Problem Solving

Artificial Intelligence

Christopher Simpkins

Problem-Solving Agents

- Goal formulation
- Problem formulation
- Search
- Execution

open-loop vs. closed-lop

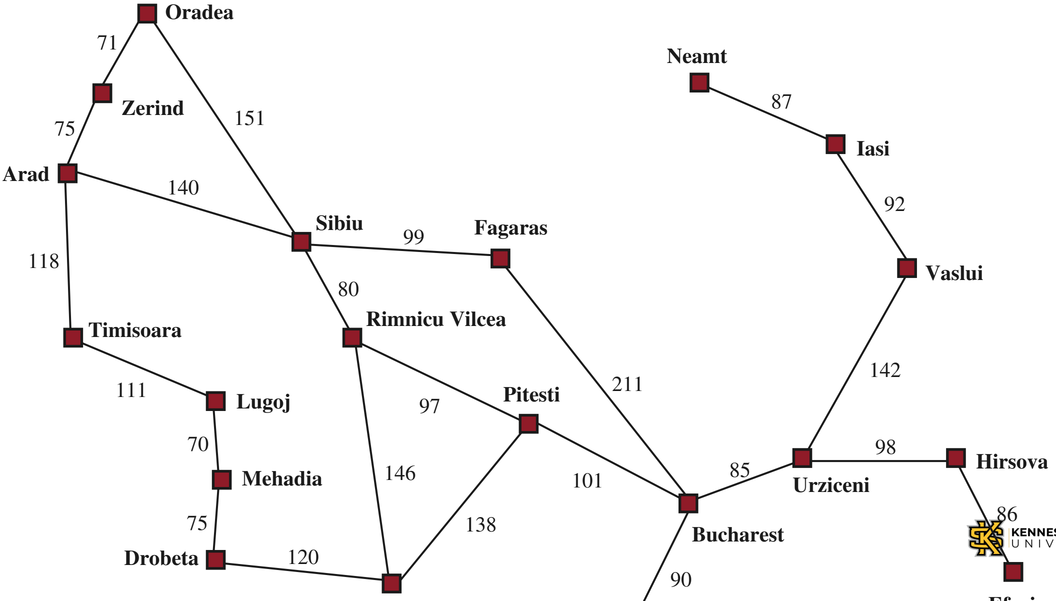
Search Problems and Solutions

Search problem:

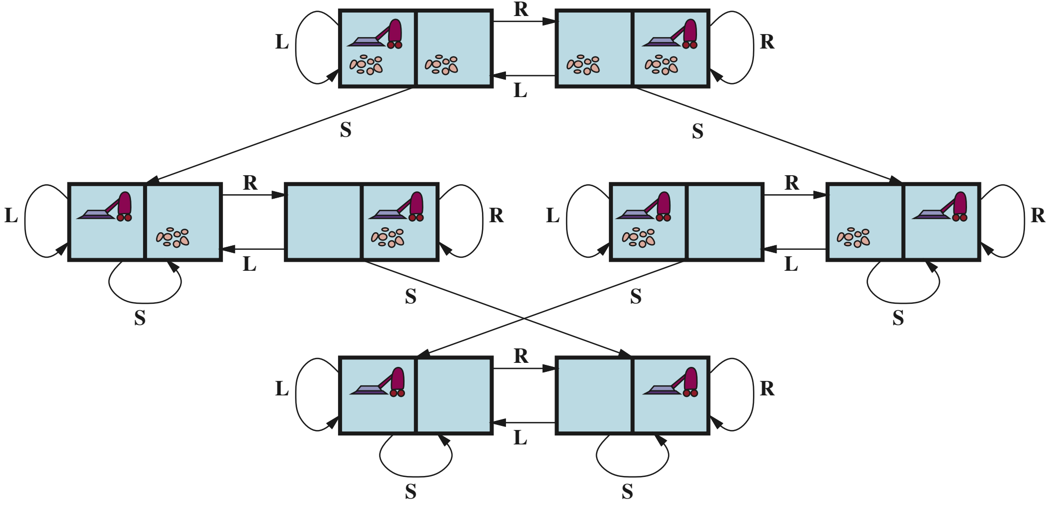
- A set of **states**, which we call a **state space**.
- **Initial state**
- A set of **goal states**
- Sets of **actions** available in each state, `ACTION(s)`
  - `ACTION(Arad)= {ToSibiu, ToTimisoara, ToZerind}`
- A **transition model**, `RESULT(s, a)`
  - `RESULT(Arad, ToZerind)= Zerind`
- An **action cost function**, `ACTION-COST(s, a, s')` or $c(s, a, s')$ which returns the cost of executing action a in state s and reaching state s'.

Solution

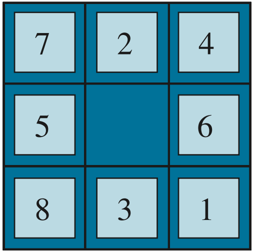
- A solution is a path from the start state to the a goal state.
- An optimal solution is a solution with lowest cost among all solutions.



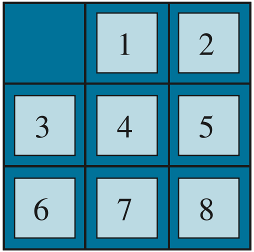
Vacuum State Space Graph



Agents



Start State

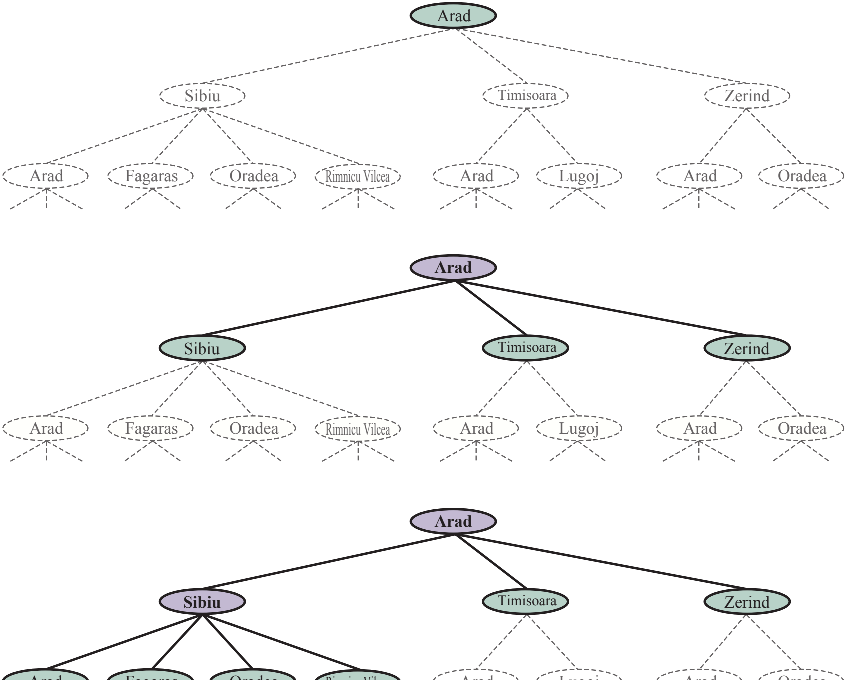


Goal State

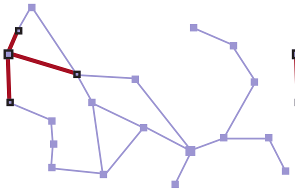
Search Algorithms

- Search tree

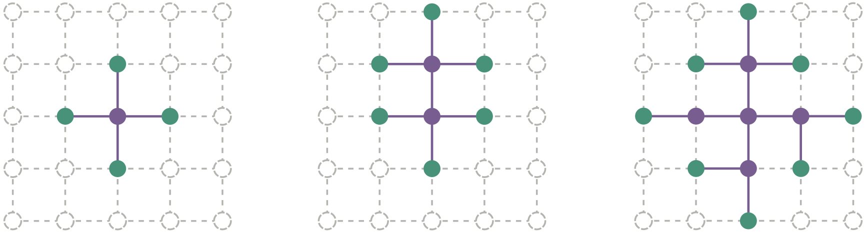
Searching State Space



Search Tree Expansion



Separation Property of Graph Search



(a) (b) (c)

- (a) Only root expanded.
- (b) Top frontier node expanded.
- (c) Remaining successors of root expanded in clockwise order.

Best-First Search Algorithm

```
function BEST-FIRST-SEARCH(problem, f) returns a solution node or failure
  node ← NODE(STATE=problem.INITIAL)
  frontier ← a priority queue ordered by f, with node as an element
  reached ← a lookup table, with one entry with key problem.INITIAL and value node
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    if problem.IS-GOAL(node.STATE) then return node
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if s is not in reached or child.PATH-COST < reached[s].PATH-COST then
        reached[s] ← child
        add child to frontier
  return failure

function EXPAND(problem, node) yields nodes
  s ← node.STATE
  for each action in problem.ACTIONS(s) do
    s′ ← problem.RESULT(s, action)
    cost ← node.PATH-COST + problem.ACTION-COST(s, action, s′)
    yield NODE(STATE=s′, PARENT=node, ACTION=action, PATH-COST=cost)
```

Search Data Structures

Node:

- `node.STATE`: the state to which the node corresponds;
- `node.PARENT`: the node in the tree that generated this node;
- `node.ACTION`: the action that was applied to the parent's state to generate this node;
- `node.PATH-COST`: the total cost of the path from the initial state to this node. In mathematical formulas, we use g(node) as a synonym for PATH-COST.

Frontier:

- `IS-EMPTY(frontier)` returns true only if there are no nodes in the frontier.
- `POP(frontier)` removes the top node from the frontier and returns it.
- `TOP(frontier)` returns (but does not remove) the top node of the frontier.
- `ADD(node, frontier)` inserts node into its proper place in the queue.

Queues used in search algorithms:

- A **priority queue** first pops the node with the minimum cost according to some evaluation function, f . It is used in best-first search.
- A **FIFO queue** or first-in-first-out queue first pops the node that was added to the queue first; we shall see it is used in breadth-first search.
- A **LIFO queue** or last-in-first-out queue (also known as a stack) pops first the most recently added node; we shall see it is used in depth-first search.

Redundant Paths

Repeated states

cycles

redundant paths

graph search

tree-like search

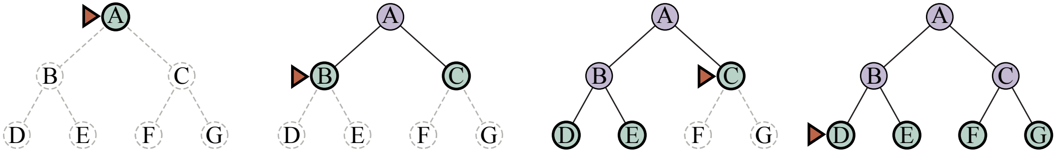
Measuring Problem-Solving Performance

- Completeness: Is the algorithm guaranteed to find a solution when there is one, and to correctly report failure when there is not?
- Cost optimality: Does it find a solution with the lowest path cost of all solutions?
- Time complexity: How long does it take to find a solution? This can be measured in seconds, or more abstractly by the number of states and actions considered.
- Space complexity: How much memory is needed to perform the search?

Uninformed Search Strategies

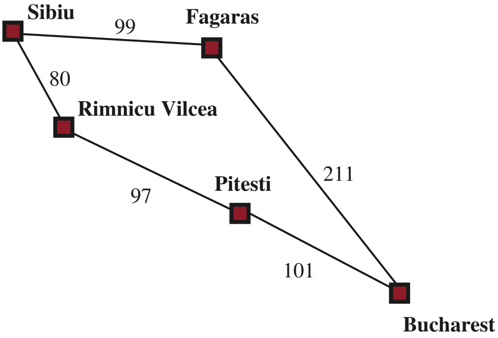
Strategy:

Breadth-First Search

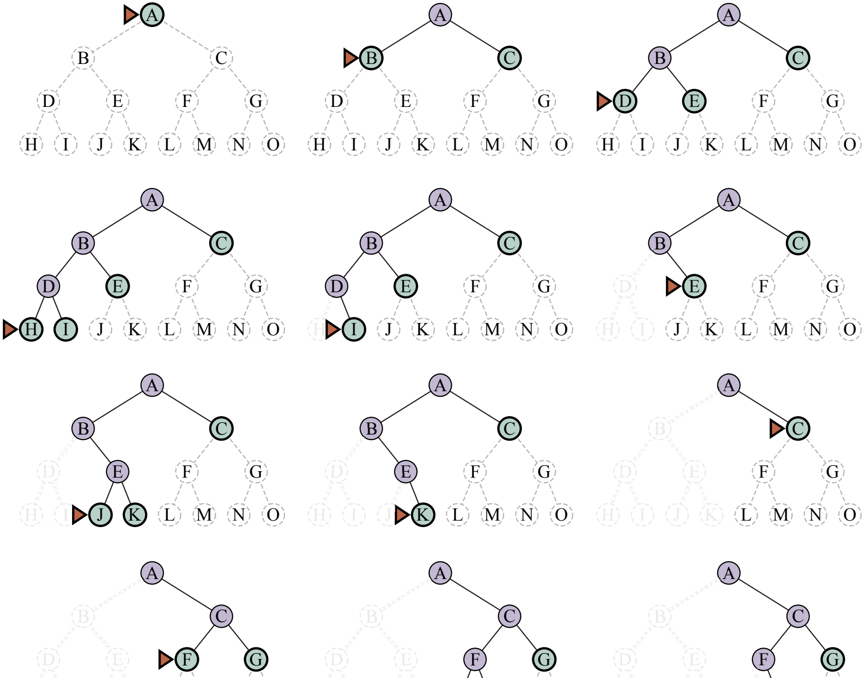



```
function BREADTH-FIRST-SEARCH(problem) returns a solution node or failure
  node ← NODE(problem.INITIAL)
  if problem.IS-GOAL(node.STATE) then return node
  frontier ← a FIFO queue, with node as an element
  reached ← {problem.INITIAL}
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if problem.IS-GOAL(s) then return child
      if s is not in reached then
        add s to reached
        add child to frontier
  return failure
```

Dijkstra's Algorithm



Depth-First Search

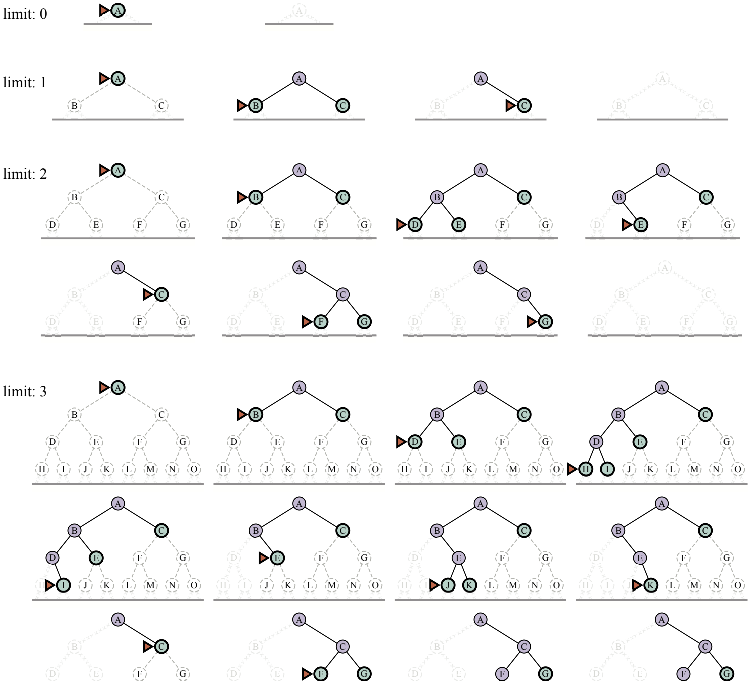


Depth-Limited Search and Iterative Deepening Search

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution node or failure
  for depth = 0 to ∞ 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 LIFO queue (stack) with NODE(problem.INITIAL) as an 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
```

Progression of Iterative Deepening Search



Bidirectional Best-First Search

```
function BIBF-SEARCH(problem_F, f_F, problem_B, f_B) returns a solution node, or failure
  node_F ← NODE(problem_F.INITIAL)                          // Node for a start state
  node_B ← NODE(problem_B.INITIAL)                          // Node for a goal state
  frontier_F ← a priority queue ordered by f_F, with node_F as an element
  frontier_B ← a priority queue ordered by f_B, with node_B as an element
  reached_F ← a lookup table, with one key node_F.STATE and value node_F
  reached_B ← a lookup table, with one key node_B.STATE and value node_B
  solution ← failure
  while not TERMINATED(solution, frontier_F, frontier_B) do
    if f_F(TOP(frontier_F)) < f_B(TOP(frontier_B)) then
      solution ← PROCEED(F, problem_F, frontier_F, reached_F, reached_B, solution)
    else solution ← PROCEED(B, problem_B, frontier_B, reached_B, reached_F, solution)
  return solution

function PROCEED(dir, problem, frontier, reached, reached_2, solution) returns a solution
        // Expand node on frontier; check against the other frontier in reached_2.
        // The variable "dir" is the direction: either F for forward or B for backward.
  node ← POP(frontier)
  for each child in EXPAND(problem, node) do
    s ← child.STATE
    if s not in reached or PATH-COST(child) < PATH-COST(reached[s]) then
      reached[s] ← child
      add child to frontier
      if s is in reached_2 then
        solution_2 ← JOIN-NODES(dir, child, reached_2[s]))
        if PATH-COST(solution_2) < PATH-COST(solution) then
```

Comparing Uninformed Search Algorithms

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes[1]	Yes[1,2]	No	No	Yes[1]	Yes[1,4]
Optimal cost?	Yes[3]	Yes	No	No	Yes[3]	Yes[3,4]
Time	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$