# CS 581

## *Advanced Artificial Intelligence*

**January 22, 2024**

# Announcements / Reminders

- **Please follow the Week 02 To Do List instructions (if you haven't already)**
- **Written Assignment #01: to be posted soon**
- **Programming Assignment #01: to be posted soon**

- **UPDATED Exam Dates:**
  - **Midterm:**
    - **WAS:** ~~**02/28/2024 during Wednesday lecture time**~~
    - **IS:** **02/21/2024 during Wednesday lecture time**
  - **Final:**
    - **WAS:** ~~**04/24/2024 during Wednesday lecture time**~~
    - **IS:** **04/22/2024 during Monday lecture time**

# Teaching Assistants

| Name | e-mail | Office hours |
|---|---|---|
| Gawade, Vishal | vgawade@hawk.iit.edu | TBD |
| Zhou, Xiaoting | xzhou70@hawk.iit.edu | TBD |
| | | |

TAs will:

- assist you with your assignments,
- hold office hours to answer your questions,
- grade your assignments (**a specific TA will be assigned to you**).

**Take advantage of their time and knowledge!**

**DO NOT email them with questions unrelated to lab grading.**

**Make time to meet them during their office hours.**

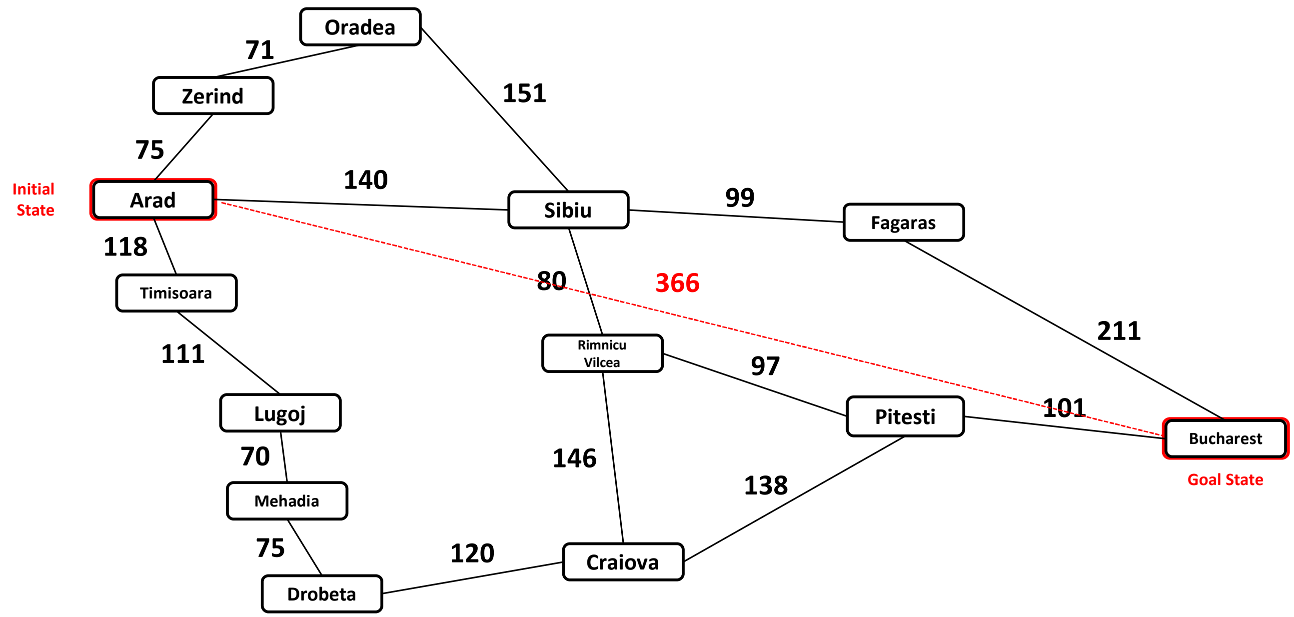
Add a **[CS581 Spring 2024]** prefix to your email subject when contacting TAs, please.

# Plan for Today

- **Solving problems by Searching**
  - **A* Algorithm: continued**
  - **Local Search Algorithms**

# What Made A* Work Well?

- Straight-line heuristics is **admissible**: it never overestimates the cost.




- An **admissible heuristics** is guaranteed to give you the optimal solution

# A* Evaluation Function

**A* Evaluation Function:**

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

**where:**

- **g(n) - initial node to node n path cost**
- **h(n) - estimated cost of the best path that continues from node n to a goal node**

**So:**

$$f(n) < C^*$$

**where:**

- **C* - optimal path cost**

# Admissible Heuristic: Proof

**An admissible heuristics h() is guaranteed to give you the optimal solution. Why? Proof by contradiction:**

- **Say: the algorithm returned a suboptimal path ($C > C^*$)**
- **So: there exist a node n on $C^*$ not expanded on C:**

**If so: $f(n) > C^*$**

$f(n) = g(n) + h(n)$ *(by definition)*

$f(n) = g^*(n) + h(n)$ *(because n is on $C^*$)*

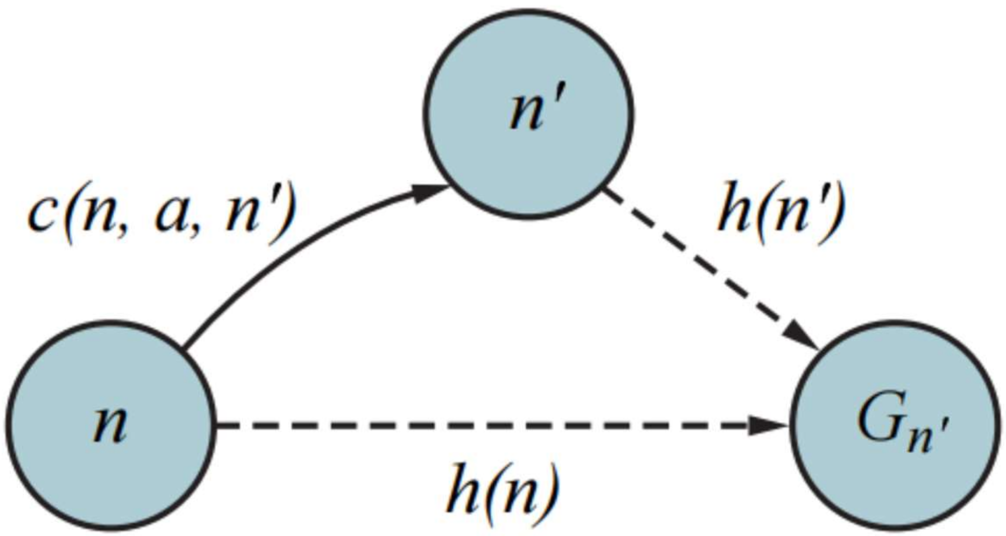
$f(n) \le g^*(n) + h^*(n)$ *(if h(n) admissible: $h(n) \le h^*(n)$)*

**But that would mean that:**

$f(n) \le C^*$ *(contradiction!)*

# What Made A* Work Well?

- **Straight-line heuristics is consistent: its estimate is getting better and better as we get closer to the goal**

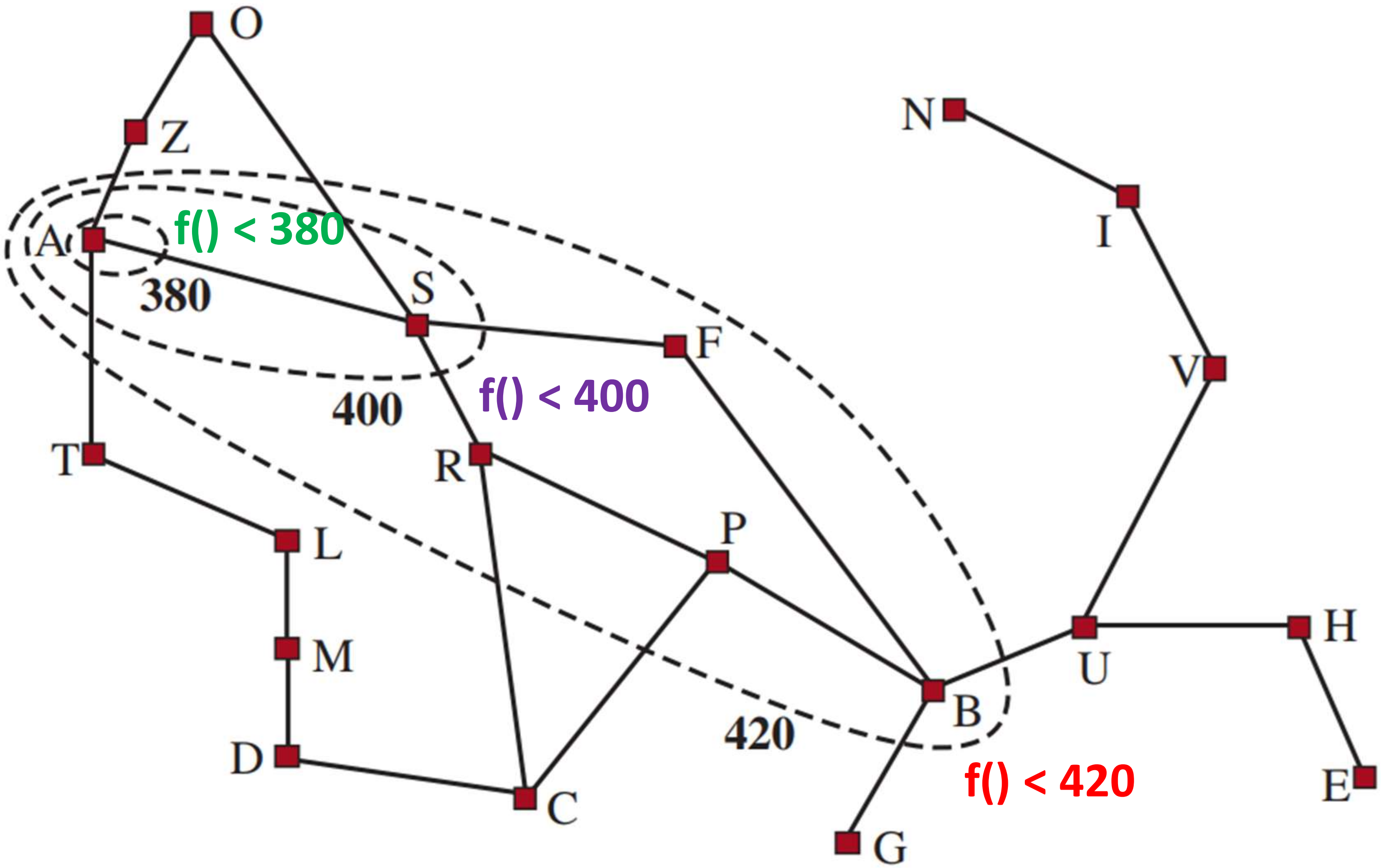



- **Every consistent heuristics is admissible heuristics, but <u>not the other way around</u>**

# A*: Search Contours

## How does A* "direct" the search progress?

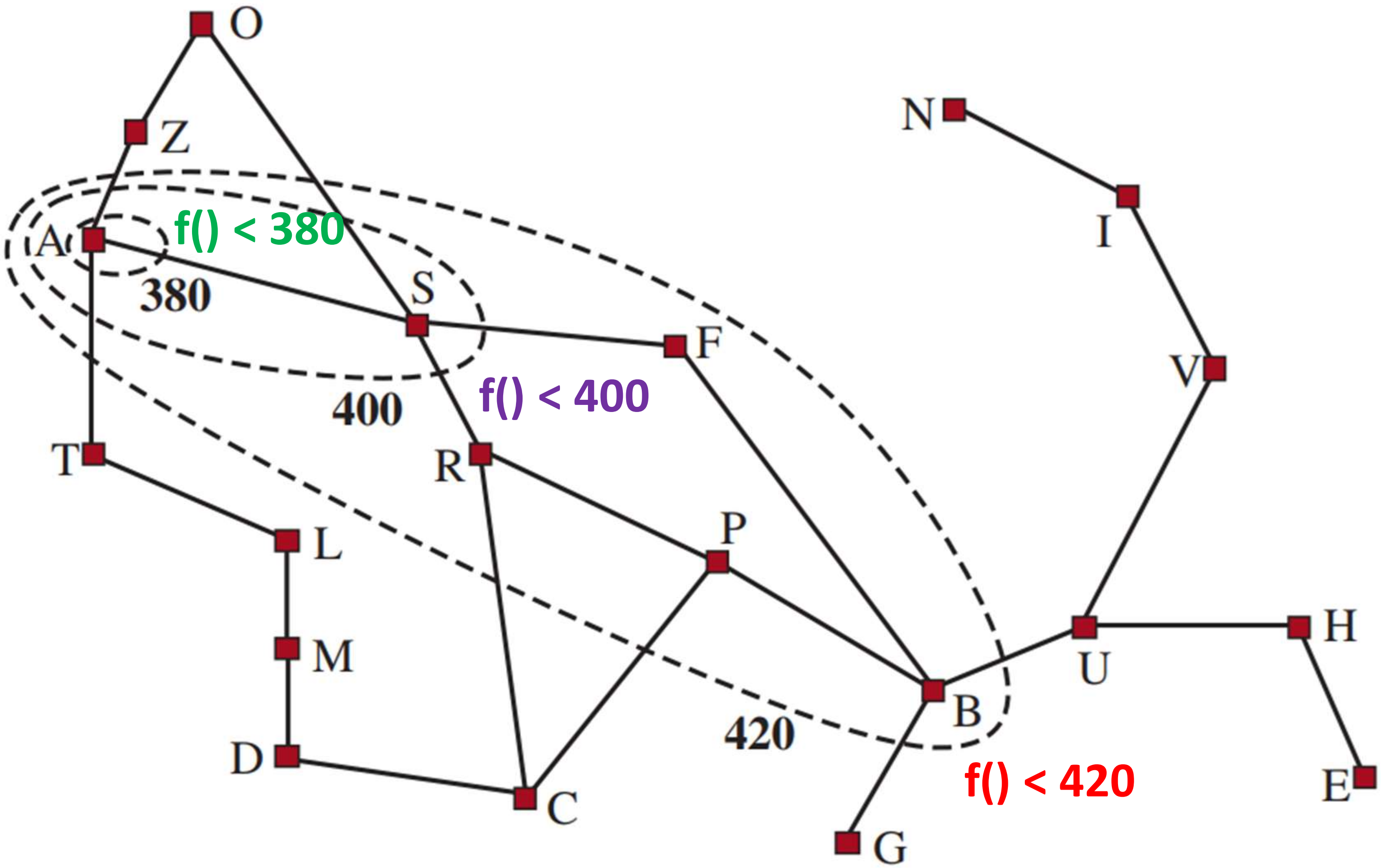
$$f(n) = \mathbf{g(State_n)} + \mathbf{h(State_n)}$$



# A*: SURELY Expanded Nodes

## Nodes INSIDE contours (ellipsoids) have:

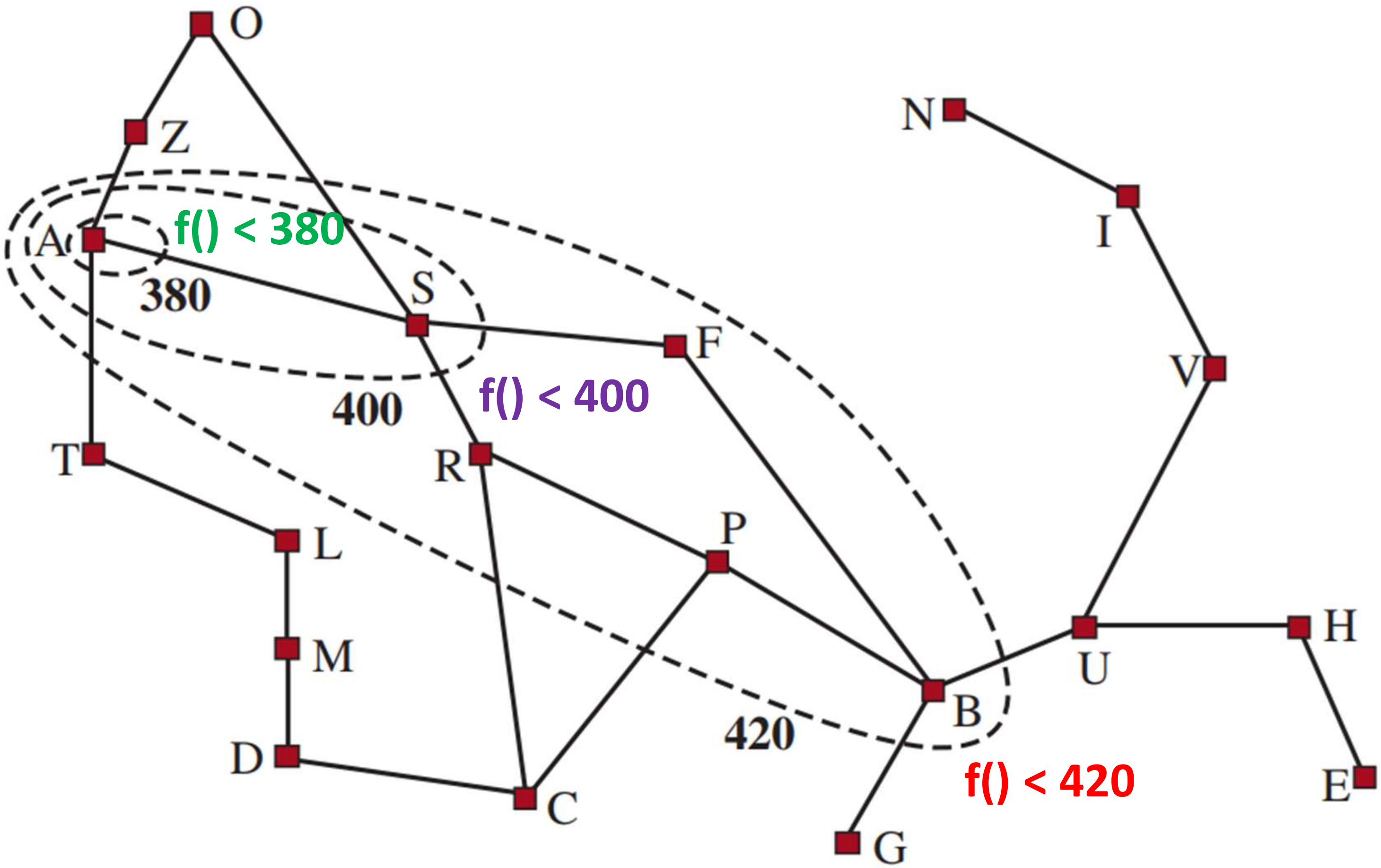
$$f(n) = \mathbf{g(State_n)} + \mathbf{h(State_n)} < \text{contour value}$$



# A*: POSSIBLY Expanded Nodes

## Nodes ON contours (ellipsoids) have:

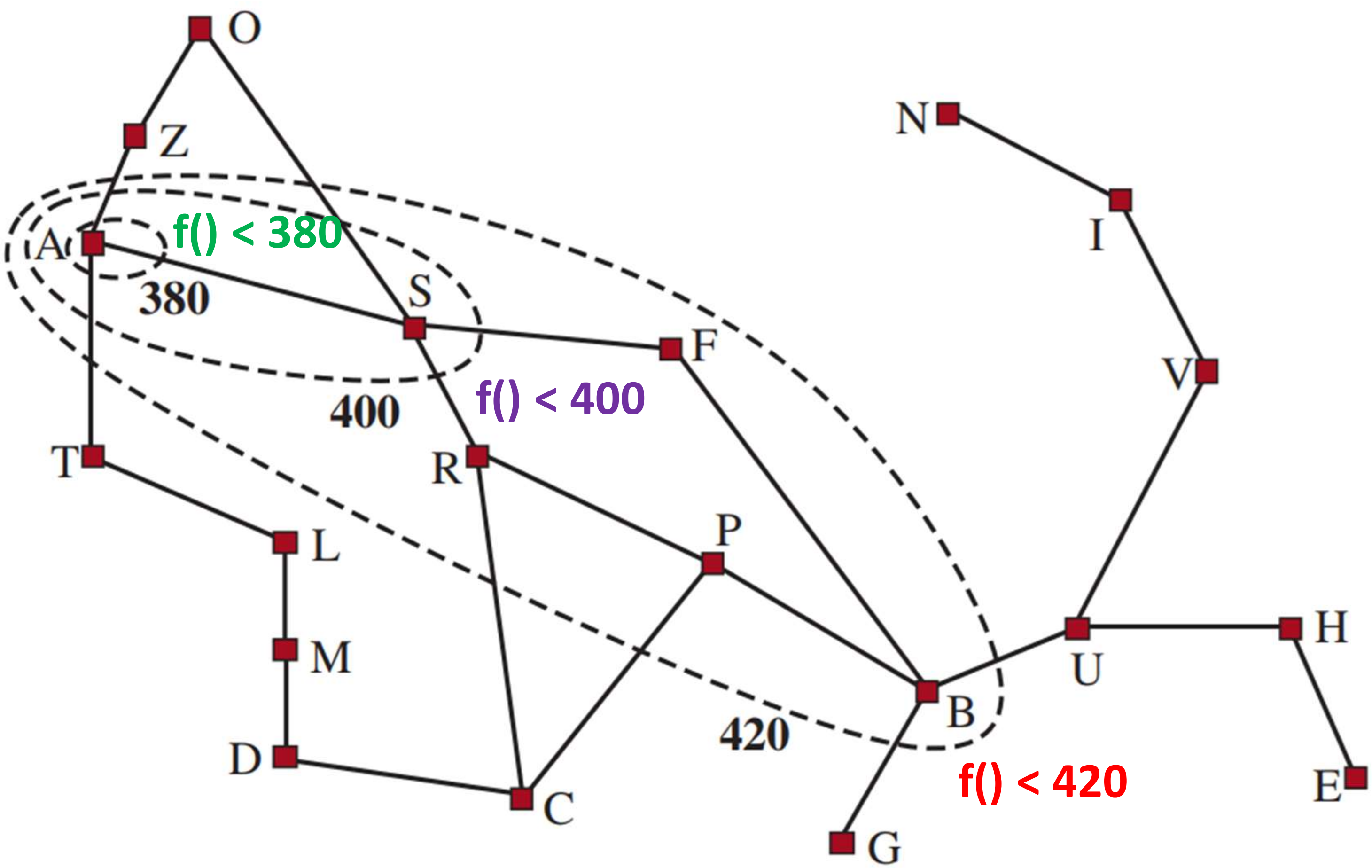
$$f(n) = \mathbf{g(State_n)} + \mathbf{h(State_n)} = \text{contour value}$$



# A*: NOT Expanded Nodes

## Nodes OUTSIDE contours (ellipsoids) have:

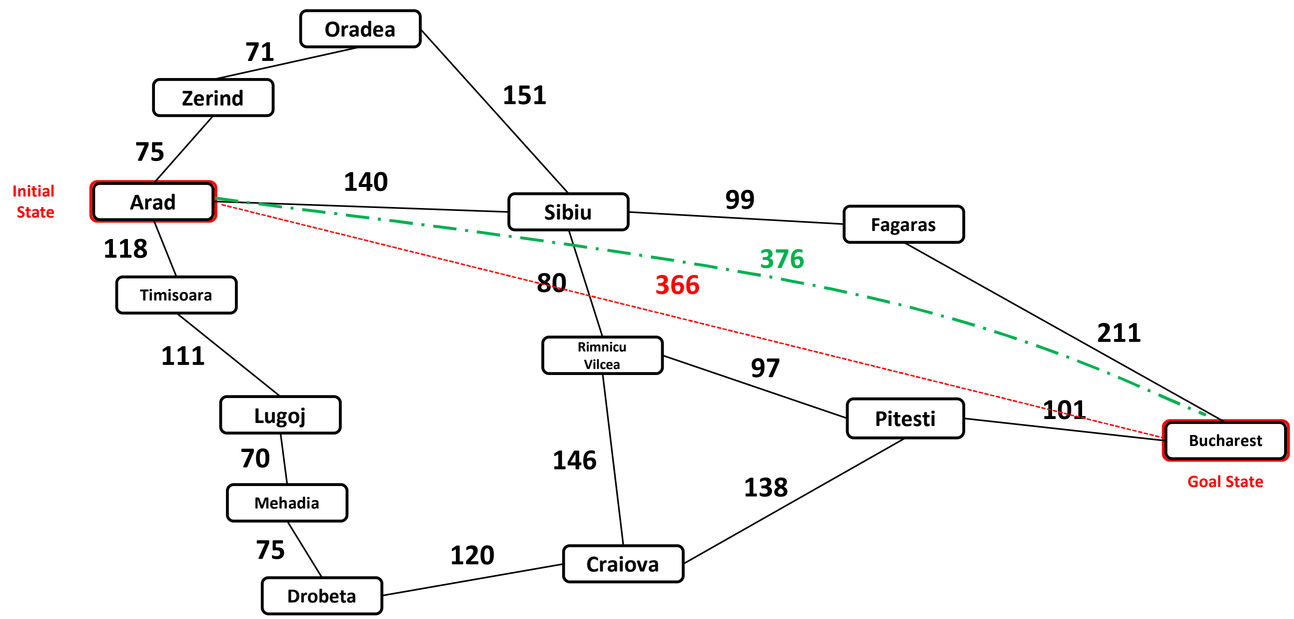
$$f(n) = \mathbf{g(State_n)} + \mathbf{h(State_n)} > \text{contour value}$$



# Dominating Heuristics

**We can have more than one available heuristics.**

**For example $h_1(n)$ and $h_2(n)$.**

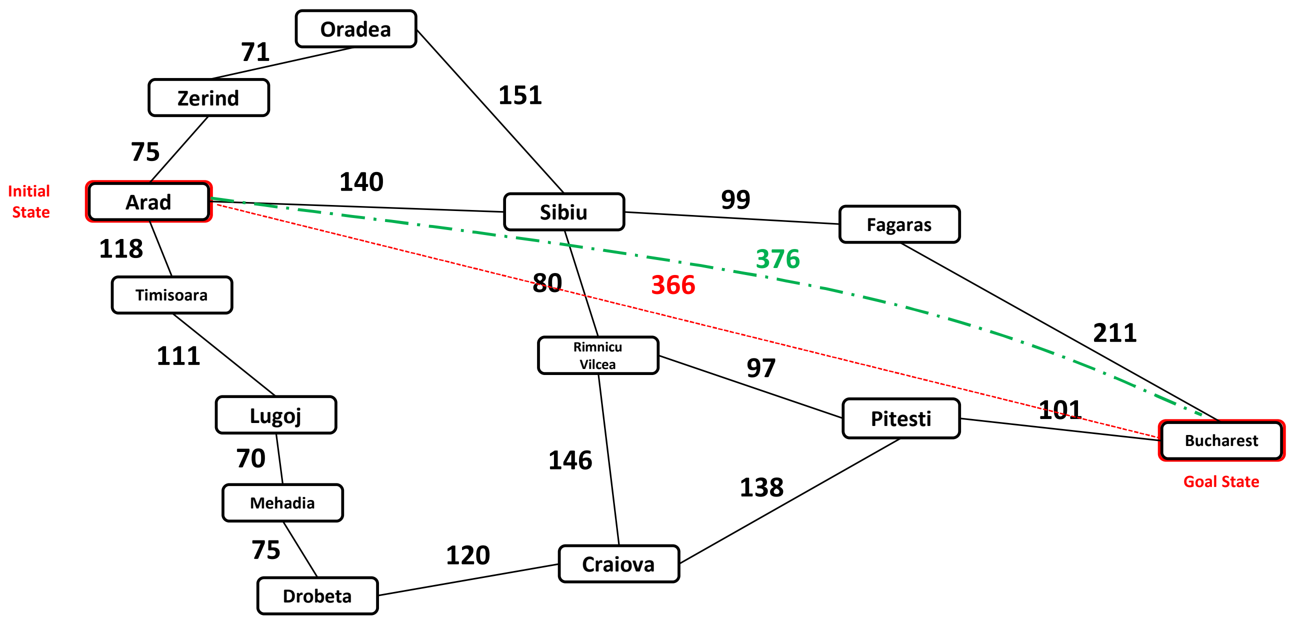



**$h_2(n)$ dominates $h_1(n)$ if and only if**

**$h_2(n) > h_1(n)$ for every $n$**

# Dominating Heuristics

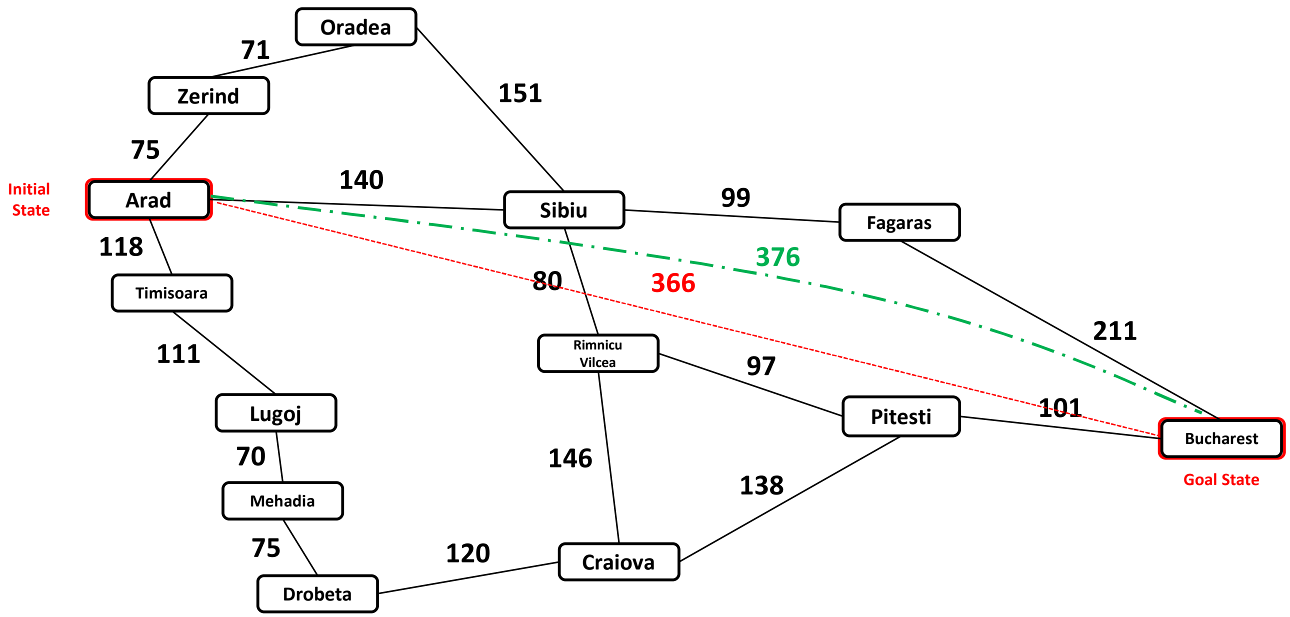
We can have more than one available heuristics. For example $h_1(n)$ and $h_2(n)$.




Heuristics $h_2(n)$ estimate is closer to actual cost than $h_1(n)$. $h_2(n)$ dominates $h_1(n)$. Use $h_2(n)$.

# Dominating Heuristics

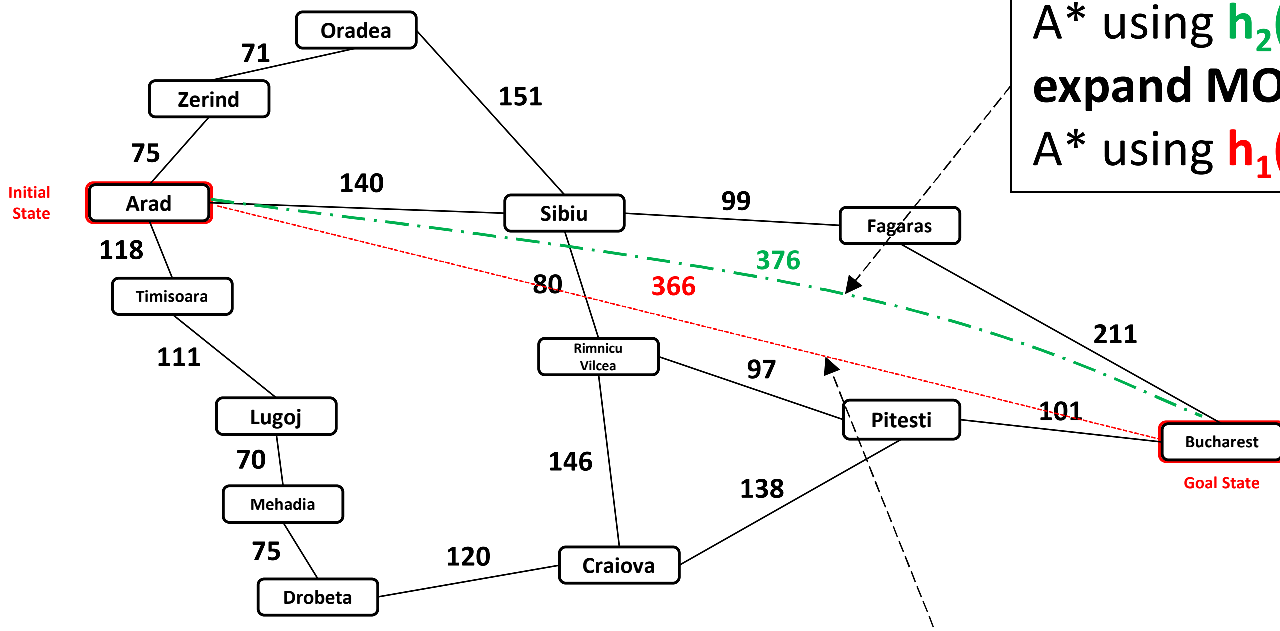
**We can have more than one available heuristics. For example $h_1(n)$ and $h_2(n)$.**




**If you have multiple admissible heuristics where none dominates the other: Let $h(n) = \max(h_1(n), h_2(n), \ldots, h_m(n))$**

# Domination → Efficiency

**Heuristics $h_2(n)$ estimate is closer to actual cost than $h_1(n)$. $h_2(n)$ dominates $h_1(n)$. Use $h_2(n)$.**




A* using $h_2(n)$ will **NEVER expand MORE** nodes than A* using $h_1(n)$*

A* using $h_1(n)$ will **NEVER expand LESS** nodes than A* using $h_2(n)$*

* Except for in the case of breaking unlucky ties

# Domination → Efficiency: Why?

**A* Evaluation Function:**

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

**where:**

- **g(n) - initial node to node n path cost**
- **h(n) - estimated cost of the best path that continues from node n to a goal node**

**So:**

$$f(n) < C^*$$

**where:**

- **C* - optimal path cost**

# Domination → Efficiency: Why?

**With**

$$f(n) < C^*$$

**and**

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

**We get:**

$$g(State_n) + h(State_n) < C^*$$

$$h(State_n) < C^* - g(State_n)$$

# Domination → Efficiency: Why?

**If**

$$h_1(State_n) < h_2(State_n)$$

**then:**

$$h_1(State_n) < h_2(State_n) < C^* - g(State_n)$$

# Domination → Efficiency: Why?

**If**

$$h_1(State_n) < h_2(State_n)$$

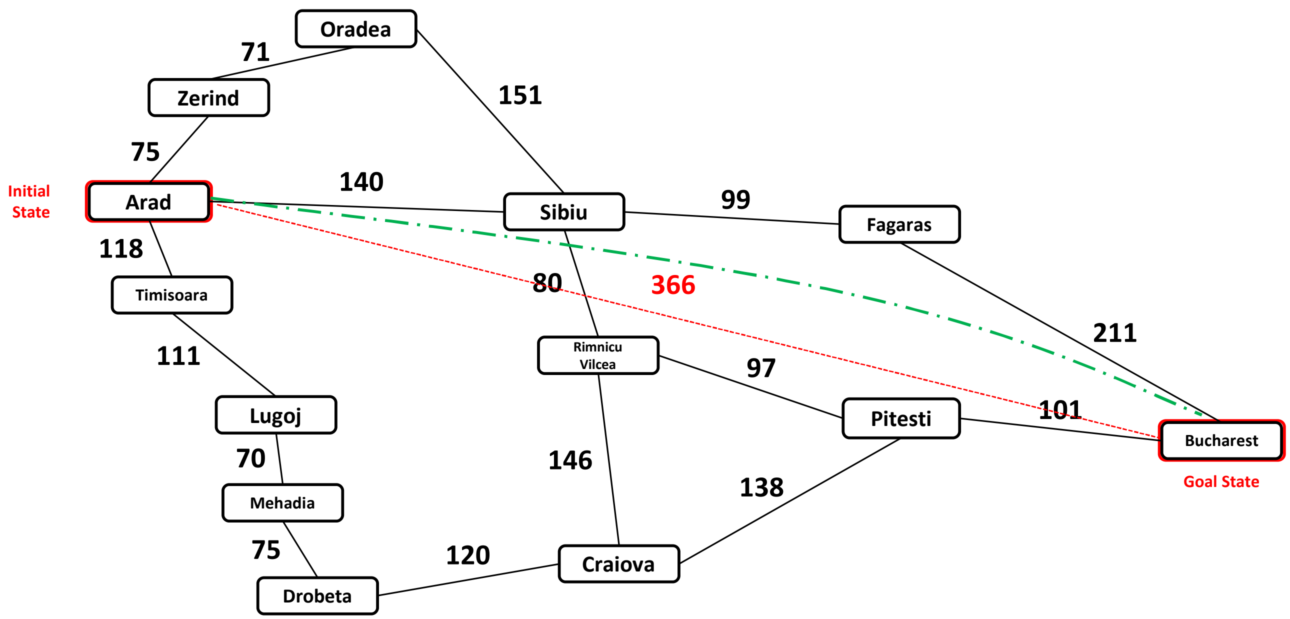
**then:**

$$h_1(State_n) < h_2(State_n) < C^* - g(State_n)$$

All A* frontier candidates selected using **$h_1(n)$** will include **AT LEAST** all A* frontier candidates selected using **$h_2(n)$**. Possibly more.

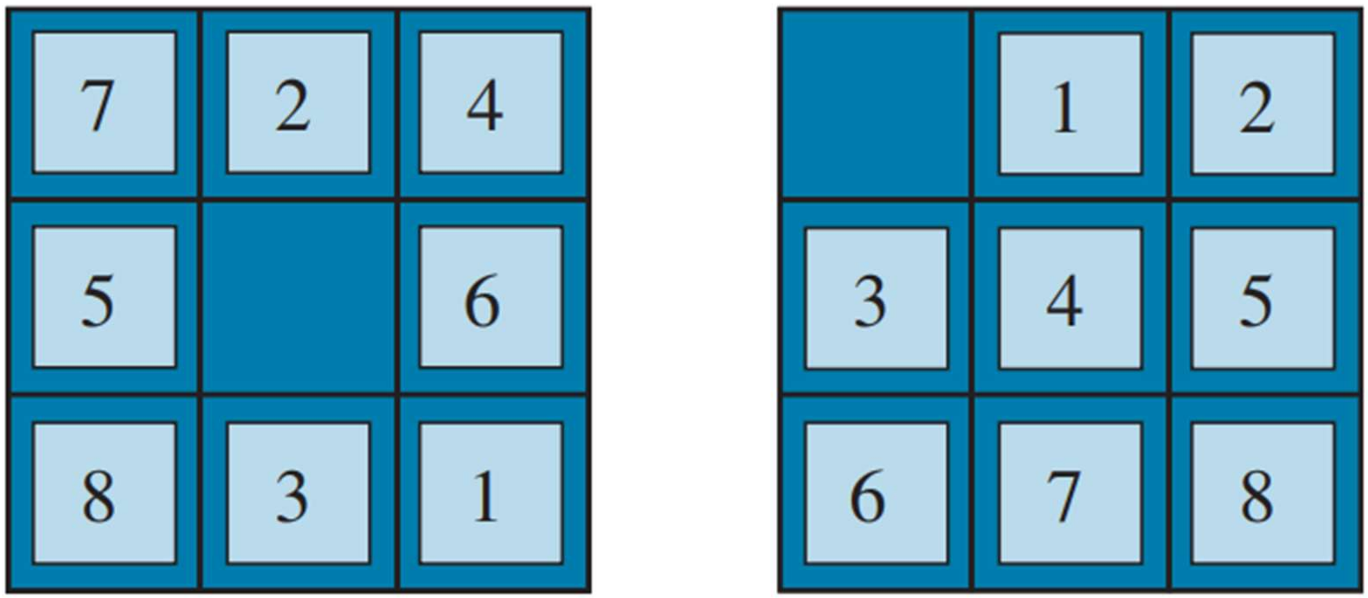
# Domination → Efficiency: But?

## $h_2(n)$ dominates $h_1(n)$. Always use $h_2(n)$?




**Generally yes, but $h_2(n)$ vs. $h_1(n)$ heuristic computation time may be a deciding factor here.**

# Heuristic and Search Performance

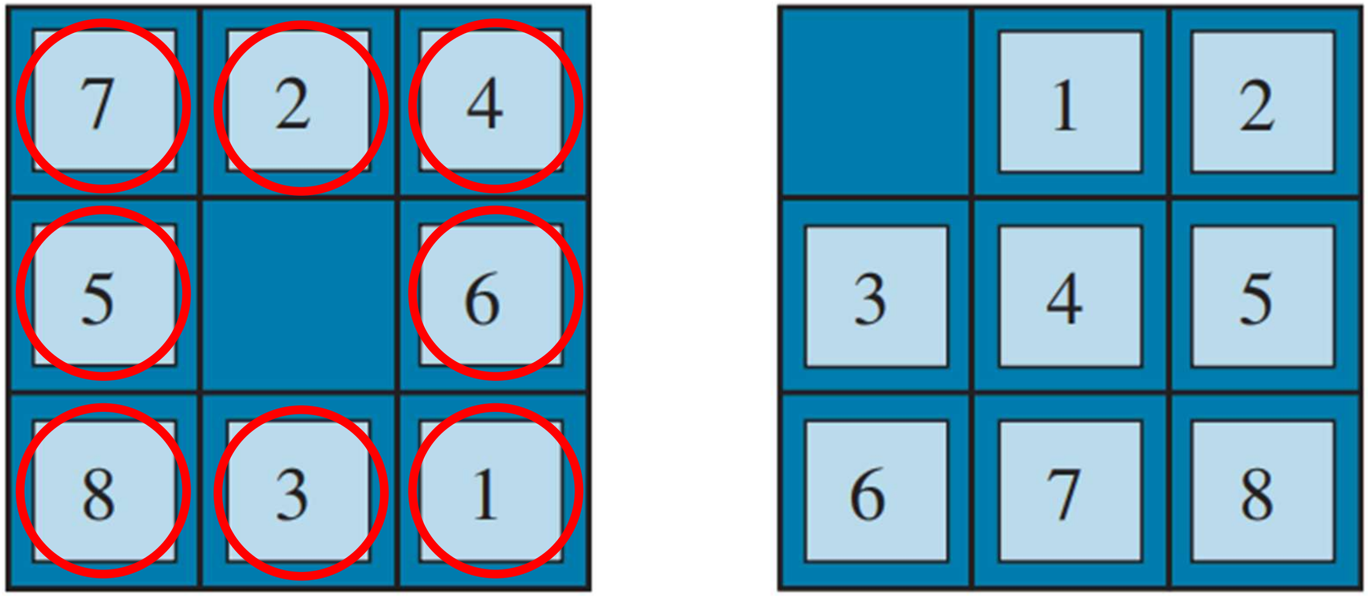



Start State Goal State

**Consider an 8-puzzle game and two admissible heuristics:**

- **$h_1()$ – number of misplaced tiles (not counting blank)**
- **$h_2()$ – Manhattan distance**

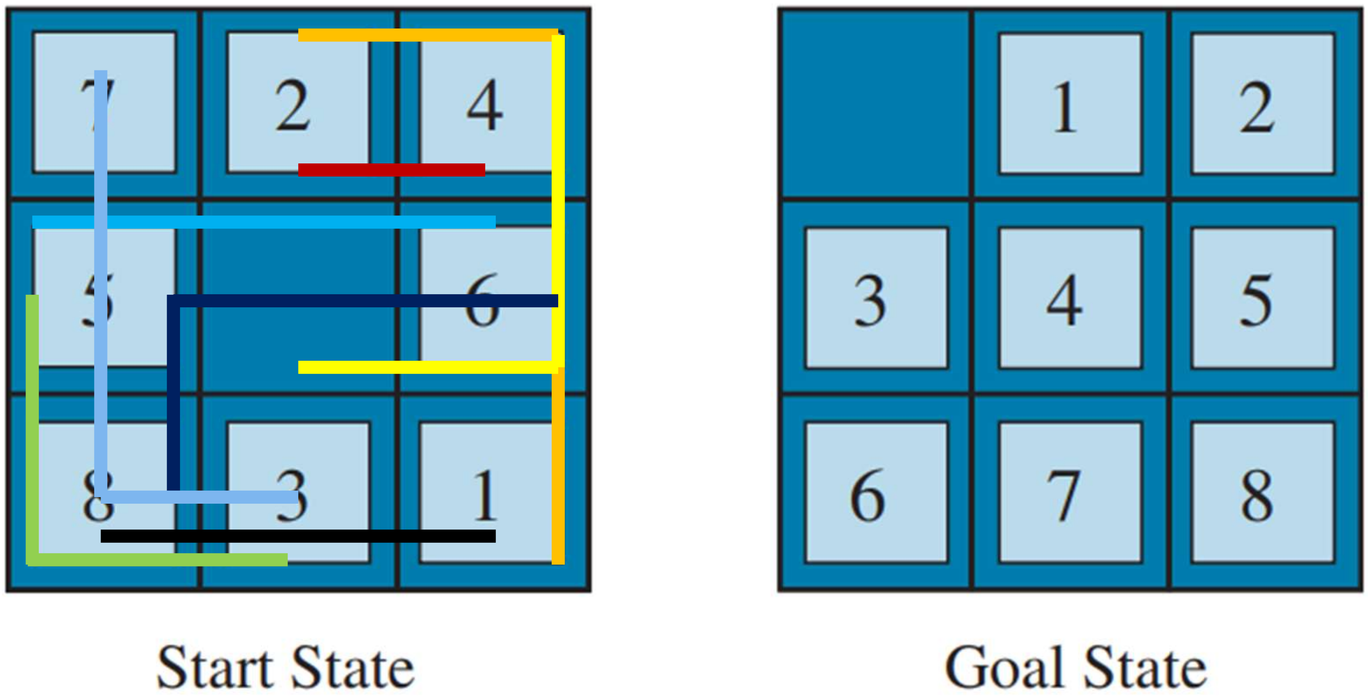
# Heuristic and Search Performance




Start State        Goal State

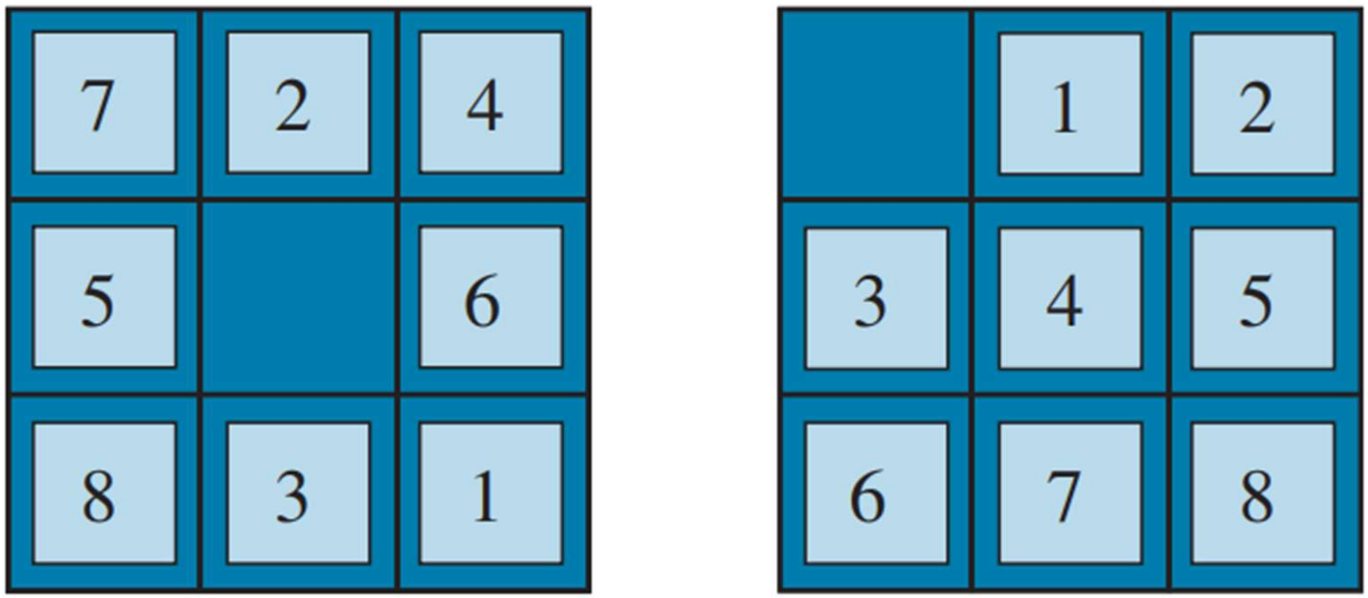
- **$h_1$(Start) = 8**        **All 8 blocks are out of place**

# Heuristic and Search Performance




- $h_1$(Start) = 8
- $h_2$(Start) = 3 + 1 + 2 + 2 + 2 + 3 + 3 + 2 = 18

# Heuristic and Search Performance

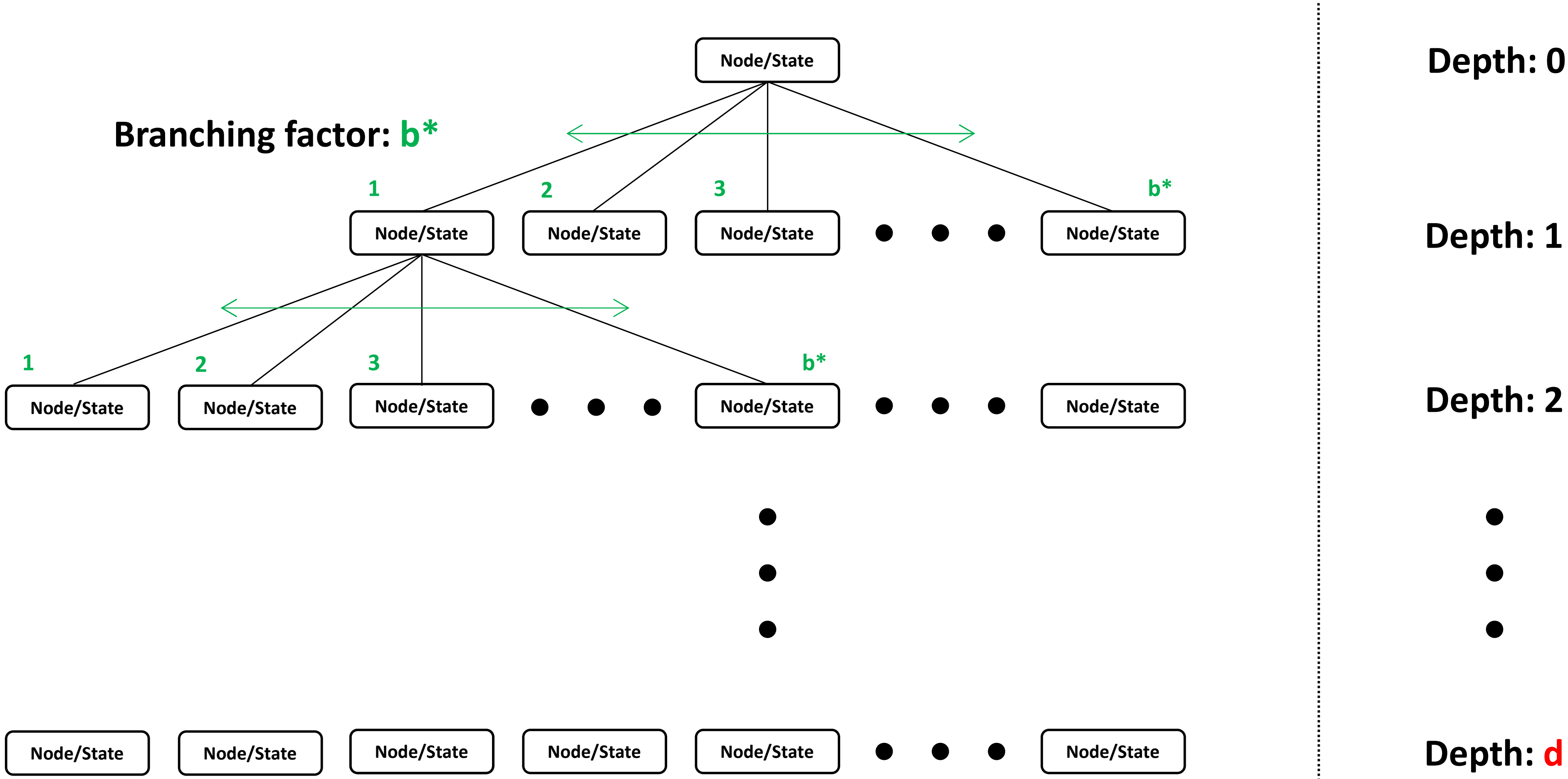



Start State Goal State

**$h_1$(Start) = 8 < $h_2$(Start) = 18 < C* = 26**

**Neither heuristic overestimates true solution cost**

# h() Quality: Effective Branching Factor




Total number of nodes / states: $1 + b^* + b^{*2} + b^{*3} + \ldots + b^{*d} \rightarrow N + 1$

# h() Quality: Effective Branching Factor

- If the total number of nodes generated is $N$ and the solution depth is $d$, then
  - $b^*$ is the branching factor that a uniform tree of depth $d$ would need to have in order to contain $N+1$ nodes

$$N+1 = 1 + b^* + (b^*)^2 + \ldots + (b^*)^d$$

- If A* finds a solution at depth 4 using 40 nodes, what is $b^*$?

$$\approx 2.182$$

- A good heuristic function achieves $b^* \approx 1$

# 8-puzzle: Heuristics Comparison

| | 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 |

**Where (data are averaged over 100 puzzles for each solution length d = 6 to 28):**

- **BFS – Breadth First Search**
- **$A^*(h_1)$ – A* using $h_1()$ – number of misplaced tiles**
- **$A^*(h_2)$ – A* using $h_2()$ – Manhattan distance**

# Can We Make A* Even Faster?

# (Sometimes at a cost!)

# Weighted A* Evaluation Function

**Weighted A* Evaluation Function:**

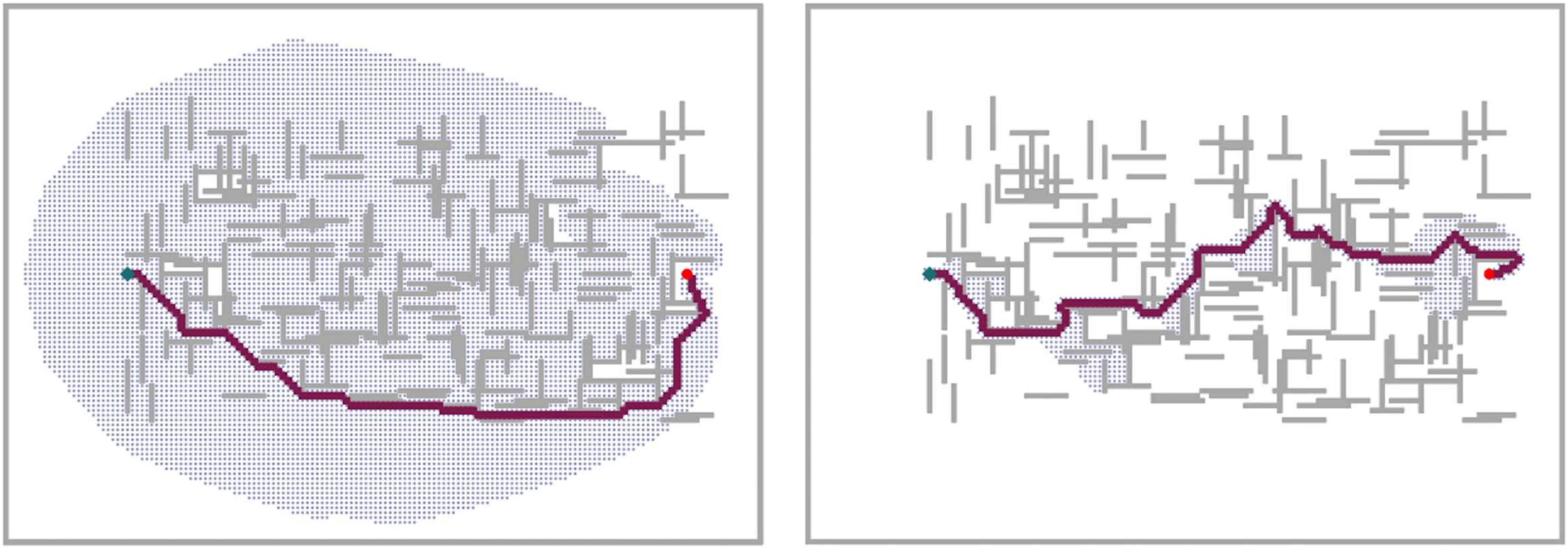
$$f(n) = g(State_n) + W * h(State_n)$$

**where:**

- **g(n) - initial node to node n path cost**
- **h(n) - estimated cost of the best path that continues from node n to a goal node**
- **W > 1**

**Here, weight W makes $h(State_n)$ (perhaps only "sometimes") inadmissible. It becomes potentially more accurate = less expansions!**

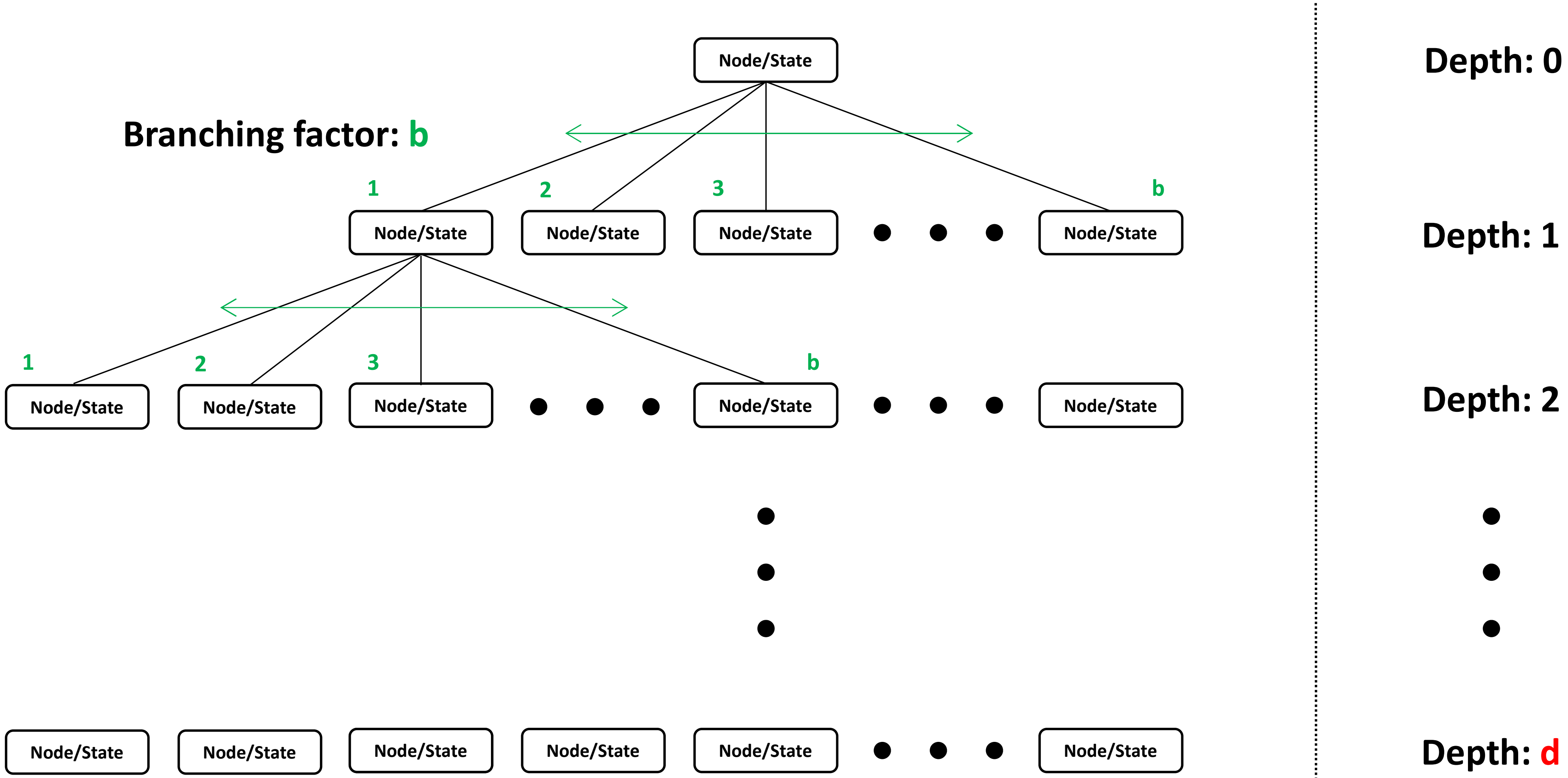
# Weighted A*: "Good Enough" Search



(a) (b)

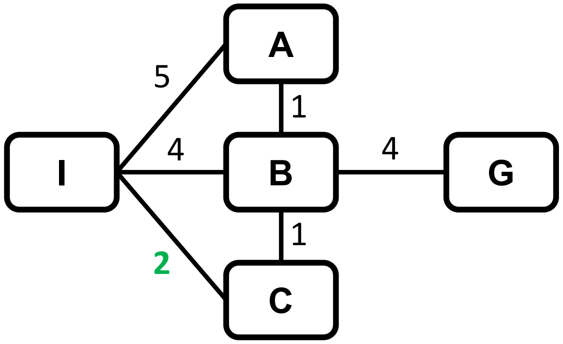
**Figure 3.21** Two searches on the same grid: (a) an A* search and (b) a weighted A* search with weight $W = 2$. The gray bars are obstacles, the purple line is the path from the green start to red goal, and the small dots are states that were reached by each search. On this particular problem, weighted A* explores 7 times fewer states and finds a path that is 5% more costly.

# Search Tree Challenges: Size




What are the possible A* search problems here? PLURAL!

# A* Search Challenges: Complexity




| Straight-line distance to Goal state | | | | | |
|---|---|---|---|---|---|
| State | I | A | B | C | G |
| h(State) | 7 | 2 | 3 | 4 | 0 |

**INITIAL STATE: I**
**GOAL STATE: G**

| Frontier | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Parent | C | I | I | | | | | |
| Node | B | B | A | | | | | |
| f(Node) | 6 | 7 | 7 | | | | | |

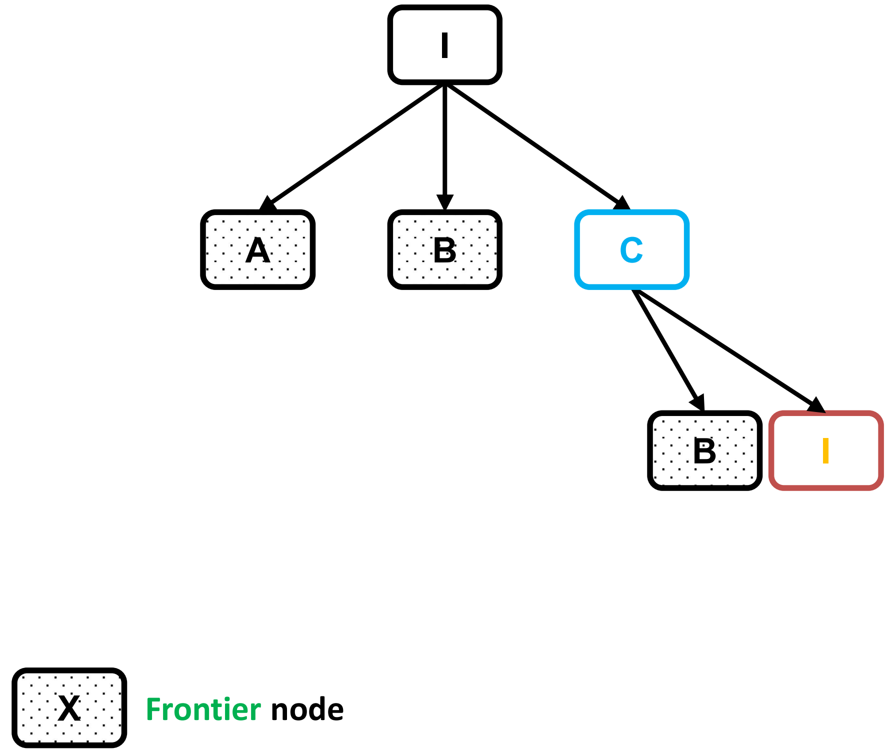
| Reached | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Parent | ---- | I | C | I | | | | |
| Key/State | I | A | B | C | | | | |
| Path cost | 0 | 5 | 3 | 2 | | | | |

State Space **Graph** | Frontier / Reached | **Search Tree**

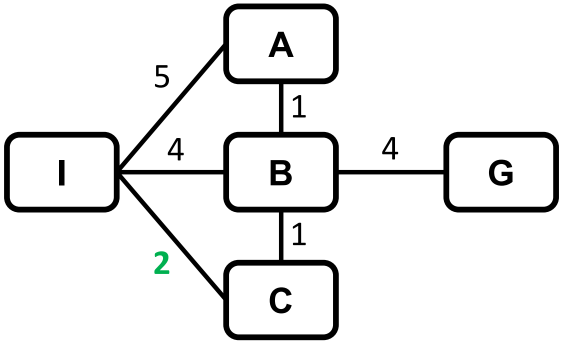
We need to STORE multiple structures in memory:

- Frontier queue
- Reached table
- search tree

This may be too much too handle…



# A* Search Challenges: Solutions




| Straight-line distance to Goal state | | | | | |
|---|---|---|---|---|---|
| State | **I** | **A** | **B** | **C** | **G** |
| h(State) | 7 | 2 | 3 | 4 | 0 |

**INITIAL STATE: I**
**GOAL STATE: G**

| Frontier | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Parent | C | I | I | | | | | |
| Node | B | B | A | | | | | |
| f(Node) | 6 | 7 | 7 | | | | | |

| Reached | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Parent | ---- | | | | | | | |
| Key/State | I | A | | C | | | | |
| Path cost | **0** | | 3 | | | | | |

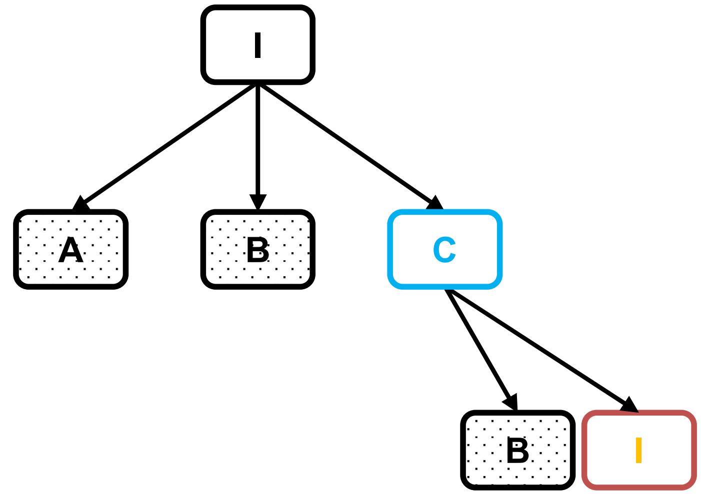
State Space **Graph**

Frontier / Reached

**Search Tree**

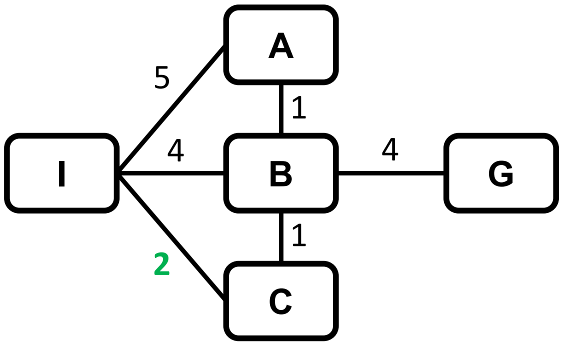
## A) "Merge" Frontier and Reached data into one structure :

- less memory used, but
- more complex algorithm
- potentially slower algorithm




**X** **Frontier node**

# A* Search Challenges: Solutions




| Straight-line distance to Goal state | | | | | |
|---|---|---|---|---|---|
| State | I | A | B | C | G |
| h(State) | 7 | 2 | 3 | 4 | 0 |

**INITIAL STATE: I**
**GOAL STATE: G**

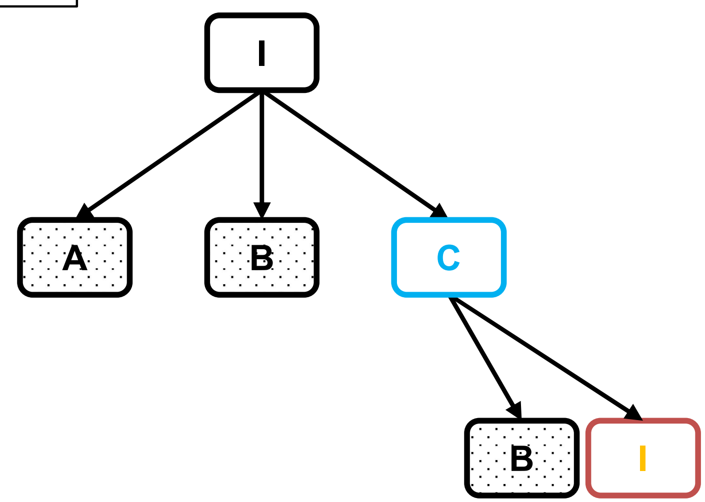
| Frontier | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Parent | C | I | I | | | | | |
| Node | B | B | A | | | | | |
| f(Node) | 6 | 7 | 7 | | | | | |

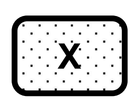
| Reached | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Parent | ---- | I | C | I | | | | |
| Key/State | I | A | B | C | | | | |
| Path cost | 0 | 5 | 3 | 2 | | | | |

State Space Graph | Frontier / Reached | Search Tree

**B) Remove nodes from Reached once we can prove that they no longer be needed**

- **ensure that such nodes will not be “touched” again**

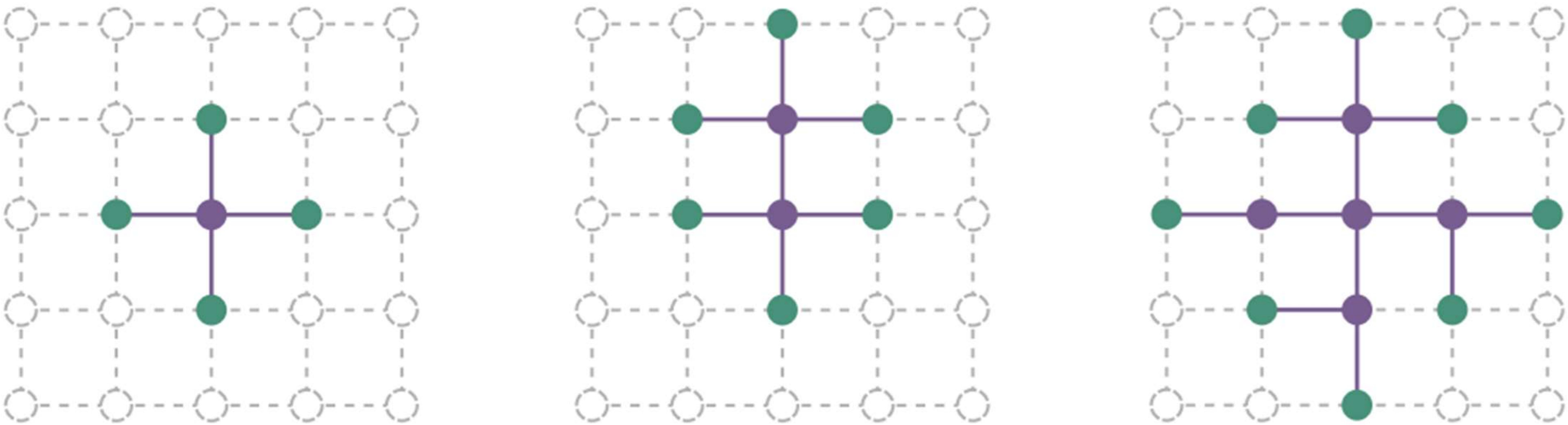


**Frontier node**

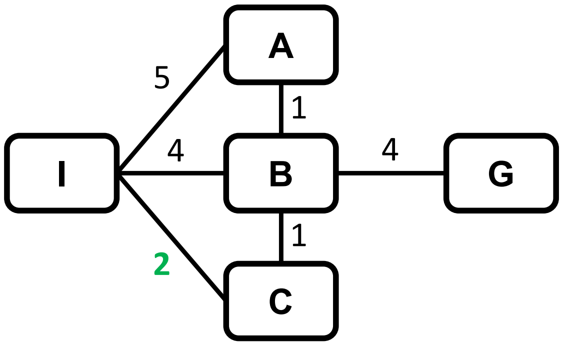
# Can We Remove It From Reached?

**State Space graph provides a list of state neighbors.**



**It can be used to decide if the state will be needed or not.**

# A* Search Challenges: Solutions




| Straight-line distance to Goal state | | | | | |
|---|---|---|---|---|---|
| State | **I** | **A** | **B** | **C** | **G** |
| h(State) | 7 | 2 | 3 | 4 | 0 |

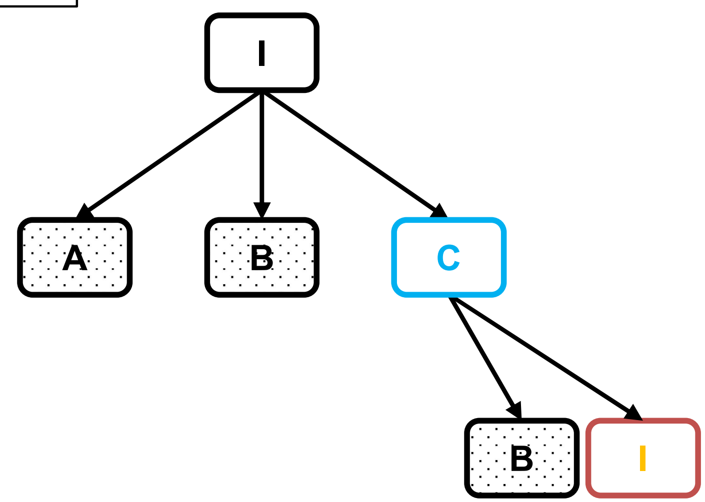
**INITIAL STATE: I**
**GOAL STATE: G**

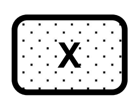
| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Frontier | Parent | C | I | I | | | | | |
| | Node | B | B | A | | | | | |
| | f(Node) | 6 | 7 | 7 | | | | | |
| Reached | Parent | ---- | I | C | I | | | | |
| | Key/State | I | A | B | C | | | | |
| | Path cost | **0** | 5 | 3 | 2 | | | | |

State Space **Graph** | Frontier / Reached | **Search Tree**

## C) Keep reference counts of the number of times a state can be reached.

- once a state has been visited **reference counts** times, it can be removed from **Reached**

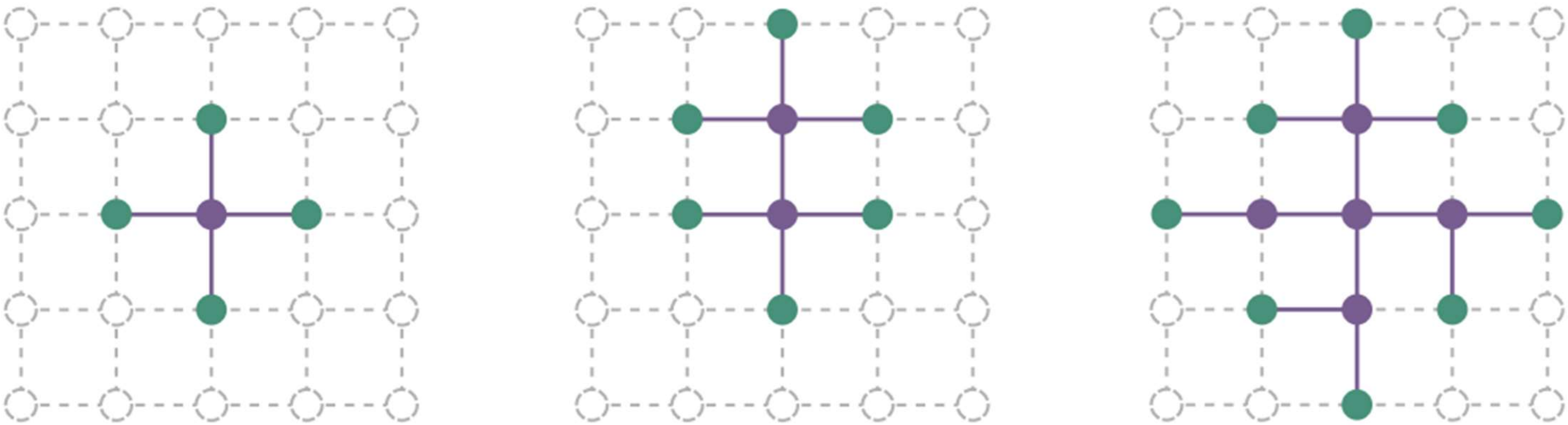



**Frontier node**

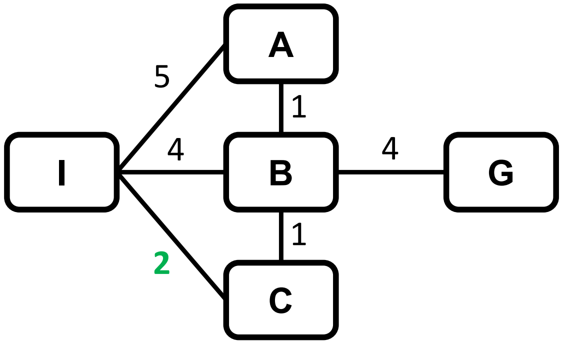
# State Space: Reference Counts

**State Space graph provides the number of neighbors for each state.**



**It can be used as a reference count.**

# A* Search Challenges: Solutions




| Straight-line distance to Goal state | | | | | |
|---|---|---|---|---|---|
| State | **I** | **A** | **B** | **C** | **G** |
| h(State) | 7 | 2 | 3 | 4 | 0 |

**INITIAL STATE: I**
**GOAL STATE: G**

| Frontier | | | | | | | |
|---|---|---|---|---|---|---|---|
| Parent | C | I | I | | | | |
| Node | B | B | A | | | | |
| f(Node) | 6 | 7 | 7 | | | | |

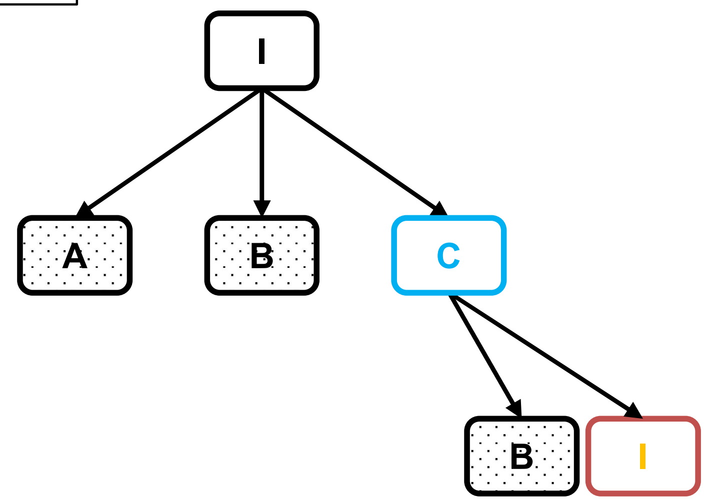
| Reached | | | | | | | |
|---|---|---|---|---|---|---|---|
| Parent | ---- | I | C | I | | | |
| Key/State | I | A | B | C | | | |
| Path cost | 0 | 5 | 3 | 2 | | | |

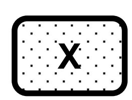
State Space **Graph** | Frontier / Reached | **Search Tree**

## D) Beam Search → restrict the size of the Frontier:

- keep $k$ best f()-score elements
- keep only elements with f()-score such that:

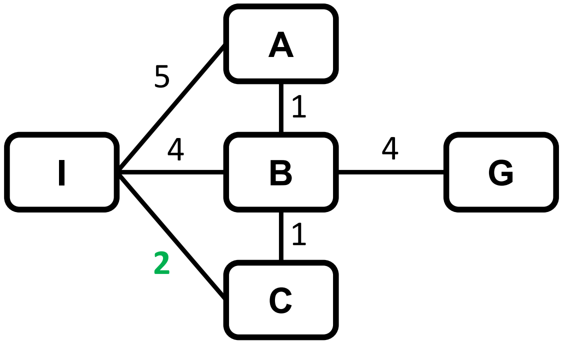
f()-score within $\delta$ of $f()_{best}$




**Frontier node**

# A* Search Challenges: Solutions




| Straight-line distance to Goal state | | | | | |
|---|---|---|---|---|---|
| State | I | A | B | C | G |
| h(State) | 7 | 2 | 3 | 4 | 0 |

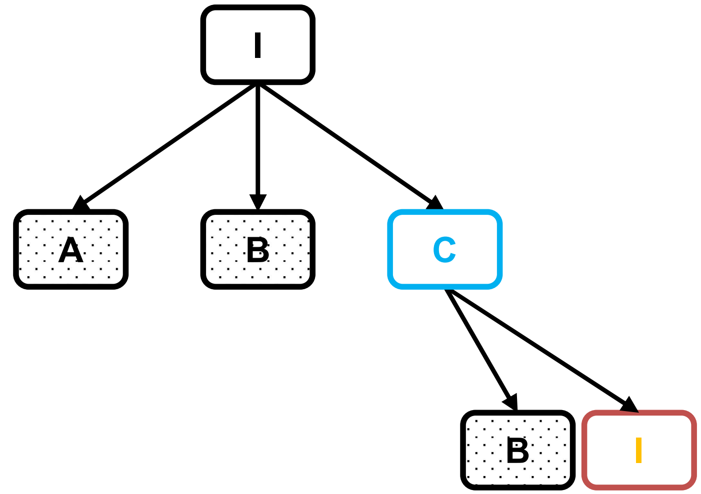
**INITIAL STATE: I**
**GOAL STATE: G**

| Frontier | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Parent | C | I | I | | | | | |
| Node | B | B | A | | | | | |
| f(Node) | 6 | 7 | 7 | | | | | |

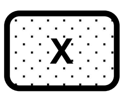
| Reached | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Parent | ---- | I | C | I | | | | |
| Key/State | I | A | B | C | | | | |
| Path cost | 0 | 5 | 3 | 2 | | | | |

State Space **Graph** | Frontier / Reached

**Search Tree**



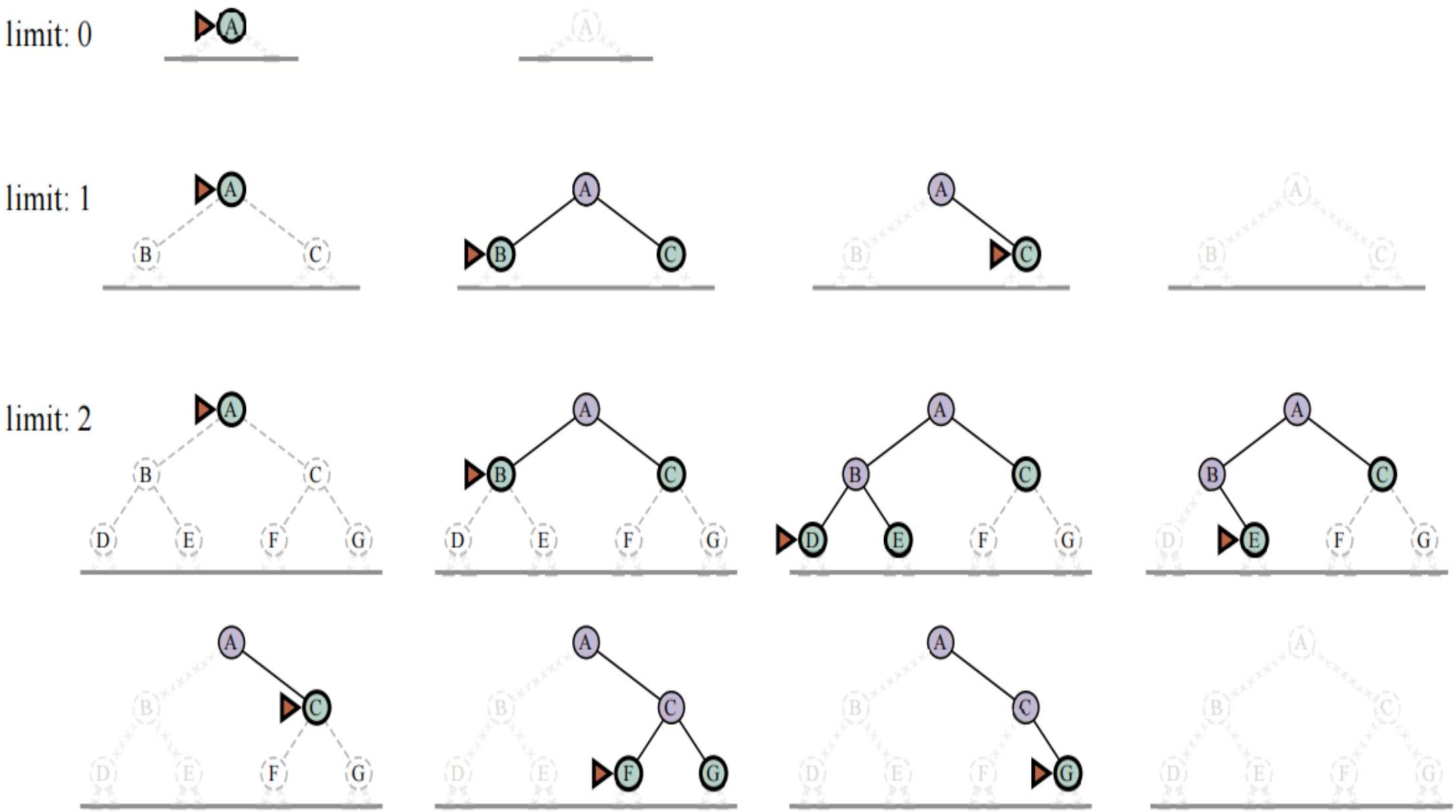

**Reduce size**



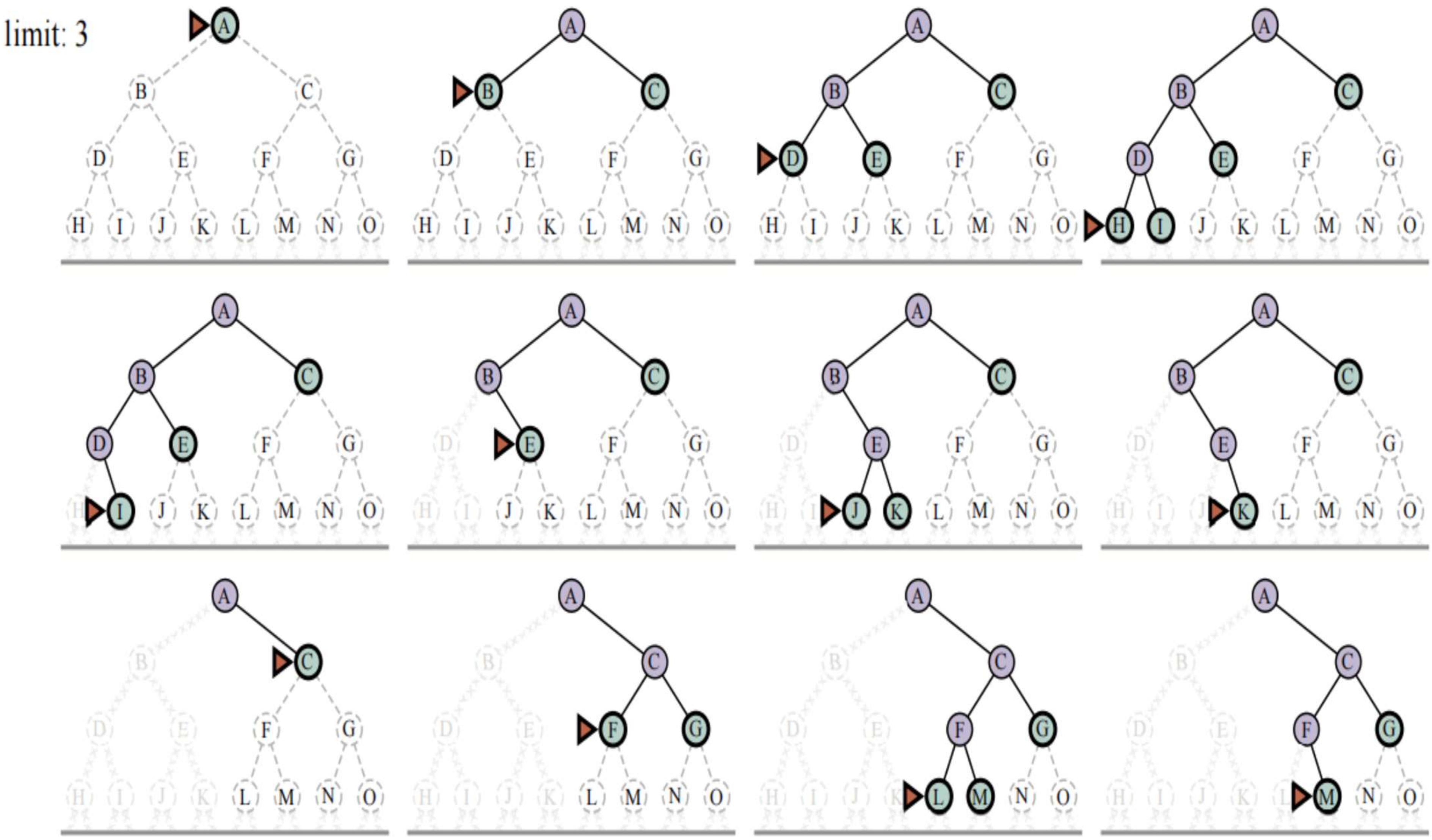

**Frontier node**

## E) Use Iterative-Deepening A* search (IDA*)

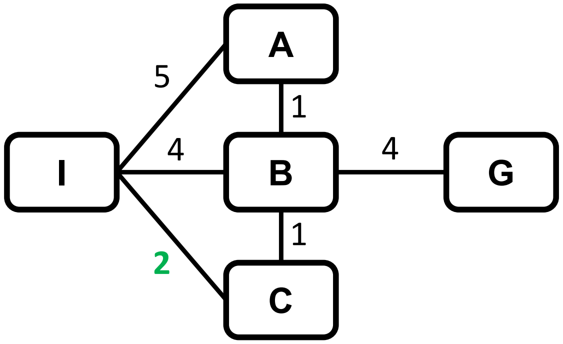
# Iterative Deepening DFS: Illustration



# Iterative Deepening DFS: Illustration



# A* Search Challenges: Solutions




| Straight-line distance to Goal state | | | | | |
|---|---|---|---|---|---|
| State | I | A | B | C | G |
| h(State) | 7 | 2 | 3 | 4 | 0 |

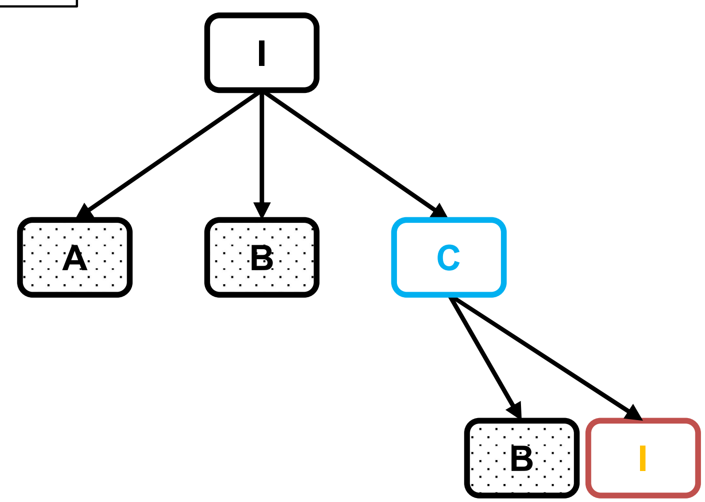
**INITIAL STATE: I**
**GOAL STATE: G**

| Frontier | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Parent | C | I | I | | | | | |
| Node | B | B | A | | | | | |
| f(Node) | 6 | 7 | 7 | | | | | |

| Reached | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Parent | ---- | I | C | I | | | | |
| Key/State | I | A | B | C | | | | |
| Path cost | 0 | 5 | 3 | 2 | | | | |

State Space **Graph** | Frontier / Reached

**Search Tree**




**Reduce size**

X **Frontier node**

## F) Use Recursive Best First Search

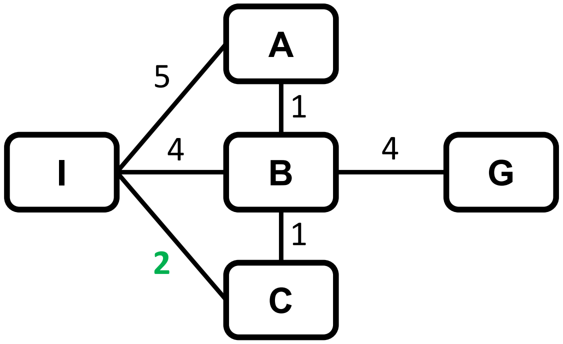
- **maintain f() limit (best alternative ancestor path f())**
- **if current node's f() exceeds f() limit, recursion unwinds**

# Recursive Best First Search

```
function RECURSIVE-BEST-FIRST-SEARCH(problem) returns a solution or failure
    solution, fvalue ← RBFS(problem, NODE(problem.INITIAL), ∞)
  return solution

function RBFS(problem, node, f_limit) returns a solution or failure, and a new f-cost limit
   if problem.IS-GOAL(node.STATE) then return node
   successors ← LIST(EXPAND(node))
   if successors is empty then return failure, ∞
   for each s in  successors do        // update f with value from previous search
       s.f ← max(s.PATH-COST + h(s), node.f))
   while true do
       best ← the node in successors with lowest f-value
       if best.f > f_limit then return failure, best.f
       alternative ← the second-lowest f-value among successors
       result, best.f ← RBFS(problem, best, min(f_limit, alternative))
       if result ≠ failure then return result, best.f
```

# A* Search Challenges: Solutions




| Straight-line distance to Goal state | | | | | |
|---|---|---|---|---|---|
| State | **I** | **A** | **B** | **C** | **G** |
| h(State) | 7 | 2 | 3 | 4 | 0 |

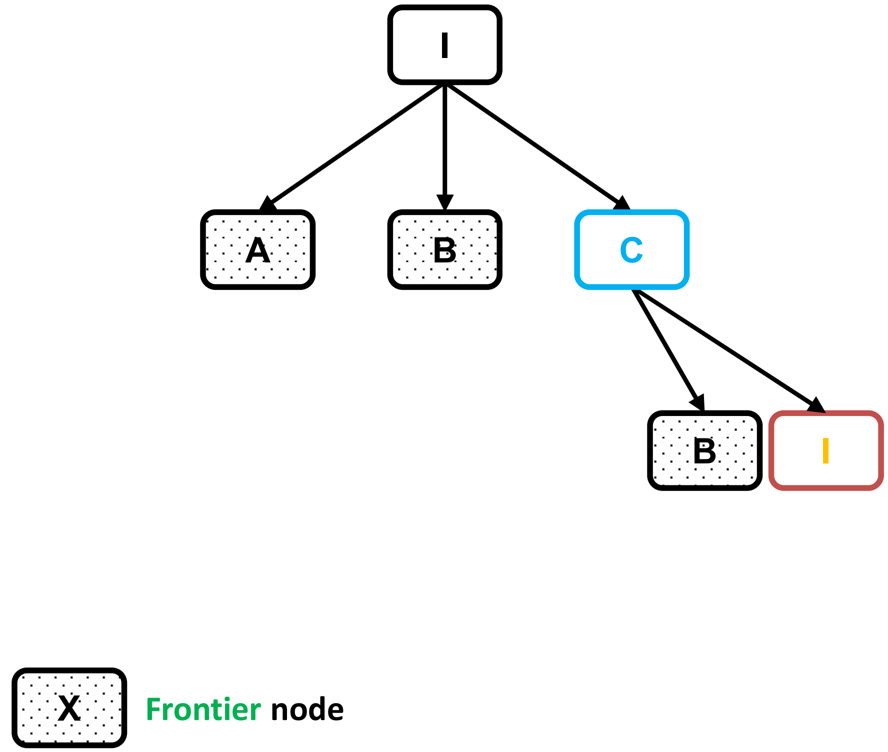
**INITIAL STATE: I**
**GOAL STATE: G**

| Frontier | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Parent | C | I | I | | | | | |
| Node | B | B | A | | | | | |
| f(Node) | 6 | 7 | 7 | | | | | |

| Reached | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Parent | ---- | I | C | I | | | | |
| Key/State | I | A | B | C | | | | |
| Path cost | 0 | 5 | 3 | 2 | | | | |

State Space Graph | Frontier / Reached | Search Tree

## G) Use Bidirectional A* Search



# Bidirectional A*

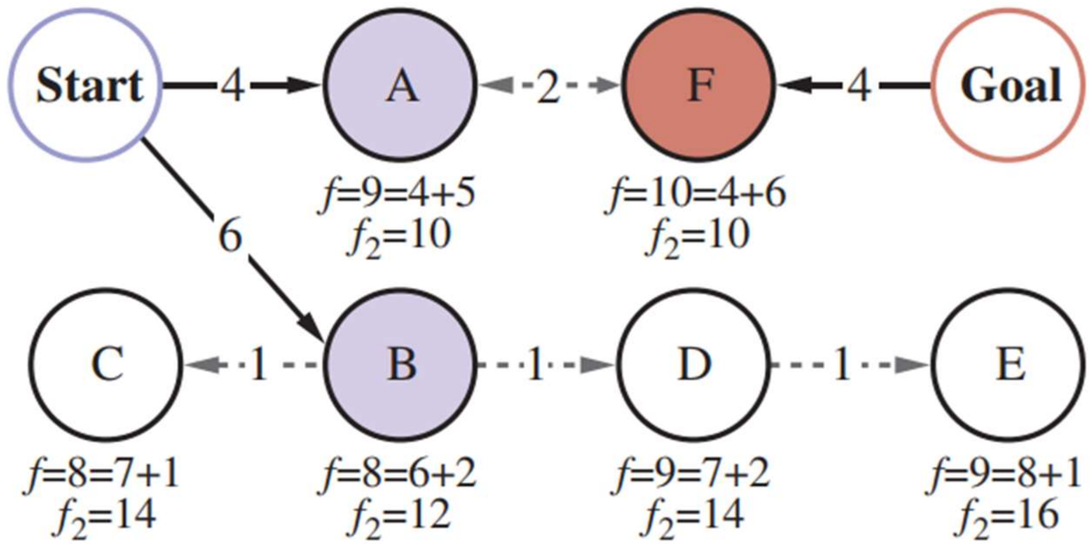



**Figure 3.24** Bidirectional search maintains two frontiers: on the left, nodes A and B are successors of Start; on the right, node F is an inverse successor of Goal. Each node is labeled with $f = g + h$ values and the $f_2 = \max(2g, g + h)$ value. (The $g$ values are the sum of the action costs as shown on each arrow; the $h$ values are arbitrary and cannot be derived from anything in the figure.) The optimal solution, Start-A-F-Goal, has cost $C^* = 4 + 2 + 4 = 10$, so that means that a meet-in-the-middle bidirectional algorithm should not expand any node with $g > \frac{C^*}{2} = 5$; and indeed the next node to be expanded would be A or F (each with $g = 4$), leading us to an optimal solution. If we expanded the node with lowest $f$ cost first, then B and C would come next, and D and E would be tied with A, but they all have $g > \frac{C^*}{2}$ and thus are never expanded when $f_2$ is the evaluation function.

# Summary

# Environment Assumptions

## "Simple Environment":

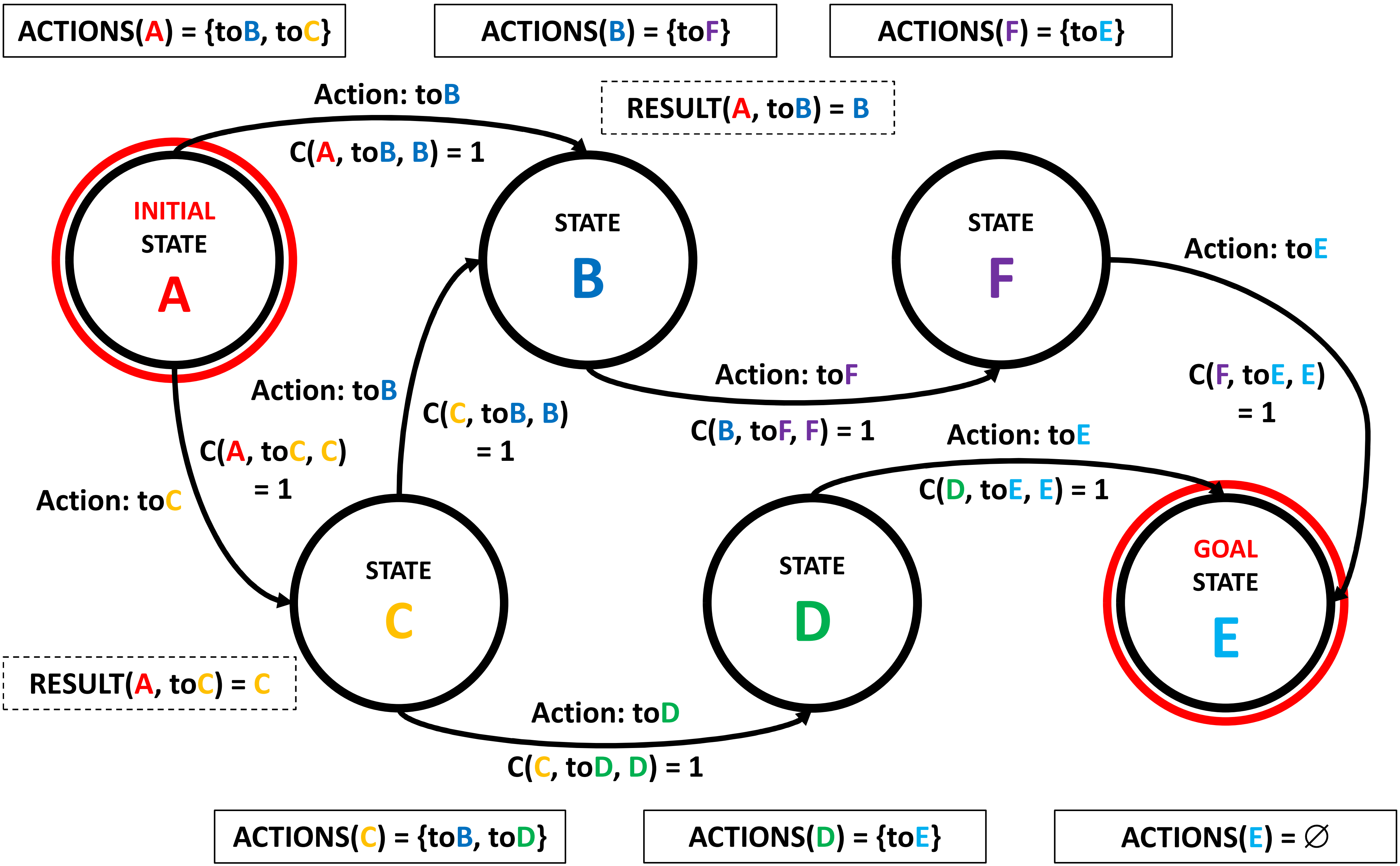
- Fully observable
- Single agent
- Deterministic
- Static
- Episodic or sequential
- Discrete
- Known to the agent

# Defining Search Problem

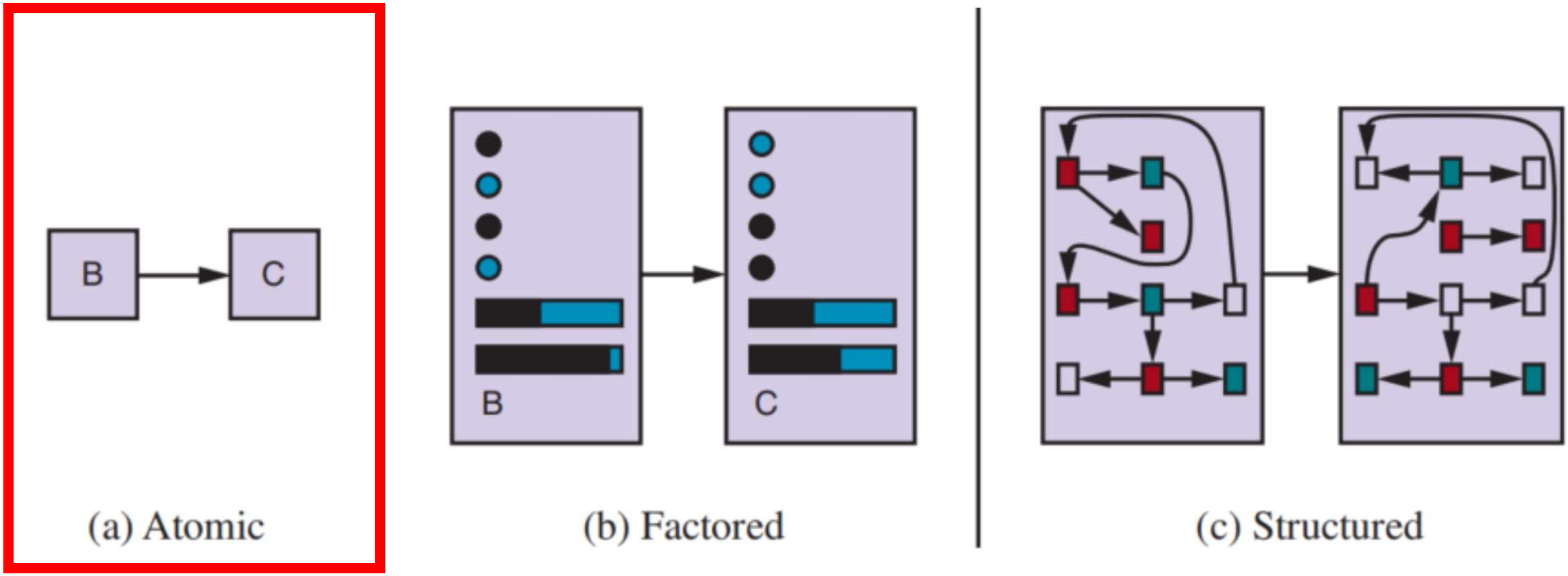
**Before agent searching, formulate a well-formed problem:**

- **Define a set of possible states: State Space**
- **Specify Initial State**
- **Specify Goal State(s) (there can be multiple)**
- **Define a FINITE set of possible Actions for EACH state in the State Space**
- **Come up with a Transition Model which describes what each action does**
- **Specify the Action Cost Function: a function that gives the cost of applying action *a* in state *s***

# State Space: A Graph



# States: Usually Atomic Representation



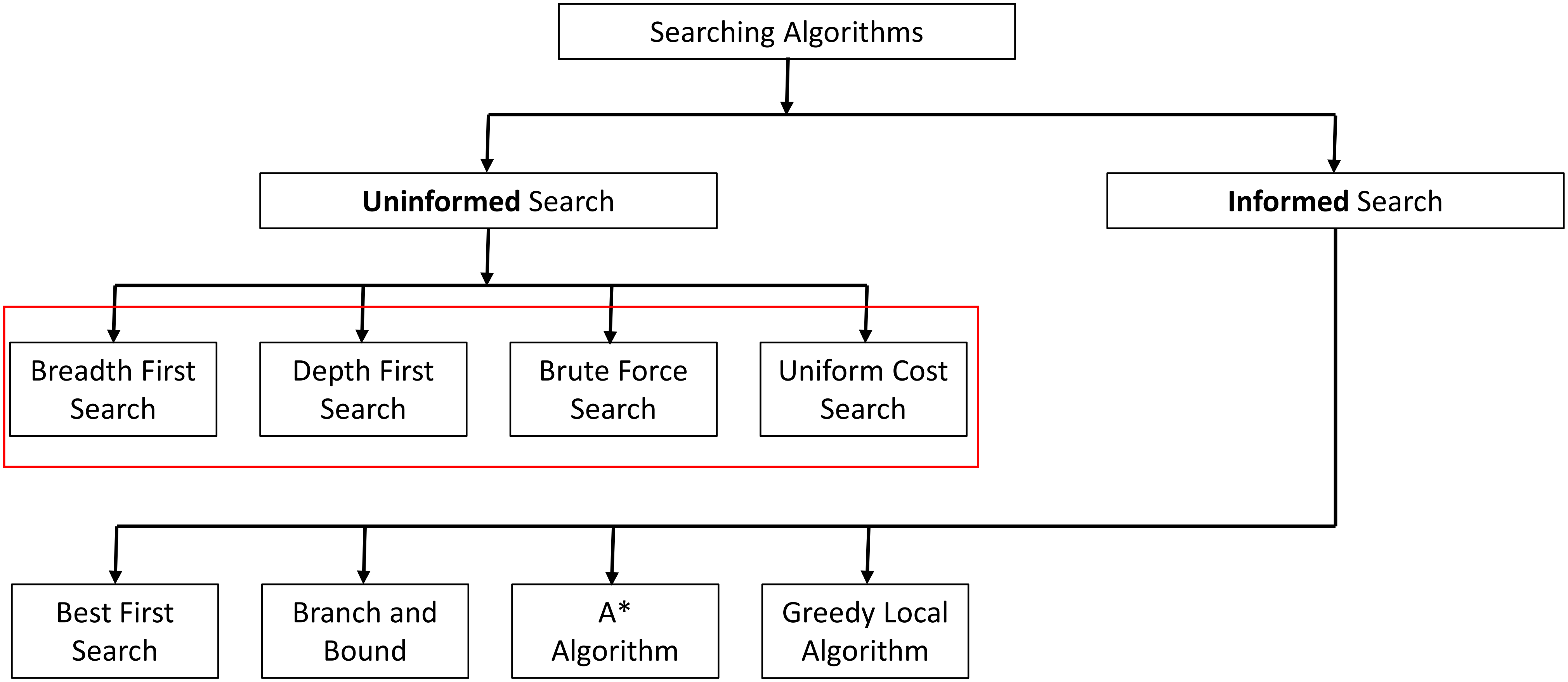

- **Atomic: state representation has NO internal structure**
- **Factored: state representation includes fixed attributes (which can have values)**
- **Structured: state representation includes objects and their relationships**

# Judging Searching Performance

Search algorithms can be judged by:

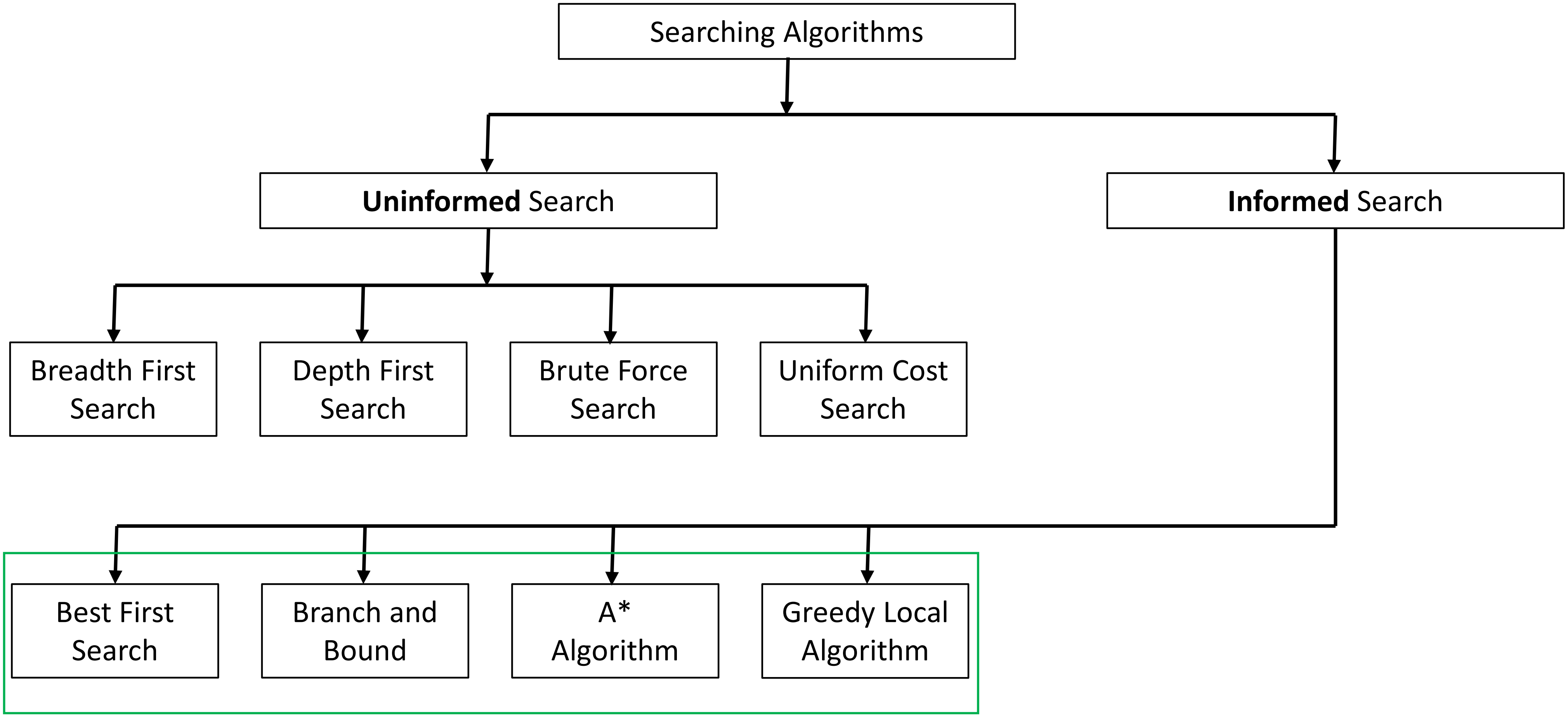
- **Completeness**: Is the algorithm guaranteed to find a solution when there is one, and to correctly report failure when there is not?
- **Cost optimality**: Does it find a solution with the lowest path cost of all solutions?
- **Time complexity**: How long does it take to find a solution? (in seconds, actions, states, etc.)
- **Space complexity**: How much memory is needed to perform the search?

# UNINFORMED Searching Algorithms




**Uninformed search:** an agent has no estimate how far it is from the goal
**Informed search:** an agent can estimate how far it is from the goal

# INFORMED Searching Algorithms




**Uninformed search:** an agent has no estimate how far it is from the goal
**Informed search:** an agent can estimate how far it is from the goal

# INFORMED Searching Algorithms

- **Greedy Best First Search** expands nodes with minimal $f(n) = h(n)$
  - complete, not optimal, but often efficient
- **A* Search** expands nodes with minimal $f(n) = g(n) + h(n)$
  - complete and optimal IF h(n) is admissible
  - space complexity an issue
- **Bidirectional A* Search** is sometimes more efficient than pure A* Search
- **Iterative Deepening A* (IDA*)** is an iterative deepening version of A*, and thus addresses the space complexity issue

# INFORMED Searching Algorithms

- RBFS (Recursive Best First Search) and SMA* (Simplified Memory-Bounded A*) are robust, optimal search algorithms that use limited amounts of memory. Given enough time, they can solve problems for which A* runs out of memory
- Weighted A* Search focuses the search towards the goal and expands fewer nodes at the expense of search optimality
- Beam Search puts a limit on the size of Frontier. That makes it incomplete and suboptimal. Often finds "good enough" solution and is faster than complete search

# Finding / Selecting Heuristics

**The performance of heuristic-based search depends on the quality of the heuristic function h().**

- **Dominating heuristic is better but can be costlier to compute**
- **Ideas for constructing good heuristic:**
  - **relax problem definition**
  - **store precomputed solution costs for subproblems in a pattern database**
  - **define landmarks**
  - **learn from experience with the problem class**

# “Hill Climbing” (Greedy Local) Search and Romanian Roadtrip Example

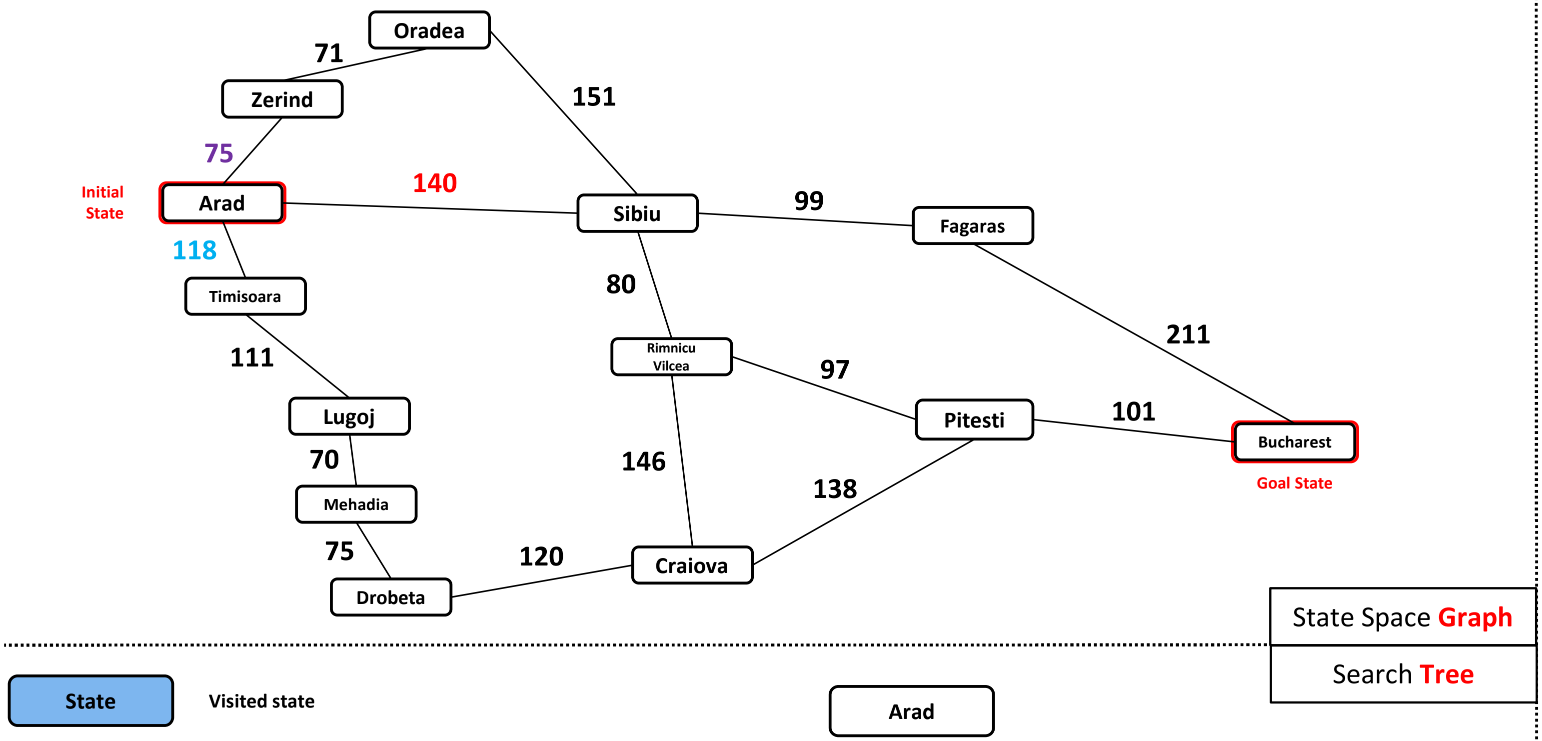
# Greedy Local: Evaluation function

**Calculate / obtain:**

f(n) = **ACTION-COST(State$_a$, toState$_n$, State$_n$)**

**A state n with minimum (or maximum) f(n) should be chosen for expansion**

# Romanian Roadtrip: Greedy Local



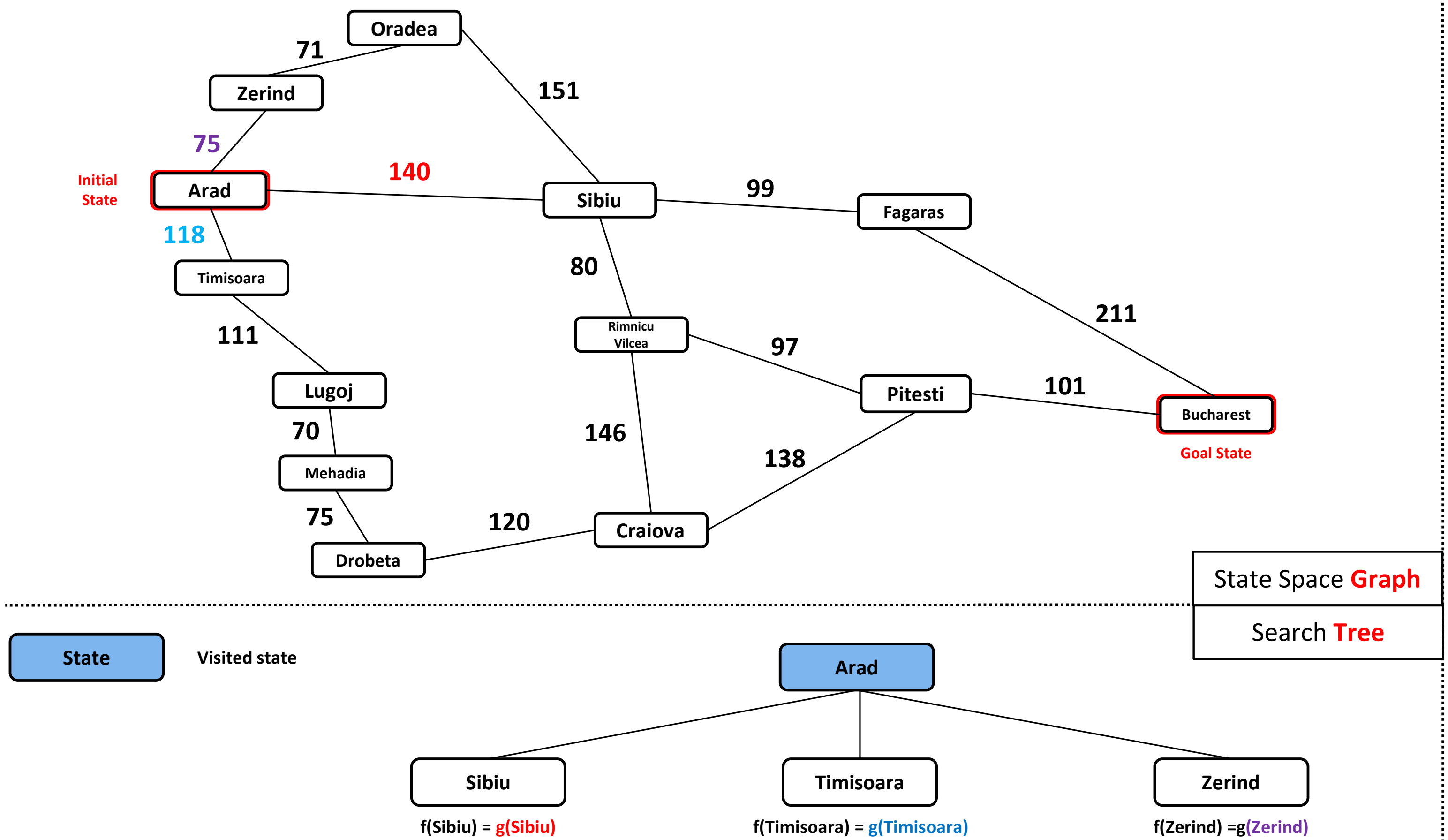

**Assumption:**
**We don't "go" to a repeated state**

**Alternatively:**

- We could go to a repeated state end
  - get stuck there

  or
  - Infinite loop

# Romanian Roadtrip: Greedy Local

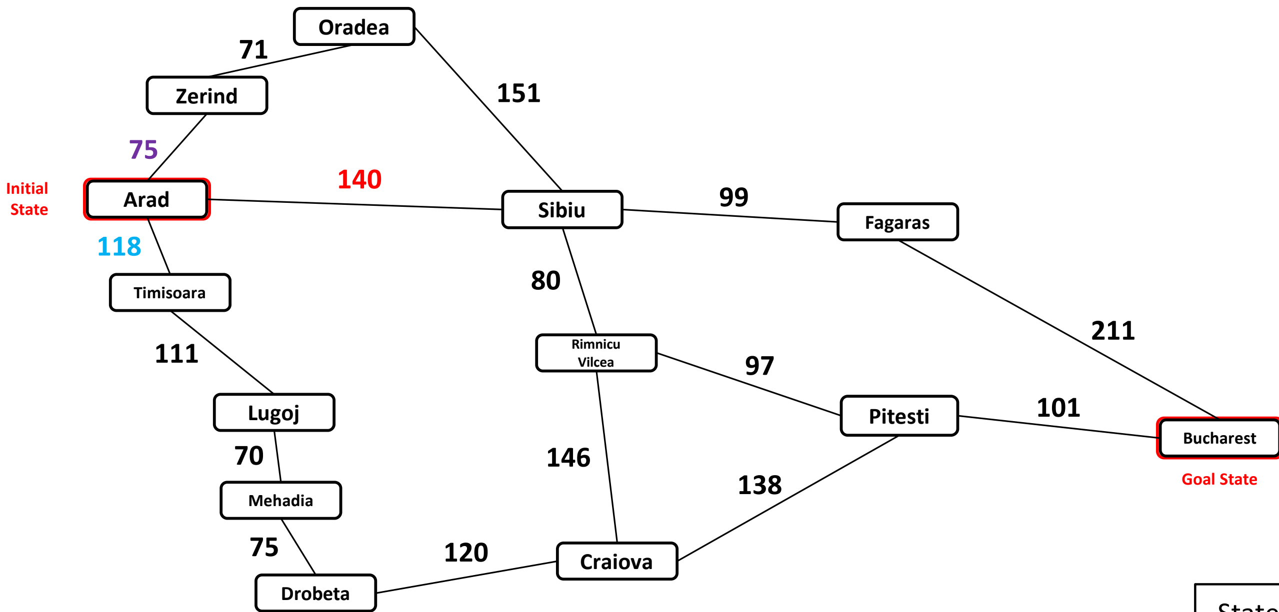


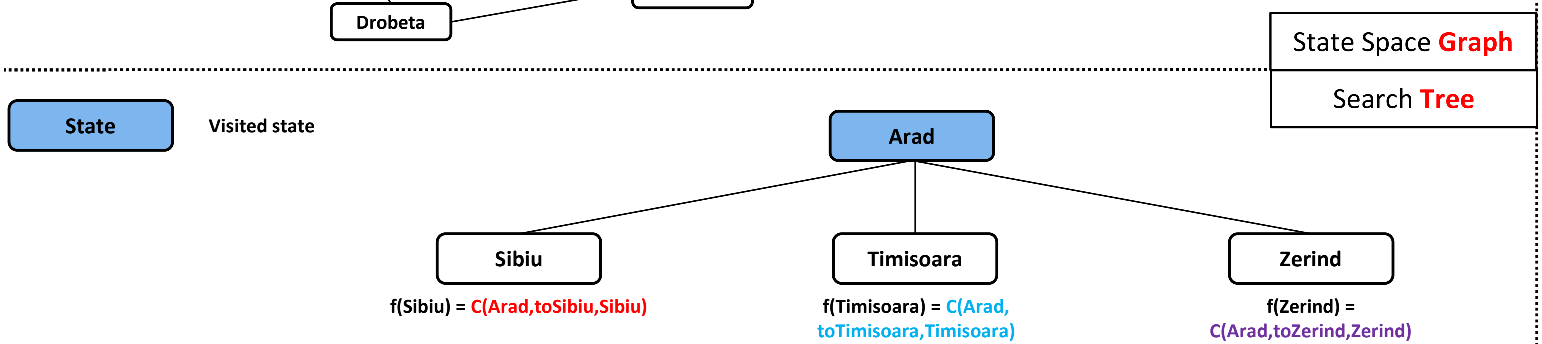

**Assumption:**
**We don't "go" to a repeated state**

**Alternatively:**
- We could go to a repeated state end
  - get stuck there

  or
  - Infinite loop

# Romanian Roadtrip: Greedy Local



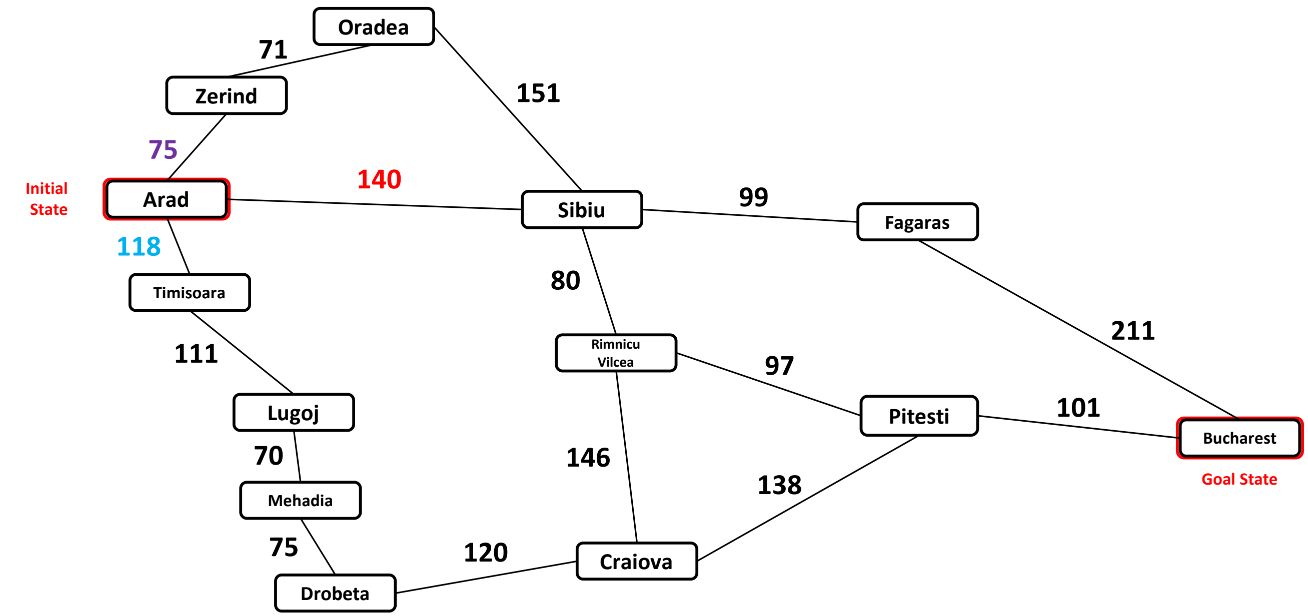



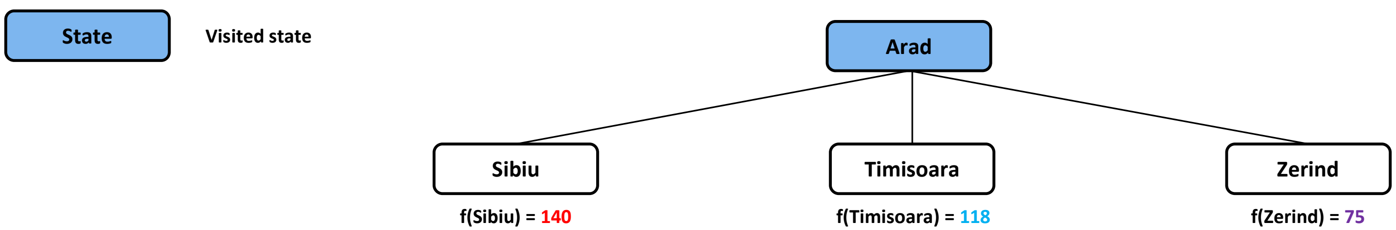

**Assumption:**
**We don't "go" to a repeated state**

**Alternatively:**
- **We could go to a repeated state end**
  - **get stuck there**

  **or**
  - **Infinite loop**

# Romanian Roadtrip: Greedy Local




**Assumption:**

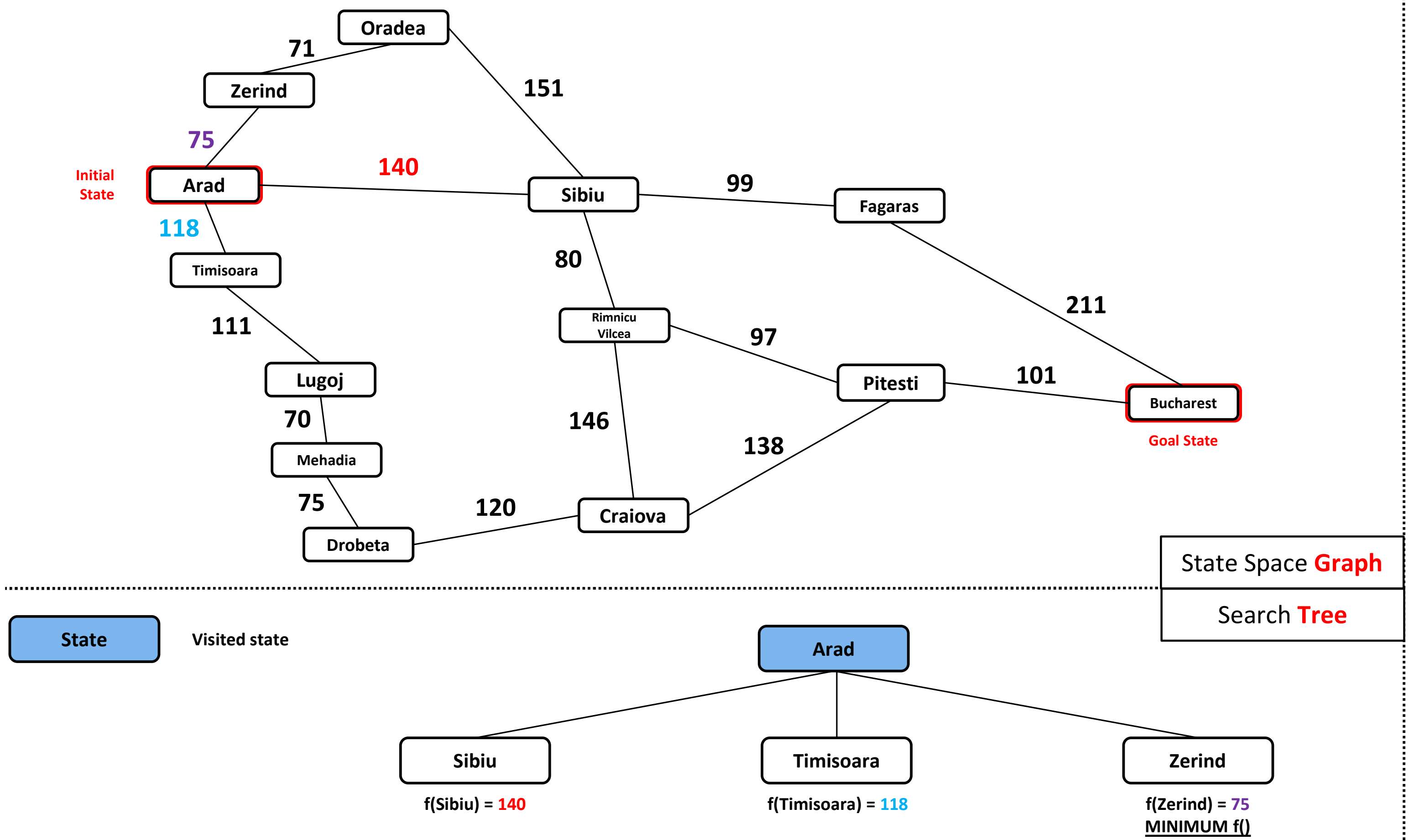
**We don't "go" to a repeated state**

**Alternatively:**

- **We could go to a repeated state end**
  - **get stuck there**

  **or**
  - **Infinite loop**

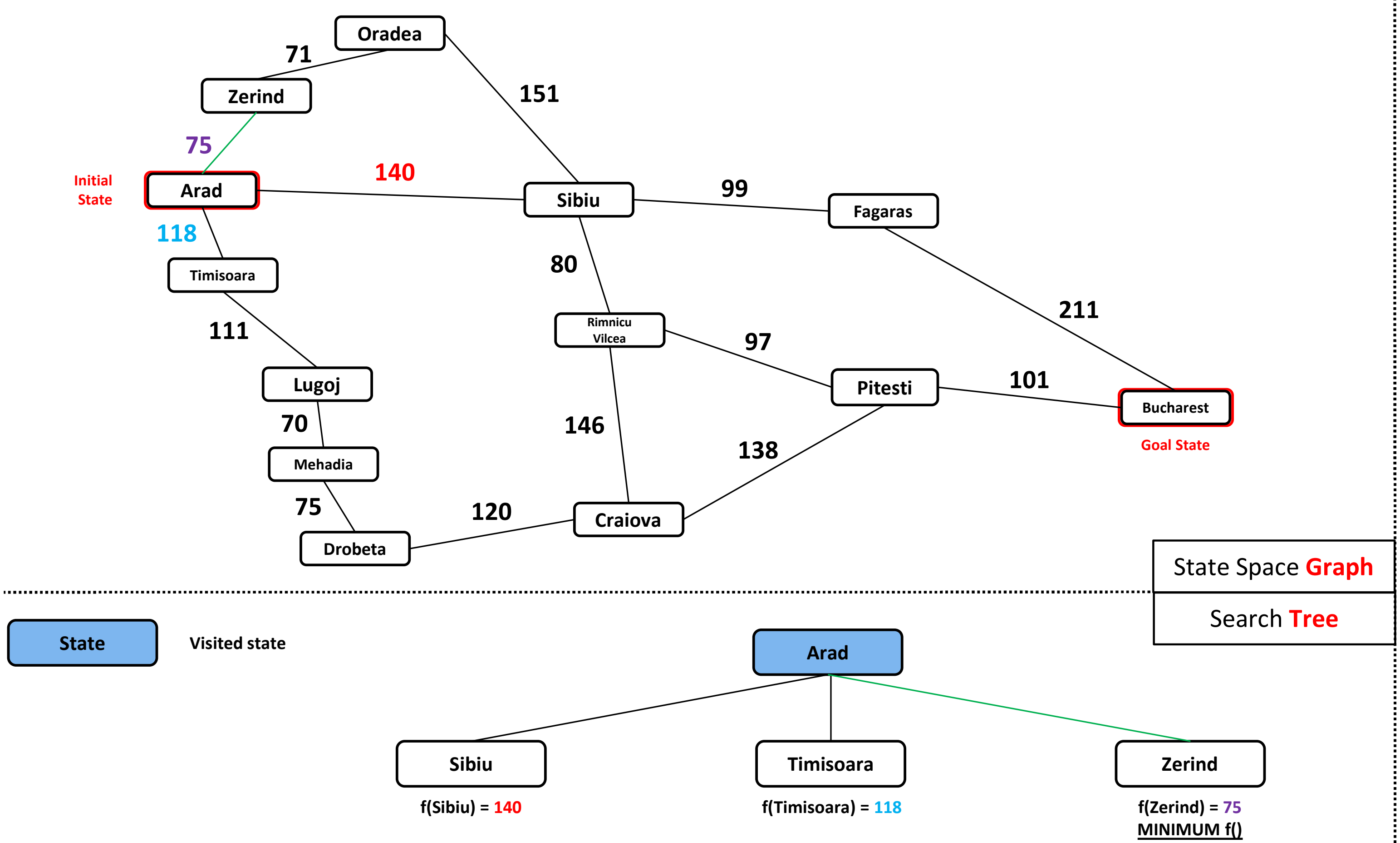
# Romanian Roadtrip: Greedy Local




**Assumption:**
**We don't "go" to a repeated state**

**Alternatively:**
- We could go to a repeated state end
  - get stuck there
  
  or
  - Infinite loop

# Romanian Roadtrip: Greedy Local

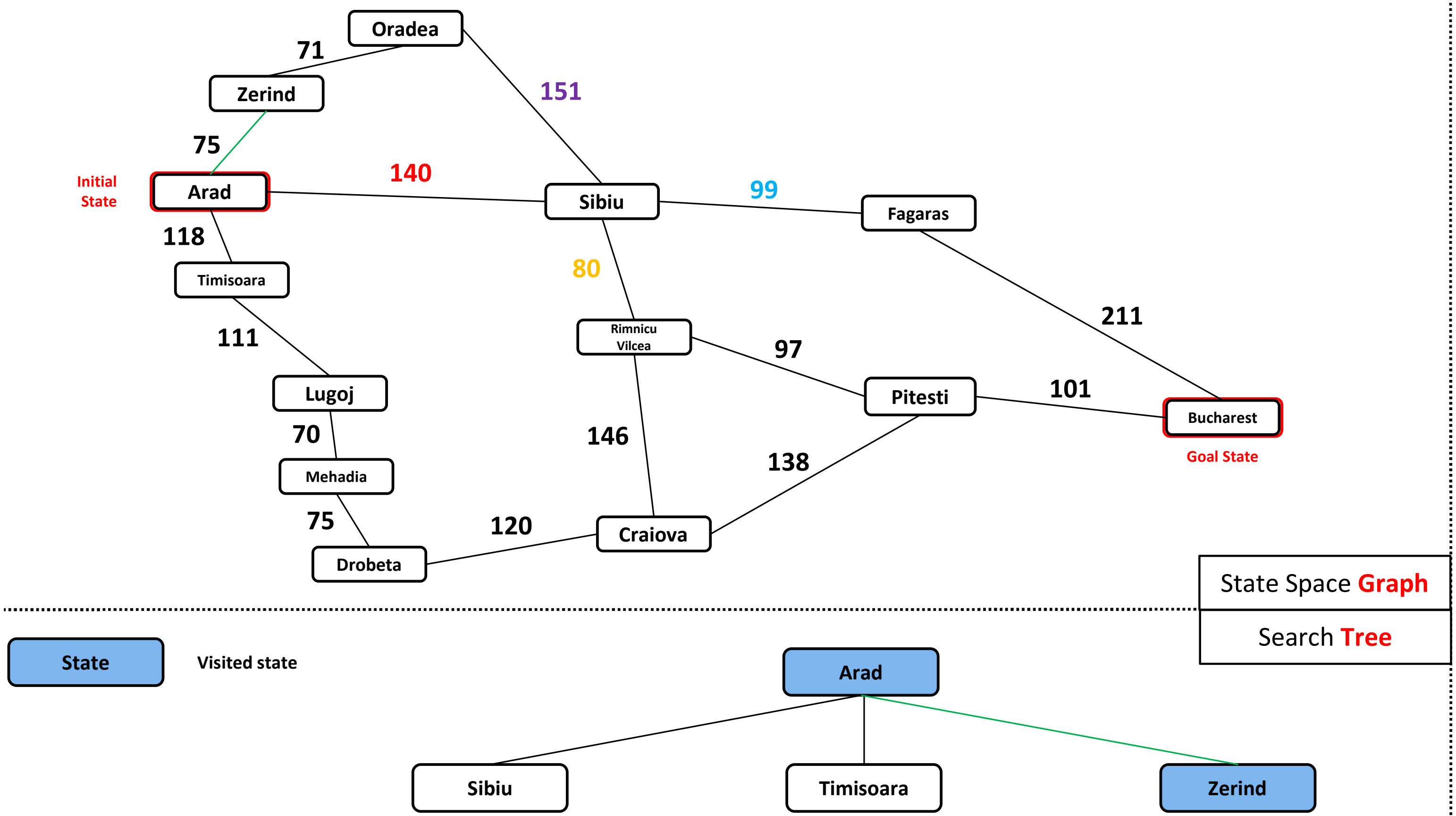



**Assumption:**
**We don't "go" to a repeated state**

**Alternatively:**
- We could go to a repeated state end
  - get stuck there

  or
  - Infinite loop

# Romanian Roadtrip: Greedy Local

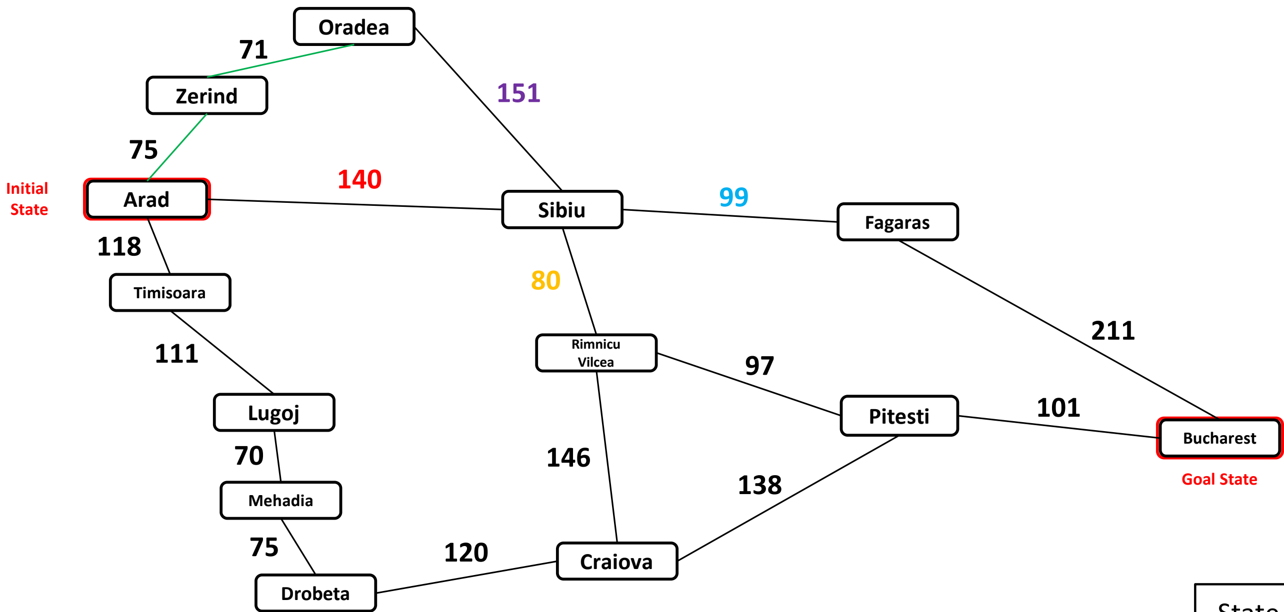



**Assumption:**
**We don't "go" to a repeated state**

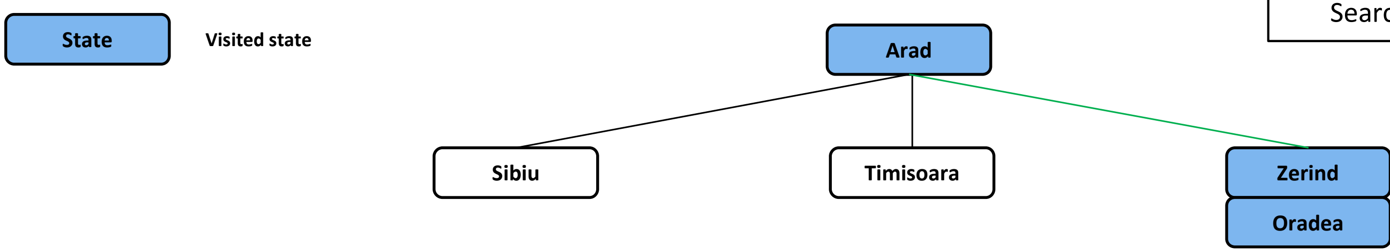
**Alternatively:**
- We could go to a repeated state end
  - get stuck there

  or
  - Infinite loop

# Romanian Roadtrip: Greedy Local




| State Space **Graph** |
|---|
| Search **Tree** |



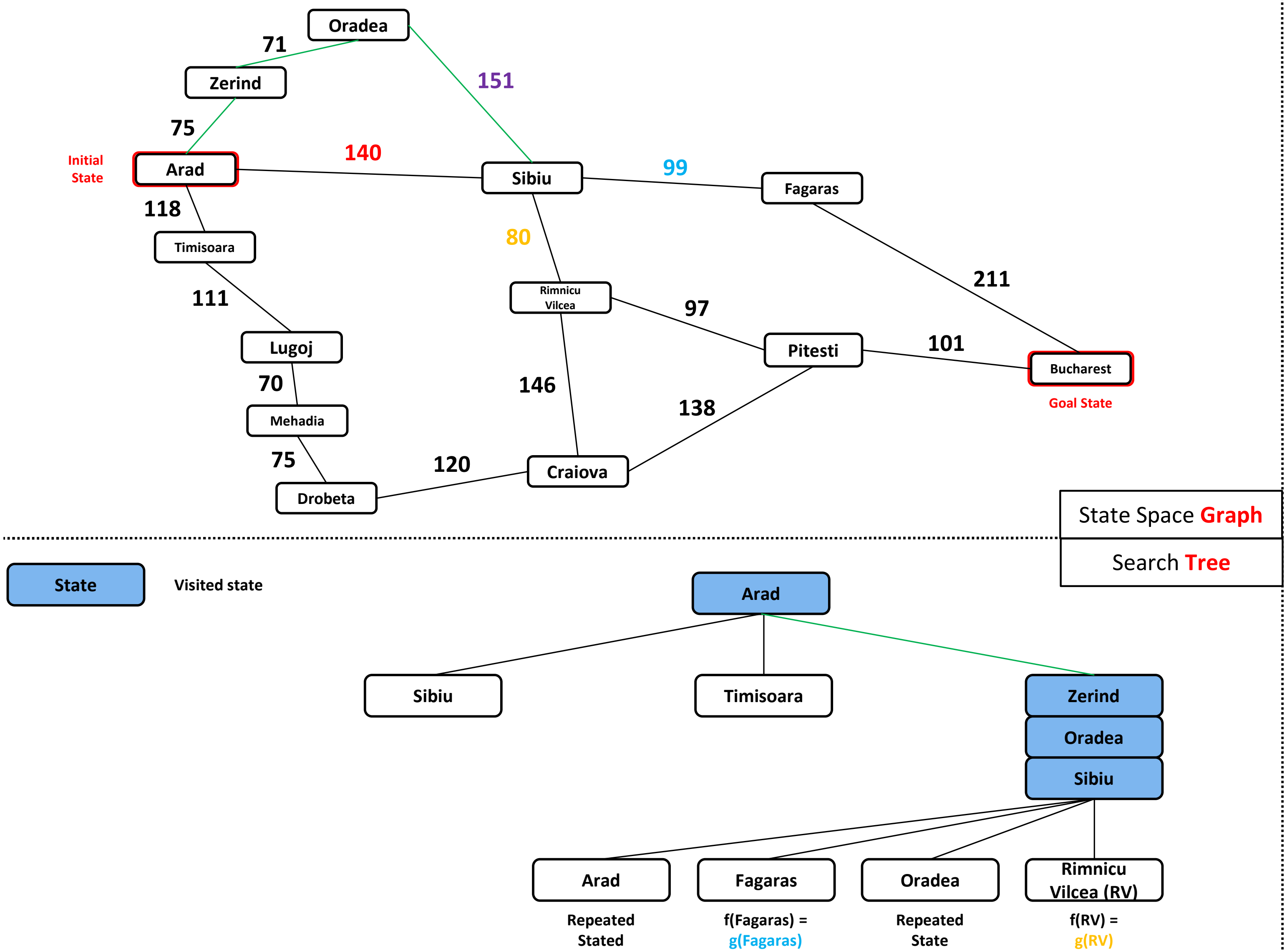

**Assumption:**
**We don't "go" to a repeated state**

**Alternatively:**

- We could go to a repeated state end
  - get stuck there

  or
  - Infinite loop

# Romanian Roadtrip: Greedy Local




**Assumption:**

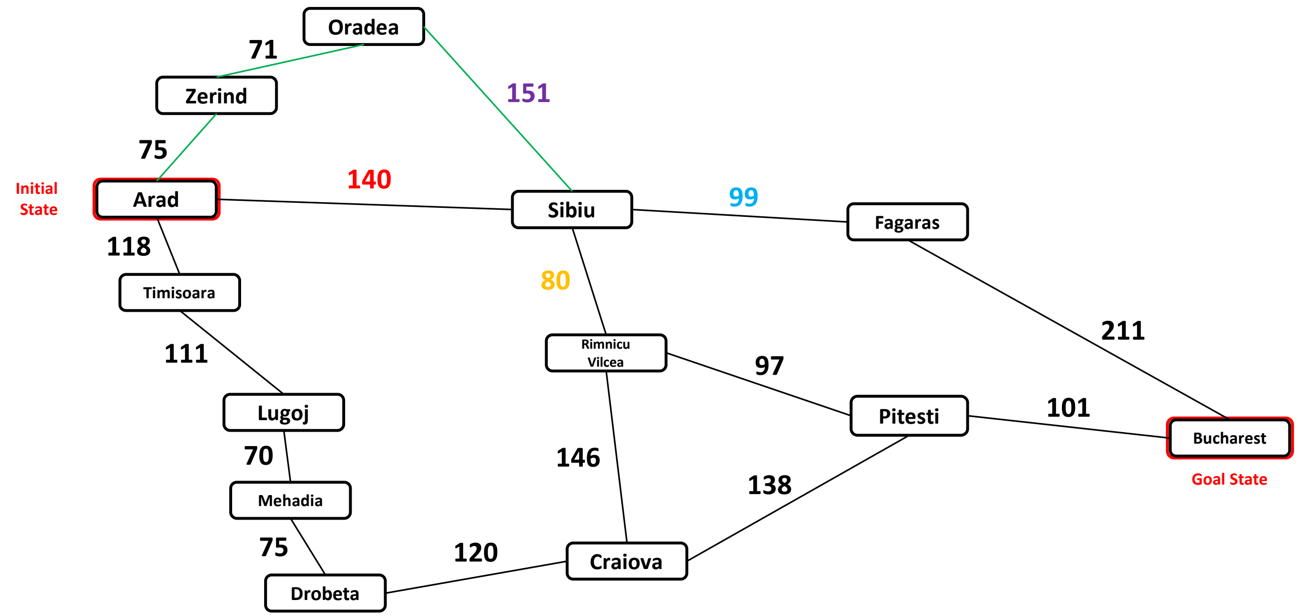
**We don't "go" to a repeated state**

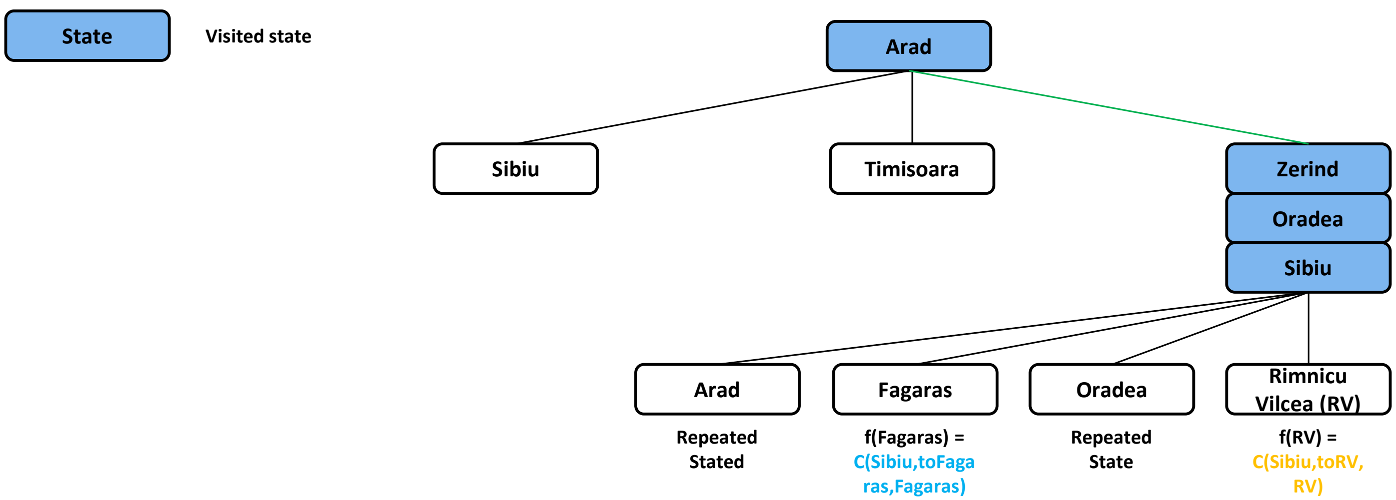
**Alternatively:**

- **We could go to a repeated state end**
  - **get stuck there**

  **or**
  - **Infinite loop**

# Romanian Roadtrip: Greedy Local




**Assumption:**

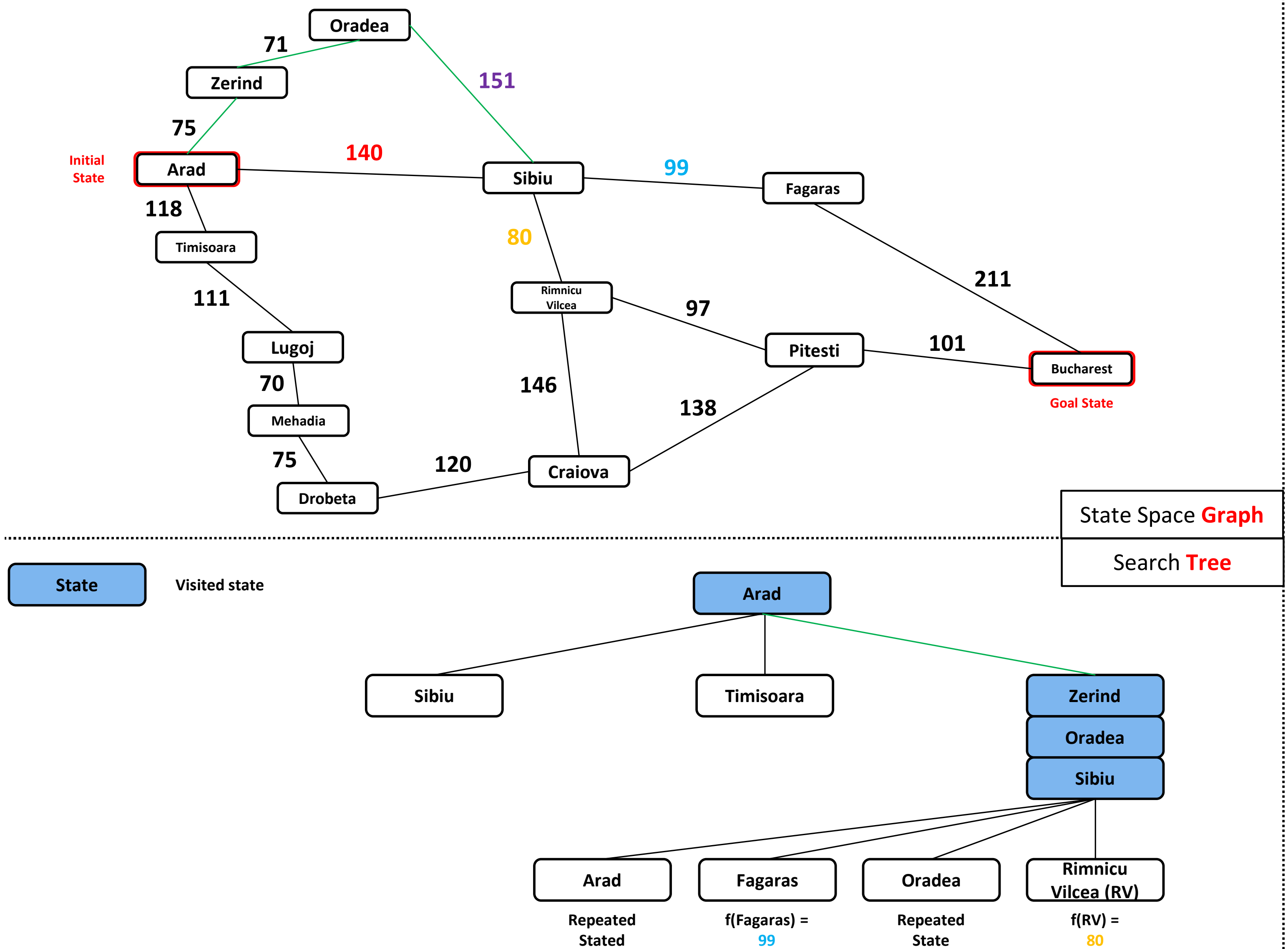
**We don't "go" to a repeated state**

**Alternatively:**

- **We could go to a repeated state end**
  - **get stuck there**

  **or**
  - **Infinite loop**

# Romanian Roadtrip: Greedy Local

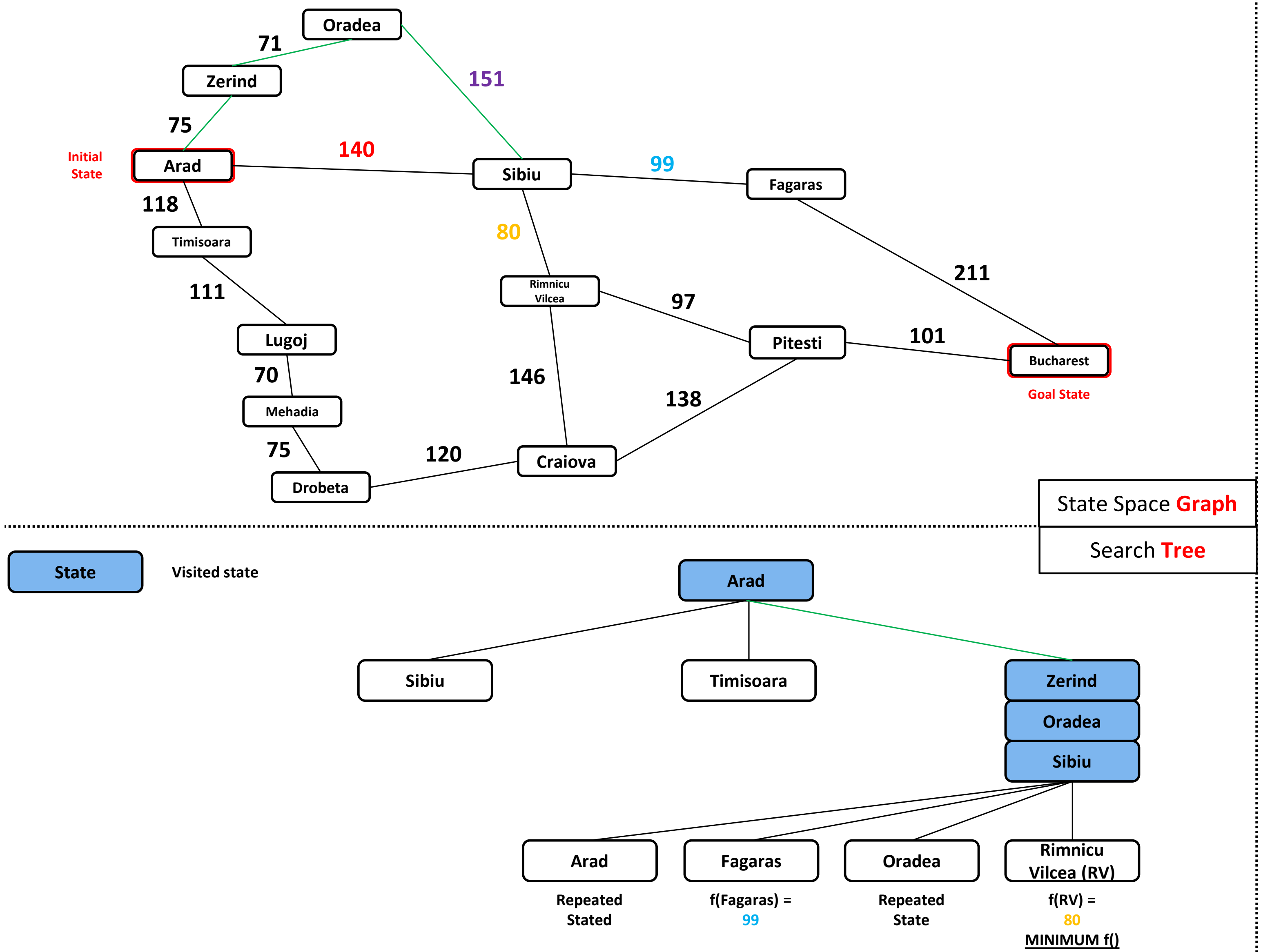



**Assumption:**
**We don't "go" to a repeated state**

**Alternatively:**
- We could go to a repeated state end
  - get stuck there

  or
  - Infinite loop

# Romanian Roadtrip: Greedy Local




**Assumption:**
**We don't "go" to a repeated state**

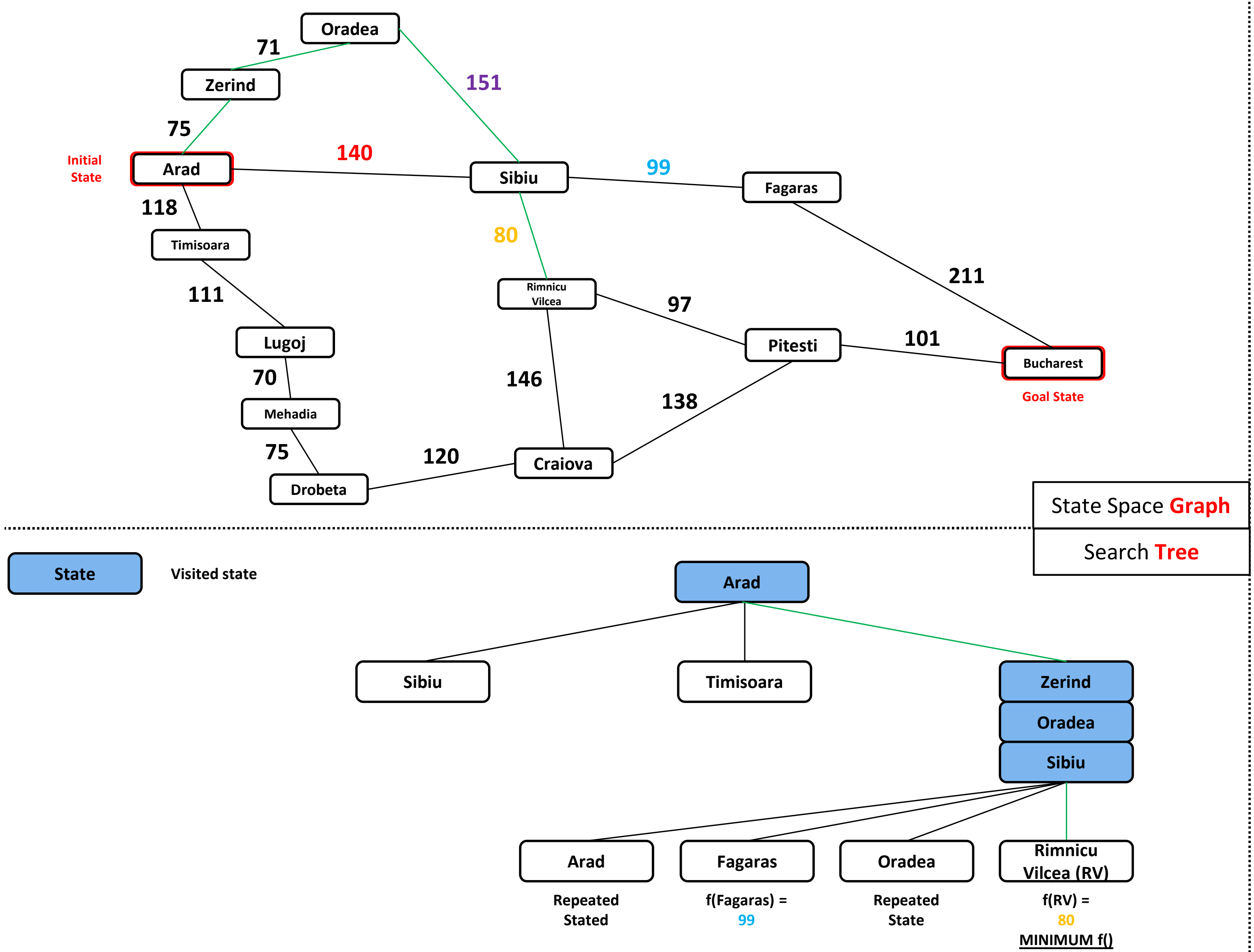
**Alternatively:**

- **We could go to a repeated state end**
  - **get stuck there**

  **or**

  - **Infinite loop**

# Romanian Roadtrip: Greedy Local

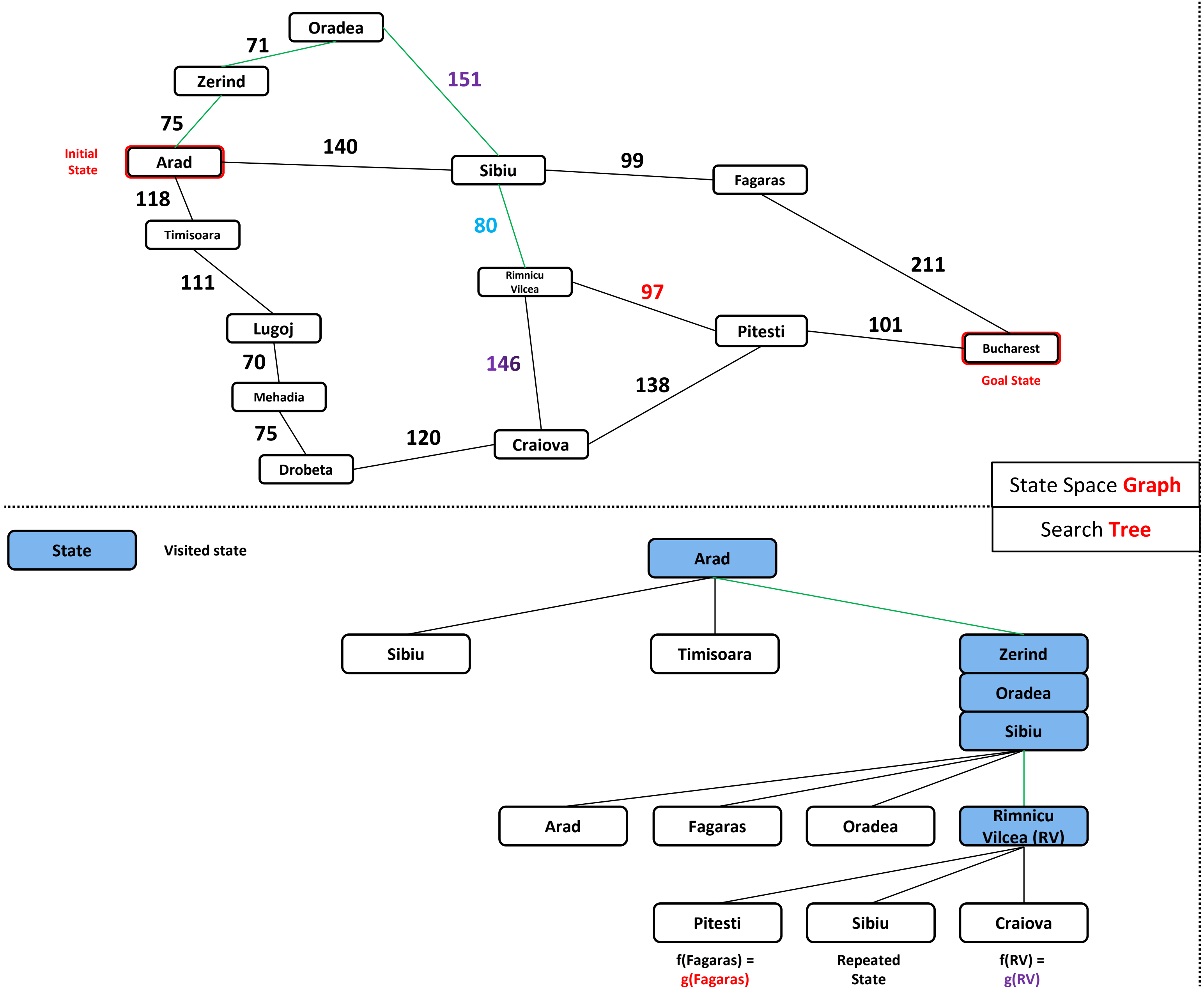



**Assumption:**
**We don't "go" to a repeated state**

**Alternatively:**
- **We could go to a repeated state end**
  - **get stuck there**

  **or**
  - **Infinite loop**

# Romanian Roadtrip: Greedy Local

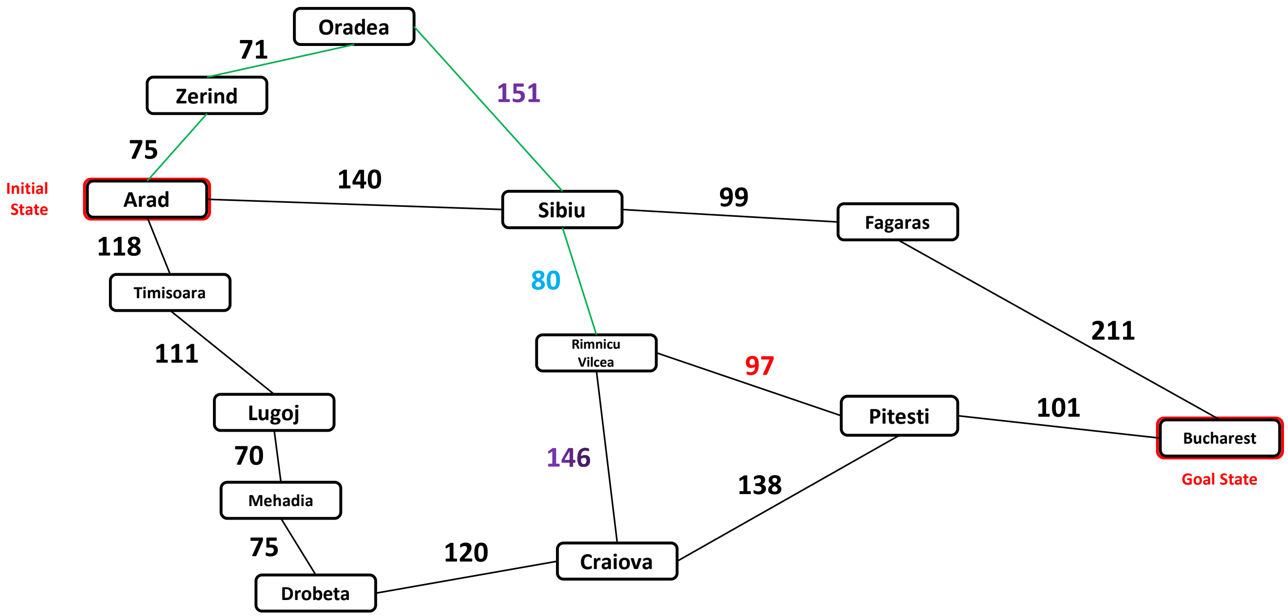


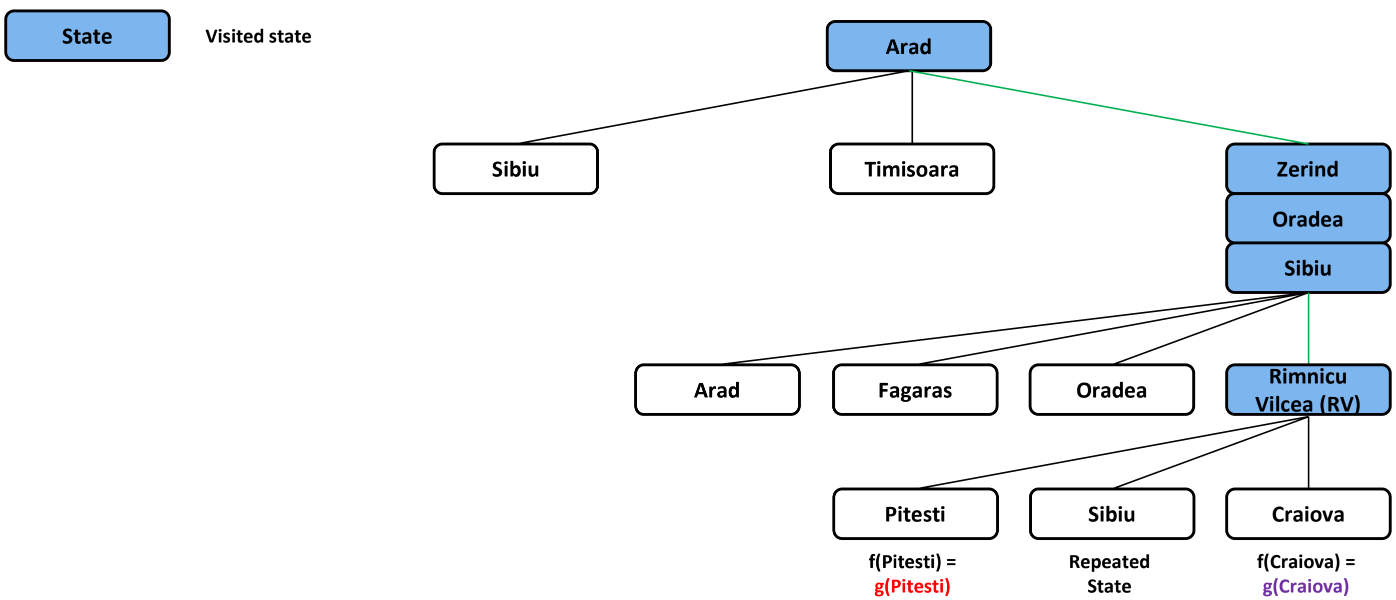

**Assumption:**
**We don't "go" to a repeated state**

**Alternatively:**
- We could go to a repeated state end
  - get stuck there

  or
  - Infinite loop

# Romanian Roadtrip: Greedy Local




**Assumption:**

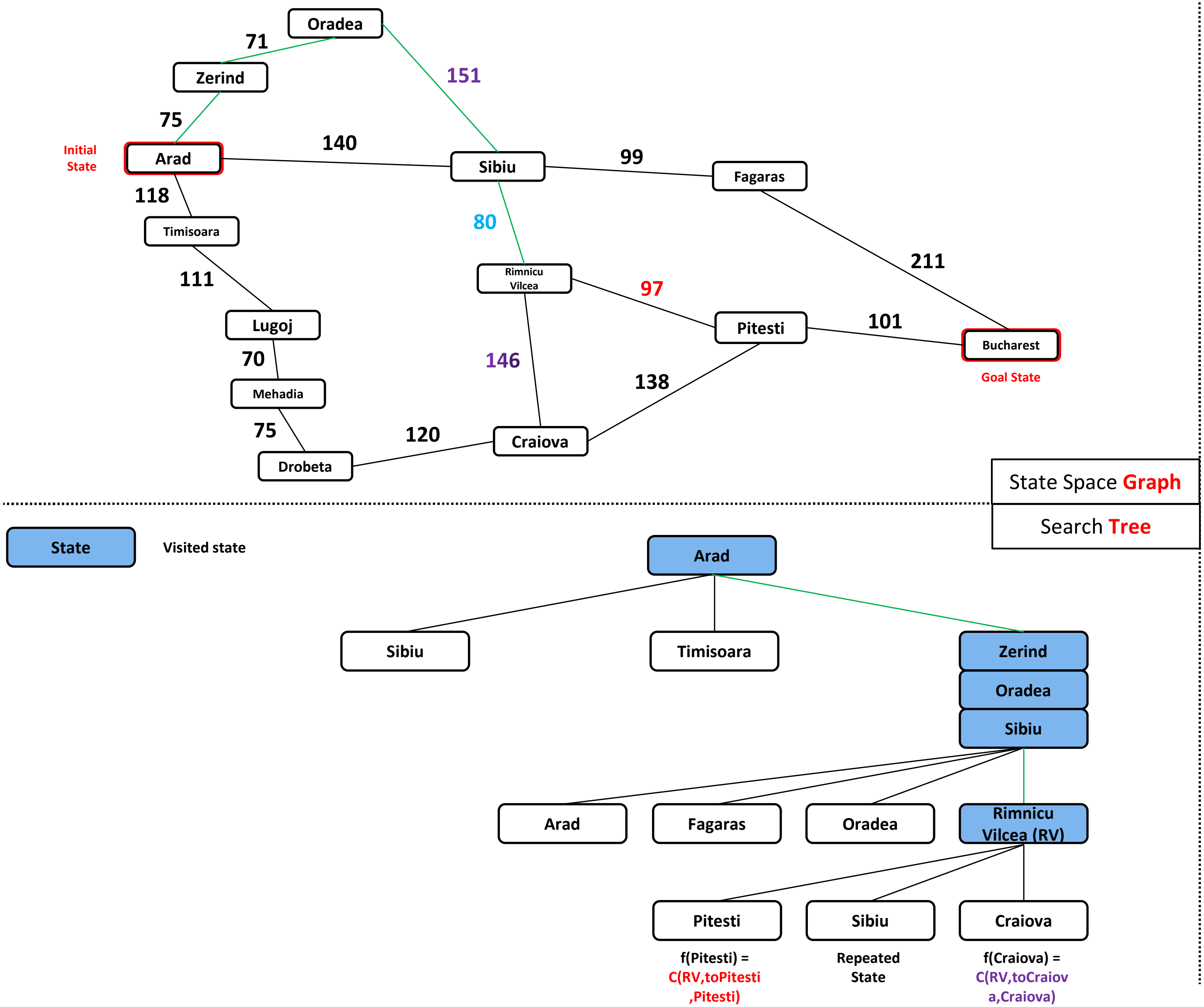
**We don't "go" to a repeated state**

**Alternatively:**

- **We could go to a repeated state end**
  - **get stuck there**

  **or**
  - **Infinite loop**

# Romanian Roadtrip: Greedy Local

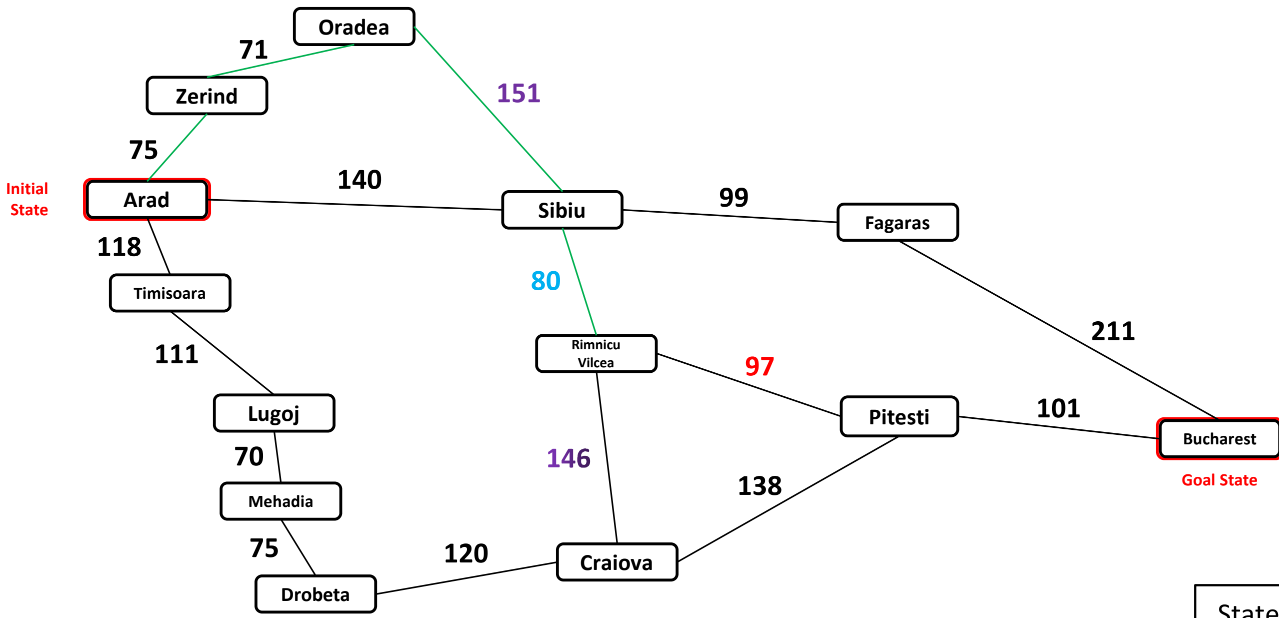


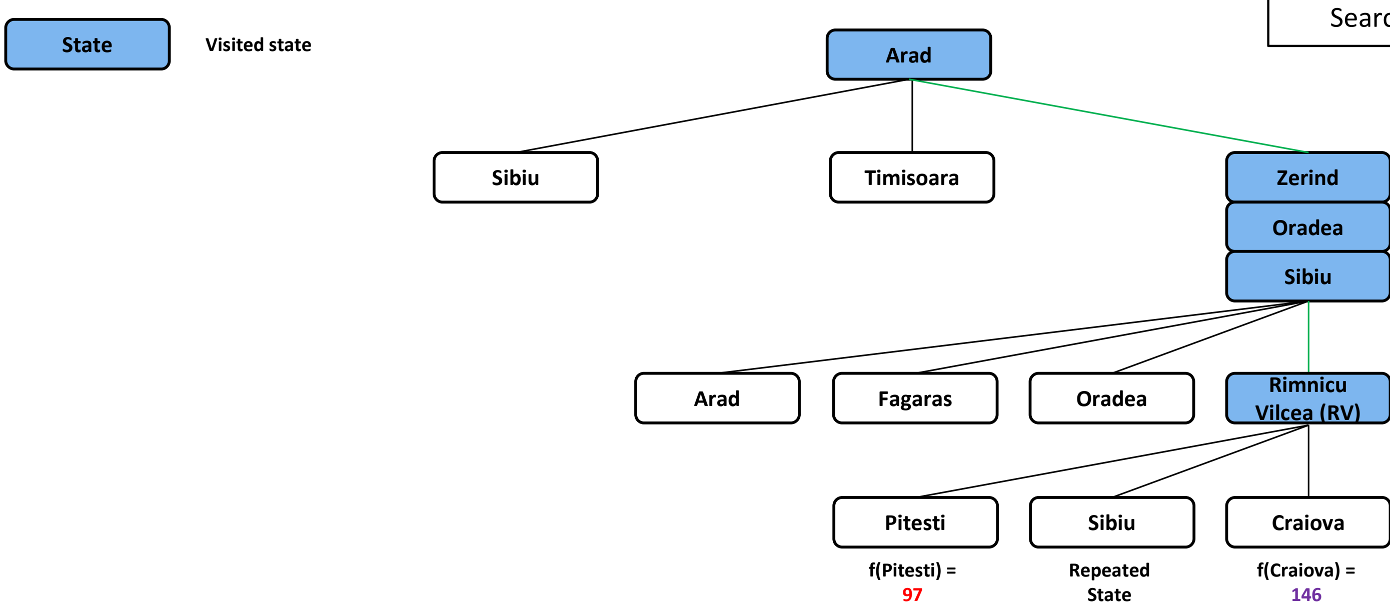

**Assumption:**
**We don't "go" to a repeated state**

**Alternatively:**
- **We could go to a repeated state end**
  - **get stuck there**

  **or**
  - **Infinite loop**

# Romanian Roadtrip: Greedy Local

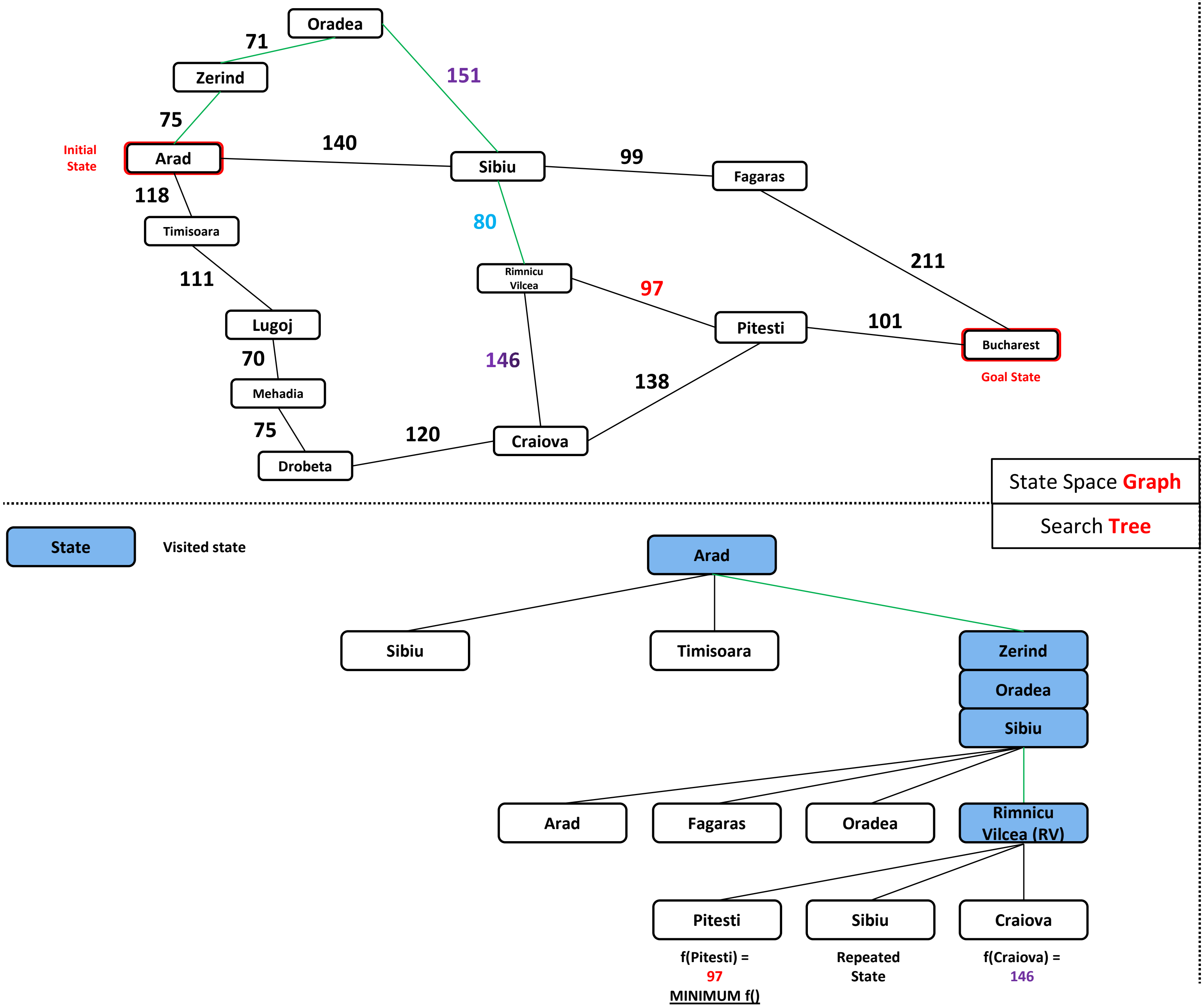



**Assumption:**
**We don't "go" to a repeated state**

**Alternatively:**
- We could go to a repeated state end
  - get stuck there
  
  or
  - Infinite loop

# Romanian Roadtrip: Greedy Local




**Assumption:**

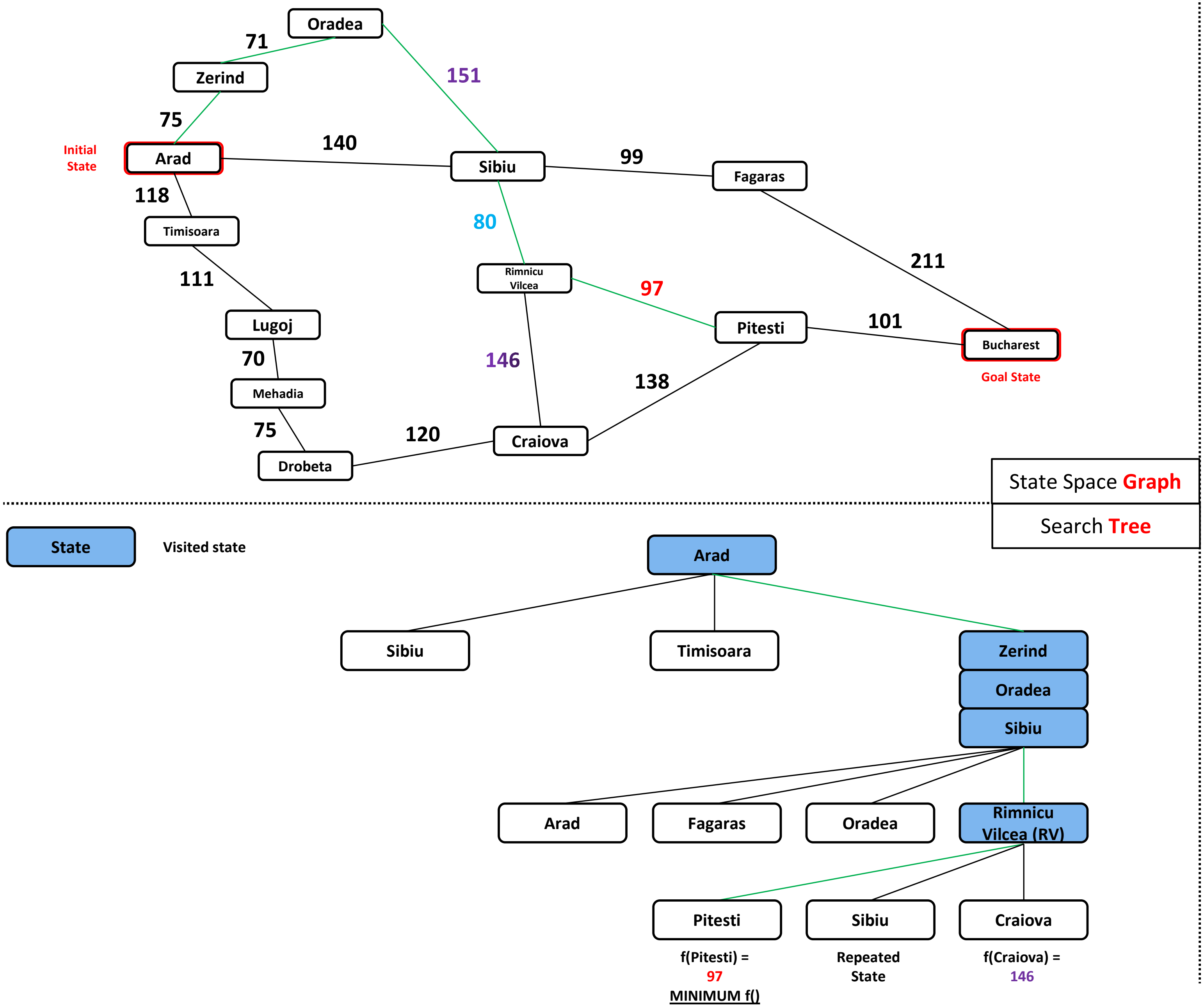
**We don't "go" to a repeated state**

**Alternatively:**

- **We could go to a repeated state end**
  - **get stuck there**

  **or**
  - **Infinite loop**

# Romanian Roadtrip: Greedy Local




**Assumption:**

**We don't "go" to a repeated state**

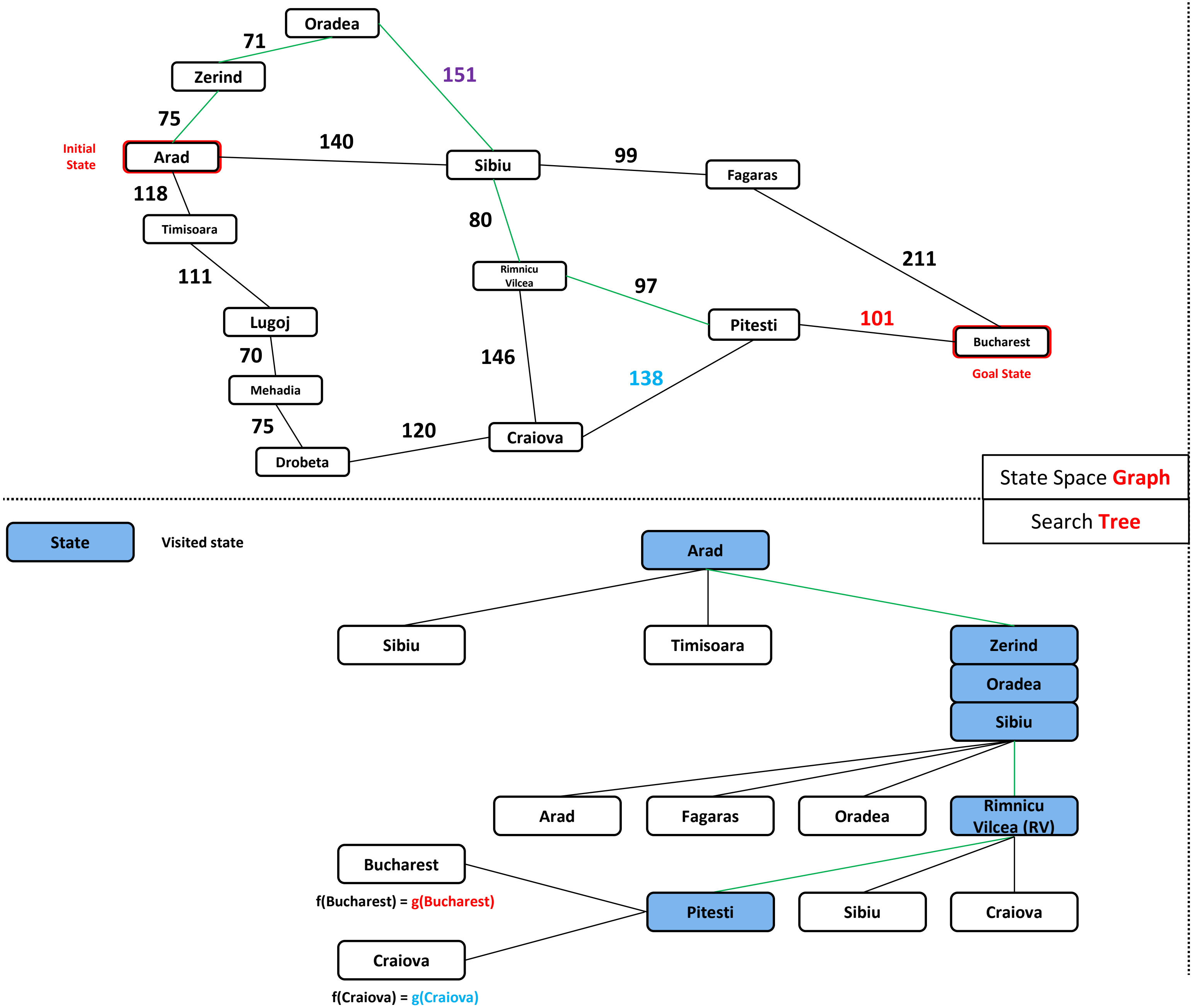
**Alternatively:**

- **We could go to a repeated state end**
  - **get stuck there**

  **or**

  - **Infinite loop**

# Romanian Roadtrip: Greedy Local

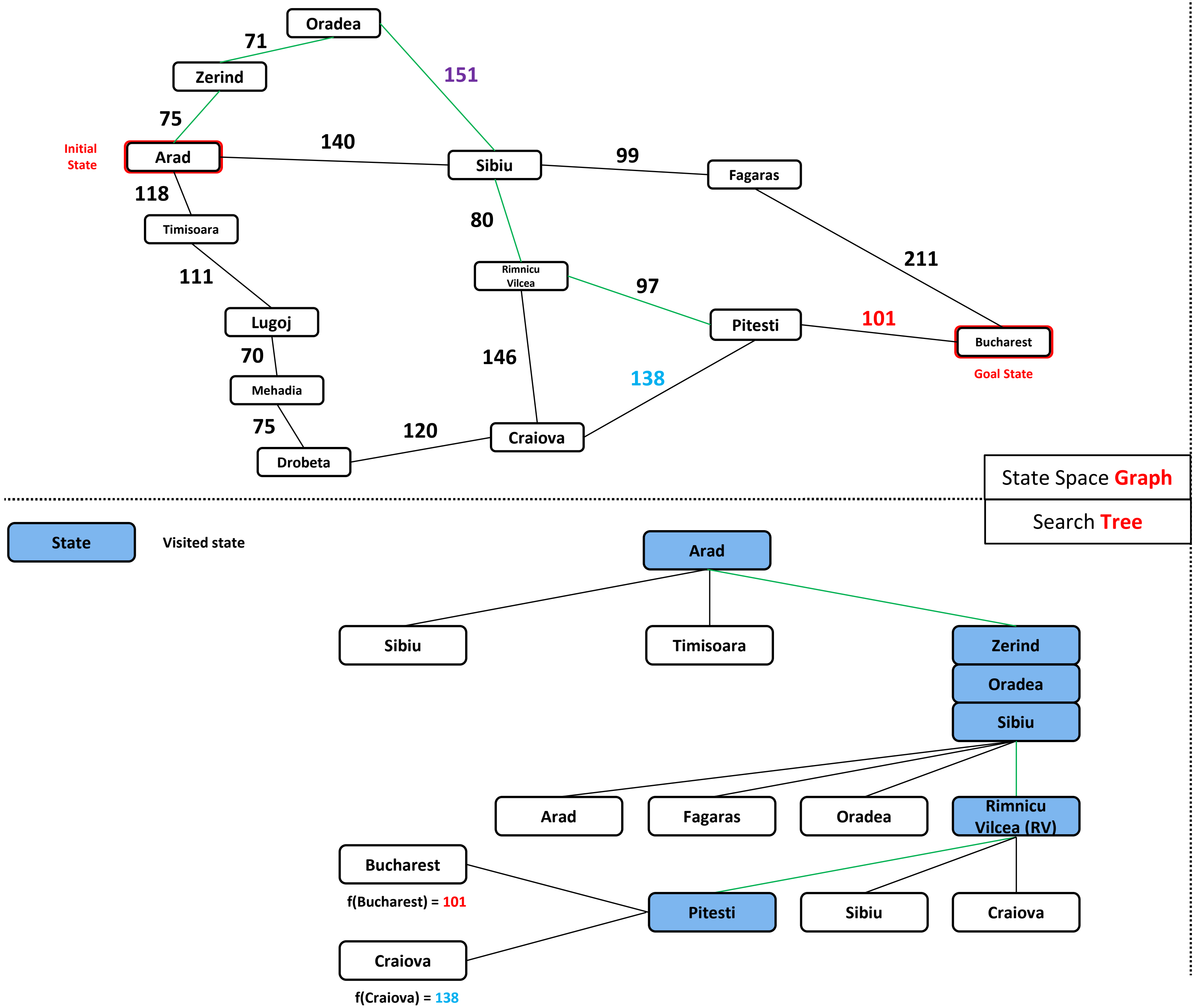



**Assumption:**
**We don't "go" to a repeated state**

**Alternatively:**
- **We could go to a repeated state end**
  - **get stuck there**

  **or**
  - **Infinite loop**

# Romanian Roadtrip: Greedy Local

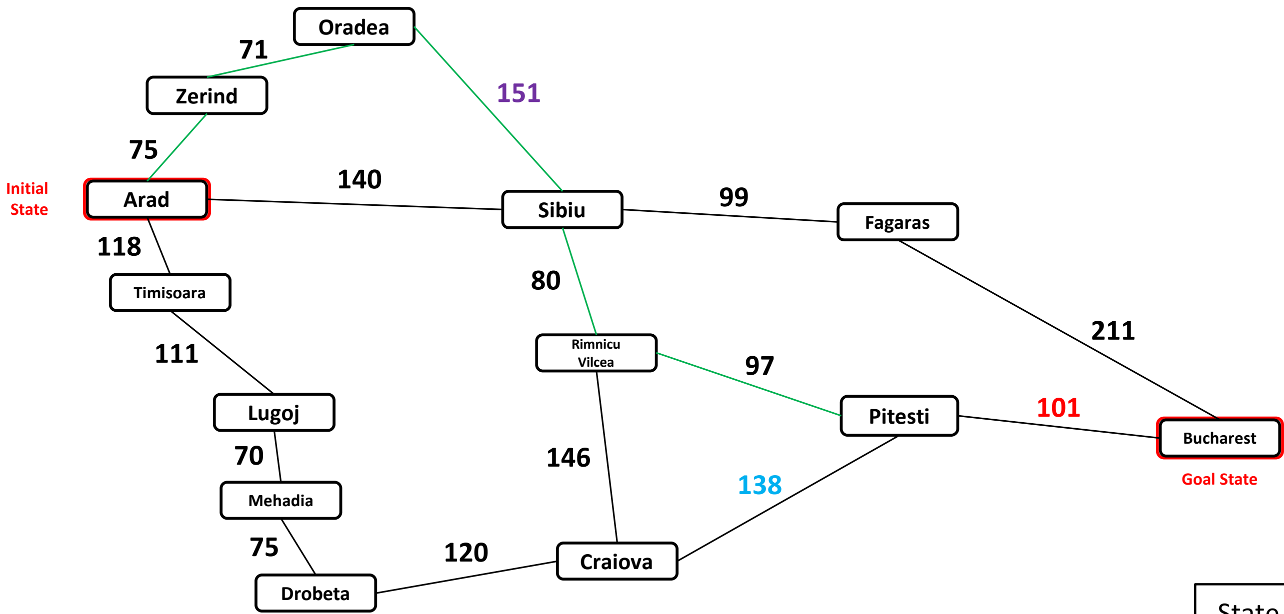



**Assumption:**
**We don't "go" to a repeated state**

**Alternatively:**
- We could go to a repeated state end
    - get stuck there

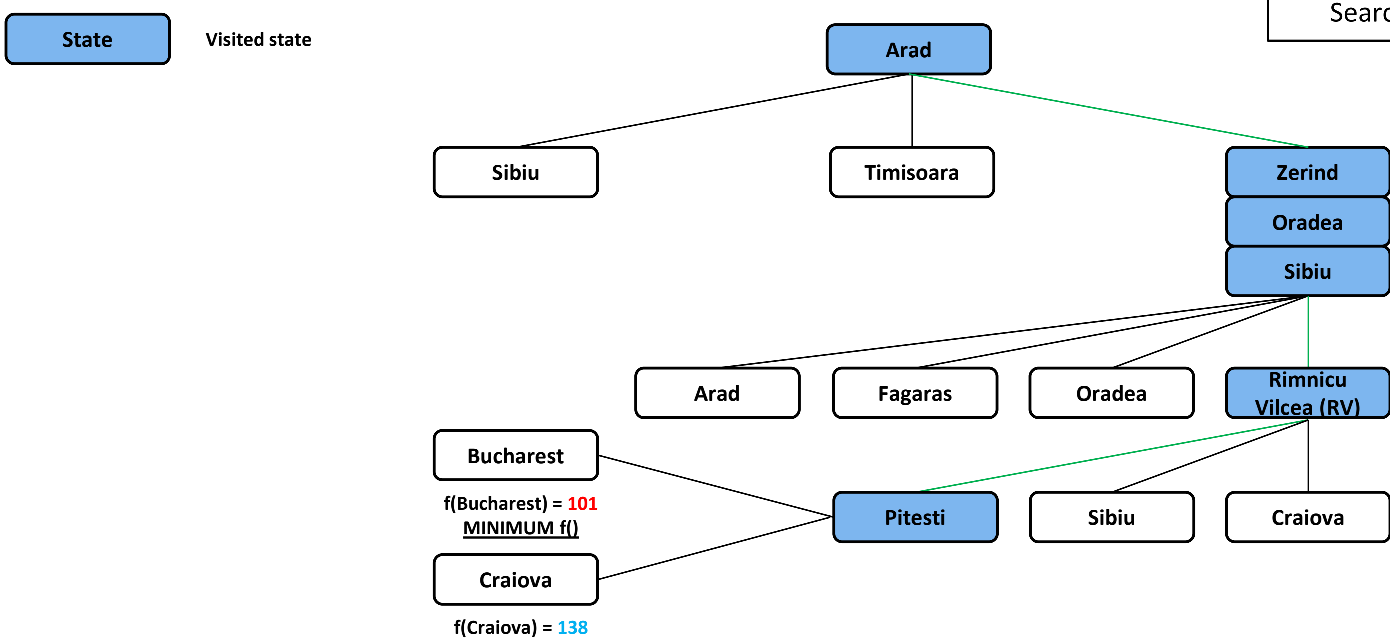
  or
    - Infinite loop

# Romanian Roadtrip: Greedy Local




State Space **Graph**

Search **Tree**




**Assumption:**

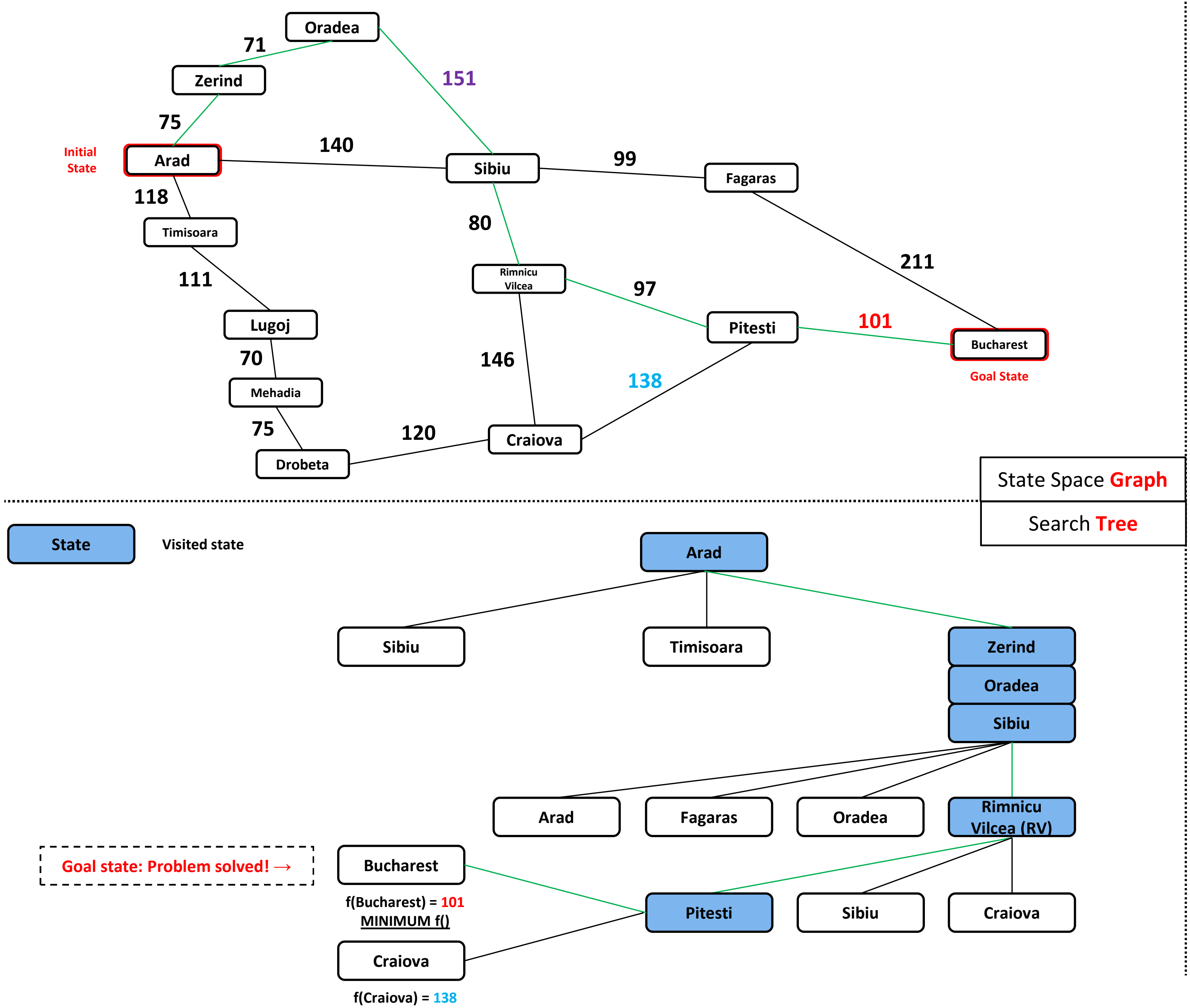
**We don't "go" to a repeated state**

**Alternatively:**

- We could go to a repeated state end
  - get stuck there

  or
  - Infinite loop

# Romanian Roadtrip: Greedy Local




**Assumption:**

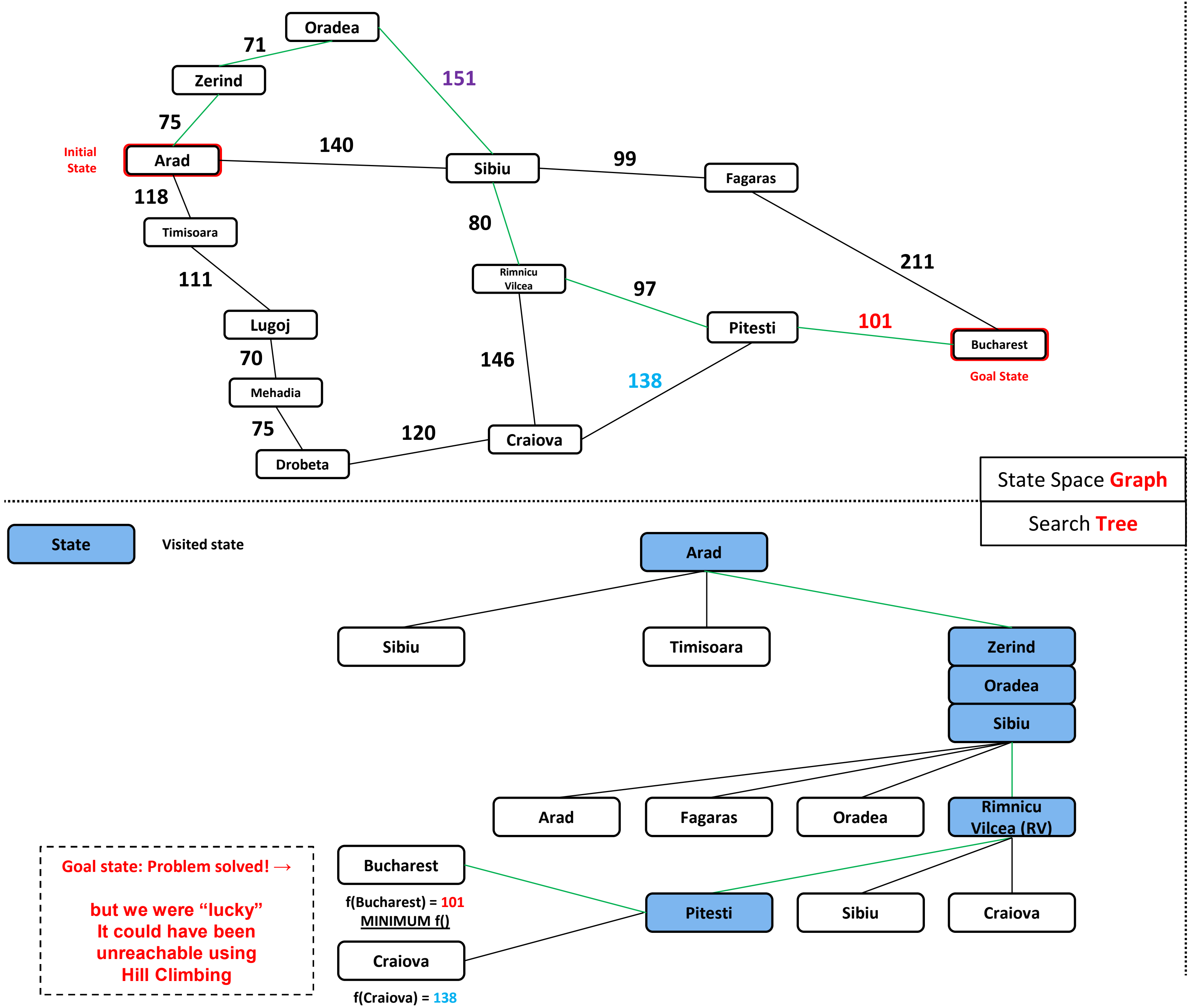
**We don't "go" to a repeated state**

**Alternatively:**

- **We could go to a repeated state end**
  - **get stuck there**

  **or**
  - **Infinite loop**

# Romanian Roadtrip: Greedy Local




**Assumption:**
**We don't "go" to a repeated state**

**Alternatively:**

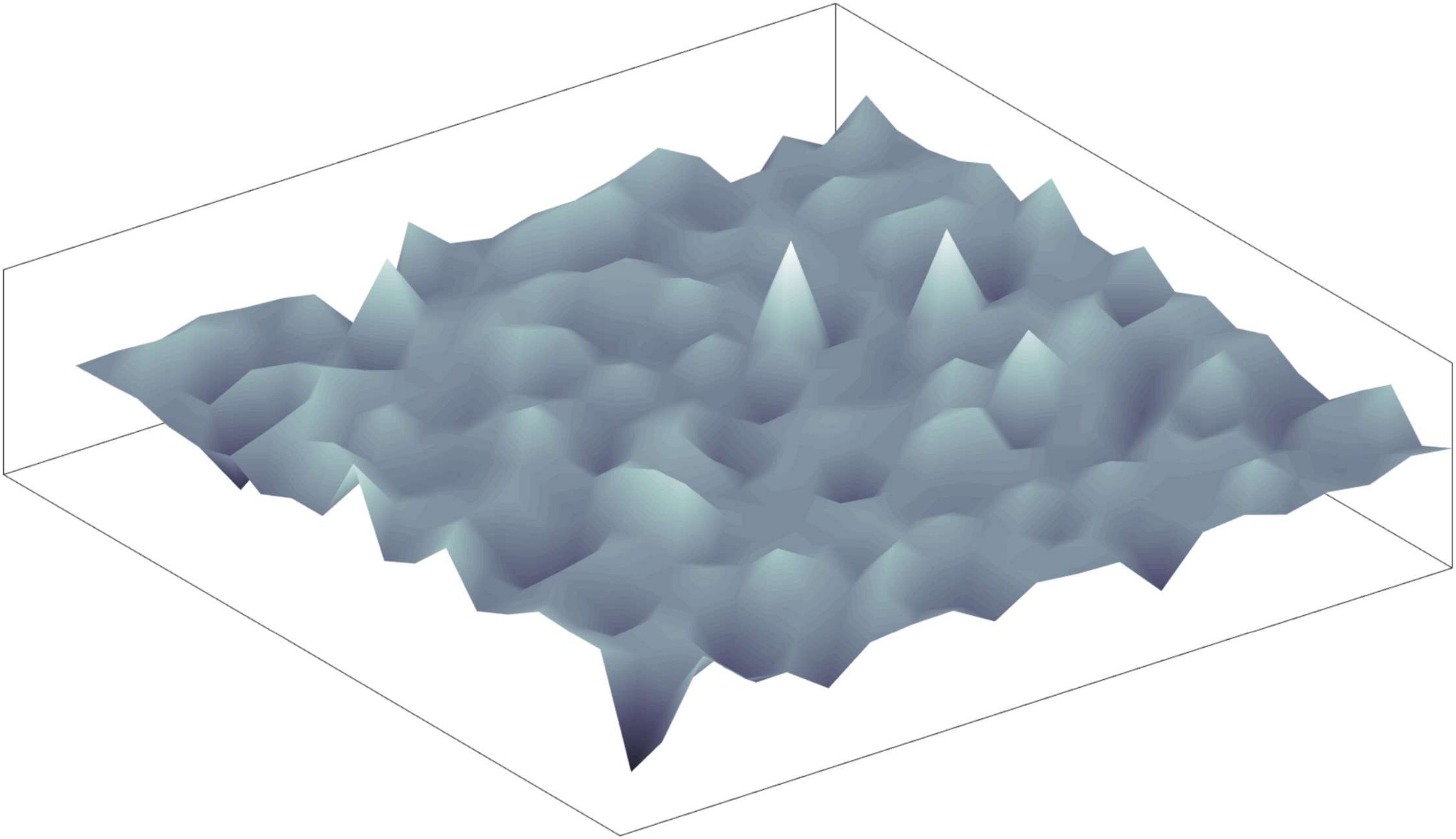
- We could go to a repeated state end
  - get stuck there

  or

  - Infinite loop

# Search In Complex Environments

# Difficult Environment / State Space

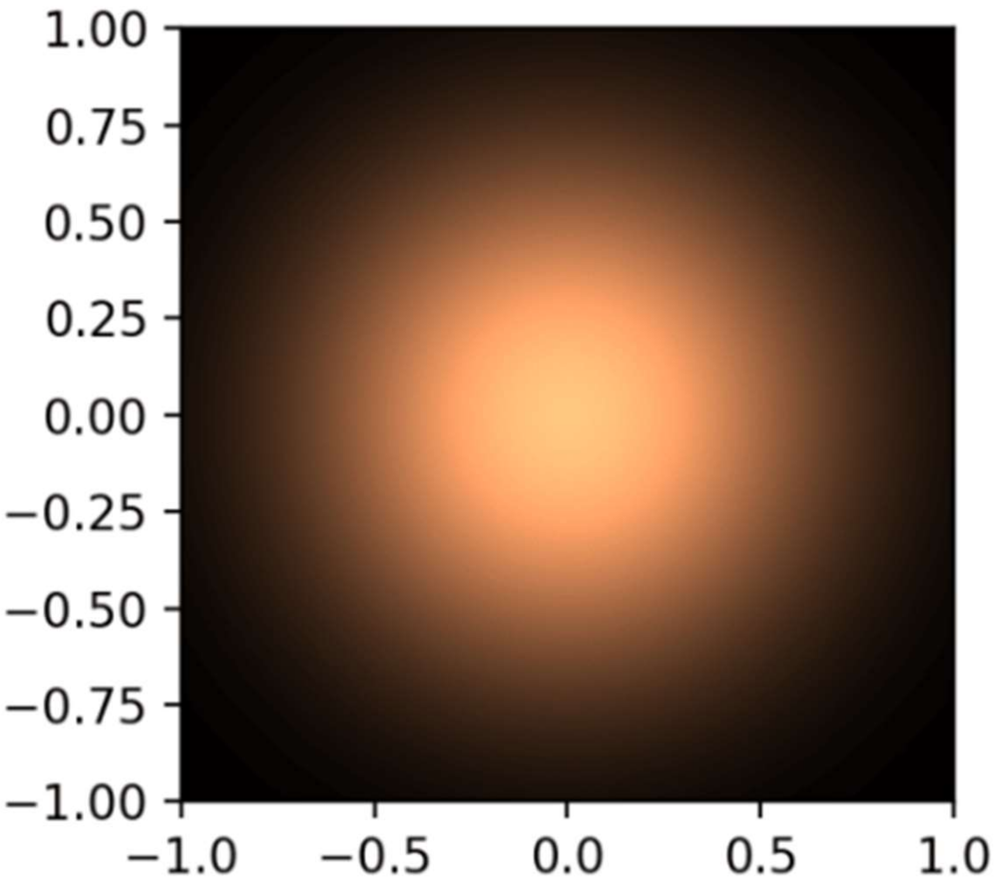


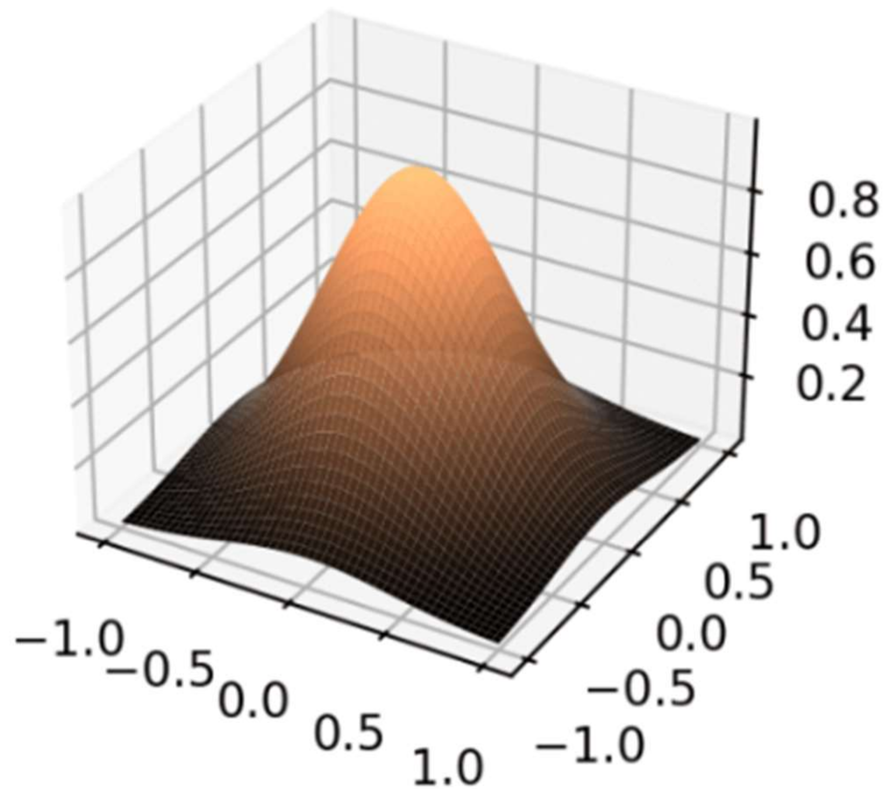
# Reality: Environment Assumptions

## Reality is not simple. What to relax?

- Fully observable?
- Single agent?
- Deterministic?
- Static?
- Episodic or sequential?
- Discrete?
- Known to the agent?

# Discrete vs. Continuous Spaces



# Local Search Algorithms:

# Can we not care about the path to the goal?

# Local Search Algorithms

**If the path to the goal does not matter, we might consider a different class of algorithms.**

**Local Search Algorithms**

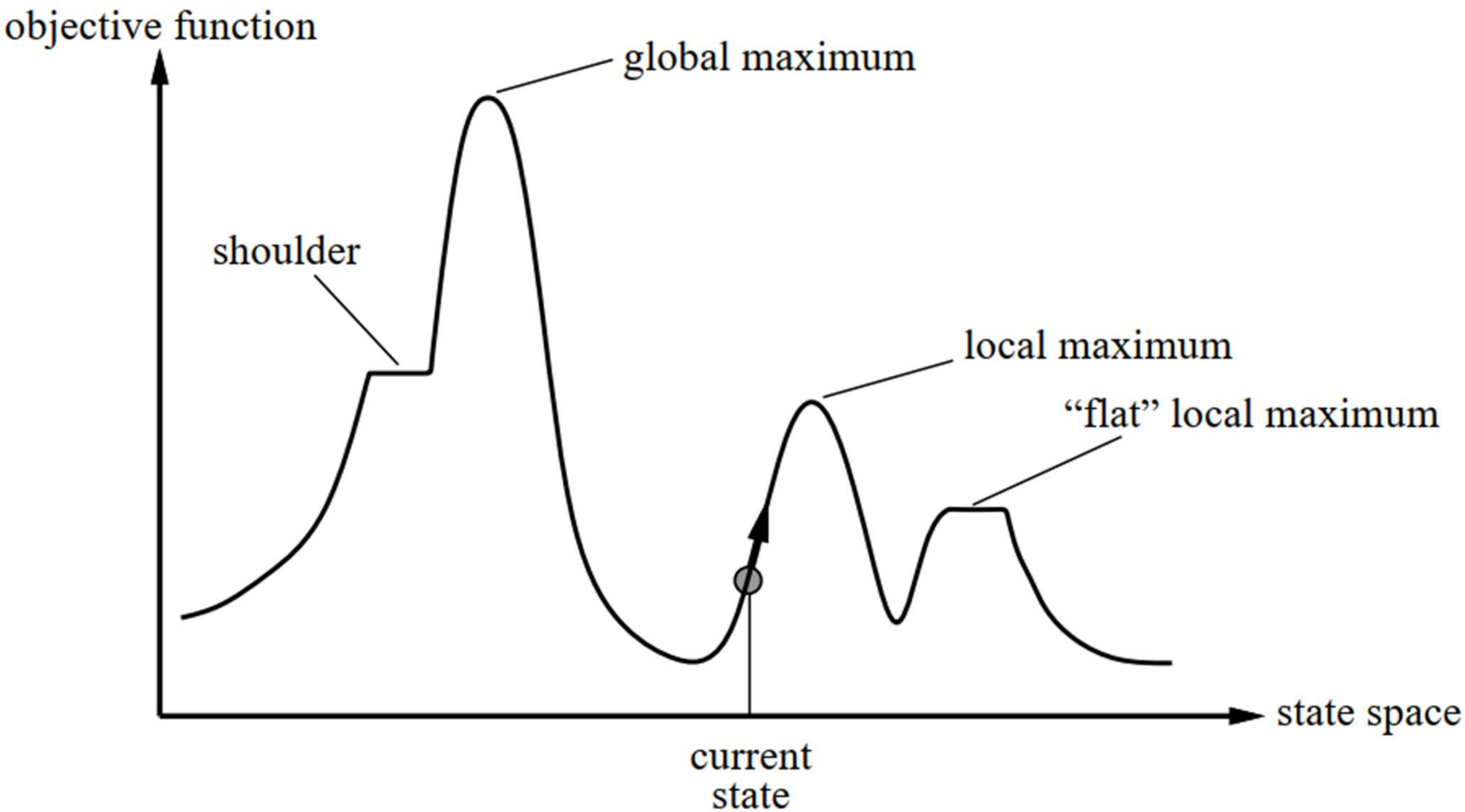
- **do not worry about paths at all.**
- **Local search algorithms operate:**
  - **using a single current state (rather than multiple paths) and generally move only to neighbors of that state.**
  - **typically, the paths followed by the search are not retained**

# Local Search Algorithms

Although local search algorithms are not systematic, they have two key advantages:

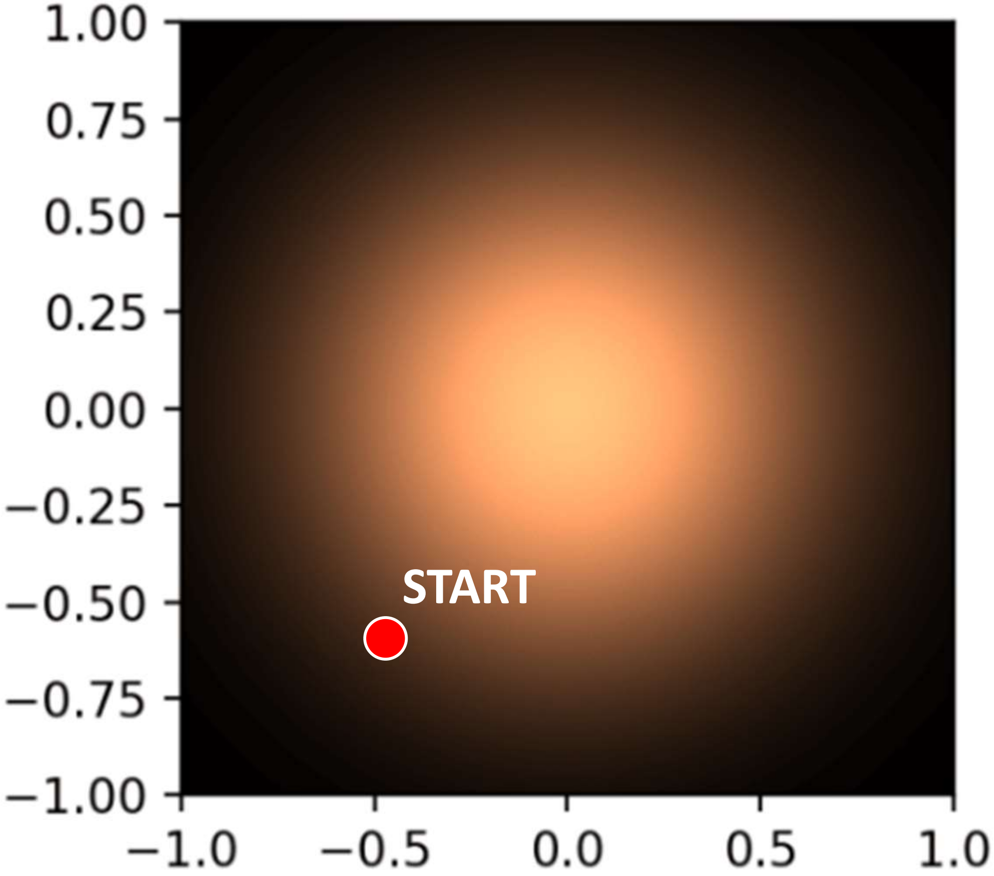
- they use very little memory—usually a constant amount; and
- they can often find reasonable solutions in large or infinite (continuous) state spaces for which systematic algorithms are unsuitable.

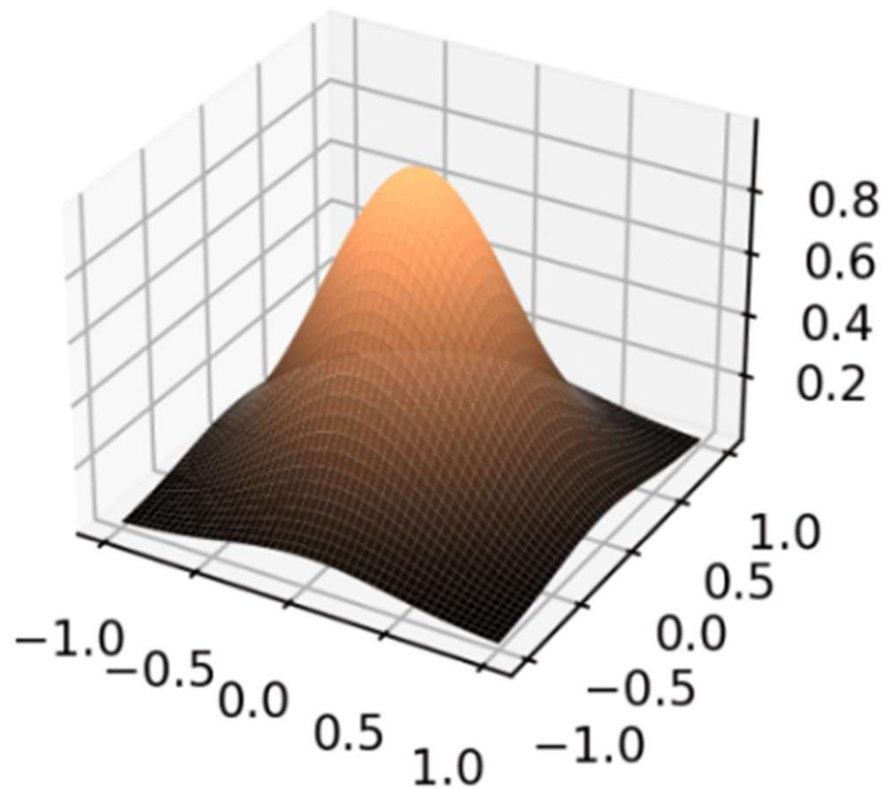
# 1D State Space Landscape



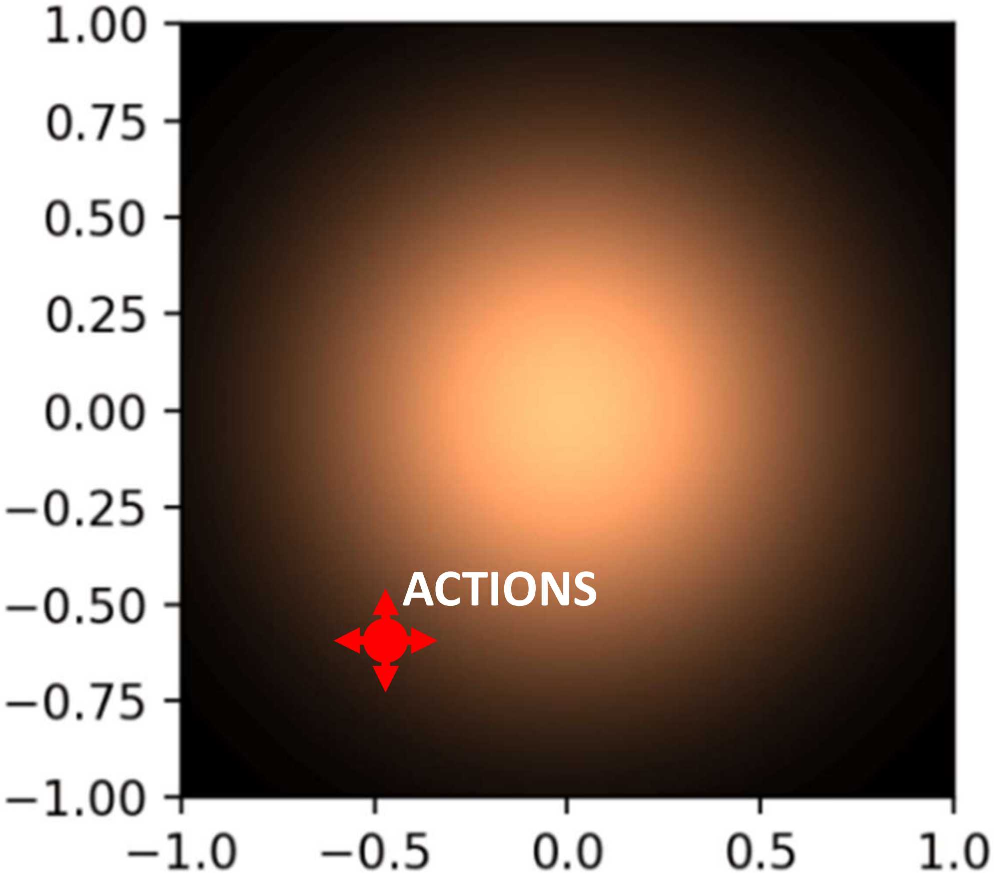

**Local search algorithms are useful for solving pure optimization problems, in which the aim is to find the best state according to an objective function.**

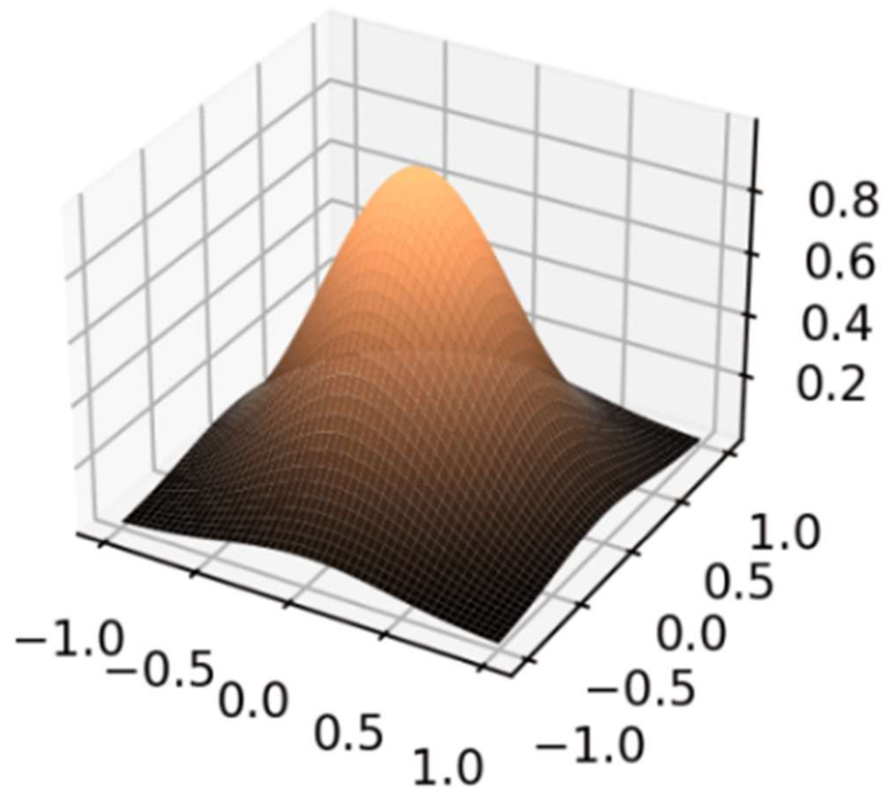
# Start “Somewhere”



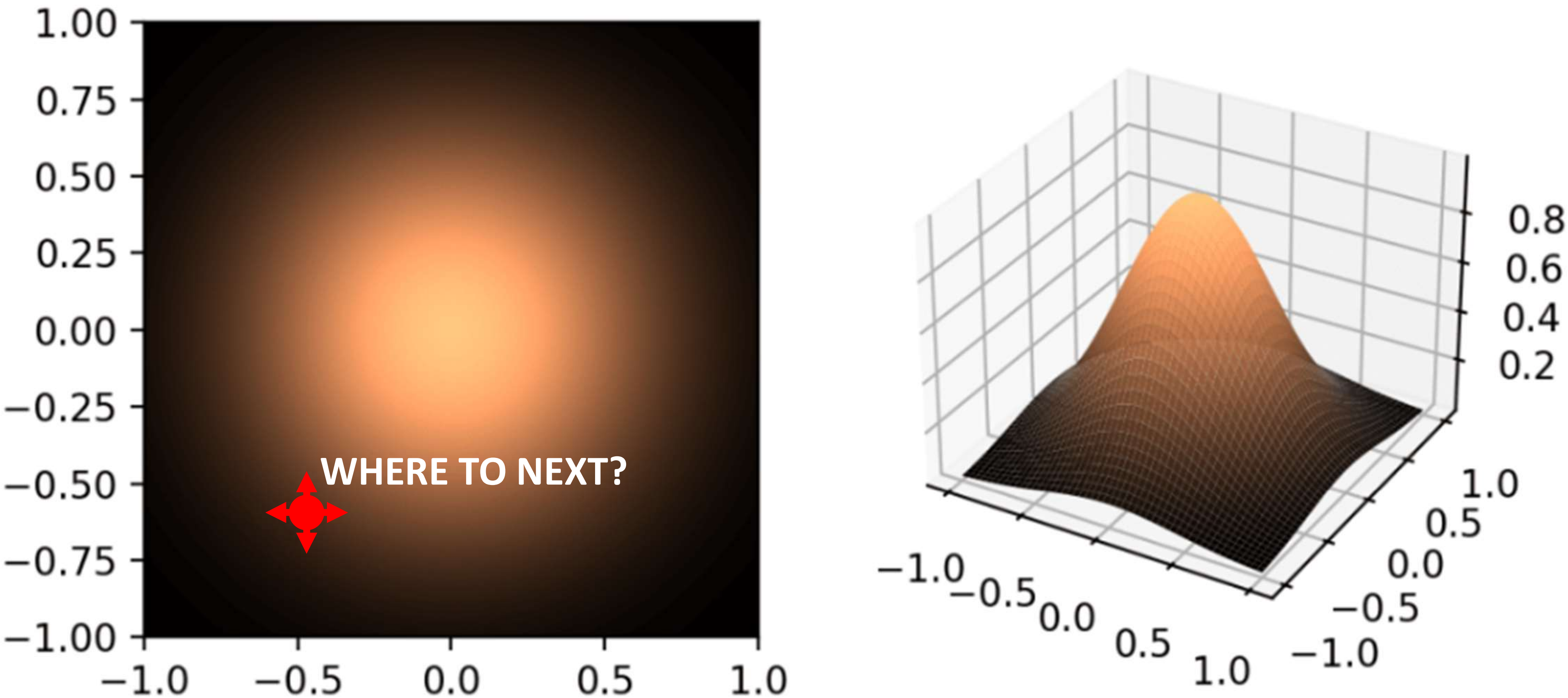



# Traverse/Explore Space With “Actions”



# Traverse/Explore Space With “Actions”



# Hill Climbing Search: Pseudocode

**function** HILL-CLIMBING(*problem*) **returns** a state that is a local maximum
  **inputs**: *problem*, a problem
  **local variables**: *current*, a node
              *neighbor*, a node

  *current* ← MAKE-NODE(INITIAL-STATE[*problem*])
  **loop do**
    *neighbor* ← a highest-valued successor of *current*
    **if** VALUE[neighbor] ≤ VALUE[current] **then return** STATE[*current*]
    *current* ← *neighbor*

# Hill Climbing and Difficult State Spaces / Environments

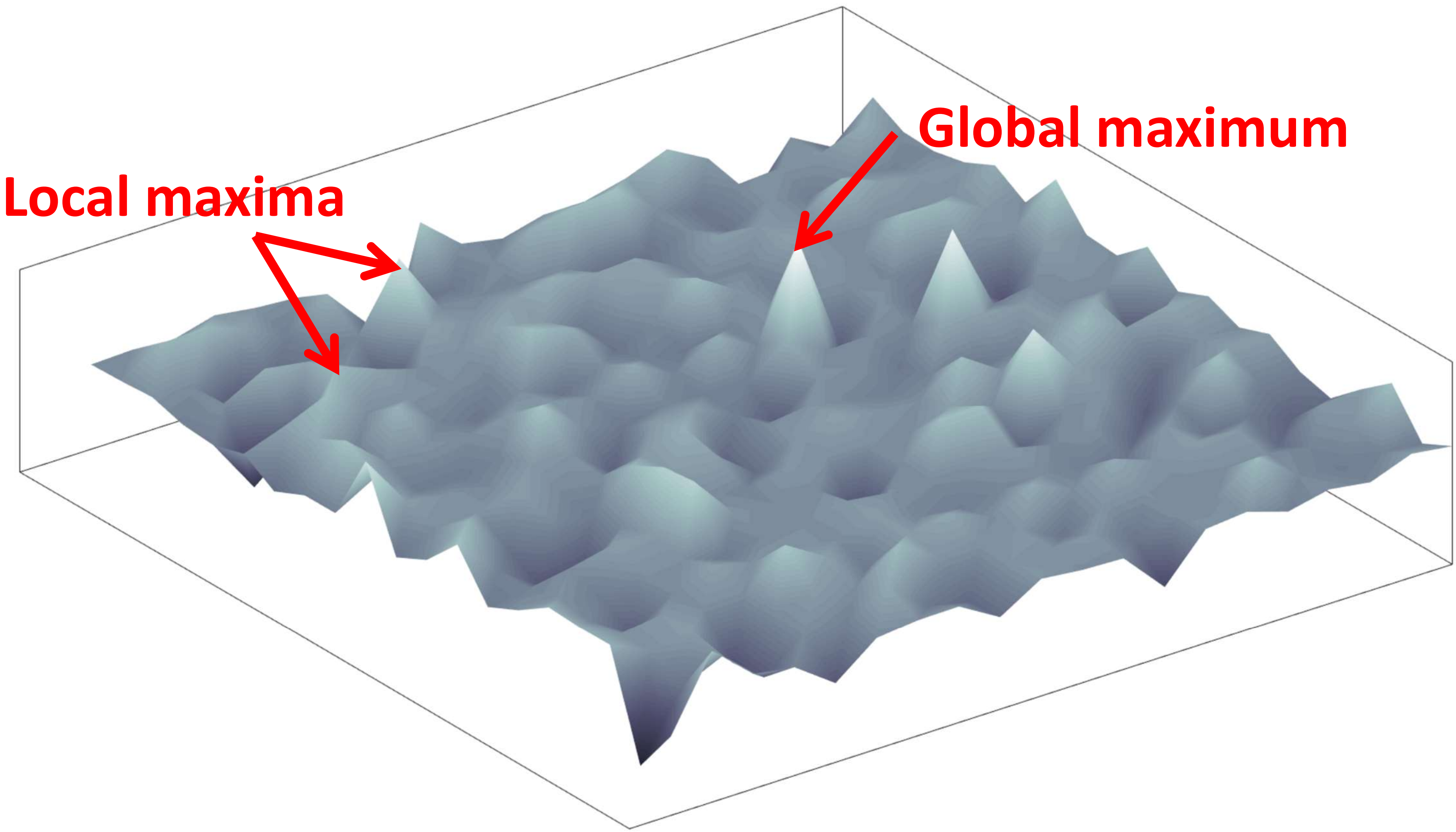
# Hill Climbing (Greedy Local) Search

- The most primitive informed search approach
  - a naive greedy algorithm
  - evaluation function: value of next state
  - does not care about the "bigger picture" (for example: total search path cost)
- Practicalities:
  - does not keep track of search history

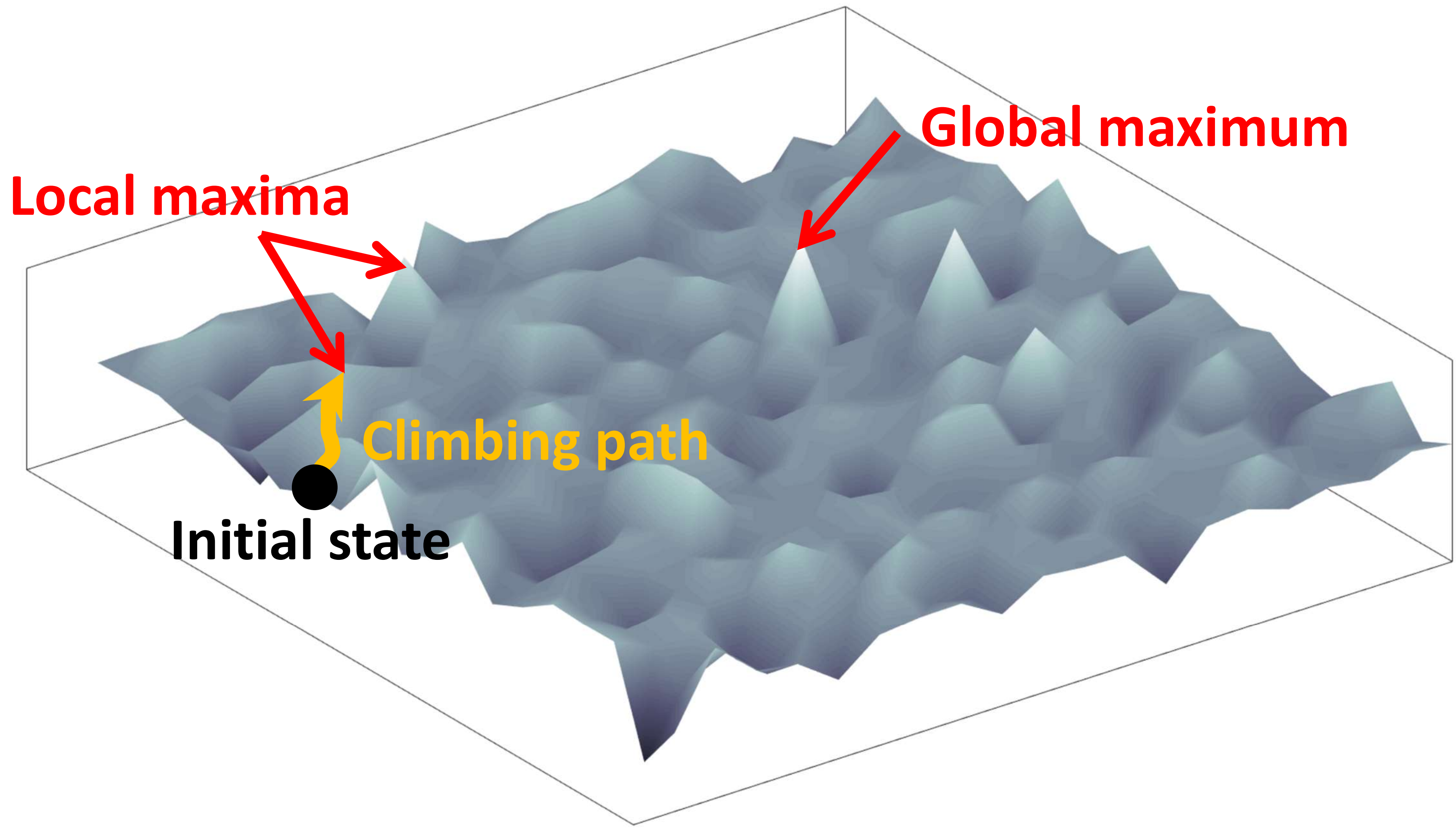
# “Getting Stuck”

- **Local maxima**: a local maximum is a peak that is higher than each of its neighboring states, but lower than the global maximum.
  - Hill-climbing algorithms that reach the vicinity of a local maximum will be drawn upwards towards the peak, but will then be stuck with nowhere else to go
- **Ridge**: ridges result in a sequence of local maxima that is
  - very difficult for greedy algorithms to navigate.
- **Plateau**: a plateau is an area of the state space landscape where the evaluation function is “flat”. It can be a flat local maximum, from which no uphill exit exists, or a shoulder, from which it is possible to make progress.
  - A hill-climbing search might be unable to find its way off the plateau

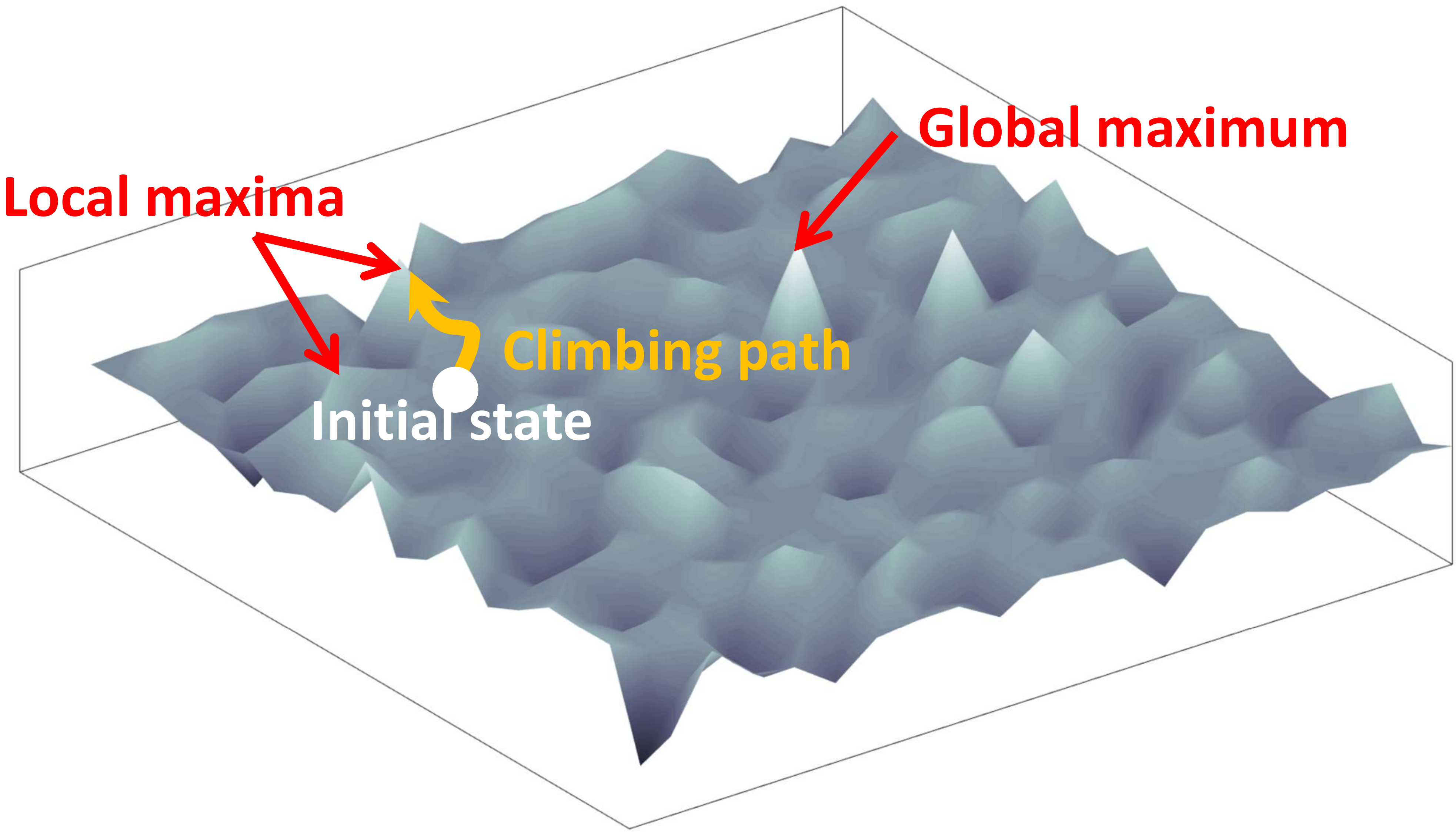
# Hill Climbing Problems: Local Maxima



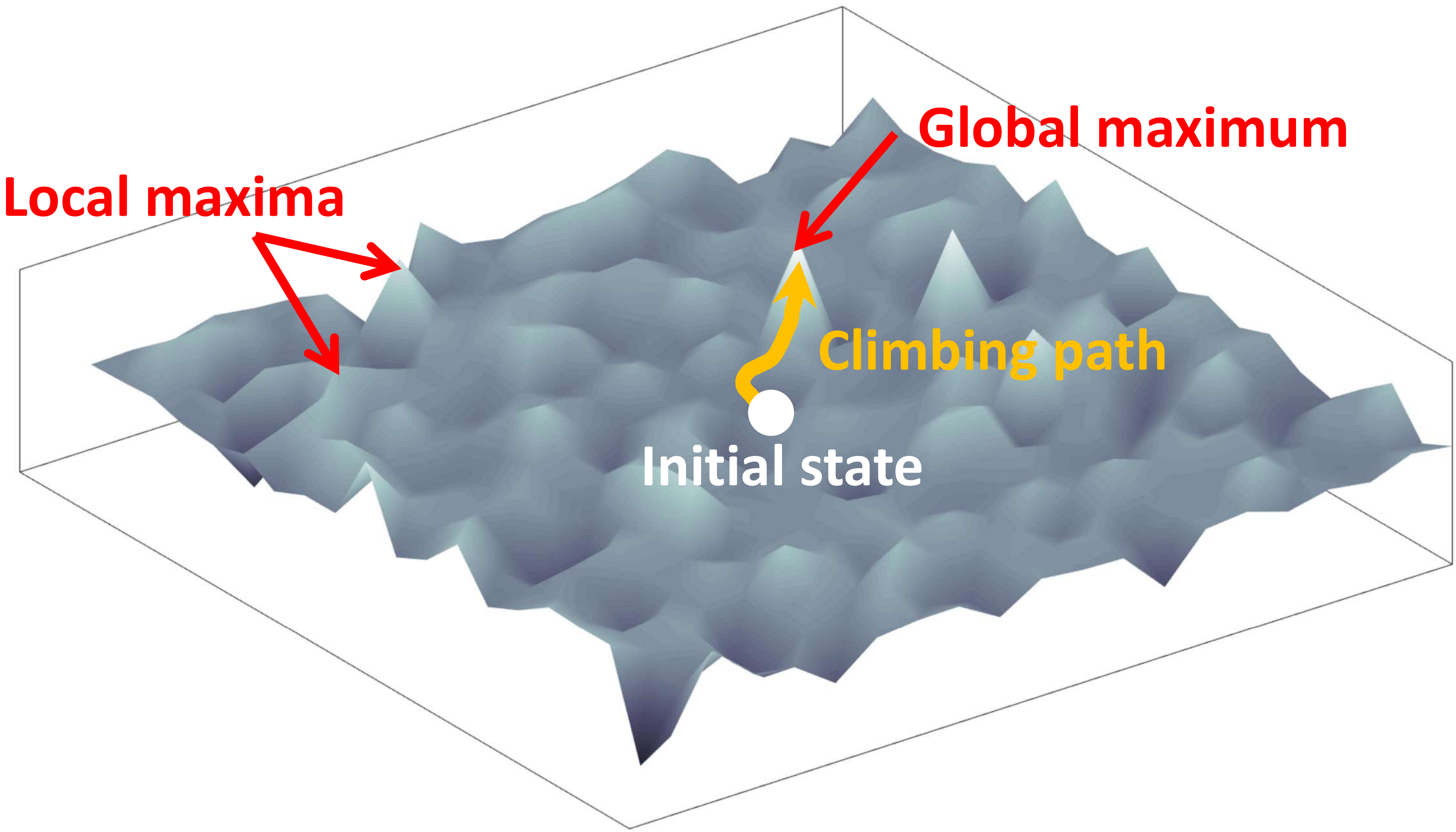
# Hill Climbing Problems: Local Maxima



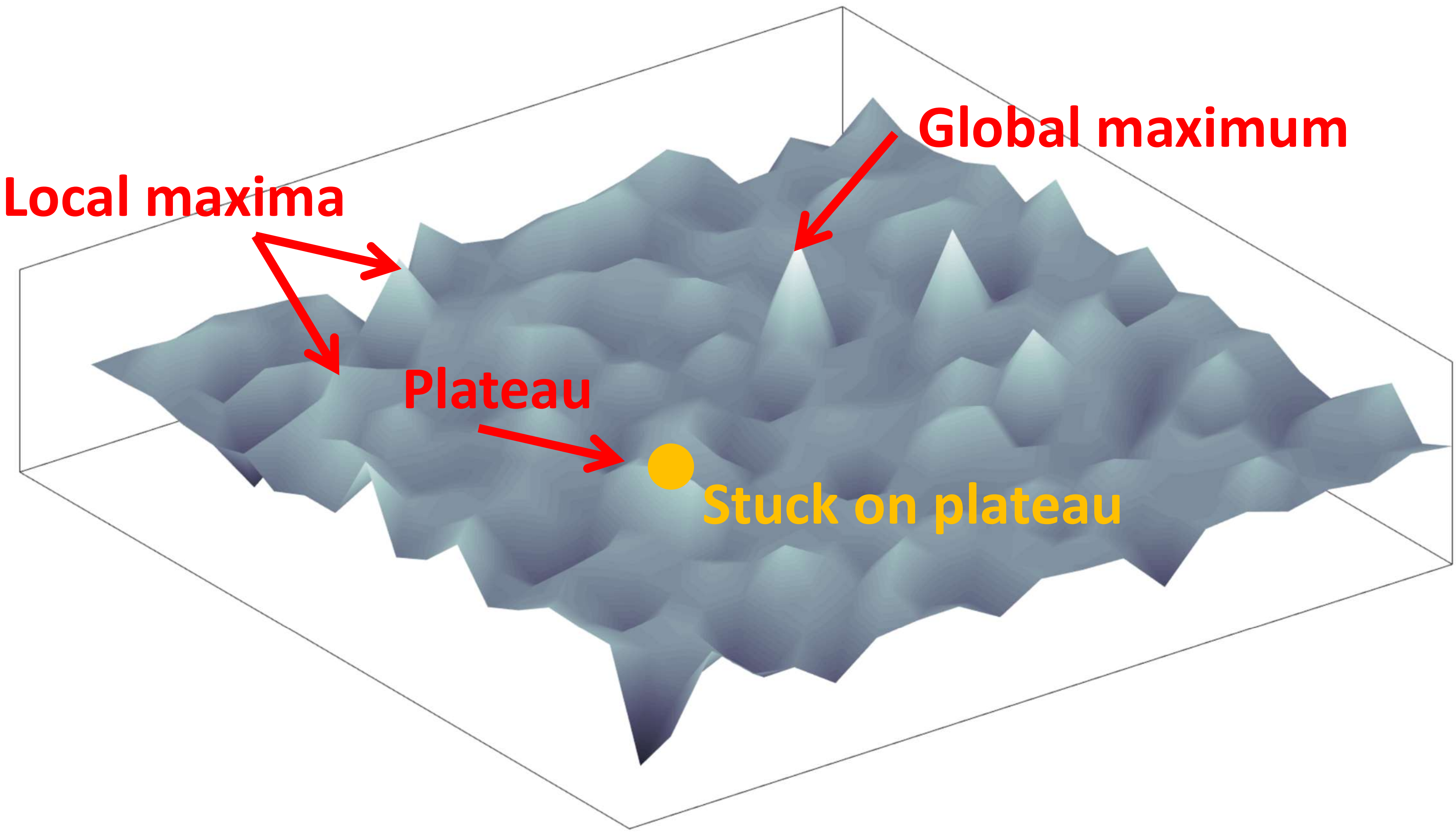
# Hill Climbing Problems: Local Maxima



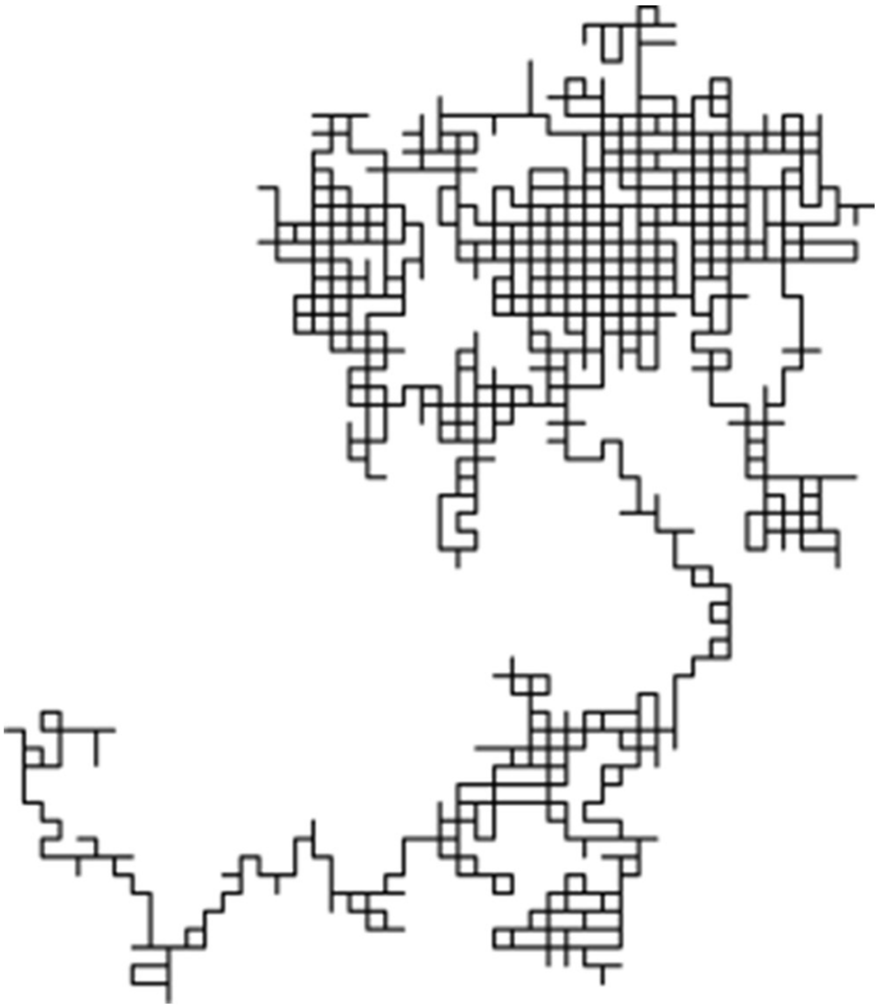
# Hill Climbing Problems: Local Maxima



# Hill Climbing Problems: Plateaus



# Random Walk



Source: *https://en.wikipedia.org/wiki/Random_walk*

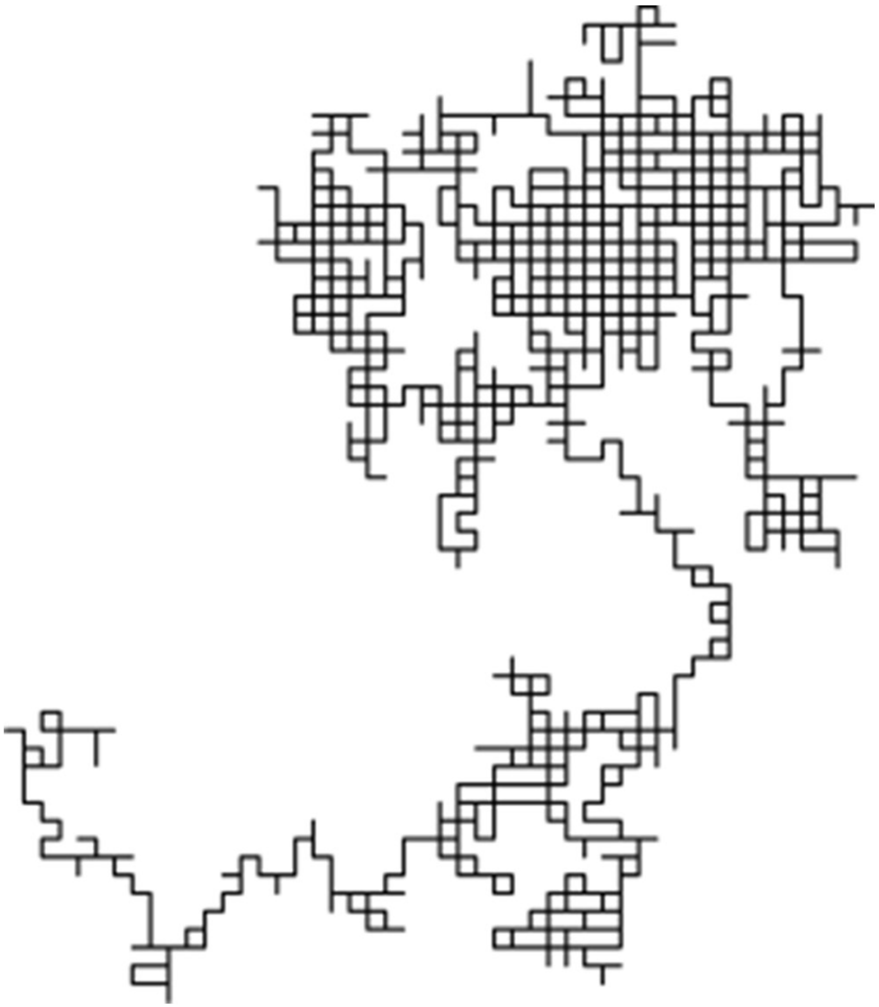
In mathematics, **a random walk**, sometimes known as **a drunkard's walk**, is a random process that describes a path that consists of a succession of random steps on some mathematical space.

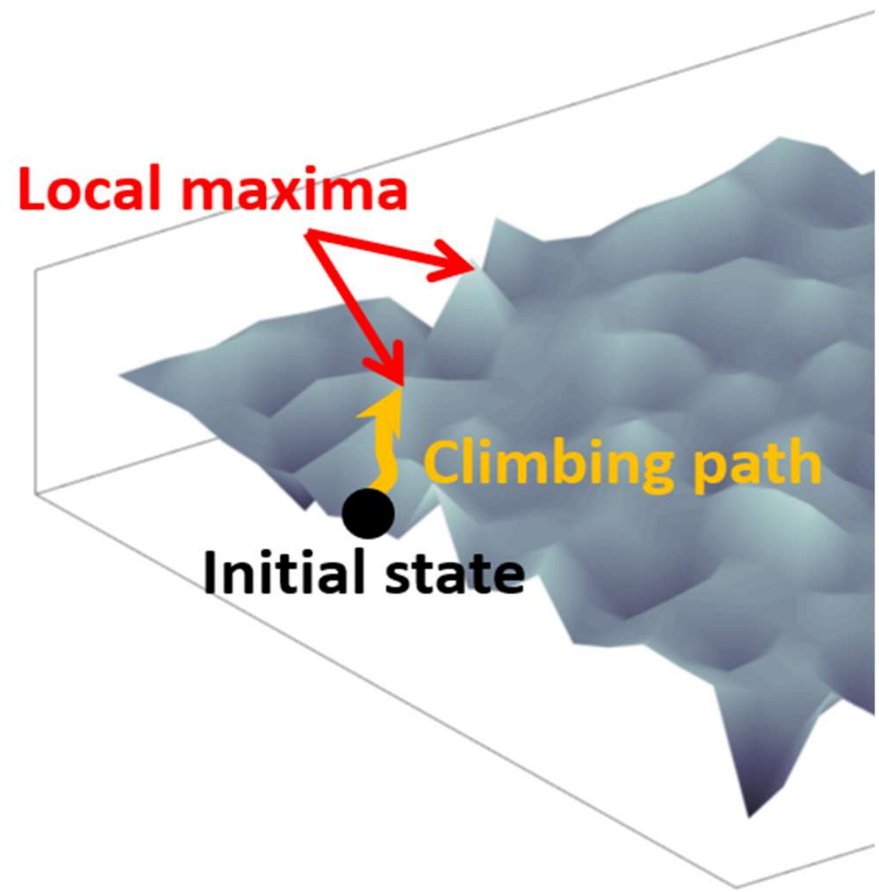
# Measuring Searching Performance

Search algorithms can be evaluated in four ways:

- **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? (in seconds, actions, states, etc.)
- **Space complexity**: How much memory is needed to perform the search?

# Random Walk vs. Hill Climbing





# Simulated Annealing: Article

Optimization by Simulated Annealing

- S. Kirkpatrick, C. D. Gelatt Jr., M. P. Vecchi
- *Science* 13 May 1983:
- Vol. 220, Issue 4598, pp. 671-680
- https://science.sciencemag.org/content/220/4598/671