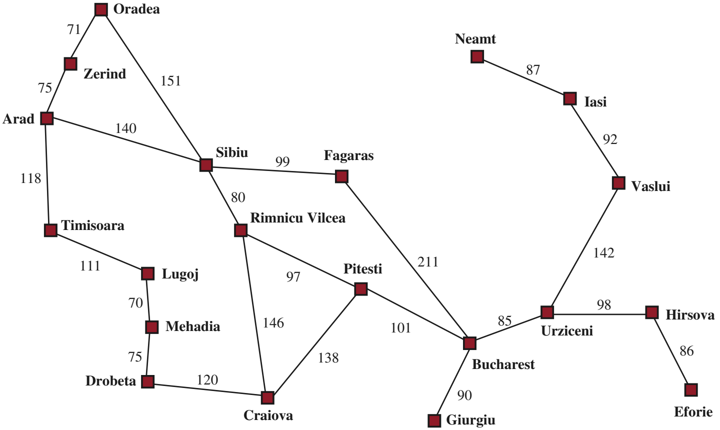
Problem Solving

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.

KENNESAW STATE UNIVERSITY

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.

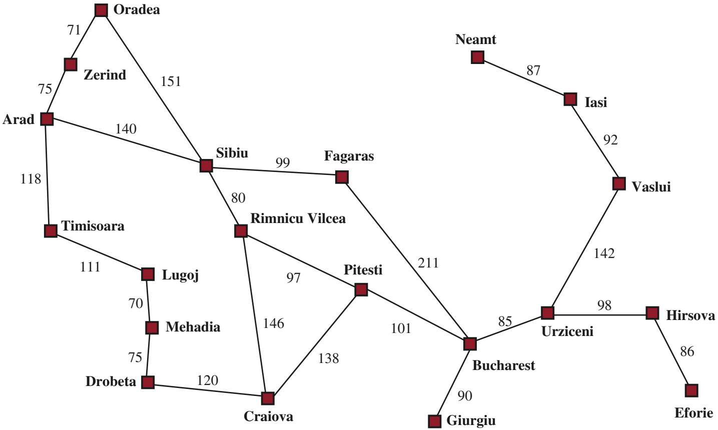
Search Problems and Solutions

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'.

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.

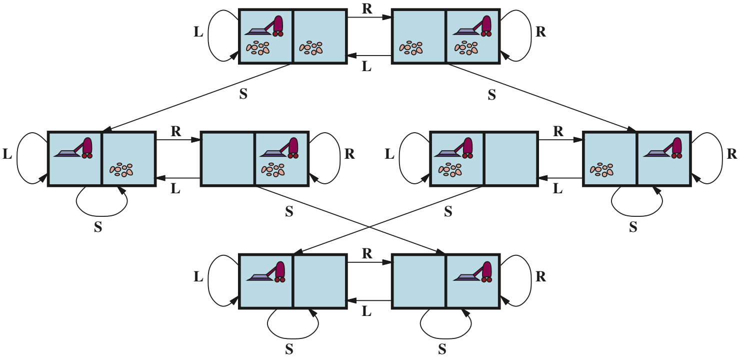


- How many paths are there from `Arad` to `Bucharest`?
- What is/are the solutions to the `Arad-to-Bucharest` problem (assume perfect information – fully observable, known dynamics, and deterministic actions)?

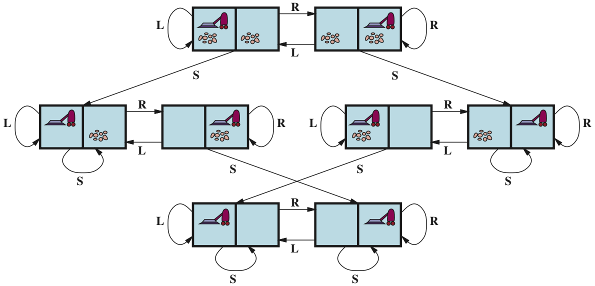
Example Problems

- **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

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

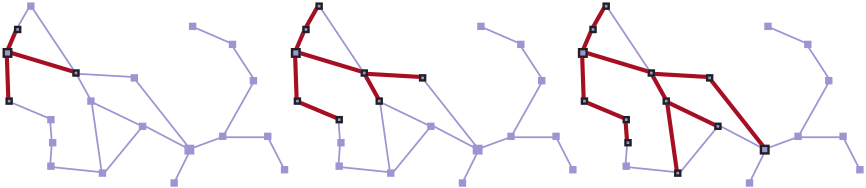
A **search algorithm** takes a search problem as input and returns a solution, or an indication of failure.

- 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

Don't confuse state space with search tree.

- State space is set of states, and actions that cause transitions between states.
- Search tree describes paths between these states, reaching towards the goal(s).
  - May be many nodes for a given state, but each path from root to node is unique.

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



KENNESAW STATE UNIVERSITY

Elements of Search Algorithms

Essence of search:

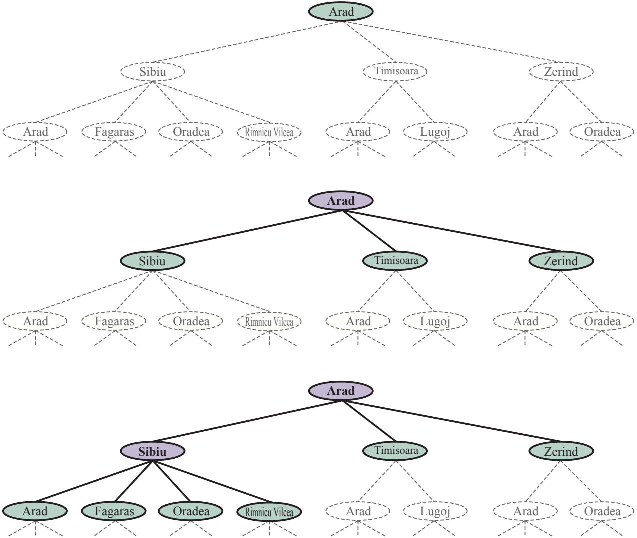
- Choose a child node to consider next. “Who’s first?”
- Put aside other nodes for later. “Who’s next?”

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). Search algorithms must keep track of:

- *Expanded* nodes. We test expanded nodes before dealing with frontier.
- *Frontier* nodes, which are generated but not yet expanded.
  - $\textit{Reached} = \textit{Expanded} + \textit{Frontier}$

KENNESAW STATE UNIVERSITY

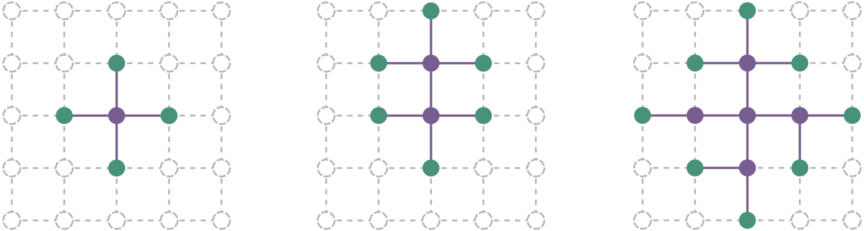
Search Progression



- *Expanded* nodes are lavender with bold letters.
- *Frontier* nodes are green with normal weight font.
- Nodes in dashed-line ovals are candidates for expansion.

Separation Property of Graph Search

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



(a) (b) (c)

Frontier is in green. Interior is lavender. Exterior is faint dashed.

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

Implementation Note: The `yield` statement

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

```
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

Best-first search is an abstract search algorithm. Name can be tricky to understand.

- *Best* way to pick the *first* node to consider next.
- We use a generalization of queues, called a *priority queue*, to store the *frontier*.
- An evaluation function, $f(node)$, imposes an ordering on the nodes in the priority queue.

> *The evaluation function considers the path to the node, not any property of the node itself. Remember, a solution to a search problem is characterized by the path from the root to the goal, not some characteristic of the goal.*

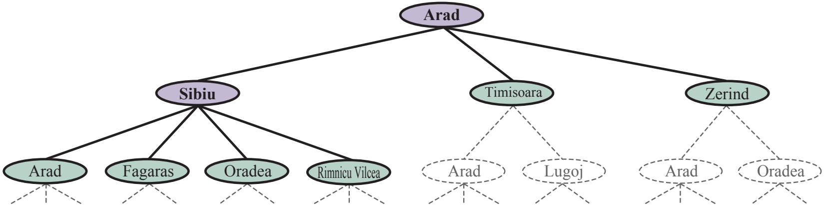
We'll now describe several uninformed search algorithms. I recommend you also look at their implementations in Python, which may be easier to follow.

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)
```

Redundant Paths



In the path from `Arad` to `Sibiu` to `Arad`,

- `Arad` is a **repeated state** and
- the path is a **cycle**, or **loopy path**.

Cycle special case of **redundant path**: multiple paths to the same state. Three approaches:

1. Remember reaches states, like best-first search. Best when reached states fits in memory.
2. Don't worry about repeated states. Works when repeated states rare or impossible.
   - **Graph search** checks for redundant paths, which occur in graphs in general.
   - **Tree-like search** does not check for redundant paths, since trees are acyclic graphs.
3. Only check for cycles, not other kinds of redundant paths.
   - E.g., search path in reverse

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?
  - Complete search algorithms must be **systematic**.
  - Easier to achieve for finite state spaces.
  - In an infinite state space with no solution, search won't terminate.
- **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?

For *explicit* graphs, like Romania, time and space complexity typically expressed in terms of number of vertices (state nodes), $|V|$, and number of edges, $|E|$ (state-action pairs, which generate $((s, a, s')$ triples).

For *implicit* state space graphs we characterize time and space complexity in terms of depth, d (number of actions in an optimal solution), and branching factor, b (number of successor nodes per node). For most of our discussions, we'll use this characterization.

Uninformed Search Strategies

Uninformed search strategies have no information about which actions are better for reaching a goal. In these cases we can only do systematic searches of the state space. We'll discuss

- Breadth-first search
- Uniform-Cost search (Djikstra's algorithm)
- Depth-first search
- Depth-limited searchand
- Iterative deepening search.
- Bidirectional search

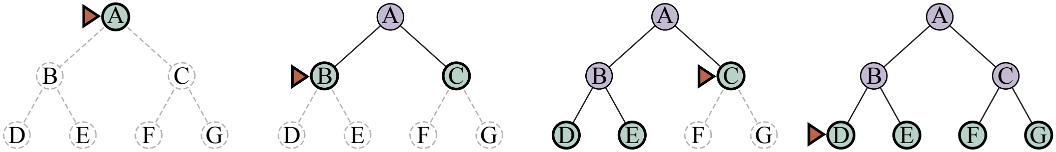
Breadth-First Search

- Good when path costs are uniform.
- Equivalent to best-first search where $f(node)$ is the depth of the node
- Guaranteed to find minimal number of actions because it evaluates depth d before generating depth $d + 1$.

But three optimizations afforded by the BFS algorithm and uniform path costs:

- FIFO queue instead of priority queue
- *Reached* is a set instead of a mapping $S \rightarrow Node$
  - With uniform path costs, as soon as BFS finds a node, it's the fastest way to it.
- **Early goal test –** as soon as we expand a node, we can test it.

BFS Algorithm



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*

Analysis of BFS

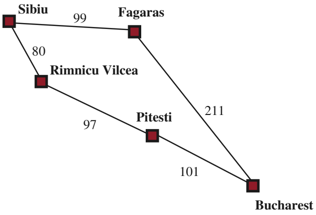
- Complete, because it generates all nodes at each depth.
- Time complexity: at each level, b nodes for each b predecessors, so
  - $1 + b + b^2 + b^3 + \cdots + b^d = O(b^d)$
- Space complexity: $O(b^d)$ because all nodes are stored while the search proceeds.

Uninformed search is not appropriate for exponential complexity problems except for smallest instances. Assuming your computer can process 1 million nodes per second and store each node in 1 Kb,

- For a problem with $b = 10$ and $d = 10$, how long will it take search and how much space will be required?
- Same problem, but with $d = 14$?

Uniform-Cost Search (Dijkstra's Algorithm)

BFS where the best-first $f(node)$ is the path cost to the current node.




```
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)

function UNIFORM-COST-SEARCH(problem) returns a solution node, or failure
  return BEST-FIRST-SEARCH(problem, PATH-COST)
```

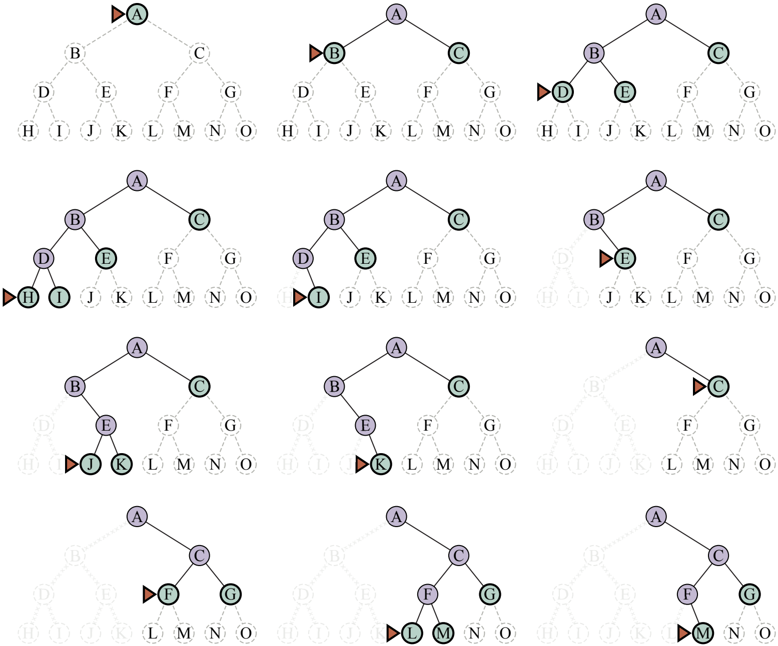
Analysis of Uniform-Cost Search

Let C^* be the cost of the optimal solution and $\epsilon > 0$ be a lower bound on the cost of each action.

- Complete, like BFS
- Cost-optimal, because a solution will be at least as low cost as any other in the frontier.
- Time and space complexity are $O(b^{1+\lfloor \frac{C^*}{\epsilon} \rfloor})$.
  - Since lower cost paths are always explored first, even when a higher cost path might be the one to lead to an optimal solution, can be worse than BFS.
  - If all action costs equal, then it's like BFS, $O(b^{1+d})$.

KENNESAW STATE UNIVERSITY

Depth-First Search - FIFO Frontier



Analysis of DFS

- Not cost-optimal – returns first solution it finds
- For state space that are finite trees:
  - Complete
  - Time $O(n)$ where n is number of states
  - Space: $O(bm)$, where b is branching factor and m is max depth of tree.
- For (acyclic) graph state spaces, may expand same state via multiple paths.
  - For cyclic graph state spaces, need to check for cycles to prevent infinite loops.
- For infinite state spaces, not complete – may get stuck in an infinite subtree.

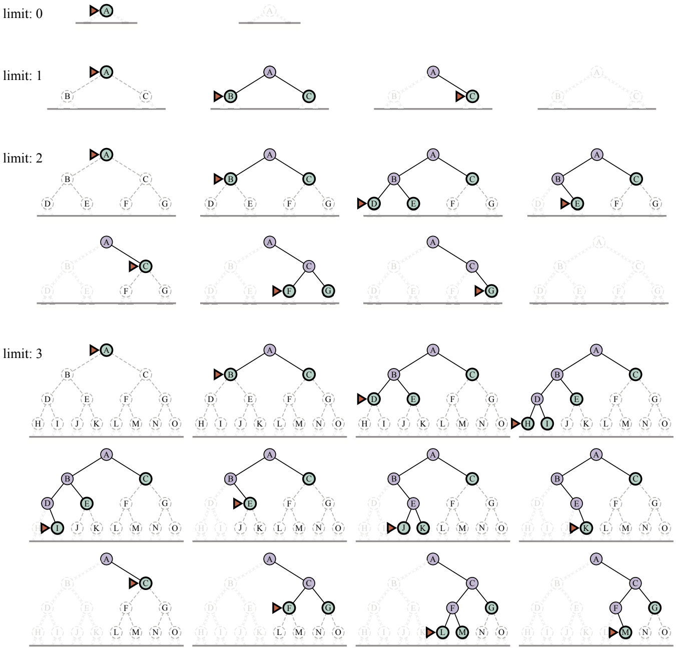
Why bother with DFS at all? **Memory efficiency**

Depth-Limited Search

```
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
```

- If limit, ℓ, too small, won't find goal.
- To guarantee completeness, choose $\ell \geq diameter$
  - Diameter of a state space graph: maximum number of actions necessary to transition from any state to any other state.

Iterative Deepening Search



Analysis of Iterative Deepening Search

- Cost-optimal for state spaces where all actions have same cost.
- For state space that are finite trees, where b is branching factor and m is max depth of tree:
  - Complete for finite acyclic spaces, or finite cyclic spaces with cycle checking
  - Space: $O(bd)$ if there is a solution, $O(bm)$ if no solution,
  - Time $O(b^d)$ if there is a solution, $O(b^m)$ if no solution.
    - $N(IDS) = (d)b^1 + (d-1)b^2 + \cdots + b^d$
- For (acyclic) graph state spaces, may expand same state via multiple paths.
  - For cyclic graph state spaces, need to check for cycles to prevent infinite loops.
- For infinite state spaces, not complete – may get stuck in an infinite subtree.

In general, iterative deepening search is the preferred uninformed search method when the search state space is larger than can fit in memory and The depth of the solution is not known.

KENNESAW STATE UNIVERSITY

Bidirectional Best-First Search

function BIBF-SEARCH($problem_F$, f_F, $problem_B$, f_B) **returns** a solution node, or *failure*
  $node_F \leftarrow$ NODE($problem_F$.INITIAL) *// Node for a start state*
  $node_B \leftarrow$ NODE($problem_B$.INITIAL) *// Node for a goal state*
  $frontier_F \leftarrow$ a priority queue ordered by f_F, with $node_F$ as an element
  $frontier_B \leftarrow$ a priority queue ordered by f_B, with $node_B$ as an element
  $reached_F \leftarrow$ a lookup table, with one key $node_F$.STATE and value $node_F$
  $reached_B \leftarrow$ a lookup table, with one key $node_B$.STATE and value $node_B$
  $solution \leftarrow failure$
  while not TERMINATED($solution$, $frontier_F$, $frontier_B$) **do**
    if f_F(TOP($frontier_F$)) $<$ f_B(TOP($frontier_B$)) **then**
      $solution \leftarrow$ PROCEED(F, $problem_F$, $frontier_F$, $reached_F$, $reached_B$, $solution$)
    else $solution \leftarrow$ 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.
  $node \leftarrow$ POP($frontier$)
  for each *child* **in** EXPAND(*problem*, *node*) **do**
    $s \leftarrow child$.STATE
    if s not in *reached* **or** PATH-COST(*child*) $<$ PATH-COST($reached[s]$) **then**
      $reached[s] \leftarrow child$
      add *child* to *frontier*
      if s is in $reached_2$ **then**
        $solution_2 \leftarrow$ JOIN-NODES(*dir*, *child*, $reached_2$[s]))
        if PATH-COST($solution_2$) $<$ PATH-COST(*solution*) **then**
          $solution \leftarrow solution_2$
  return *solution*

Motivation: $b^{\frac{d}{2}} + b^{\frac{d}{2}} \ll b^d$.

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

Notes:

- 1: complete if b is finite, and the state space either has a solution or is finite
- 2: complete if all action costs are $\geq \epsilon > 0$
- 3: cost-optimal if action costs are all identical
- 4: if both directions are breadth-first or uniform-cost

Informed (Heuristic) Search Strategies

- Use domain-specific hints about “distance” from goals
- Hints encapsulated in a **heuristic function**, $h(node)$:
  - $h(node) =$ estimated cost of cheapest path from $node$ to a goal state
  - h is really a function of $state$, not $node$. We use $h(node)$ to be consistent with $f(node)$ in best-first search, and path cost, $g(node)$.
  - Book uses $f(n)$, $g(n)$ and $h(n)$. I use $node$ instead of n to clearly distinguish from n as an index in problem size, N.

Example Heuristic for Romania, h_{SLD}:

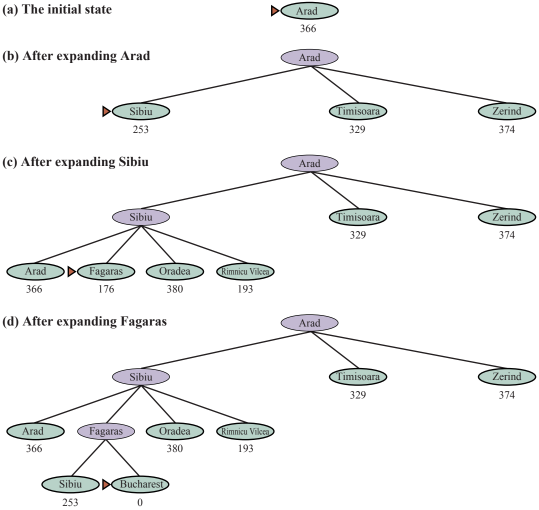
Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Drobeta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

Straight line distances to Bucharest from each of the cities in Romania.

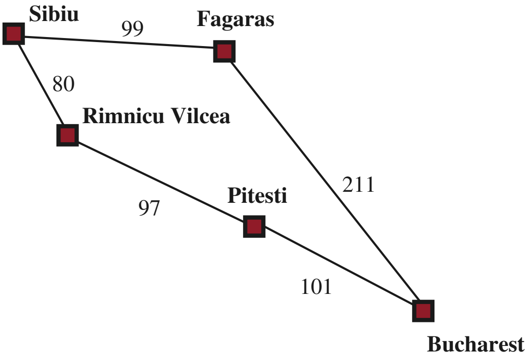
KENNESAW STATE UNIVERSITY

Greedy Best-First Search

- Recall that best-first search uses a priority queue for its frontier, ordered by $f(node)$
- Greedy best-first search uses $f(node) = h(node)$
- Greediness: get as close to the goal as possible in each step



Optimality of Greedy Best-First Search



- Greedy best-first search returns the path via Sibiu and Fagaras to Bucharest.
- The path through Rimnicu Vilcea and Pitesti is 32 miles shorter.

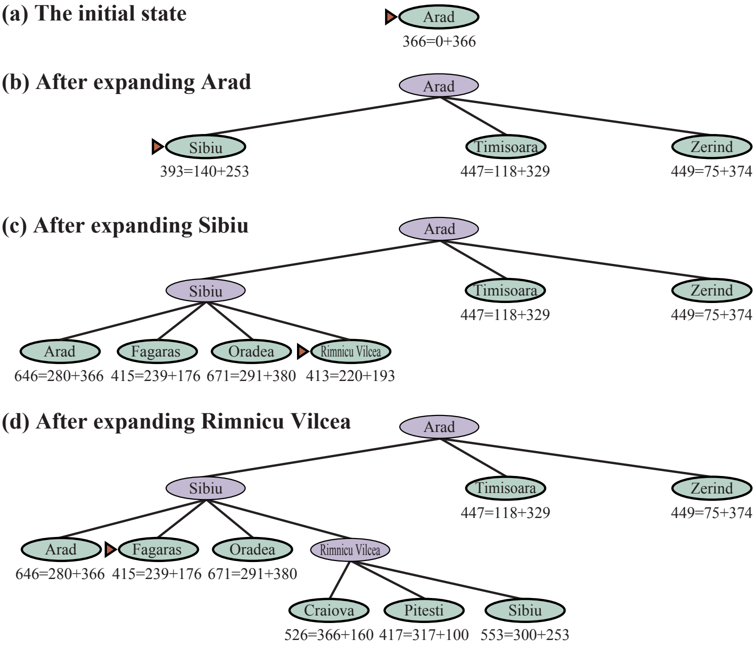
A^* Search

$$f(node) = g(node) + h(node)$$

- Complete
- Optimal with an admissible heuristic
- Relatively efficient, but can generate exponential number of nodes for some problems.

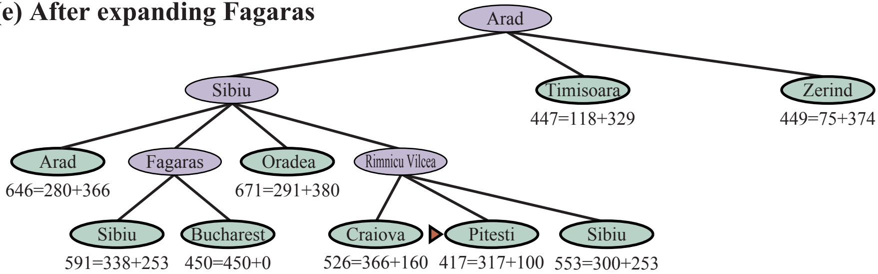
Heavily dependent on quality of heuristic function.

A^* Progress Part 1

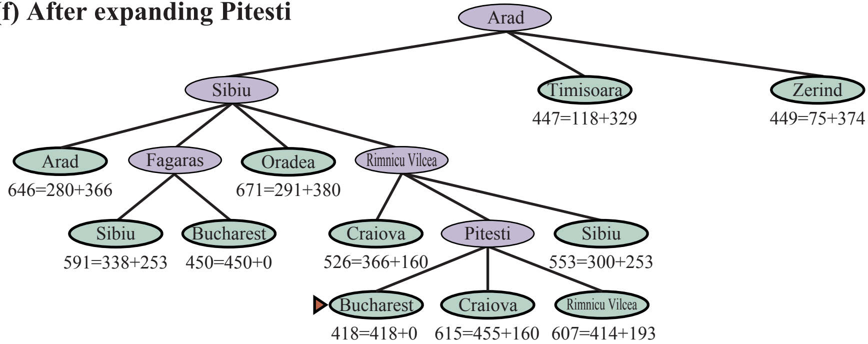


A^* Progress Part 2

(e) After expanding Fagaras

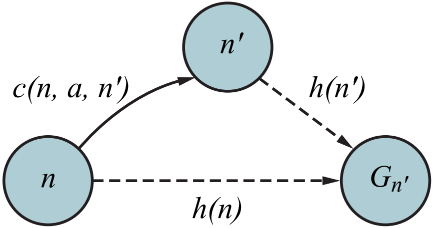


(f) After expanding Pitesti



Admissibility and Consistency

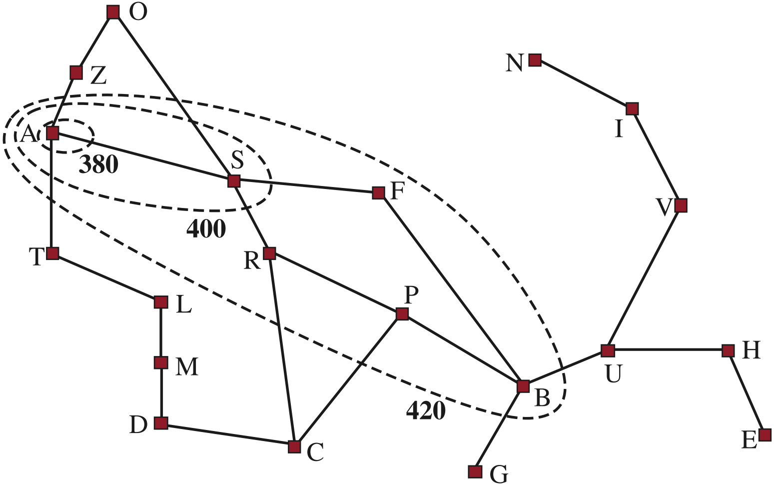
- An **admissible** heuristic never overestimates the cost to reach a goal.
- A **consistent** heuristic is a kind of local admissibility: for every node $node$ and successor $node'$ generated by action a: $h(node) \leq c(node, a, node') + h(n')$. This is a form of **triangle inequality**.



- Admissibility is required to guarantee cost-optimality in A^*.
- Consistency improves performance by guaranteeing that the first time we reach a node, it is on the optimal path – so we don't re-evaluate multiple paths to the same node.

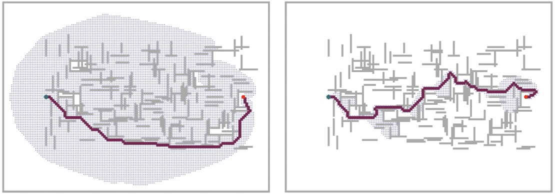
Search Contours

- In a topographical map, countours indicate a constant elevation
- In a search contour of a state space, a contour indicates an upper bound on path cost in a region
  - In the 400 countour, each node has $f(node) = g(node) + h(node) \leq 400$.



Satisficing Search: A^* vs Weighted A^*

- Detour index: multiplier applied to straight-line distance to account for curvature of roads. E.g., detour index of 1.3 means a road connecting locations 10 miles apart would be estimated as 13 miles long.
- Weighted A^* search: apply a wieght, like detour index, to $h(node)$
  - $f(n) = g(n) + w \cdot h(n)$, for some $w > 1$
- Results in inadmissible heuristic (overestimates), but can improve search speed.

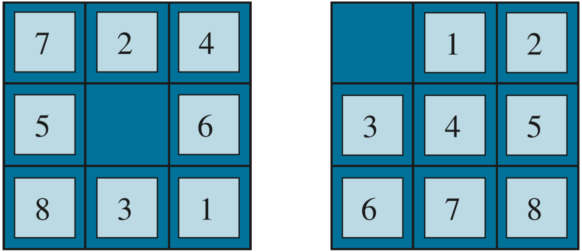


(a) (b)

(a) an A^* search and (b) a weighted A^* search with weight $w = 2$.

- The gray bars are obstacles, the purple line is the path from the green start to red goal, and the small dots are states that were reached by each search.
- On this particular problem, weighted A^* explores 7 times fewer states and finds a path that is 5% more costly.

Heuristic Functions



Start State Goal State

- Misplaced tiles, $h_1 = 8$.
- Manhattan distance, $h_2 = 3 + 1 + 2 + 2 + 2 + 3 + 3 + 2 = 18$

True solution cost is 26, so neither heristic overestimates.

Heuristic Accuracy and Performance

- Effective branching factor: for N nodes, branching factor of uniform tree of depth d that would contain $N+1$ nodes. Want close to 1.

	Search Cost (nodes generated)			Effective Branching Factor		
d	BFS	$A^*(h_1)$	$A^*(h_2)$	BFS	$A^*(h_1)$	$A^*(h_2)$
6	128	24	19	2.01	1.42	1.34
8	368	48	31	1.91	1.40	1.30
10	1033	116	48	1.85	1.43	1.27
12	2672	279	84	1.80	1.45	1.28
14	6783	678	174	1.77	1.47	1.31
16	17270	1683	364	1.74	1.48	1.32
18	41558	4102	751	1.72	1.49	1.34
20	91493	9905	1318	1.69	1.50	1.34
22	175921	22955	2548	1.66	1.50	1.34
24	290082	53039	5733	1.62	1.50	1.36
26	395355	110372	10080	1.58	1.50	1.35
28	463234	202565	22055	1.53	1.49	1.36

- h_2 dominates h_1 because for any $node$, $h_2(node) \geq h_1(node)$
- We want a heuristic that underestimates, but by as little as possible.

KENNESAW STATE UNIVERSITY

Designing Heuristic Functions

- Relaxing the problem definition
- Storing precomputed solution costs for subproblems in a pattern database
- Defining landmarks
- Learning from experience

Heuristic functions are a way to inject domain knowledge into a problem solver.