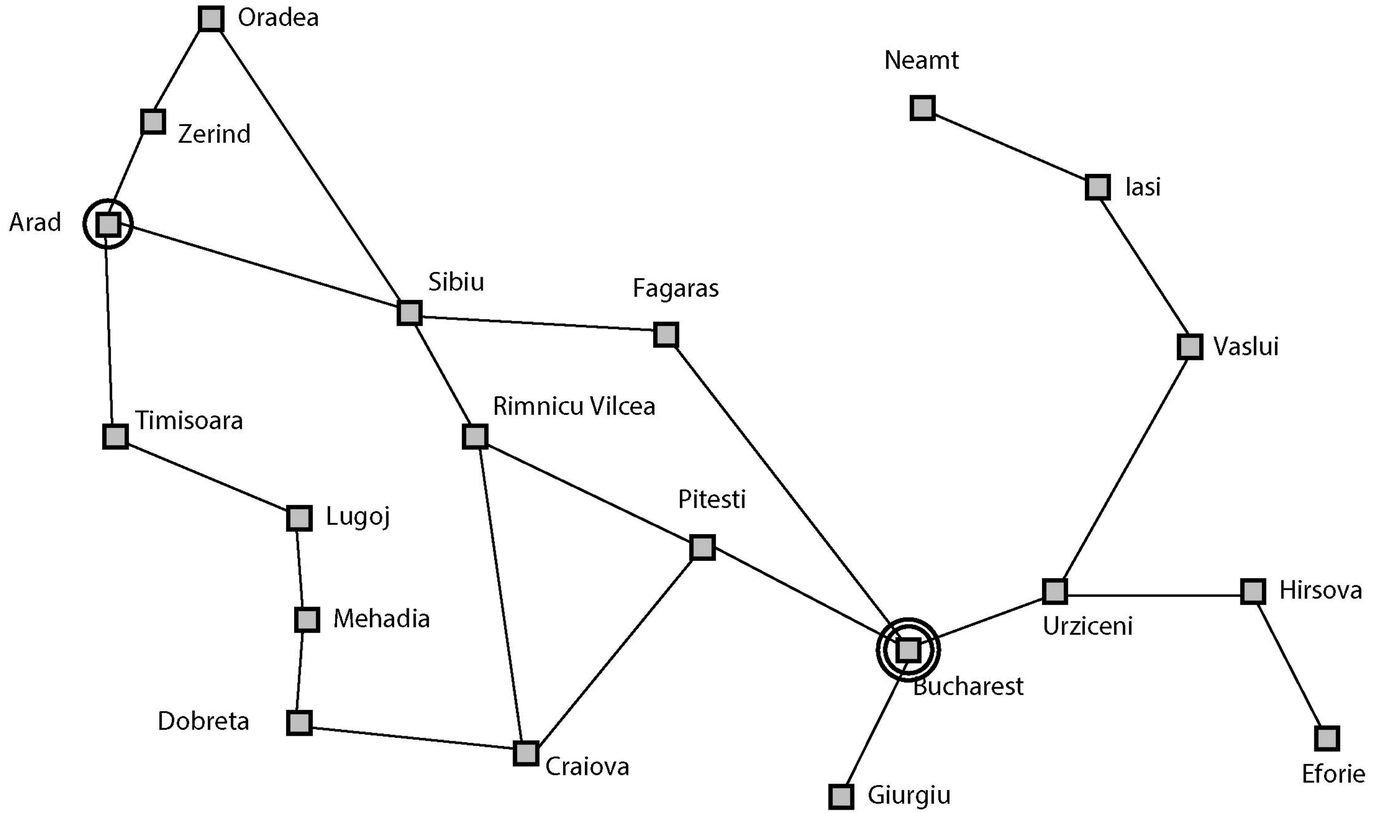
ROMANIA MAP

- Task: go from Arad to Bucharest

- What is a state?
- Start state?
- Goal state?
- What are the actions?
- What is the solution(s)?



STATE-SPACE GRAPH FOR ROMANIAN STREET PATH PROBLEM

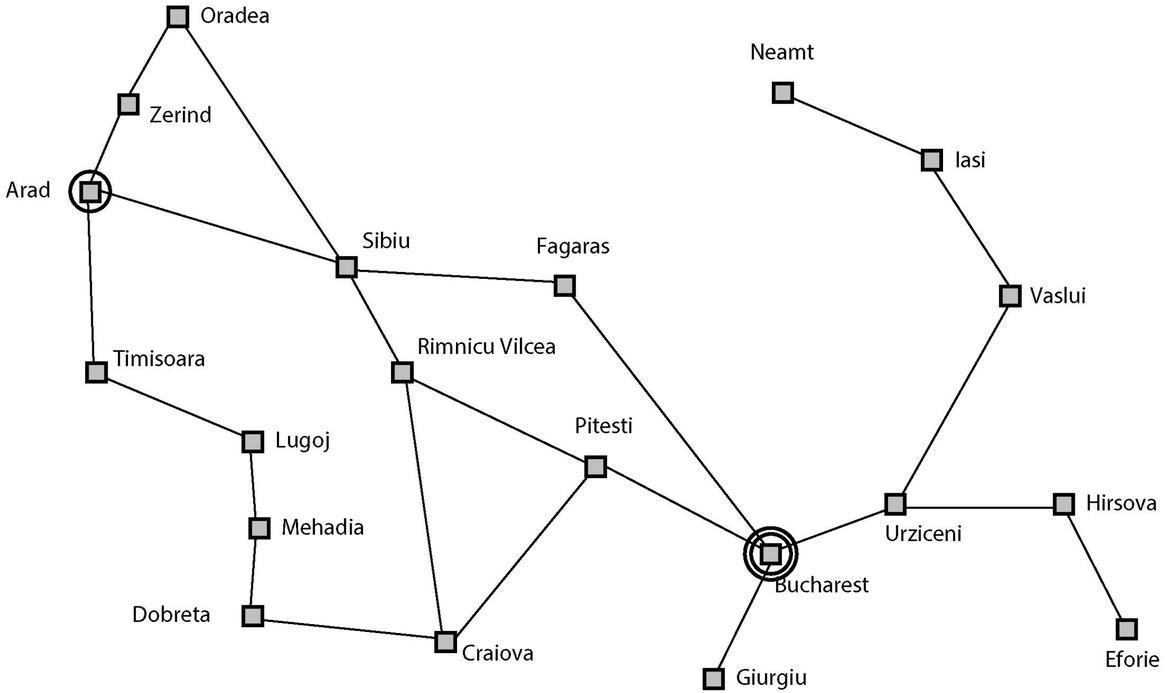


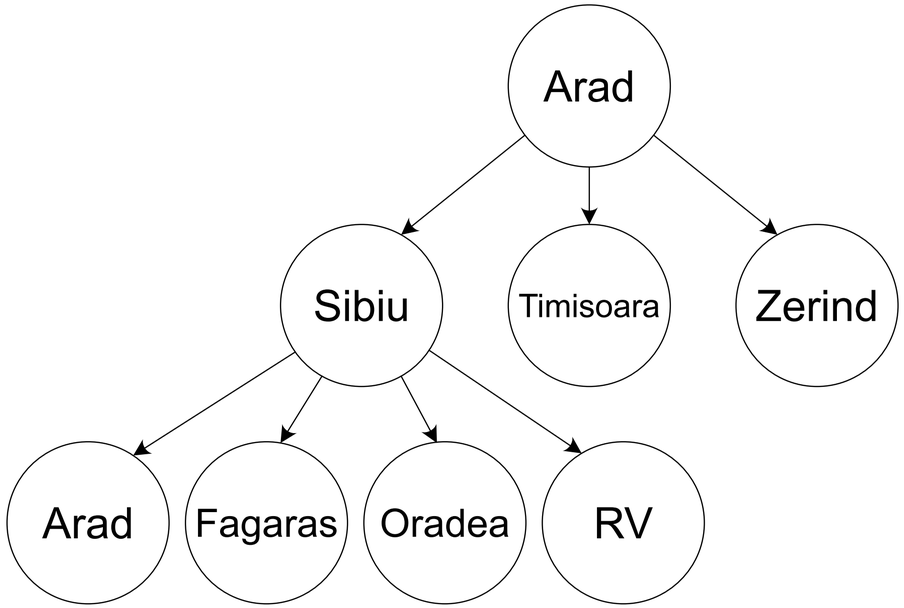
This can be modelled as a state-space search problem, where the states are locations.

SEARCH TREE

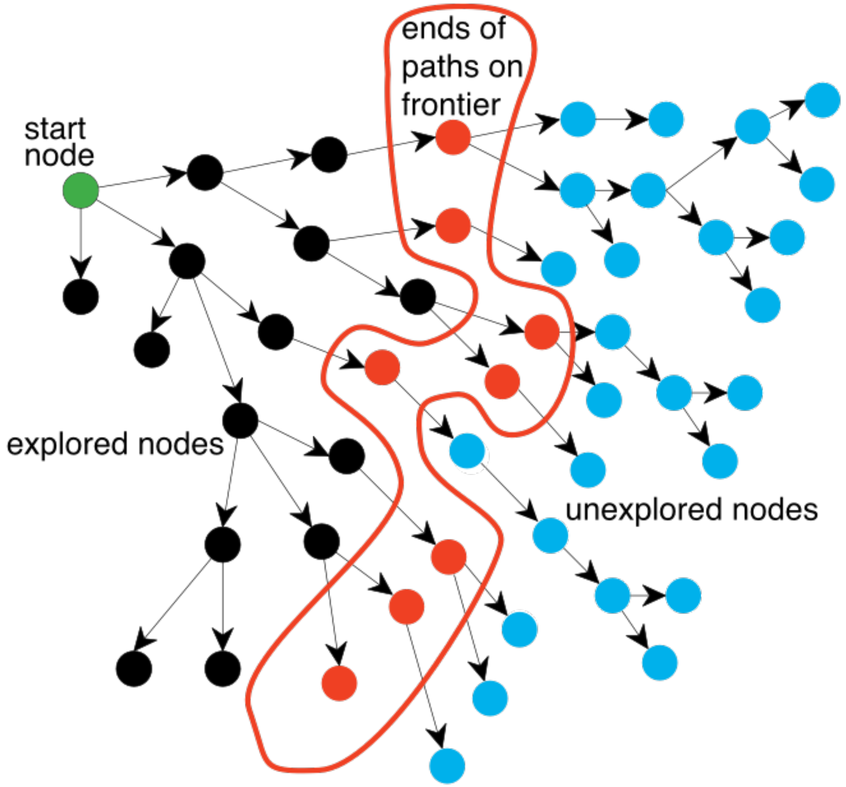
- Search tree: superimposed over the state space.
- Root: search node corresponding to the initial state.
- Leaf nodes: correspond to states that have no successors in the tree because they were not expanded or generated no new nodes.

ROMANIA MAP - GRAPH SEARCHING





PROBLEM SOLVING BY GRAPH SEARCHING

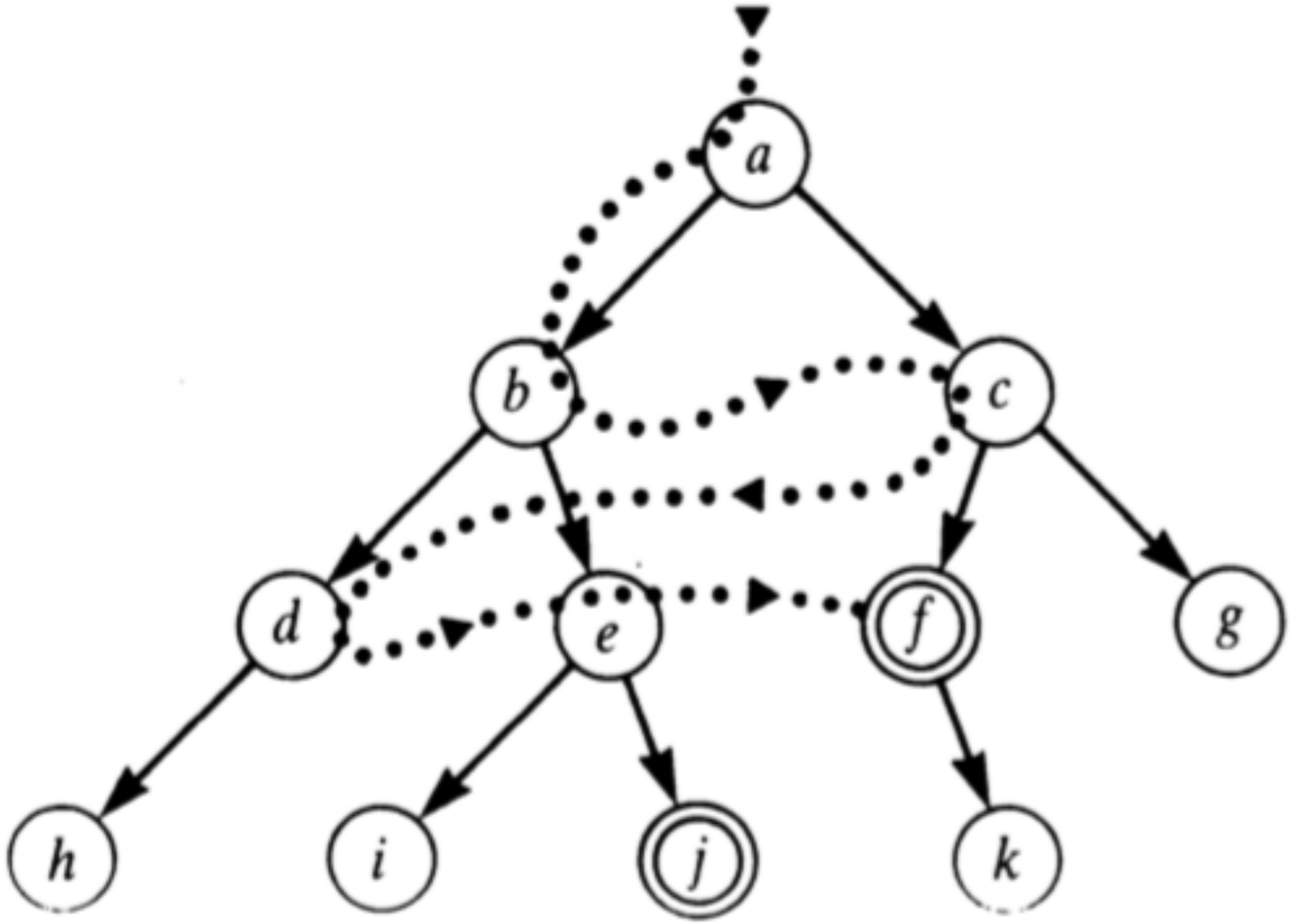


Search strategy — way in which frontier expands

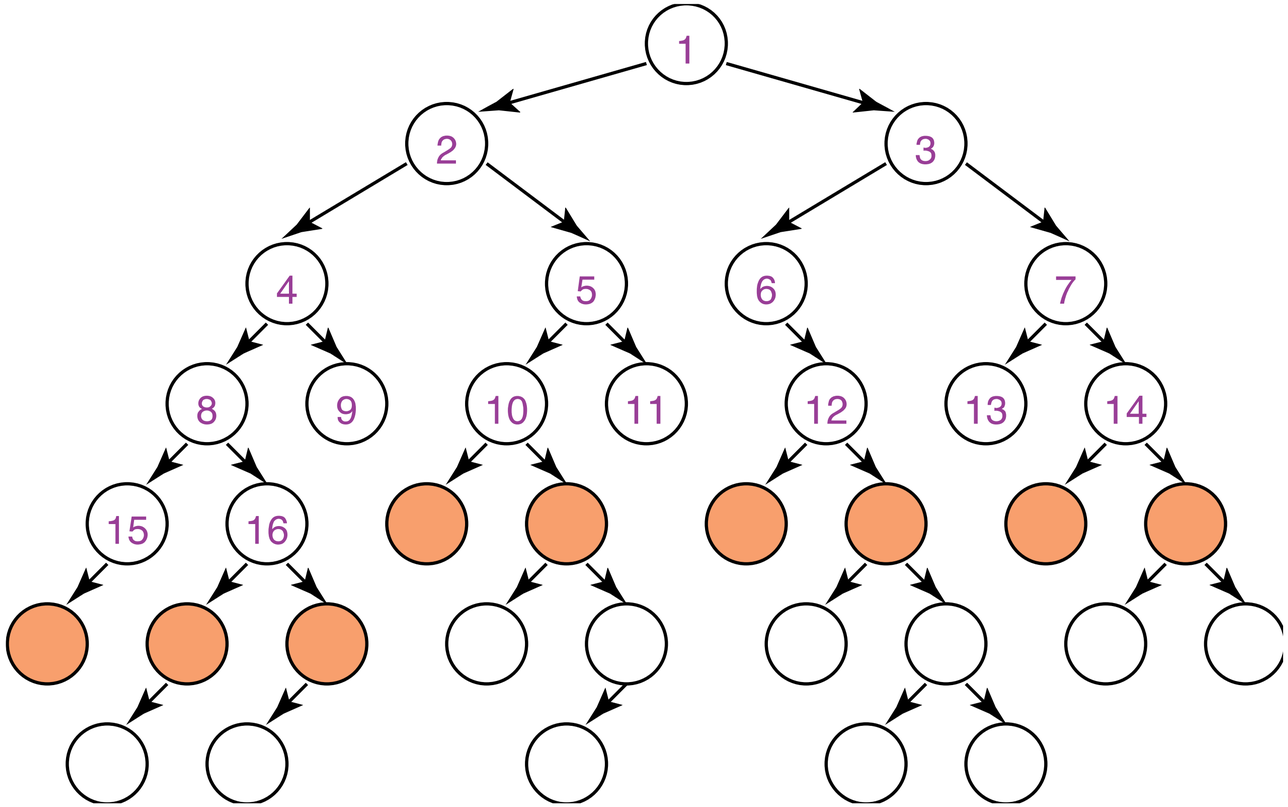
BREADTH-FIRST SEARCH

- Breadth-first search treats the frontier as a queue.
- It always selects the earliest element added to the frontier.
- If the list of paths on the frontier is [p1,p2,...,pr]:
  - p1 is selected. Its neighbours are added to the end of the queue, after pr .
  - p2 is selected next.

BREADTH-FIRST SEARCH



BREADTH-FIRST SEARCH

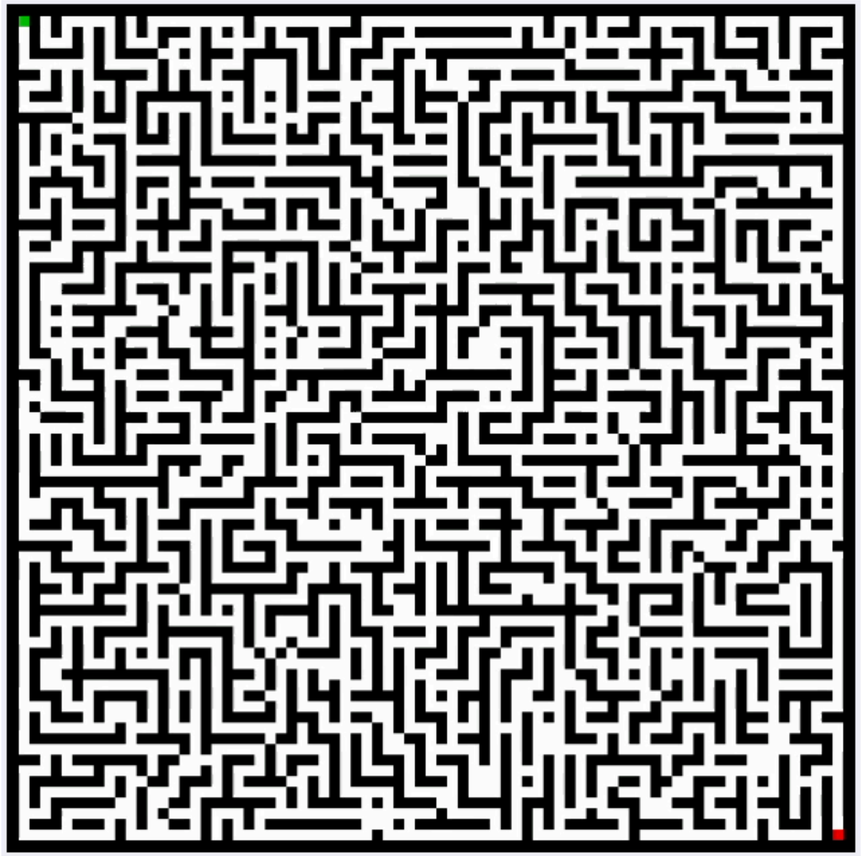


BREADTH-FIRST SEARCH

- All nodes are expanded at a given depth in the tree before any nodes at the next level are expanded
- Can be implemented by using a queue to store frontier nodes
  - put newly generated successors at end of queue
- Stop when node with goal state is generated
- Include check that generated state has not already been explored
  - Needs a new data structure for set of explored states
- Very systematic
- Finds the shallowest goal first

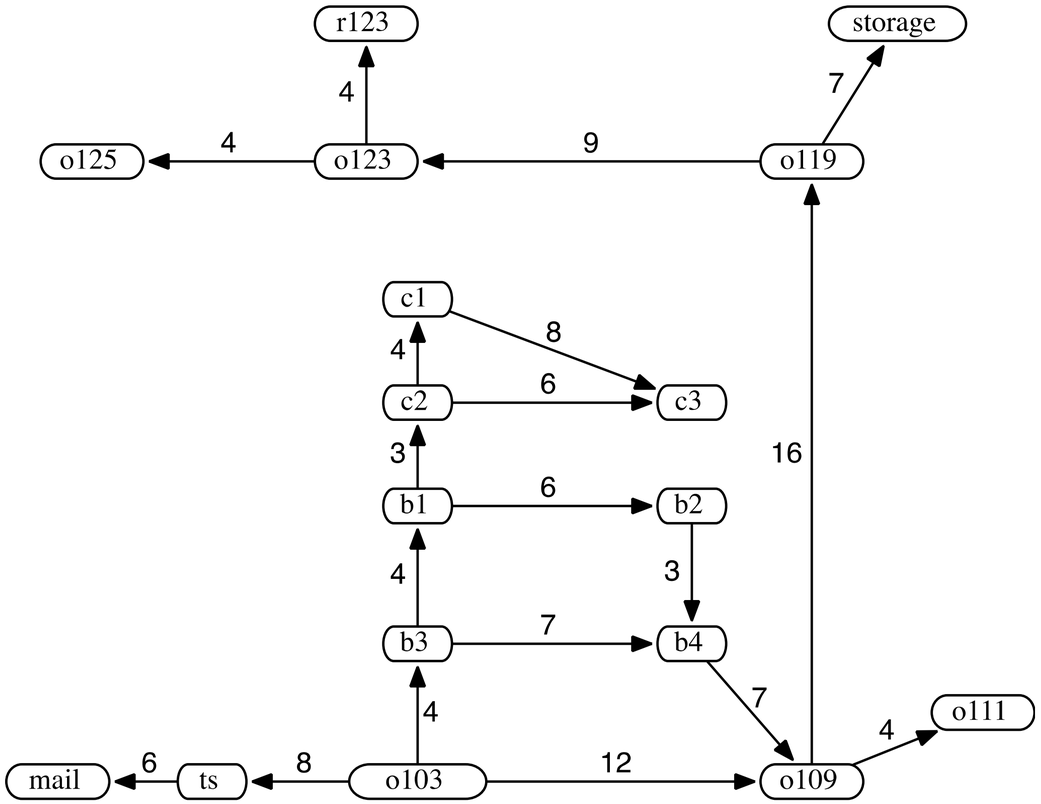
GUESS THE SEARCH PATH

Start

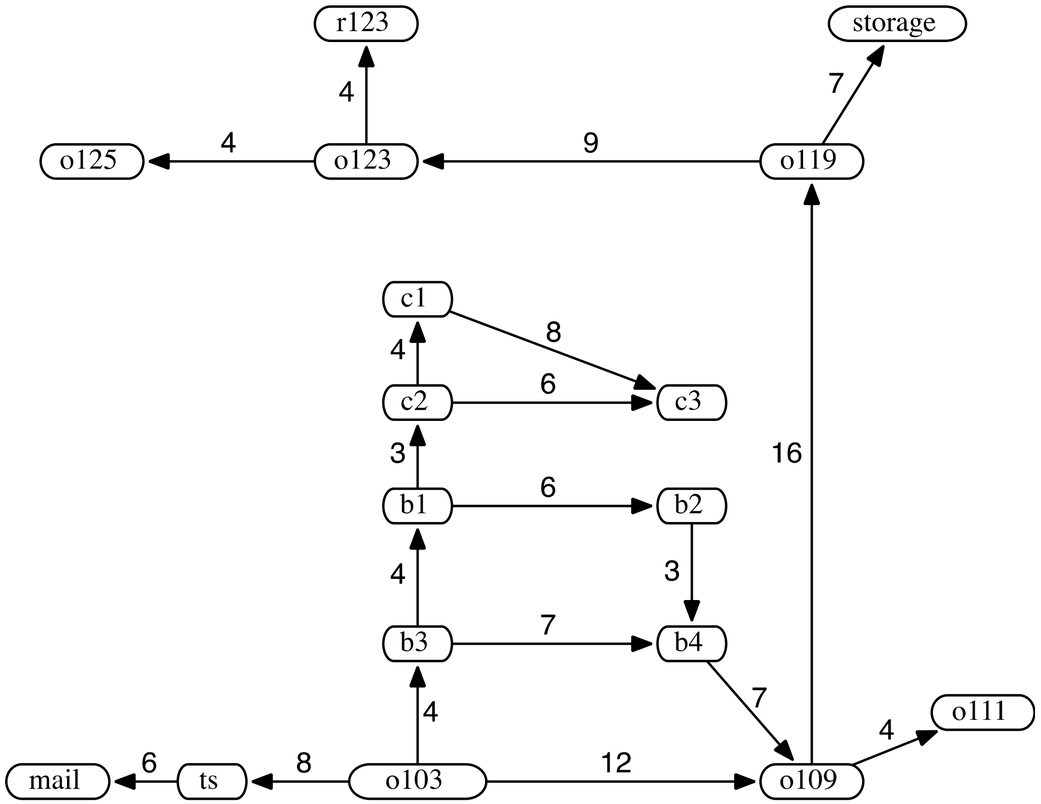


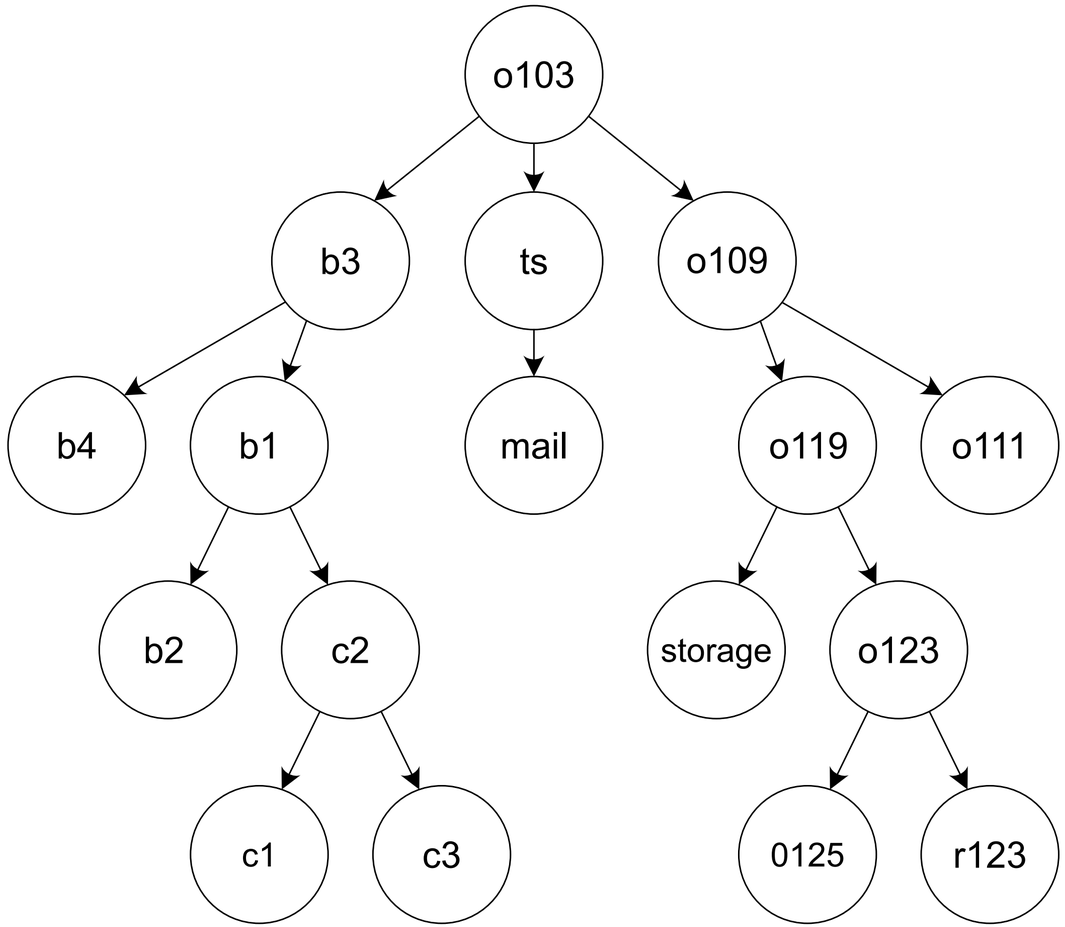
Goal

BREADTH-FIRST SEARCH



BREADTH-FIRST SEARCH





COMPLEXITY OF BREADTH-FIRST SEARCH

- Does breadth-first search guarantee to find the path with fewest arcs?
- What happens on infinite graphs or on graphs with cycles if there is a solution?
- What is the time complexity as a function of the length of the path selected?
- What is the space complexity as a function of the length of the path selected?
- How does the goal affect the search?

PROPERTIES OF BREADTH-FIRST SEARCH

Algorithm	Completeness	Admissibility	Space	Time
Breadth-First Search	guaranteed, if b is finite	guaranteed, if arcs have the same cost	$O(b^d)$	$O(b^d)$

- Complete? Yes, *it terminates with a solution when one exists*
- Time? $b^d+b^2+b^3+\ldots+ b^d = O(b^d)$
- Space? $O(b^d)$ *keeps every node in memory; generate all nodes up to level d*
- Optimal? Yes, but only if all actions have the same cost
- Space is the big problem for BFS; it grows exponentially with depth!

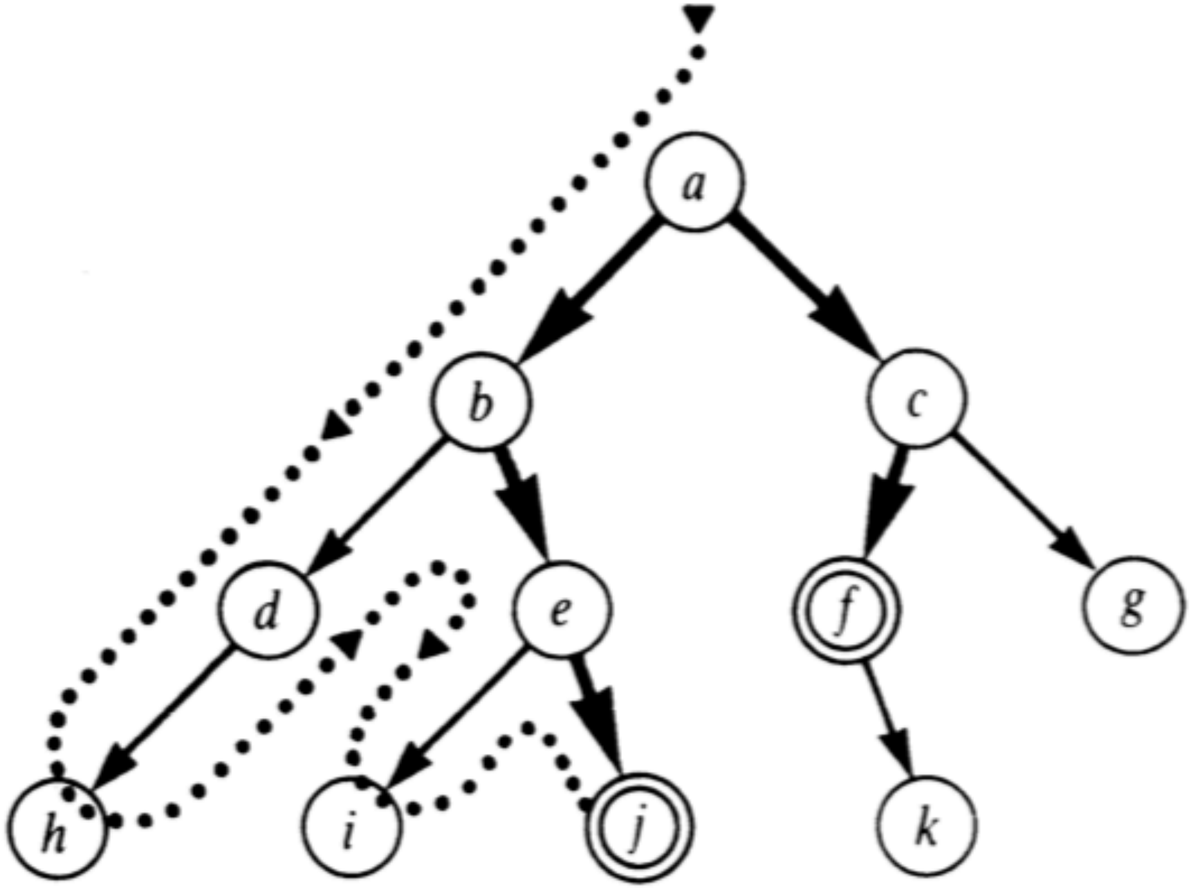
DEPTH FIRST SEARCH

- Expands one of the nodes at the deepest level of the tree

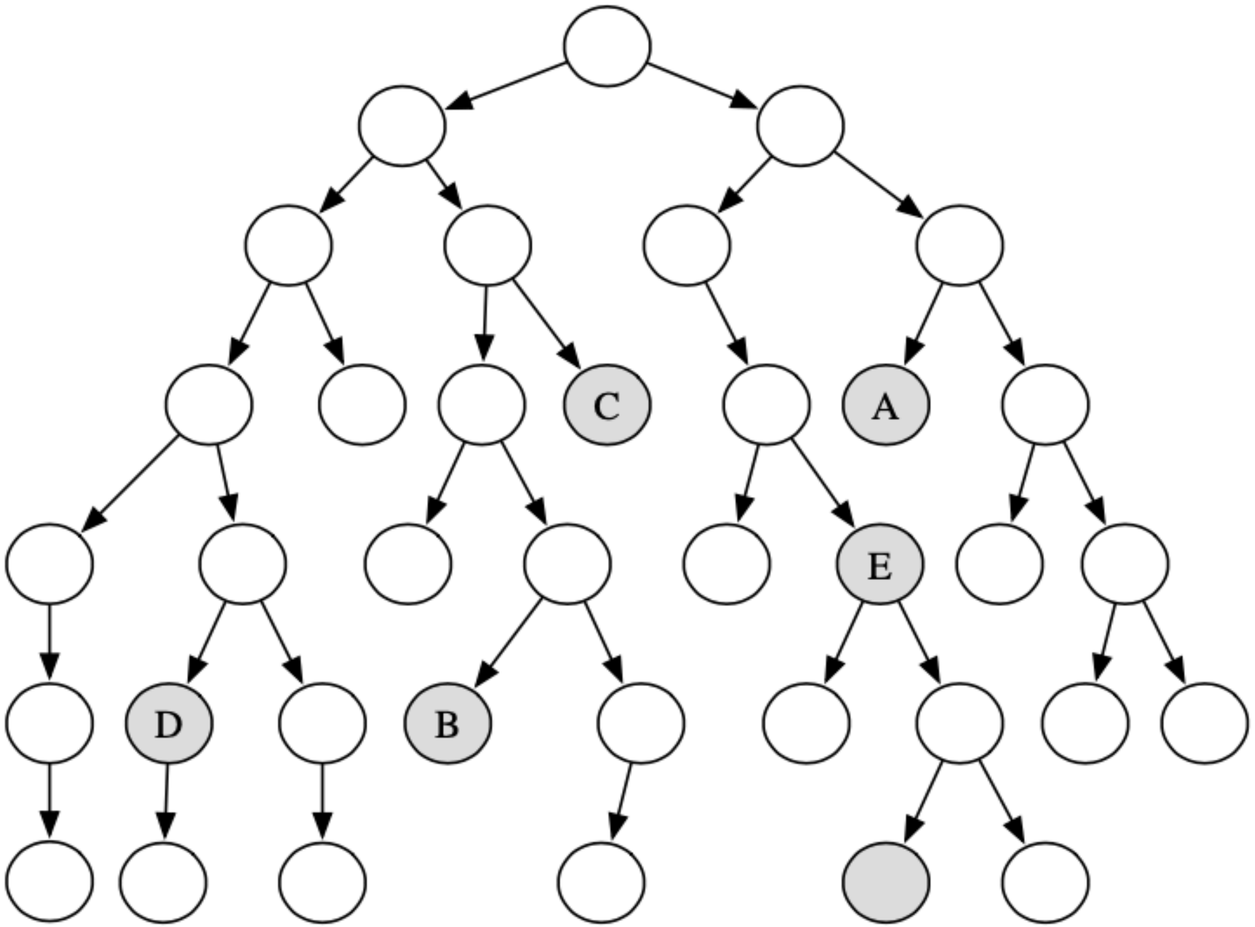
- Implementation:
  - Implementing the frontier as a stack = insert newly generated states at the front of the queue (thus making it a stack)
  - can alternatively be implemented by recursive function calls

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

DEPTH-FIRST SEARCH - DFS



WHICH SHADED GOAL WILL DEPTH-FIRST SEARCH FIND FIRST?

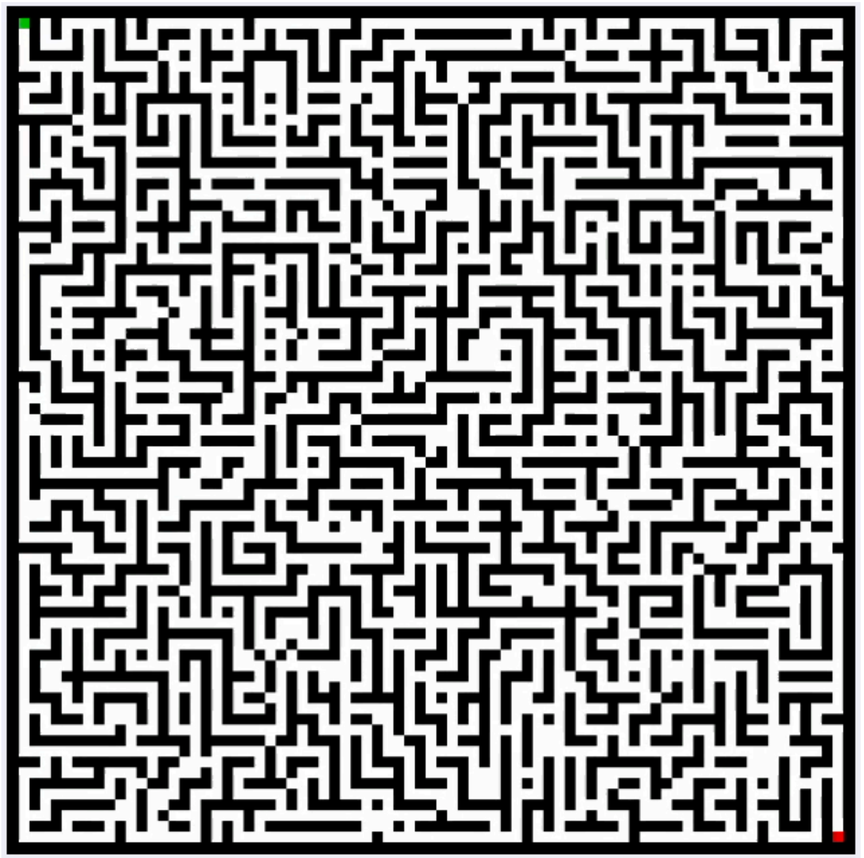


DEPTH FIRST SEARCH

- Idea: Always expand node at deepest level of tree and when search hits a dead-end return back to expand nodes at a shallower level
- Can be implemented using a stack of explored + frontier nodes
- At any point depth-first search stores single path from root to leaf
  - together with any remaining unexpanded siblings of nodes along path
- Stop when node with goal state is expanded
- Include check that generated state has not already been explored along a path – cycle checking

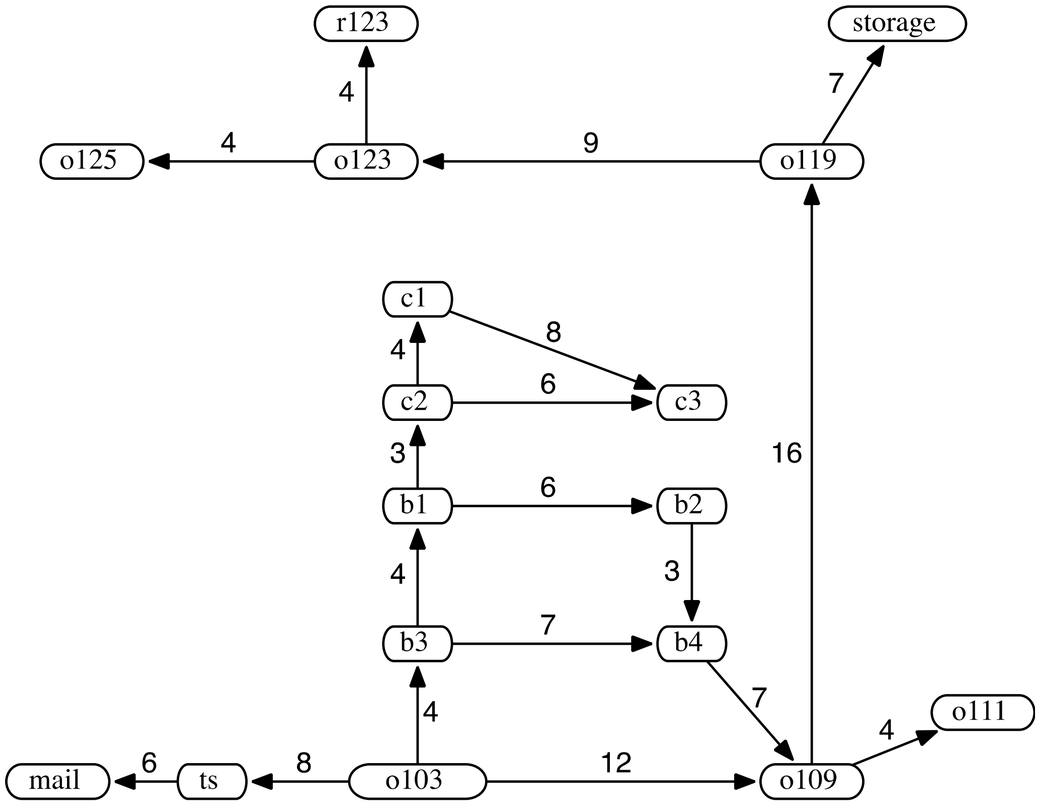
GUESS THE SEARCH PATH

Start

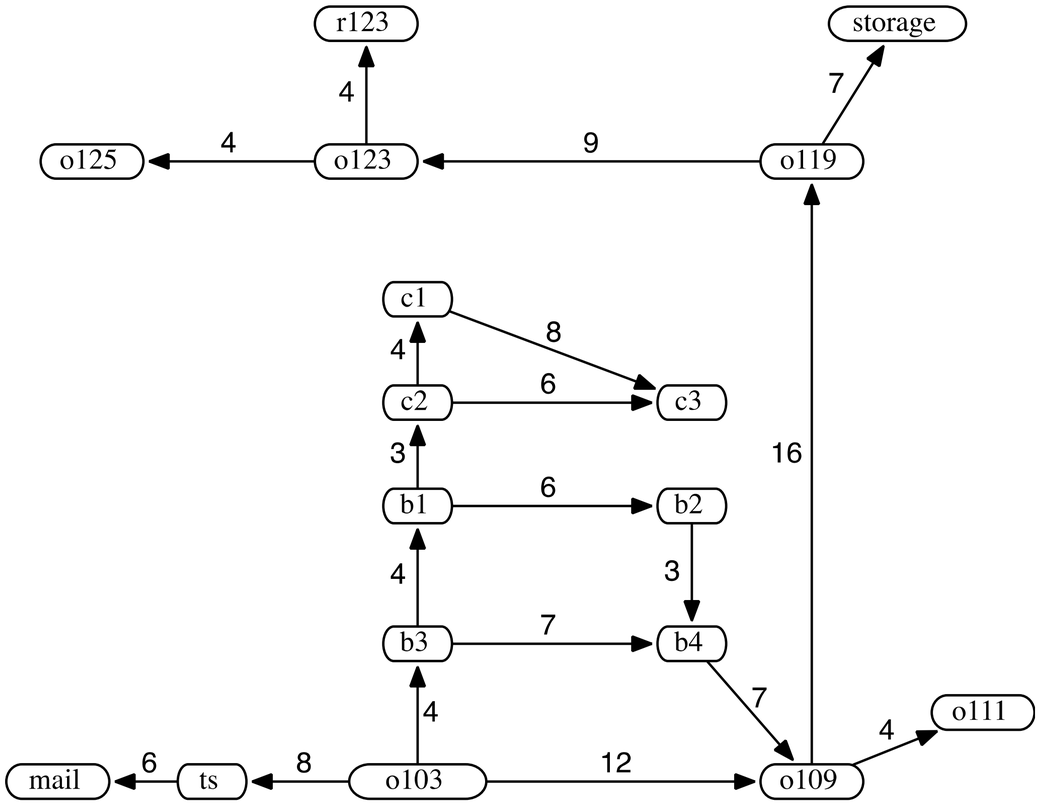


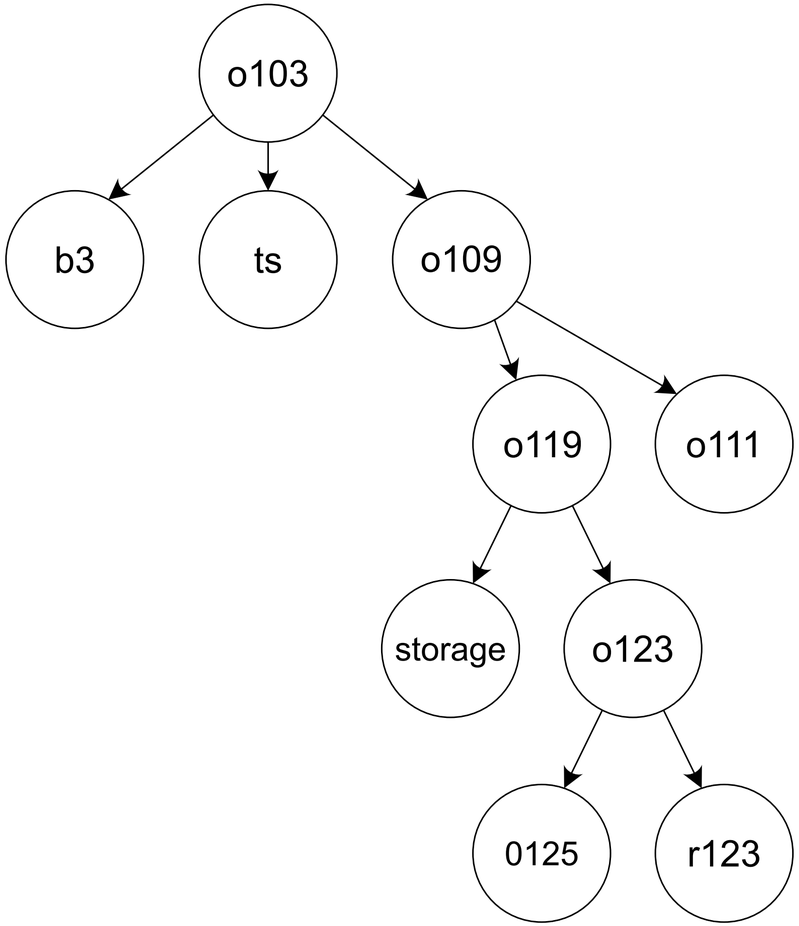
Goal

DEPTH-FIRST SEARCH



DEPTH-FIRST SEARCH





PROPERTIES OF DEPTH-FIRST SEARCH

Algorithm	Completeness	Admissibility	Space	Time
Depth-First Search	not guaranteed	not guaranteed	O(bm)	$O(b^m)$

- Complete? No! fails in infinite-depth spaces, spaces with loops
  - Modify to avoid repeated states along path in finite spaces
- Time? O(bm) , m = maximum depth of search tree, b = branching factor
  - terrible if m is much larger than b
  - but if solutions are dense, can be much faster than breadth-first search
- Space? O(bm), i.e., linear space!
- Optimal? No, can find suboptimal solutions first.

DEPTH-FIRST SEARCH — ANALYSIS

- In cases where problem has many solutions, depth-first search may outperform breadth-first search because there is a good chance it will find a solution after exploring only a small part of the space

- However, depth-first search may get stuck following a deep or infinite path even when a solution exists at a relatively shallow level

- Therefore, depth-first search is not complete and not optimal
  - Avoid depth-first search for problems with deep or infinite path

LOWEST-COST-FIRST SEARCH - UNIFORM-COST SEARCH

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

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

- An optimal solution is one with minimum cost.
- For many domains, arcs have non-unit costs, the aim is to find an optimal solution, a solution such that no other solution has a lower total cost.
  - For example, for a delivery robot, the cost of an arc may be resources (e.g., time, energy) required by the robot to carry out the action represented by the arc, and the aim is for the robot to solve a given goal using fewest resources.

LOWEST-COST-FIRST SEARCH - UNIFORM-COST SEARCH

- The simplest search method that is guaranteed to find a minimum cost path is Lowest-Cost-First Search or Uniform-Cost Search
  - is similar to breadth-first search, but it selects a path with the lowest cost.
  - the frontier is implemented as a priority queue ordered by the **cost function**.

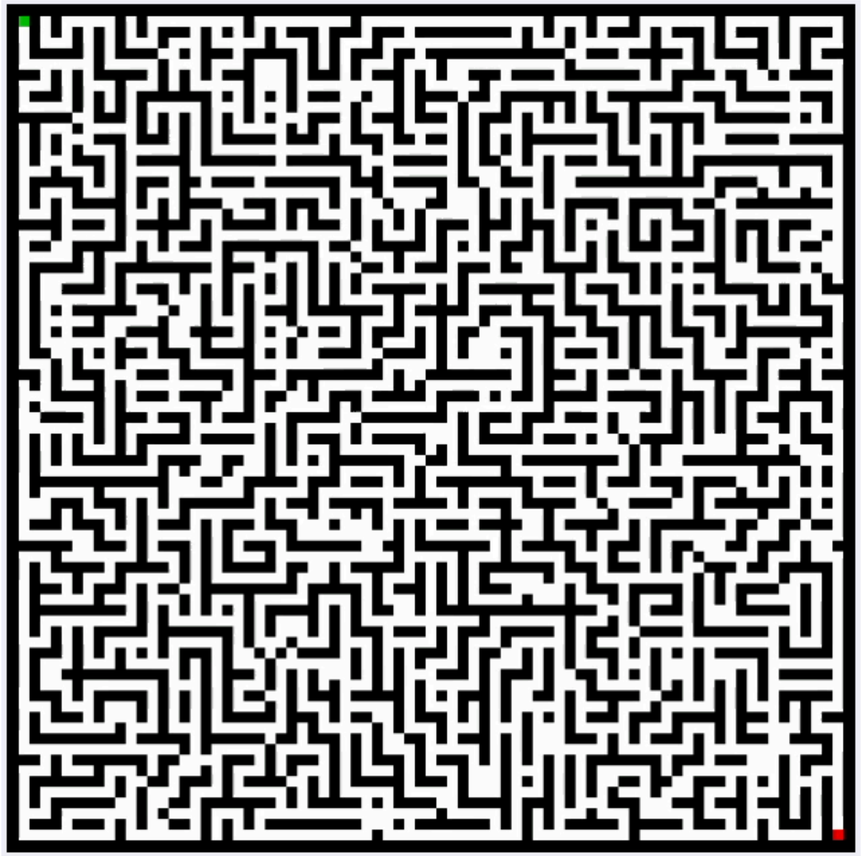
$$cost(\langle n_0, \ldots, n_k \rangle) = \sum_{i=1}^{k} cost(\langle n_{i-1}, n_i \rangle)$$

UNIFORM-COST SEARCH

- Expand root first, then expand least-cost unexpanded node
- Implementation: priority queue = insert nodes in order of increasing path cost - (lowest path cost g(n)).
- Reduces to Breadth First Search when all actions have same cost
- Finds the cheapest goal provided path cost is monotonically increasing along each path (i.e. no negative-cost steps)

GUESS THE SEARCH PATH

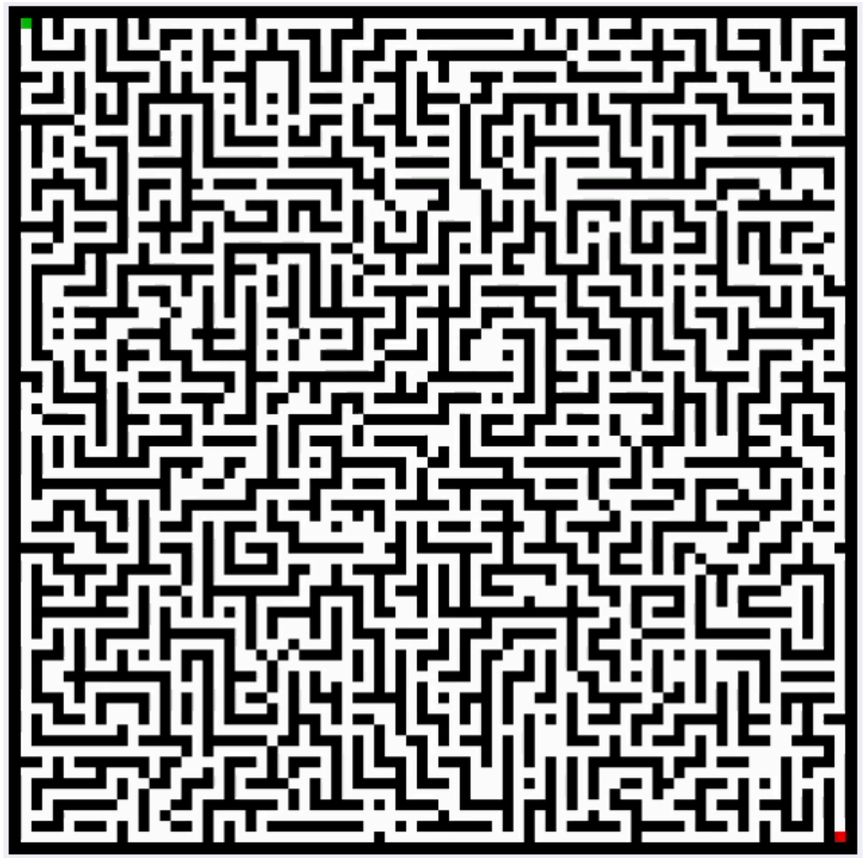
Start



Goal

GUESS THE SEARCH PATH

Start

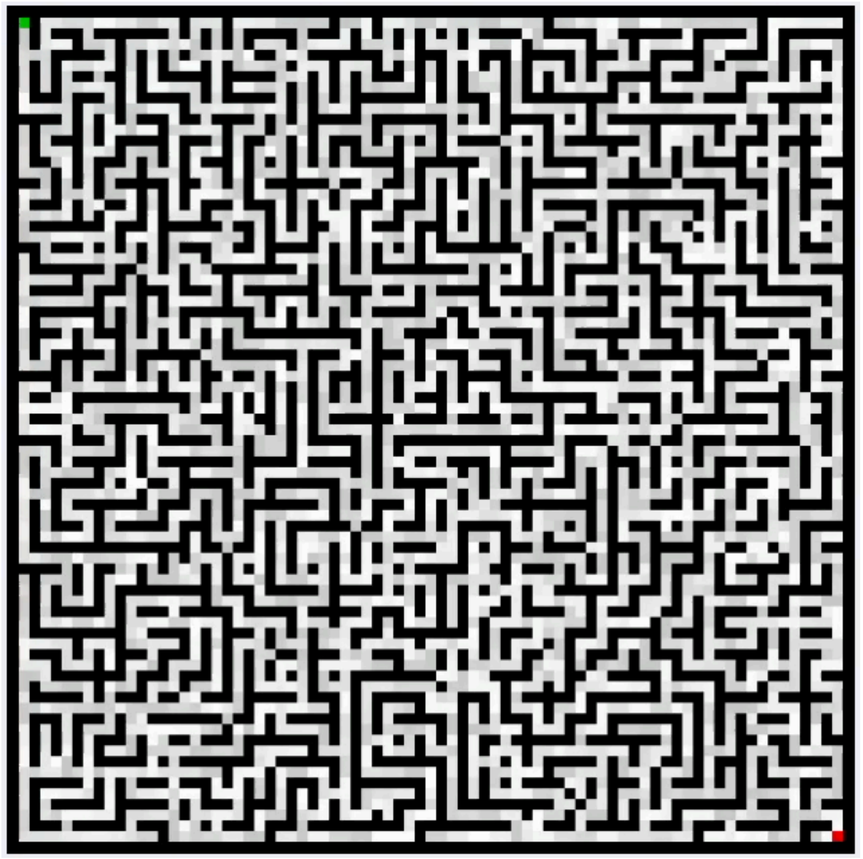


Goal

I cheated:
I just copied this video from the Breadth-First Search slide

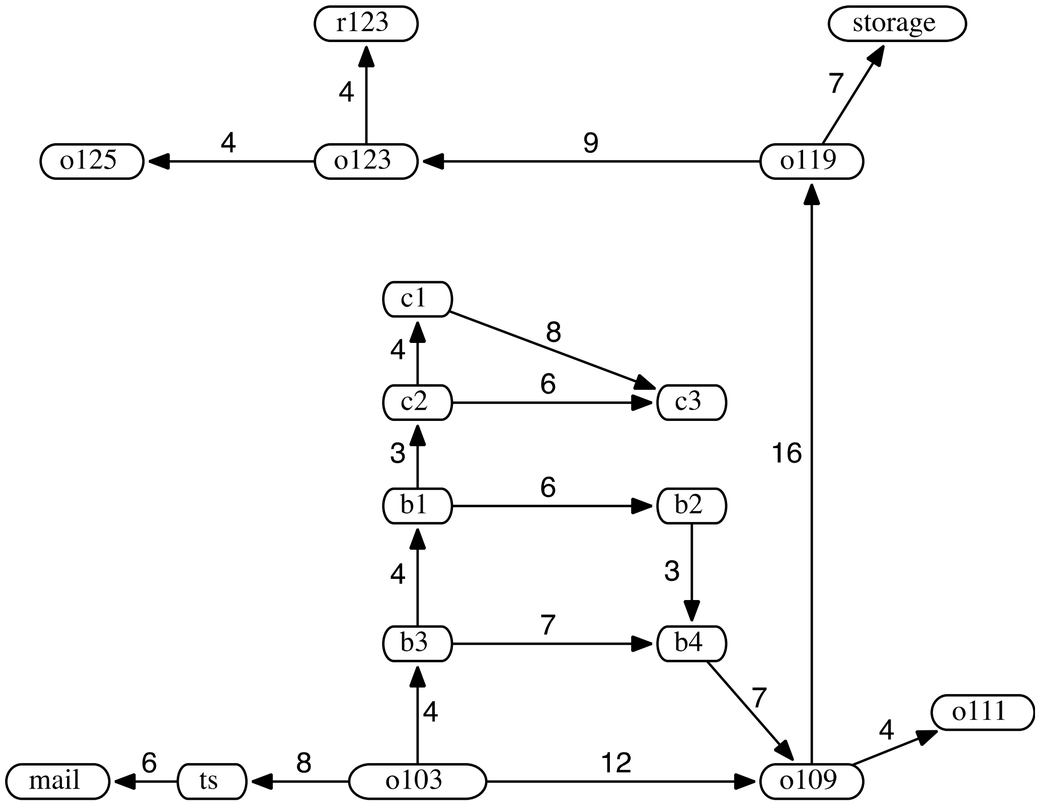
GUESS THE SEARCH PATH

Start

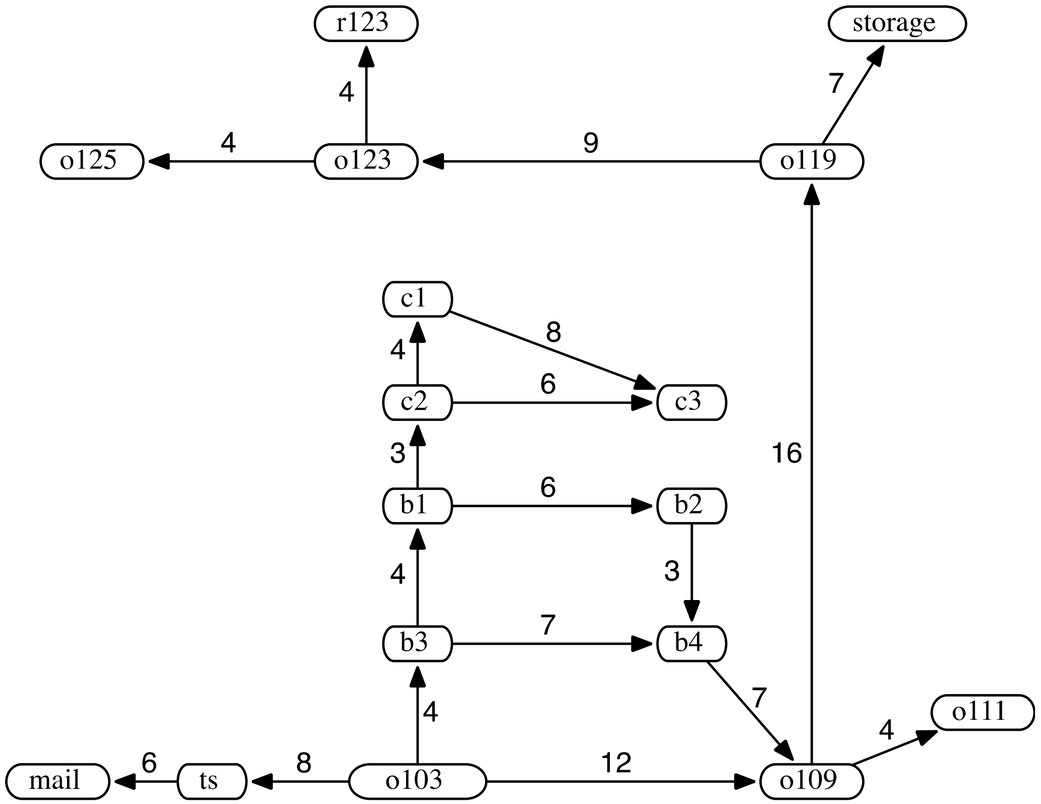


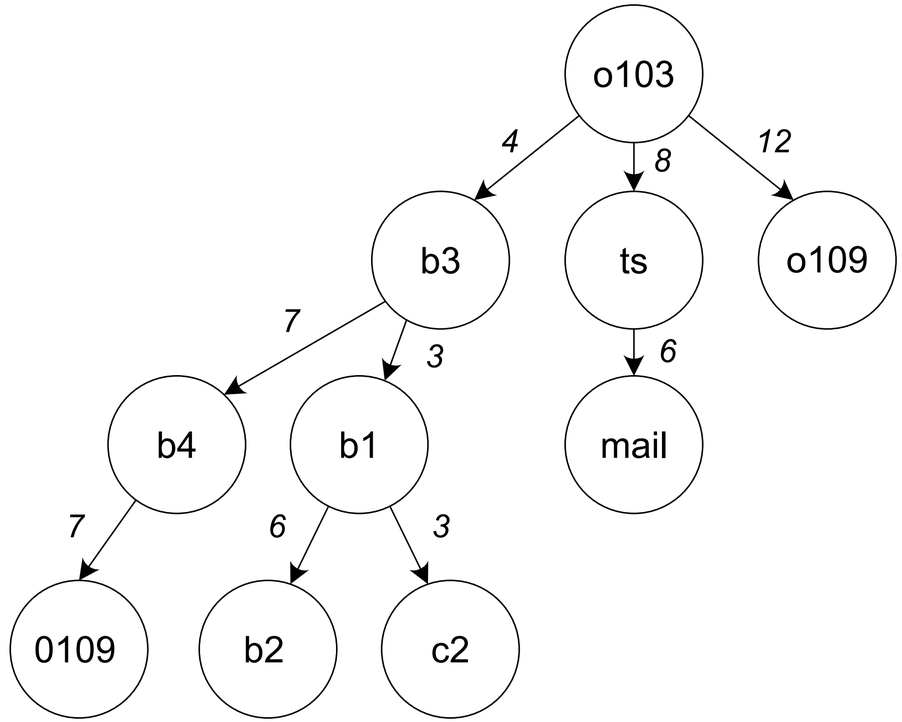
Goal

UNIFORM-COST SEARCH



UNIFORM-COST SEARCH





PROPERTIES OF UNIFORM-COST SEARCH

Algorithm	Completeness	Admissibility	Space	Time
Uniform-Cost Search	guaranteed, if b is finite and cost ≥ ε	guaranteed	$O(b^{[1 + C / \varepsilon]})$	$O(b^{[1 + C / \varepsilon]})$

- Complete? Yes, if b is finite and if step cost ≥ ε with ε > 0
- Time? $O(b^{[1 + C / \varepsilon]})$ where C* = cost of the optimal solution and assume every action costs at least ε
- Space? $O(b^{[1 + C / \varepsilon]})$
- Optimal? Yes – nodes expanded in increasing order of g(n)

DEPTH LIMITED SEARCH

Algorithm	Completeness	Admissibility	Space	Time
Depth Limited Search	guaranteed, if b is finite	not guaranteed	O(bk)	O(bk)

- Expands nodes like Depth First Search but imposes a cutoff on the maximum depth of path.
- Complete? Yes (no infinite loops anymore)
- Time? $O(b^k)$ where k is the depth limit
- Space? O(bk), i.e., linear space similar to DFS!
- Optimal? No, can find suboptimal solutions first.

Problem: How to pick a good limit ?

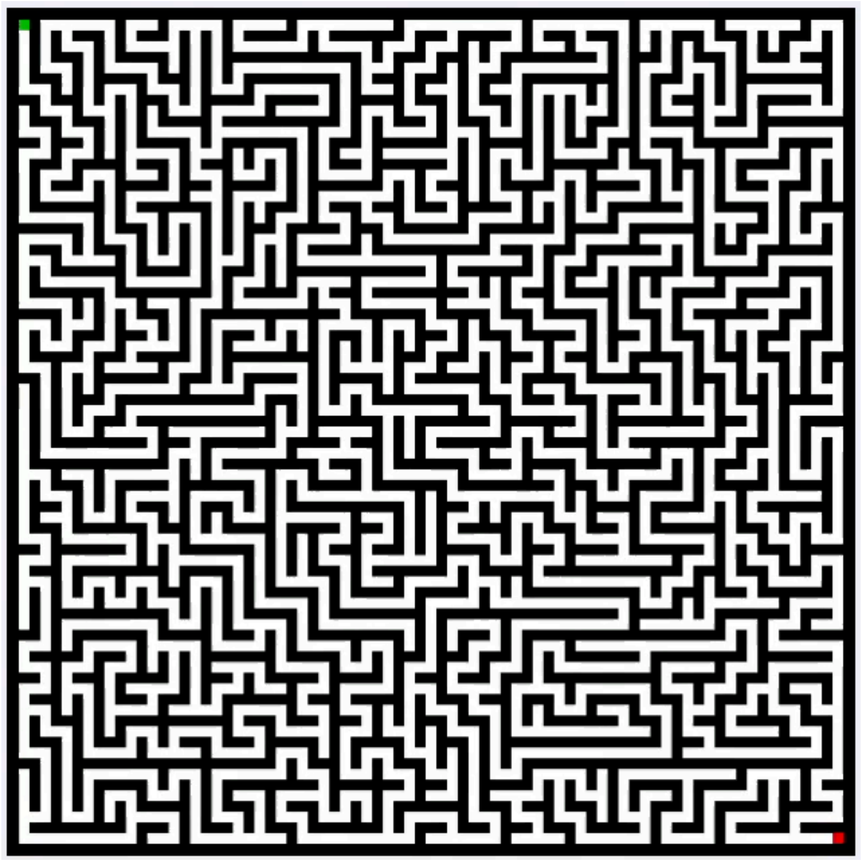
ITERATIVE DEEPENING SEARCH

- Tries to combine the benefits of depth-first (low memory) and breadth-first (optimal and complete) by doing a series of depth- limited searches to depth 1, 2, 3, etc.

- Early states will be expanded multiple times, but that might not matter too much because most of the nodes are near the leaves.

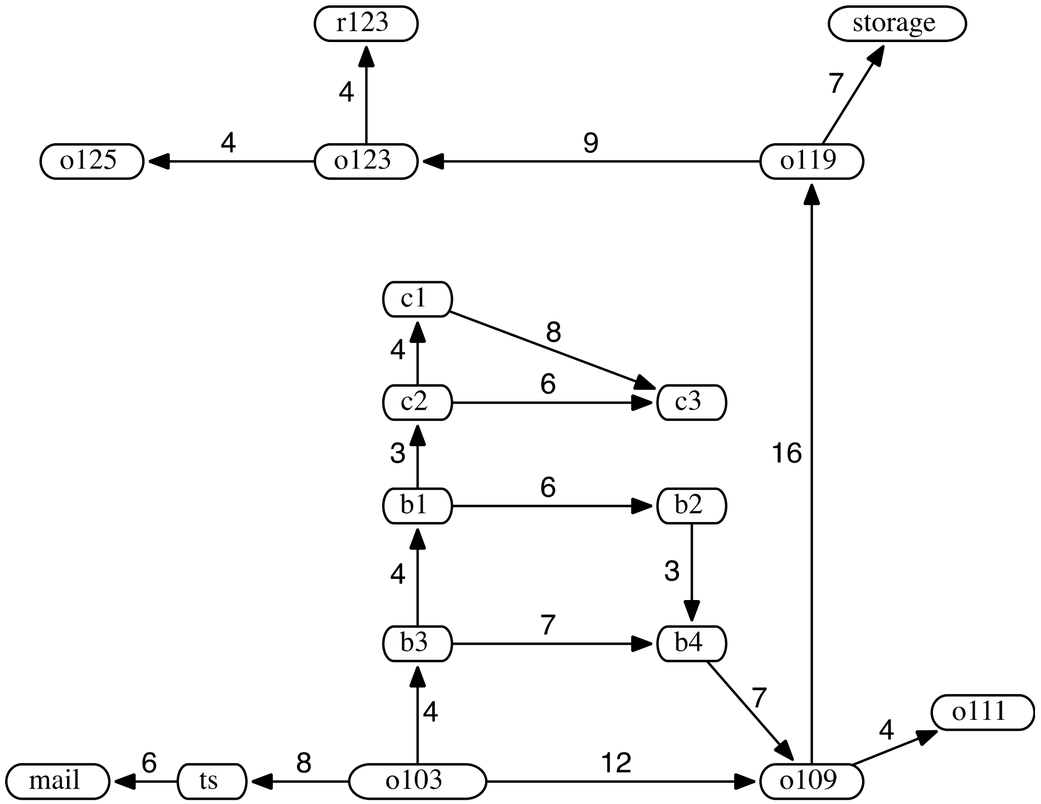
GUESS THE SEARCH PATH

Start

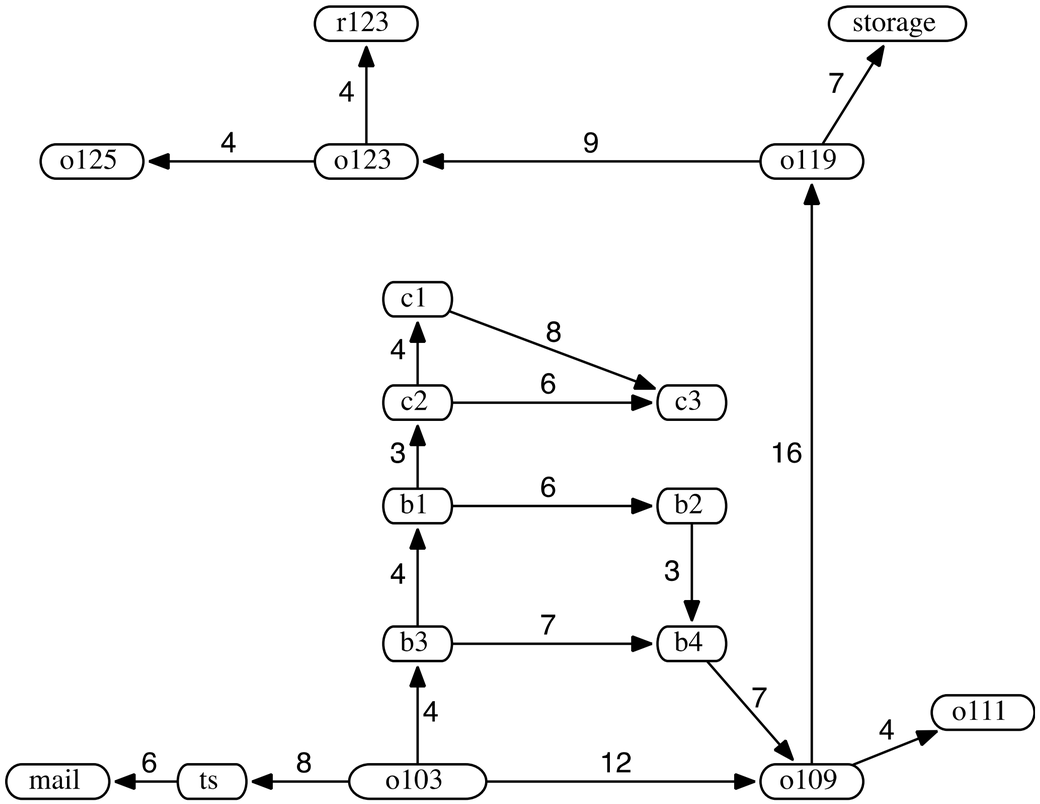


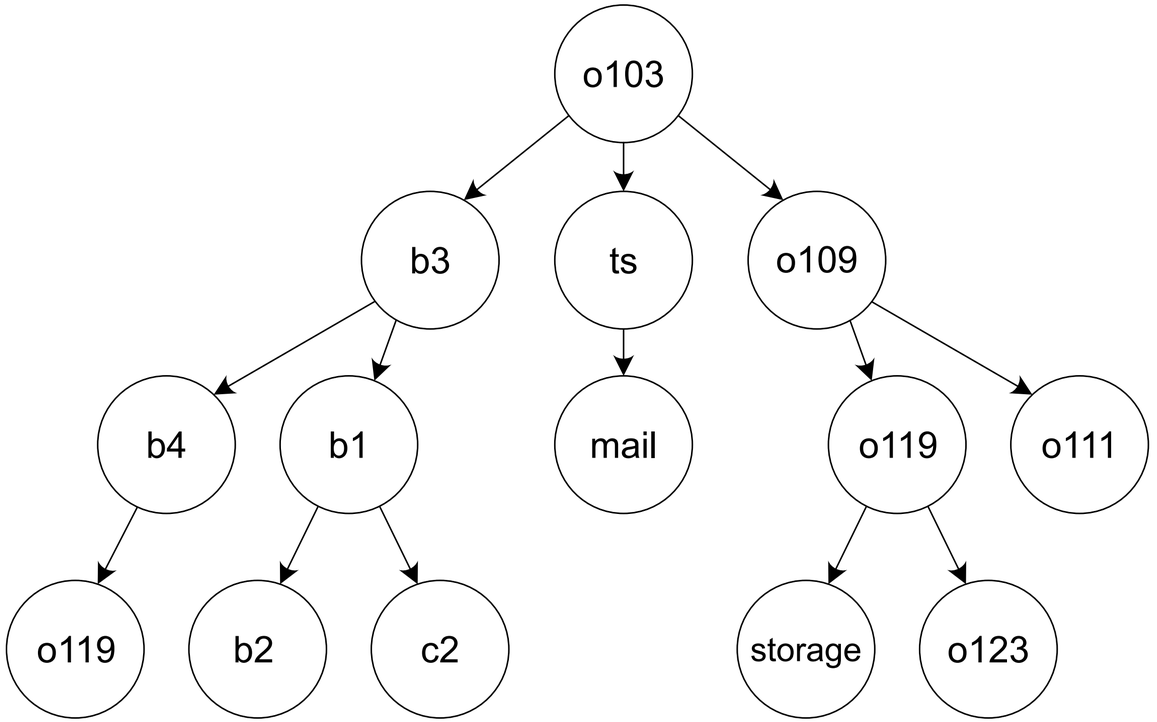
Goal

ITERATIVE DEEPENING SEARCH



ITERATIVE DEEPENING SEARCH





ITERATIVE DEEPENING SEARCH

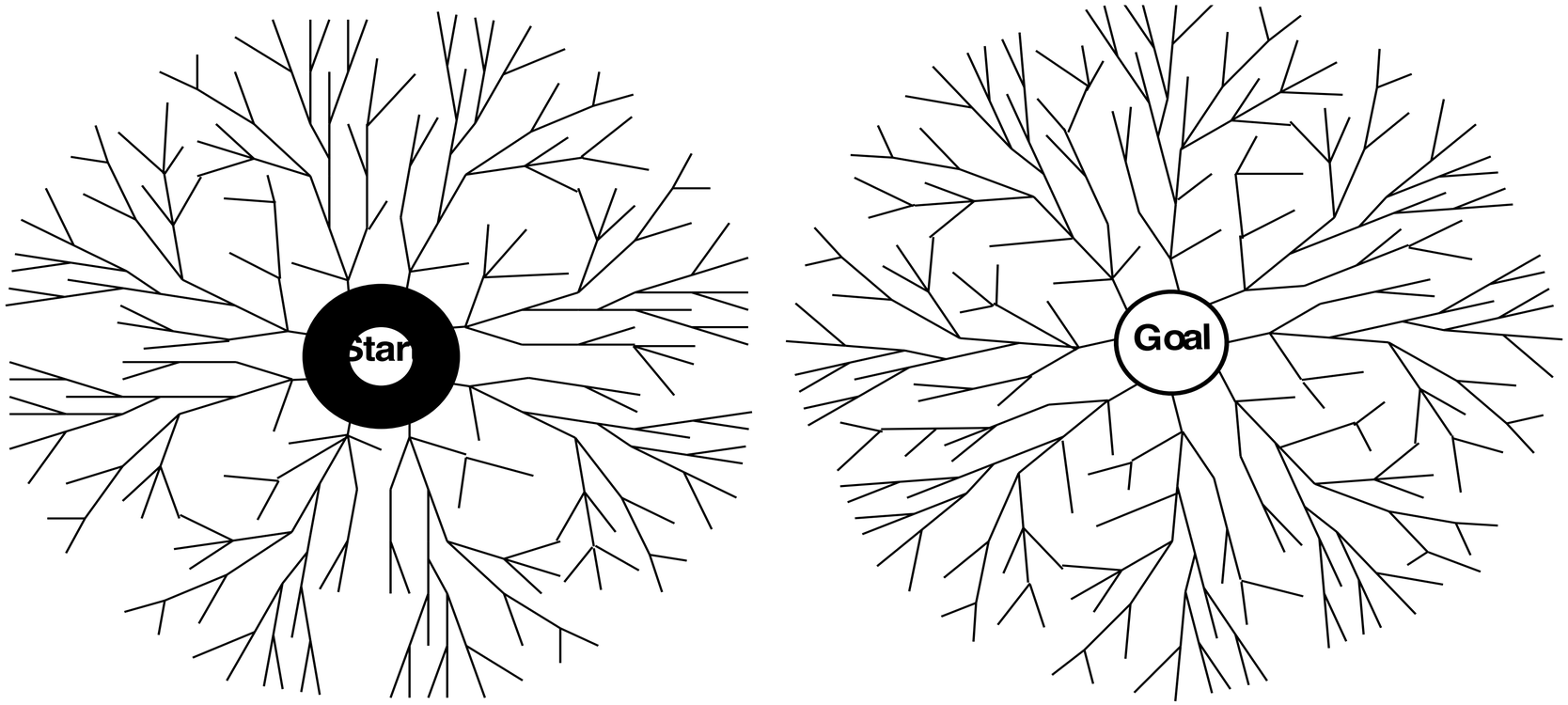
- It can be very difficult to decide upon a depth limit for search
- The maximum path cost between any two nodes is known as the diameter of the state space
- This would be a good candidate for a depth limit but it may be difficult to determine in advance
- Idea: Try all possible depth limits in turn
- Combines benefits of depth-first and breadth-first search

PROPERTIES OF ITERATIVE DEEPENING SEARCH

Algorithm	Completeness	Admissibility	Space	Time
IDS	guaranteed, if b is finite	guaranteed, if arcs have the same cost	O(bd)	$O(b^d)$

- Complete? Yes.
- Time: $O(b^d)$ nodes at the top level are expanded once, nodes at the next level twice, and so on
- Space? O(bd)
- Optimal? Yes, if step costs are identical.
- In general, iterative deepening is the preferred search strategy for a large search space where depth of solution is not known

BIDIRECTIONAL SEARCH



BIDIRECTIONAL SEARCH

- Idea: Search both forward from the initial state and backward from the goal, and stop when the two searches meet in the middle.

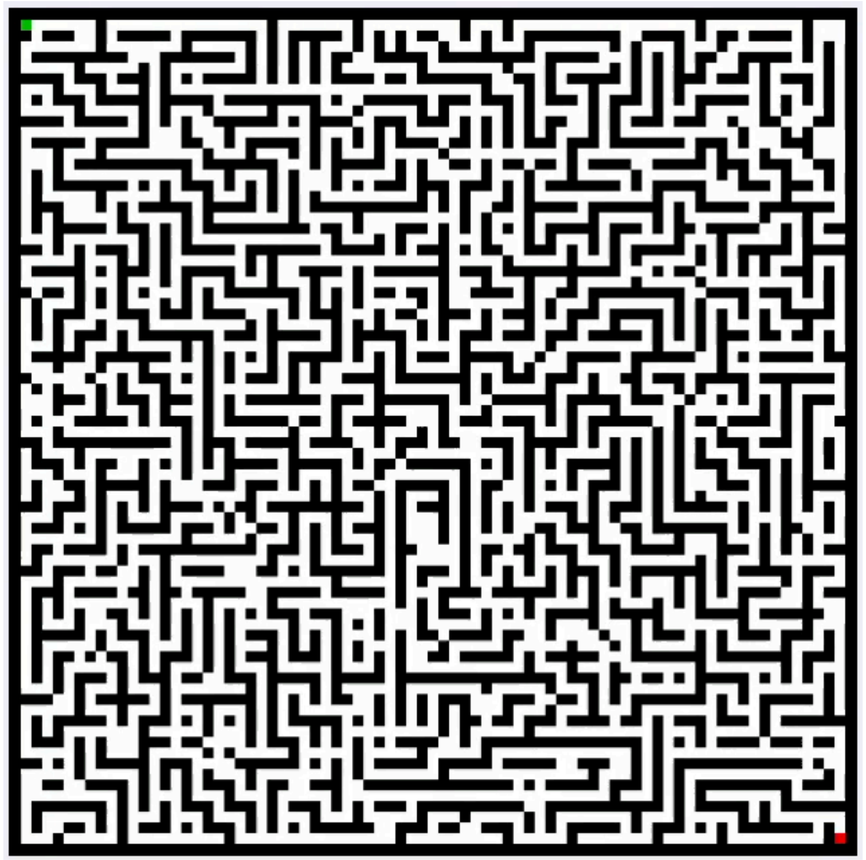
- We need an efficient way to check if a new node already appears in the other half of the search. The complexity analysis assumes this can be done in constant time, using a Hash Table.

- Assume branching factor = b in both directions and that there is a solution at depth = d. Then bidirectional search finds a solution in $O(2b^{d/2}) = O(b^{d/2})$ time steps.

- To check for intersection it must have all states from one of the searches in memory, therefore space complexity is $O(b^{d/2})+O(b^{d/2}) = 2O(b^{d/2}) = O(b^{d/2})$

GUESS THE SEARCH PATH

Start



Goal

BIDIRECTIONAL SEARCH — ANALYSIS

Algorithm	Completeness	Admissibility	Space	Time
Bidirectional Search	depends on search algorithms used	depends on search algorithms used	$O(b^{d/2})$	$O(b^{d/2})$

- Complete? Depends on the search algorithm used. If BFS, then yes.
- Time: $O(b^{d/2})$
- Space? $O(b^{d/2})$
- Optimal? Depends on the search algorithm used. If BFS, then yes.

COMPLEXITY RESULTS FOR UNINFORMED SEARCH

Algorithm	Completeness	Admissibility	Space	Time
Breadth-First Search	guaranteed, if b is finite	guaranteed, if arcs have the same cost and >=0	$O(b^d)$	$O(b^d)$
Depth-First Search	not guaranteed	not guaranteed	$O(bm)$	$O(b^m)$
Uniform Cost Search	guaranteed, if b is finite and cost ≥ ε	guaranteed	$O(b^{[1 + C / \varepsilon]})$	$O(b^{[1 + C / \varepsilon]})$
Depth Limited Search	guaranteed, if b is finite	not guaranteed	$O(bk)$	$O(bk)$
Iterative Deepening Search	guaranteed, if b is finite	guaranteed, if arcs have the same cost and >=0	$O(bd)$	$O(b^d)$
Bidirectional Search	depends on search algorithms used	depends on search algorithms used	$O(b^{d/2})$	$O(b^{d/2})$

SUMMARY

- Problem formulation usually requires abstracting away real-world details to define a state space that can feasibly be explored.
- Variety of Uninformed search strategies
- Iterative Deepening Search uses only linear space and not much more time than other Uninformed algorithms.