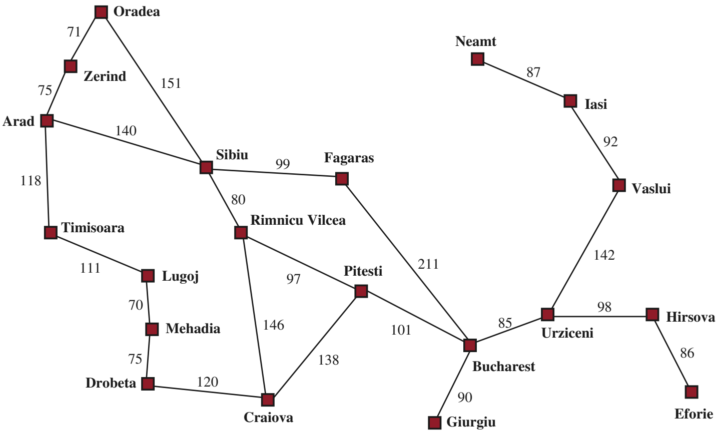
Problem Solving

Artificial Intelligence

Christopher Simpkins

KENNESAW STATE UNIVERSITY

Problem-Solving Agents



- In this lesson we consider a *state* to be our location in one of these cities.
- A *goal* is a state in which we are located in a particular city.

This is the essence of problem solving: transforming a current state into a goal state. The first family of algorithms we'll study for problem solving are *search* algorithms.

Problem Solving Process

To solve a problem, we

- Formulate a **goal**, e.g., "reach Bucharest"
- Formulate the **problem** as a set of states and actions that move us from one state to another.
  - Problem is a **model** – an *abstract* mathematical description.
  - Abstraction is essence and ignorance.
  - Key skill in problem formulation is finding the right **level of abstraction**.
- **Search** the possible sequences of action in our problem model that transforms our state from the current state to the goal state. A sequence of actions that gets us to the goal state is called a *solution*. May be many; pick one.
- **Execute** the actions in the solution.

Open-Loop vs. Closed-Loop

- In an **open-loop** system the agent gets no feedback, i.e., sensor input, after executing an action.
  - If the agent's model is perfect and actions are deterministic, then the agent can operate in an open-loop fashion, simply executing the actions in the solution one after the other.
- In a **closed-loop** system gets sensory feedback after every action, so it can check whether the action had the expected effect.
  - If the environment is partially observable or actions are nondeterministic, closed-loop control is necessary.
  - Say the agent executes to `ToSibiu` action but ends up in `Zerind`. Closed-loop feedback will alert the agent to this fact so it can re-plan.

Search Problems and Solutions

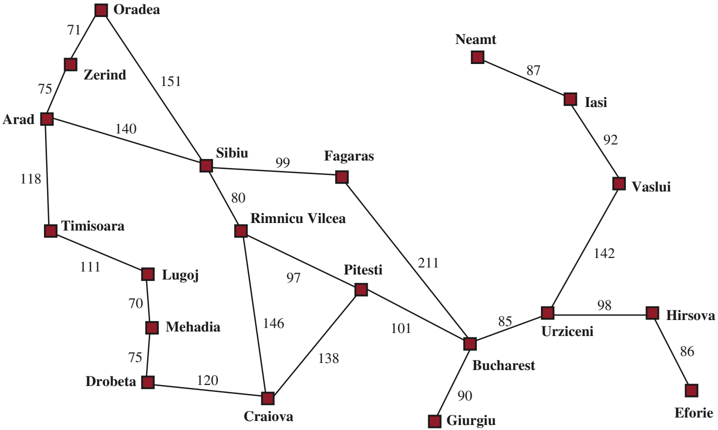
A search problem consists of:

- A set of **states**, which we call a **state space**.
- **Initial state**
- A set of **goal states**.
  - Typically use an `IS-GOAL(s)` predicate function to identify 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'.

KENNESAW STATE UNIVERSITY

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.

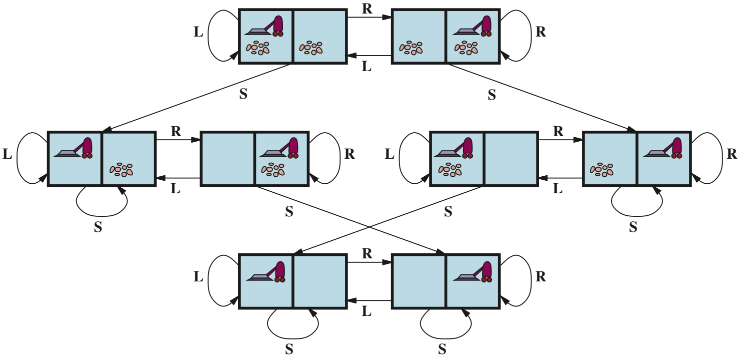


- How many paths are there from `Arad` to `Bucharest`?
- What is/are the solutions to the `Arad-to-Bucharest` problem (assume perfect information – fully observable, known dynamics, and deterministic actions)?

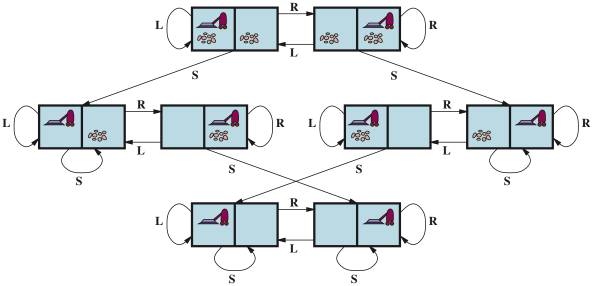
Example Problems

- **Standardized problems** use idealized environments designed to illustrate or exercise various problem-solving methods. See, for example, Gymnasium.
  - A **grid world** is an standardized environment whose states are organized as a grid, and whose actions include moving between adjacent grids.
- **Real-world problems** are formulated for specific real-world tasks, like the problem specification used for Rhoombas.

Here's a standardized environment for the vacuum cleaner agent, formulated as a grid world:



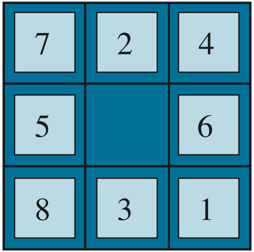
Vacuum Cleaner Grid World



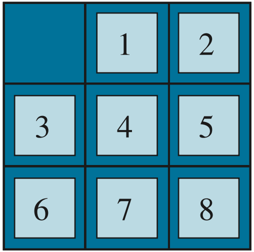
- **States** include both the agent's location, and characteristics of the environment. For the vacuum world, that's $2 \cdot 2^2 = 8$ states.
- **Initial state** is an arbitrary choice of the possible states. Sometimes this choice is important.
- **Actions** for this vacuum world are are `L`, `R`, and `Suck`.
  - For 2D grids we can choose between
    - **absolute** movement, like `Up` and `Right`, a.k.a., cardinal directions, or
    - **egocentric** movement, like `TurnRight`, `MoveForward`. How does this affect the state description?
- **Goal states** are those in which every location is clean.
- **Action cost** (path cost) is 1.

KENNESAW STATE UNIVERSITY

Agents



Start State



Goal State

Route Finding

- **States**: a location (e.g., an airport) and the time.
  - If action cost (e.g., a flight segment) depends on previous segments, fares, etc., the state must include these details.
- **Initial state**: The user's home airport.
- **Actions**: Take any flight from the current location, in any seat class, leaving after the current time, or for connecting flights, after sufficient in-airport transfer time.
- **Transition model**: The state resulting from taking a flight will have the flight's destination as the new location and the flight's arrival time as the new time.
  - Example $T(s, a, s')$: `T(S(ATL, 10:00), A(DL875), S(LGA, 12:00))` (DL875 has a flight time of 2 hours).
- **Goal state**: A destination city. Sometimes the goal can be more complex, such as arrive at the destination on a nonstop flight. (Remember, a solution is a path, i.e., sequence of actions.)
- **Action cost**: A combination of monetary cost, waiting time, flight time, customs and immigration procedures, seat quality, time of day, type of airplane, frequent-flyer reward points, and so on.

Real-World Problems

- **Touring problems**
- **VLSI layout –** minimize area, minimize circuit delays, minimize stray capacitances, and maximize manufacturing yield
  - Cell layout – place cells on chip so they don't overlap and have room for connections
  - Channel routing – find routes for each wire between cells
- **Robot navigation**
- **Automatic assembly sequencing –** standard practice in manufacturing since the 1970s.
  - Solving some automatic assembly problems could earn you a Nobel Prize!

Search Algorithms

A **search algorithm** takes a search problem as input and returns a solution, or an indication of failure.

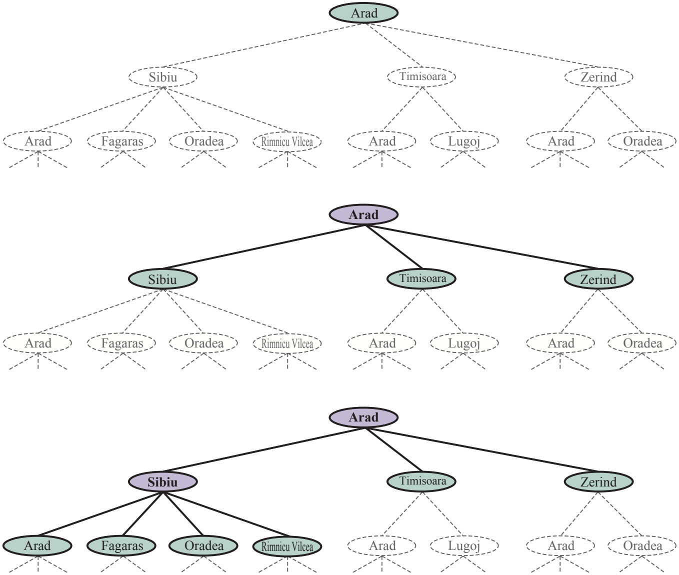
- In general, the states and actions of a problem create a state space graph.
- Here we consider algorithms that superimpose a **search tree** over the state-space graph.
- **Nodes** correspond to states, **edges** correspond to actions
  - May be many nodes for a given state, but each path is unique.

Don't confuse state space with search tree.

- State space describes the set of states and actions that case transitions from one state to another.
- Search tree describes paths between these states, reaching towards the goal(s).

Searching State Space

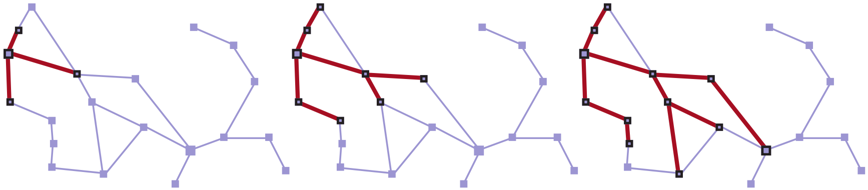
Root node is initial state. At each node we can **expand** the node, which grows the tree, by taking actions (adding edges) that lead to successor states (generate successor/child nodes).



KENNESAW STATE UNIVERSITY

Search Tree Expansion

Here is a search tree being imposed on the Romania state space graph by a search algorithm.

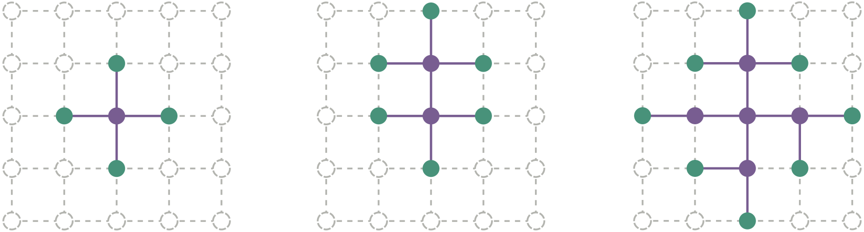


Essence of search:

- Choose a child node to consider next.
- Put aside other nodes for later.

Separation Property of Graph Search

The **frontier** separates the interior region of expanded nodes from the exterior region of unexpanded nodes.



(a) (b) (c)

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

The `yield` statement

A function containing a `yield` statement is a **generator**. Use a generator to turn a data generating process into an iterator.

```
In [36]: def by_twos(start: int, end: int):
    ...:     x = start
    ...:     while x < end:
    ...:         yield x
    ...:         x += 2
    ...:

In [37]: by_twos(1, 9)
Out[37]: <generator object by_twos at 0x109010ee0>

In [38]: list(Out[37])
Out[38]: [1, 3, 5, 7]

In [39]: for x in by_twos(1, 10):
    ...:     print(f"{x=}")
    ...:
x=1
x=3
x=5
x=7
x=9
```

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 a **queue** with operations:

- `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.

Best-First Search Algorithm

$f(node)$ is an evaluation function, which imposes an ordering on the nodes in the priority queue.

```
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)
```

Redundant Paths

Repeated states

cycles

redundant paths

graph search

tree-like search

KENNESAW STATE UNIVERSITY

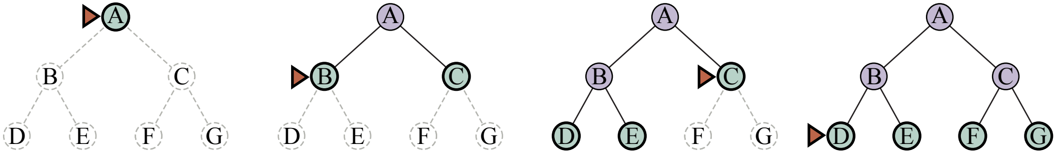
Measuring Problem-Solving Performance

- **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:

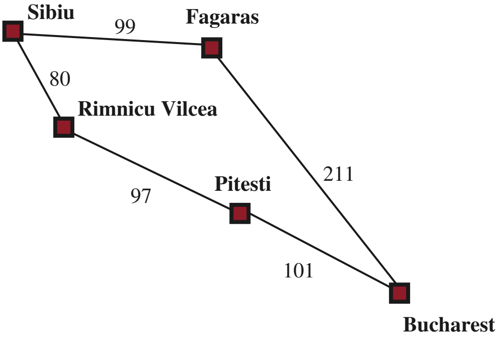
Breadth-First Search



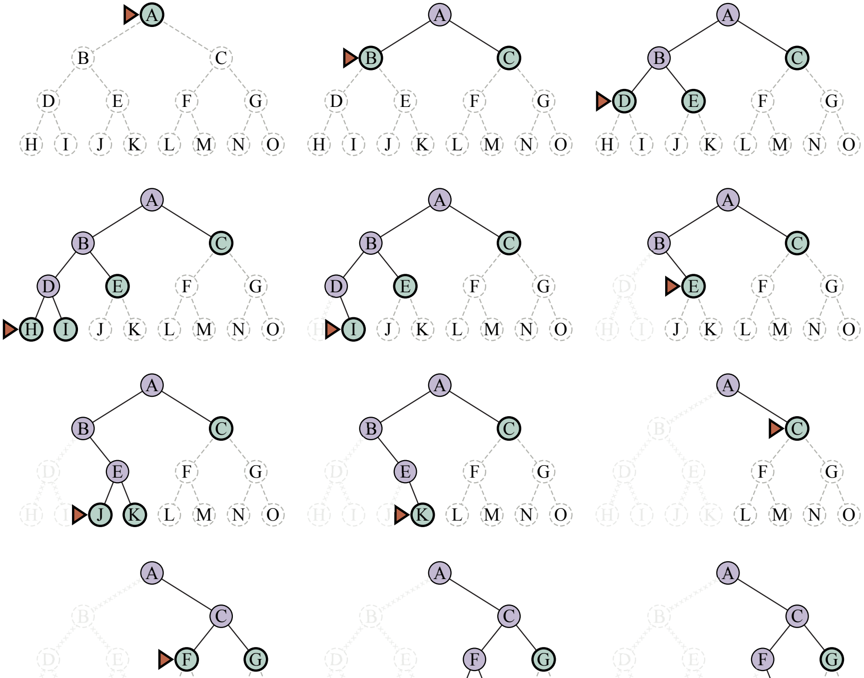
function BREADTH-FIRST-SEARCH(*problem*) **returns** a solution node or *failure*
  node ← NODE(*problem*.INITIAL)
  if *problem*.IS-GOAL(*node*.STATE) **then return** *node*
  frontier ← a FIFO queue, with *node* as an element
  reached ← {*problem*.INITIAL}
  while not IS-EMPTY(*frontier*) **do**
    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*

KENNESAW STATE UNIVERSITY

Dijkstra's Algorithm



Depth-First Search

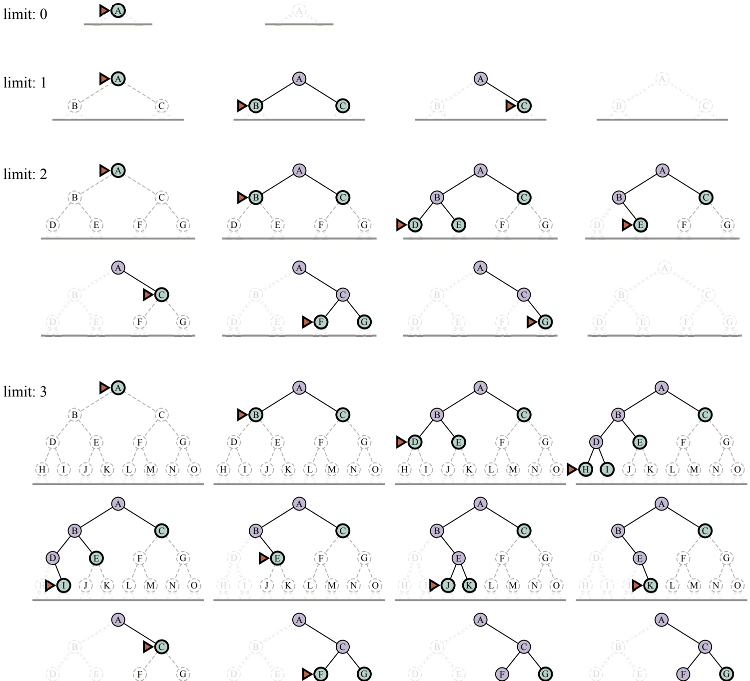


Depth-Limited Search and Iterative Deepening Search

function ITERATIVE-DEEPENING-SEARCH(*problem*) **returns** a solution node or *failure*
  for *depth* = 0 **to** ∞ **do**
    result ← DEPTH-LIMITED-SEARCH(*problem*, *depth*)
    if *result* $\neq$ *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
```

KENNESAW STATE UNIVERSITY

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	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$

KENNESAW STATE UNIVERSITY