



CSC460/660

Artificial Intelligence: Principles & Techniques

Faisal N. Abu-Khzam

Department of Computer Science & Mathematics

Lebanese American University

Beirut, Lebanon

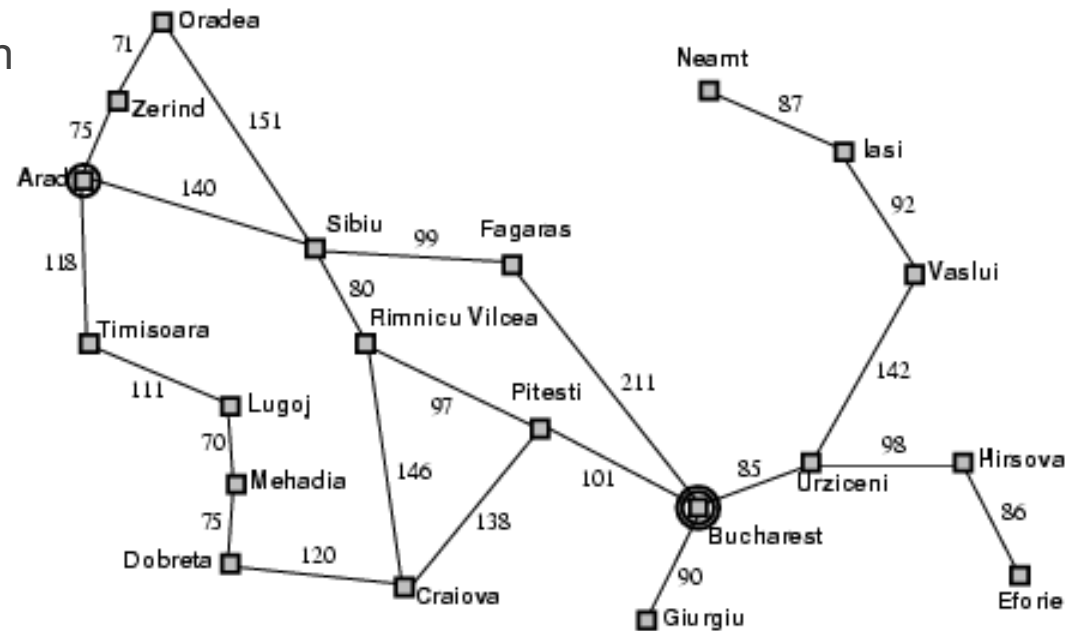


Overview

- Problem-solving agents
- Uninformed search strategies
- Informed search and heuristics
- Greedy best-first search
- A* search

A task environment: route planning

- Performance: reach the destination with minimum travel cost
- Environment: road network and travel conditions
- Actuators: movement or route recommendations
- Sensors: location and available road information



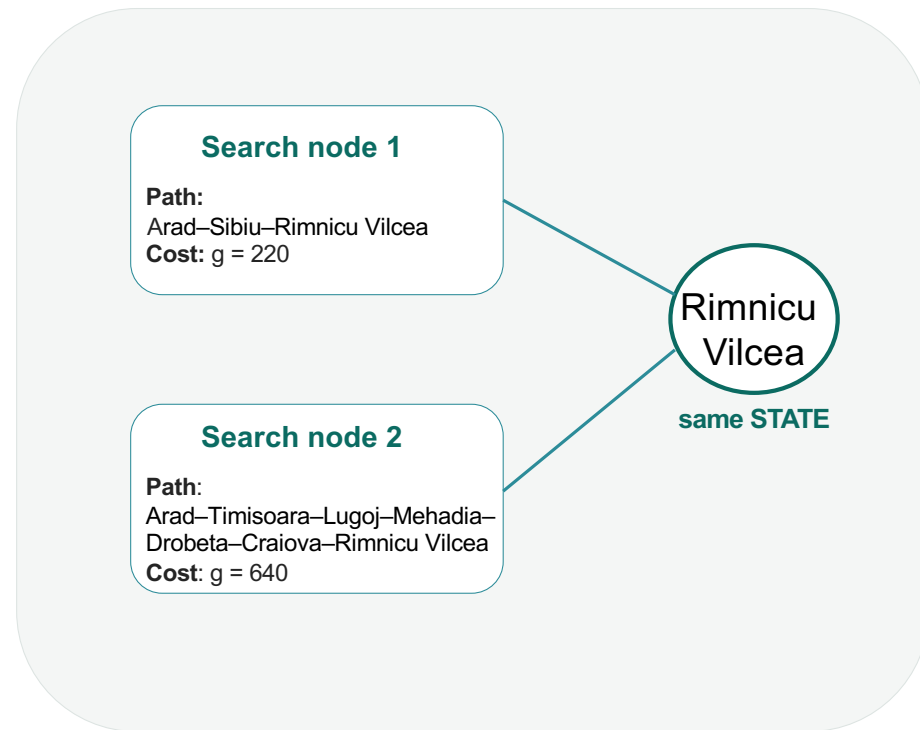
Arad → Bucharest

Problem formulation

- Initial state: where the agent starts
- Actions and transition model: legal moves and their results
- Goal test: whether a state satisfies the objective
- Path cost: the sum of the chosen action costs

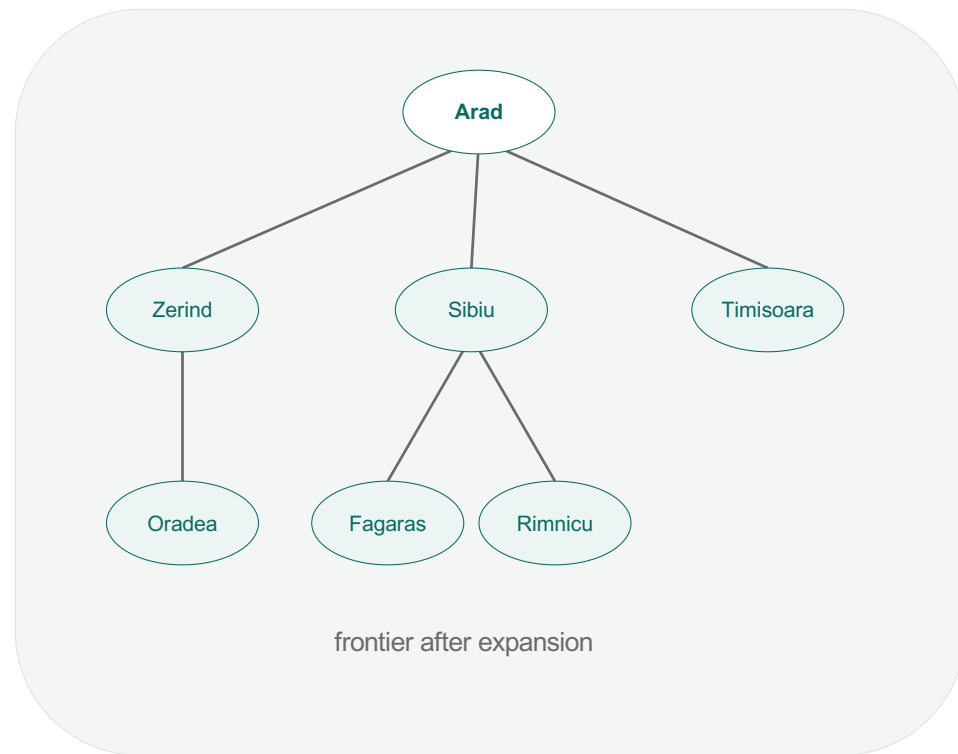
States and search nodes

- A state describes a configuration of the problem
- A search node also records a parent, action, depth and path cost g
- *Arad-Sibiu-Rimnicu Vilcea* and *Arad-Timisoara-Lugoj-Mehadia-Drobeta-Craiova-Rimnicu Vilcea* reach the same state *Rimnicu Vilcea*
- They create different tree nodes with costs 220 and 640



The search framework

- The frontier stores generated nodes waiting for expansion
- Select a node according to the search strategy
- Check for a goal, then generate legal successors
- Track repeated states and reconstruct the path through parents



How we evaluate search

- Quality measures:
 - **Complete**: finds a solution whenever one exists
 - **Optimal**: returns a solution of minimum path cost
- Efficiency measures
 - **Time**: number of nodes generated or expanded
 - **Space**: maximum number of stored nodes
- Depend on the problem and the implementation

Search Strategies

- A search strategy is defined by picking the **order of node expansion**
- Strategies are evaluated along the following dimensions:
 - completeness: does it always find a solution if one exists?
 - time complexity: number of nodes generated
 - space complexity: maximum number of nodes in memory
 - optimality: does it always find a least-cost solution?
- Time and space complexity are measured in terms of
 - b : maximum branching factor of the search tree
 - d : depth of the least-cost solution
 - m : maximum depth of the state space (may be ∞)

Uninformed Search Strategies

Use only the information available in the problem definition

- Breadth-first search
 - Depth-first search
 - Depth-limited search
 - Iterative deepening search
 - Uniform-cost search
- 
- A decorative horizontal bar at the bottom of the slide, consisting of a thin teal line above a thicker teal rectangle.

Worked example: a weighted graph

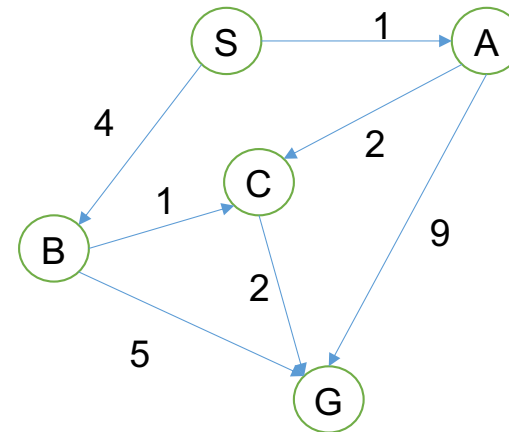
Directed edges and costs:

- S–A: 1
- S–B: 4
- A–C: 2
- A–G: 9
- B–C: 1
- B–G: 5
- C–G: 2
- Start S. Goal G. No other edges...
- Candidate routes?

Worked example: a weighted graph

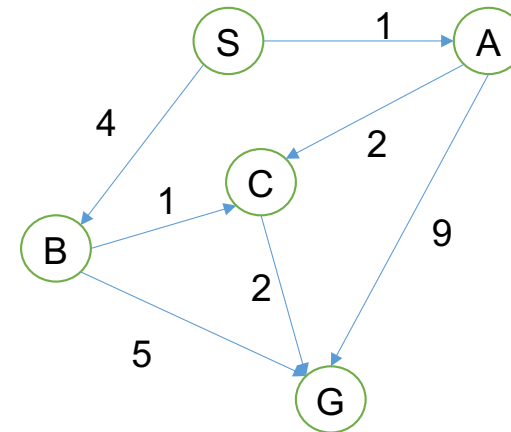
Directed edges and costs:

- S–A: 1
- S–B: 4
- A–C: 2
- A–G: 9
- B–C: 1
- B–G: 5
- C–G: 2
- Start S. Goal G. No other edges.
- Candidate routes: S–A–G costs 10. S–A–C–G costs 5.



Breadth-first search

- Expand the shallowest frontier node using a FIFO queue
- For the example: remove S, A, B, C, G
- The first G represents S–A–G, with cost 10
- BFS minimizes the number of actions

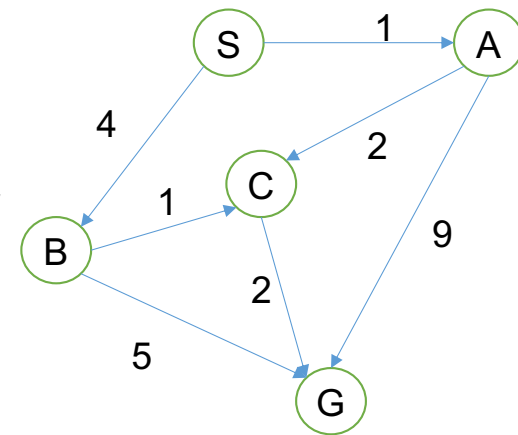


BFS: guarantees and memory

- Complete with finite branching and a finite-depth goal
- Optimal when all step costs are equal and positive
- Frontier storage grows exponentially with solution depth
- Unequal costs can make the shallowest solution expensive

Depth-first search

- Expand the deepest node using a stack or recursion
- Alphabetical branch choice gives S, A, C, G
- This example happens to return cost 5
- A different successor order can change the result completely



Depth limits and iterative deepening

- Depth-limited search stops at a chosen depth limit
- A cutoff differs from proving that no solution exists
- Iterative deepening repeats limits 0, 1, 2, ...
- It combines shallowest-goal search with modest memory

Uniform-cost search

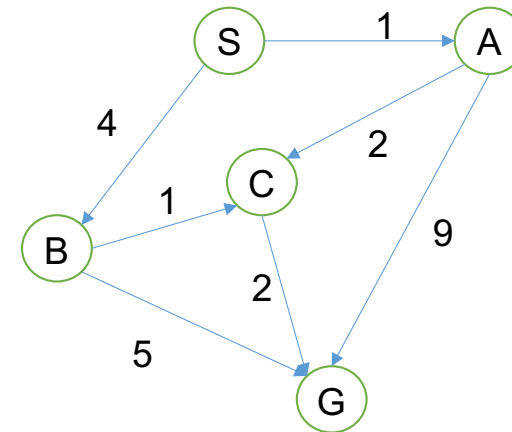
- Expand the frontier node with the smallest path cost g
 - Use a priority queue and keep the best known g for each state
 - Update a state when a cheaper path reaches it
 - Accept a goal when it is removed from the priority queue
-
- Why did they call it “uniform-cost search”? What is uniform about it?

Uniform-cost search

- Why did they call it “uniform-cost search”? What is uniform about it?
- Subtle reason: search proceeds in **uniformly increasing path-cost contours**.
- UCS always expands the frontier node having the smallest accumulated cost $g(n)$. So it explores the state space according to **cost**, rather than according to depth.
- Think of the contrast:
 - **BFS**: expands all nodes at depth 0, then depth 1, then depth 2, ...
 - **UCS**: expands nodes in increasing g -cost, regardless of how many edges were used.
- One way to think about “uniform”
 - Search expands outward from the start as a uniform cost wavefront: everything reachable cheaply is explored before anything more expensive.
- BFS is the special case of uniform-cost search when all step costs are equal.

Uniform-cost search: trace

- After S: A(1), B(4)
- After A: C(3), B(4), G(10)
- After C: B(4), G(5)
- After B: G(5). The alternatives cost 5 to C and 9 to G.
- Remove G(5): return S–A–C–G

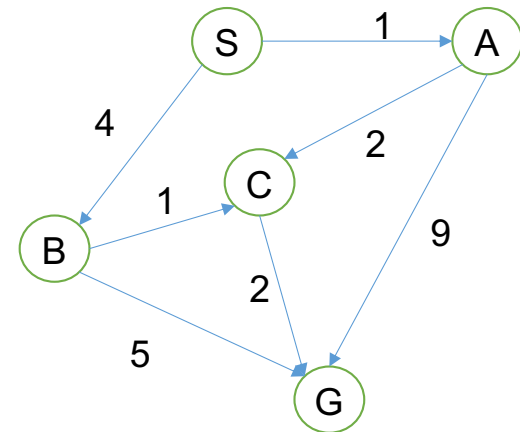


Uninformed search: choice of strategy

- BFS: shallowest solutions when each action has the same cost
- DFS: limited memory, with no general optimality guarantee
- Iterative deepening: unknown depth and equal action costs
- UCS: minimum-cost paths with suitable nonnegative costs

Discussion: search without heuristics

- Suppose the successors of S are explored in the order B, A , and the successors of B in the order G, C .
- What path does DFS return?
- Would BFS necessarily return a cheapest path?



- Why can UCS discard a more expensive path to a state already reached more cheaply?

Exercise: Missionaries and Cannibals

- **Problem:**

Three missionaries and three cannibals must cross a river using a boat that carries at most two people.

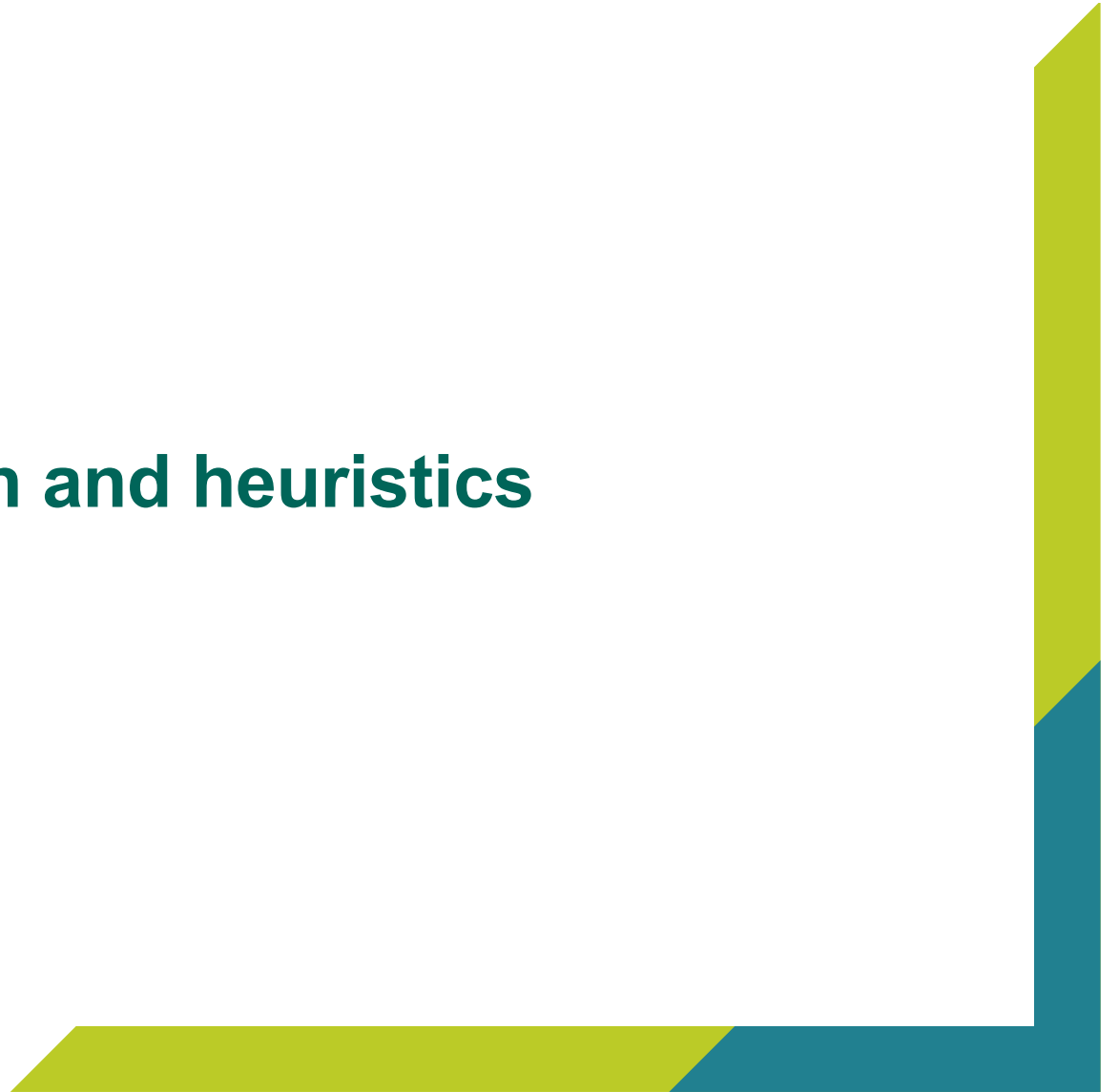
- **Rules**

- The boat cannot cross empty.
- On either bank, if missionaries are present, they cannot be outnumbered by cannibals.
- Everyone starts on the left bank.

- **Questions**

- How should we represent a state?
 - What are the possible actions?
 - Which states are legal?
 - Which uninformed search strategy should we use?
 - Find a shortest sequence of crossings.
- 

Informed search and heuristics



Why informed search?

- Uninformed search uses only the problem definition.
- UCS knows the cost already incurred, $g(n)$.
- But it has **no idea which frontier node is closer to the goal**.
- Can additional knowledge guide the search toward promising states?

Heuristic function

- A heuristic $h(n)$ estimates the cost of the cheapest path from state/node n to a goal.
- $h(n)$ uses **problem-specific knowledge** not contained in the basic search strategy.
- $h(\text{goal}) = 0$.
- It is an **estimate**, not necessarily the true remaining cost.
- Example: for route finding, **straight-line distance to the destination**.

Greedy Best-First Search

- Expand the frontier node with the smallest $h(n)$.
- Evaluation function: $f(n) = h(n)$.
- It chooses the node that **appears closest to the goal**.
- It ignores the cost already incurred, $g(n)$.
- Therefore it can be fast, but is **not generally optimal**.

From UCS and Greedy to A*

- **UCS:**
“How much have I spent?” $\rightarrow g(n)$
- **Greedy:**
“How much seems to remain?” $\rightarrow h(n)$
- **Why not use both?**

A* Search

- Combines the actual cost so far with the estimated cost remaining
- Evaluation function: $f(n) = g(n) + h(n)$
- $g(n)$: actual path cost from the start to n
- $h(n)$: estimated cost from n to a goal
- $f(n)$: estimated total cost of a solution through n
- Expand the frontier node with the smallest $f(n)$

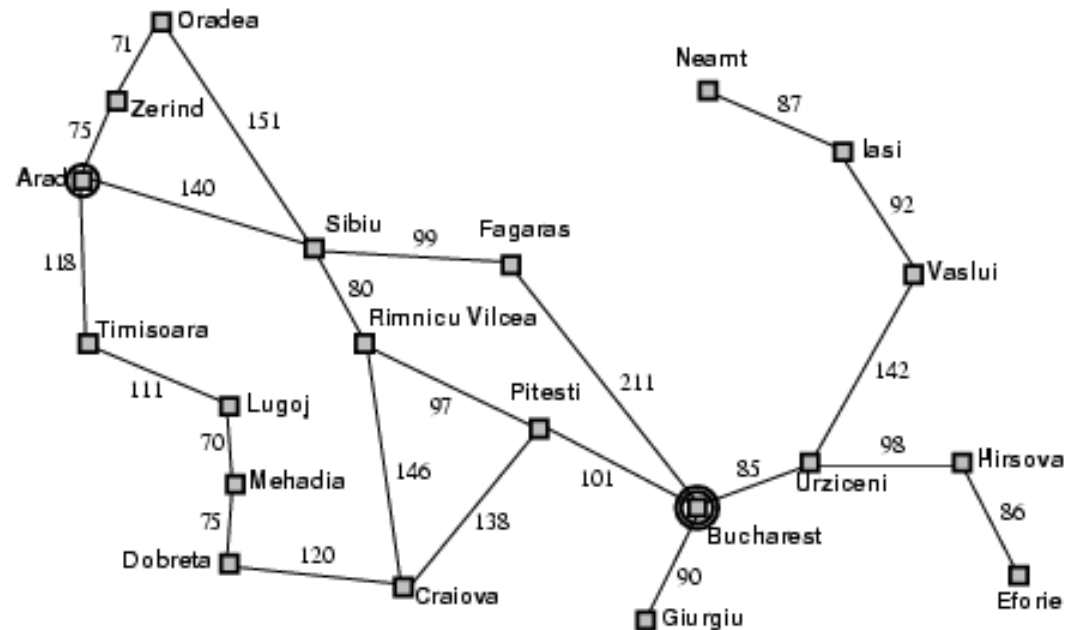
Romania again: a heuristic

Let $h(n)$ be the straight-line distance from city n to Bucharest.

- Examples:

- $h(\text{Arad}) = 366$
- $h(\text{Sibiu}) = 253$
- $h(\text{Fagaras}) = 176$
- $h(\text{Rimnicu Vilcea}) = 193$
- $h(\text{Bucharest}) = 0$

- Why might this be a useful estimate of the remaining road distance?



Greedy Best-First Search: Romania

Start: **Arad**; Goal: **Bucharest**

After expanding Arad:

- Zerind: $h = 374$
- Sibiu: $h = 253$
- Timisoara: $h = 329$

Greedy chooses the node with the smallest h :

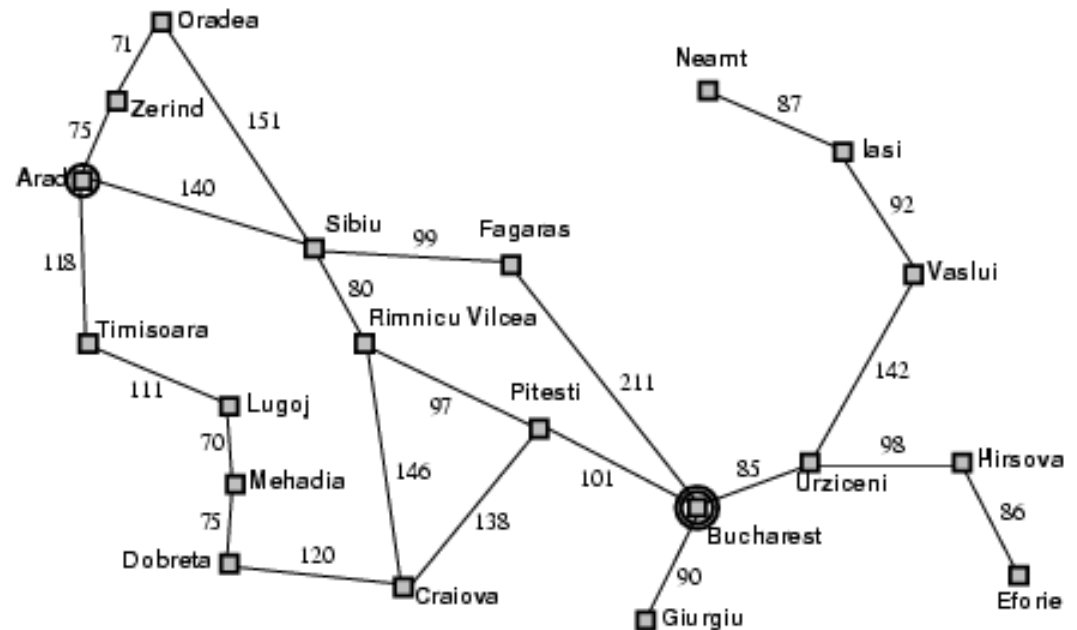
- **Arad → Sibiu → Fagaras → Bucharest**

Path cost:

- $140 + 99 + 211 = 450$

Is this the cheapest route?

- No. A cheaper route has cost **418**.



A* Search: Romania

Node n	$g(n)$	$h(n)$	$f(n) = g(n) + h(n)$
Zerind	75	374	449
Sibiu	140	253	393
Timisoara	118	329	447

- A* uses $f(n) = g(n) + h(n)$ to decide which node to expand next. Therefore it expands **Sibiu** next.
- Continuing the search gives: **Arad** → **Sibiu** → **Rimnicu Vilcea** → **Pitesti** → **Bucharest**
- Path cost: **140 + 80 + 97 + 101 = 418**
- Compare:
Greedy: 450 A*: 418

Tree search vs. graph search

- Each node represents a **path** from the initial state. The same state may therefore appear in **several different search nodes**.
- **Tree search:** treats these search nodes separately and does not detect repeated states.
 - Two different paths produce two different search nodes, even if both nodes contain the same underlying state. It does **not** merge them.
- **Graph search:** keeps track of reached states and avoids or updates repeated states.
 - Underlying **state space is a graph**. If two search nodes represent the same state, it recognizes the repetition and can discard/update accordingly.

Tree search vs. graph search

- **Tree search:** treats these search nodes separately and does not detect repeated states.
 - Two different paths produce two different search nodes, even if both nodes contain the same underlying state. It does **not** merge them.
- **Graph search:** keeps track of reached states and avoids or updates repeated states.
 - Underlying **state space is a graph**. If two search nodes represent the same state, it recognizes the repetition and can discard/update accordingly.

But with A*, a question arises:

- What if we expand a state and later discover a better path to it?
- The properties of the heuristic determine whether this can cause a problem.

When is A* Optimal?

- An important property of a heuristic is **admissibility**.
- Let $h^*(n)$ be the true cost of the cheapest path from n to a goal.
- A heuristic is **admissible** if it never overestimates the true remaining cost:

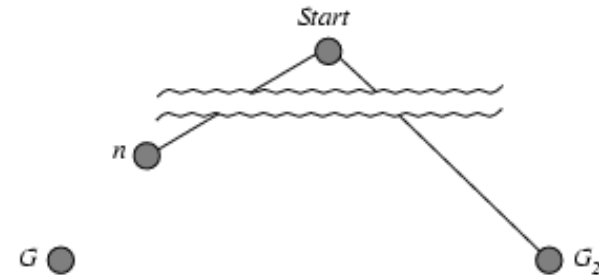
$$0 \leq h(n) \leq h^*(n)$$

- Straight-line distance is admissible for the Romania example:
 - a road route cannot be shorter than the straight-line distance between its endpoints.
- With an admissible heuristic, A* **tree search** is optimal.

Optimality of A* (proof)

- Suppose some suboptimal goal G_2 has been generated and is in the fringe. Let n be an unexpanded node in the fringe such that n is on a shortest path to an optimal goal G .

- $f(G_2) = g(G_2)$ since $h(G_2) = 0$
- $g(G_2) > g(G)$ since G_2 is suboptimal
- $f(G) = g(G)$ since $h(G) = 0$
- $f(G_2) > f(G)$ from above



A subtle problem: repeated states

- Recall: the **same state can be reached by different paths**.
- A search algorithm may reach a state, expand it, and later discover a cheaper path to that same state.
- What should A* do then?
- Can we impose a condition on h that prevents this from causing trouble?

Consistent heuristics

- A heuristic function h is consistent if, for every transition $n \rightarrow n'$ with cost $c(n, n')$:

$$h(n) \leq c(n, n') + h(n')$$

- The estimate from n cannot exceed: **cost of reaching n' + estimate from n'**
- This is a form of the **triangle inequality**.
- Consequently, along any path:

$$f(n') = g(n') + h(n') \geq g(n) + h(n) = f(n)$$

- **So f -values never decrease along a path.**
- **Note: every consistent heuristic is admissible** (assuming $h(G) = 0$).

Why does consistency imply admissibility?

- Suppose $n = n_0, n_1, \dots, n_k = G$ is a cheapest path from n to a goal.
- Consistency says on every edge:

$$h(n_i) \leq c(n_i, n_{i+1}) + h(n_{i+1}).$$

- Apply it repeatedly:

$$\begin{aligned} h(n) &\leq c(n, n_1) + h(n_1) \\ &\leq c(n, n_1) + c(n_1, n_2) + h(n_2) \\ &\leq \dots \\ &\leq c(n, n_1) + \dots + c(n_{k-1}, G) + h(G) \end{aligned}$$

- Since $h(G) = 0$, we get: $h(n) \leq \text{cost of a cheapest path from } n \text{ to } G = h^*(n)$.
- Therefore h never overestimates:

consistent $\Rightarrow$ admissible

Why consistency matters for A*

- With a consistent heuristic, $f(n)$ never decreases along a path.
- Therefore, when A* removes a node n from the priority queue, the cheapest path to its state has already been found.
 - The state does not need to be reopened later.
- When a goal G is removed from the priority queue:

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

because $h(G) = 0$.

- Any frontier node on an optimal path has an f -value no greater than the cost of the optimal solution.
 - *Therefore, A* graph search with a consistent heuristic is optimal.*
- 

How efficient is A*

- A* can dramatically reduce the number of expanded nodes when h is informative.
- But in the worst case, A* can still require **exponential time and space**.
- Its major practical limitation is often **memory**: it may keep a very large frontier and reached set.
- A better heuristic may reduce the number of expanded nodes, but computing the heuristic itself also has a cost.

UCS, Dijkstra and A*

- **Dijkstra's algorithm**

- Computes shortest-path distances from a source in a weighted graph with nonnegative edge costs.
- Traditionally continues until all required shortest paths are determined.

- **Uniform-cost search**

- Uses the same fundamental idea: expand the state with minimum $g(n)$.
- Can stop when the desired goal is removed from the priority queue.

- **A* Search**

- Adds goal-specific information:

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

- With $h(n) = 0$, **A* becomes UCS/Dijkstra-style search.**

- A good heuristic can make A* explore much less of the graph when we want one particular goal.
- *Dijkstra/UCS knows where it has been. A* additionally has information about where it wants to go.*

Bidirectional search

- Instead of searching only forward from the initial state:

Arad → → → → **Bucharest**

- Search simultaneously:

Arad → → → ← ← ← **Bucharest**

- One search proceeds forward from the initial state, another proceeds backward from the goal.
- When the searches meet, we obtain a candidate solution
 - with BFS expanded level by level, an appropriate stopping condition guarantees shortest-path optimality.
- Reached states from each direction can be stored in a hash table/set...

Bidirectional search

- If the solution has depth d and branching factor b :
 - Ordinary BFS explores roughly $O(b^d)$ nodes.
 - Bidirectional BFS explores roughly $O(b^{d/2}) + O(b^{d/2}) = O(b^{d/2})$
 - **Huge difference!**
- 

Why can bidirectional search be so much faster?

- Suppose:

$$b = 10, d = 6.$$

- One-direction BFS, roughly:

$$10^6 = 1,000,000$$

- Bidirectional BFS, approximately:

$$2 \times 10^3 = 2,000.$$

- **Same problem. Same solution. Much less search.**

A* Search Examples

- The Traveling Salesman Problem (TSP)
- Vertex Cover
- Independent Set