Computational Intelligence

Searching for Paths

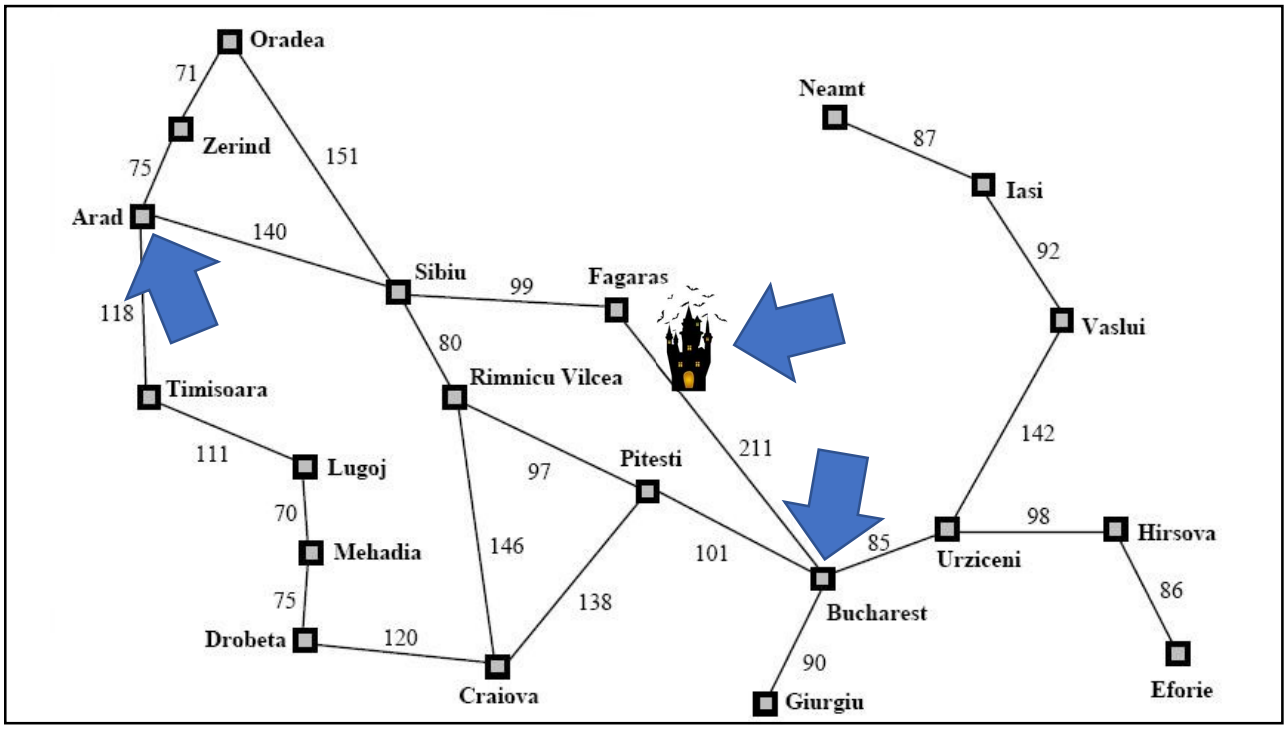
Giovanni Squillero
squillero@polito.it

Searching for Paths

- The goal is to find a **sequence of actions** to solve a problem
  - In each step one must decide what to do next ("it's all about choiches", Patrick Winston)
  - Find a **path** from the *initial* state to the *goal* state (in the solution space)
- Problem space **vs.** Solution space
- Solving → Searching

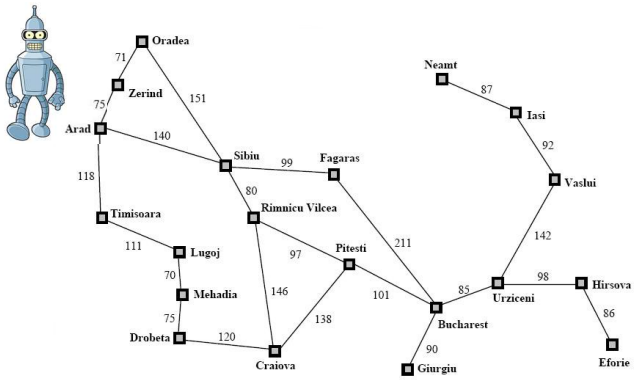
Basic Assumptions

- Problem characteristics:
  - **Sequential** (the solution is as a sequence of steps)
  - **Observable** (all relevalt details are known)
  - **Deterministic** (the effects of actions are predictable)
  - **Static** (time is not relevant for choiches)
  - **Discrete** (the alternative actions can be enumerated)
- Only one actor is active
  - Implied by deterministic and static



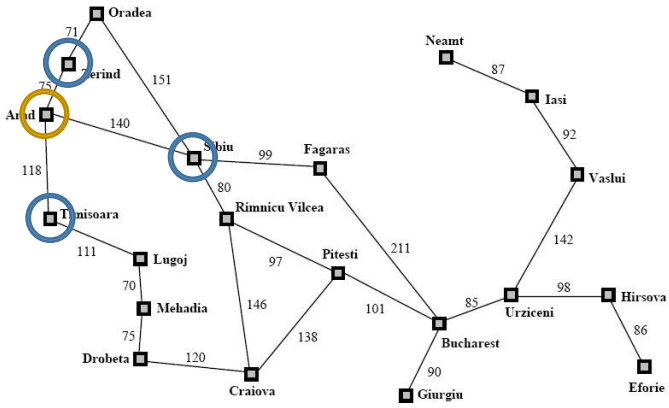
Searching for a Path

- Bender is in Arad and wants to go to Bucarest
- In each step, it considers the possible alternatives and picks one



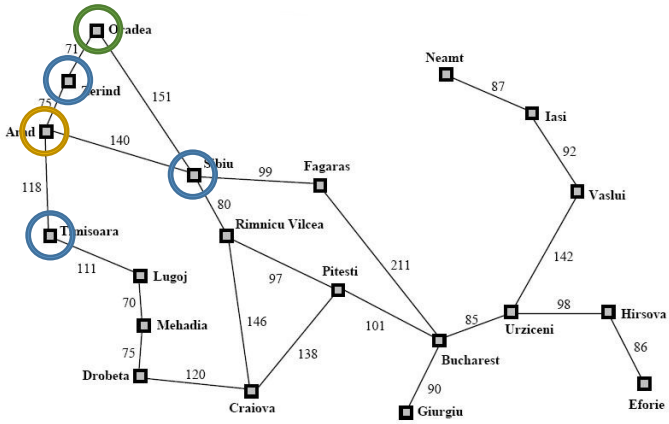
Searching for a Path

- In Arad, Bender can go to Zerind, Sibiu, or Timisoara



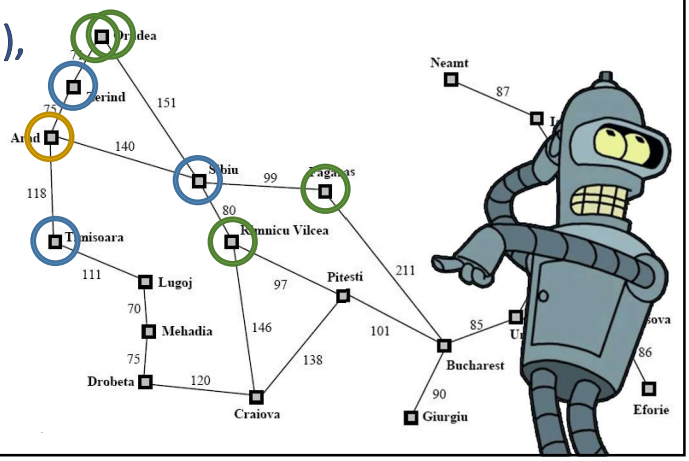
Searching for a Path

- In Arad, Bender can go to Zerind, Sibiu, or Timisoara
- After Arad→Zerind: only Oradea (going back is useless)



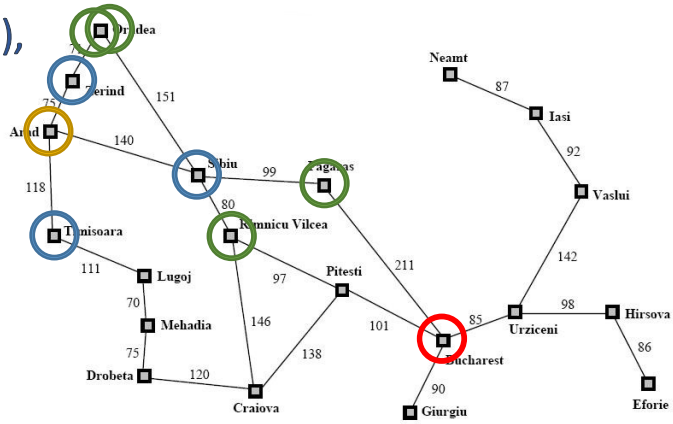
Searching for a Path

- In Arad, Bender can go to Zerind, Sibiu, or Timisoara
- After Arad→Zerind: only Oradea (going back is useless)
- After Arad→Sibiu: Oradea (!), Fagaras, Rimnicu Vilcea



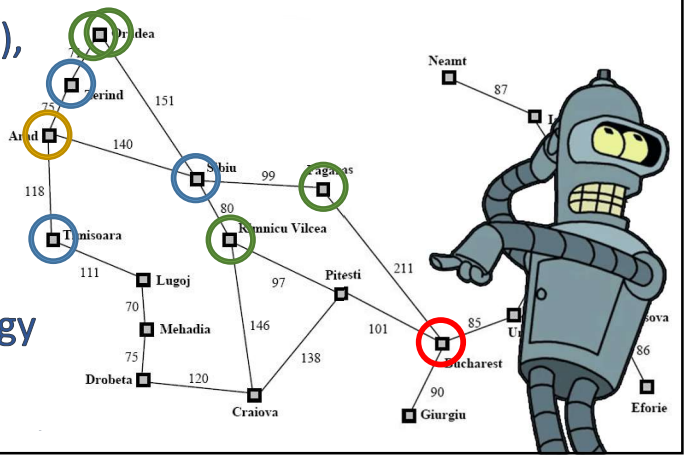
Searching for a Path

- In Arad, Bender can go to Zerind, Sibiu, or Timisoara
- After Arad→Zerind: only Oradea (going back is useless)
- After Arad→Sibiu: Oradea (!), Fagaras, Rimnicu Vilcea
- After Arad→Sibiu→Fagaras: Bender can reach Bucarest!
- …



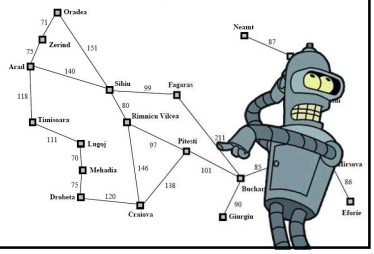
Searching for a Path

- In Arad, Bender can go to Zerind, Sibiu, or Timisoara
- After Arad→Zerind: only Oradea (going back is useless)
- After Arad→Sibiu: Oradea (!), Fagaras, Rimnicu Vilcea
- After Arad→Sibiu→Fagaras: Bender can reach Bucarest!
- …
- Need for a systematic strategy to explore the territory!



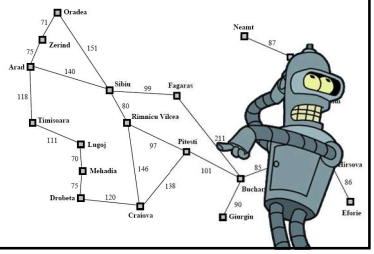
Find a Path (Greedy Best First)

- Move forward to the next city trying to head in the direction of Bucarest
- Don't go back to a previously visited city



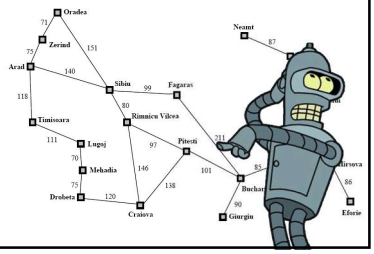
Find a Path (Greedy Best First)

- Move forward to the next city trying to head in the direction of Bucarest
- Don't go back to a previously visited city
- **Backtrack** if stuck
- Requires **extra knowledge** about the problem



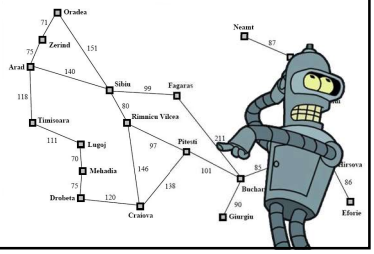
Find a Path (Generate 'n Test)

- Move forward to the next city at random
- Never go back to a previously visited city



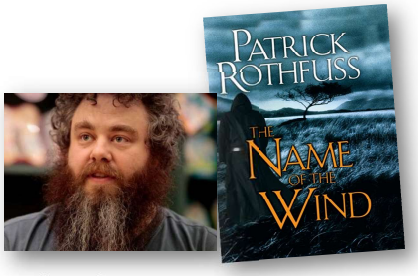
Find a Path (Generate 'n Test)

- Move forward to the next city at random
- Never go back to a previously visited city
- **Backtrack** if stuck
- **Don't require any knowledge** about the problem
- Perfectly **reasonable** strategy — in some cases



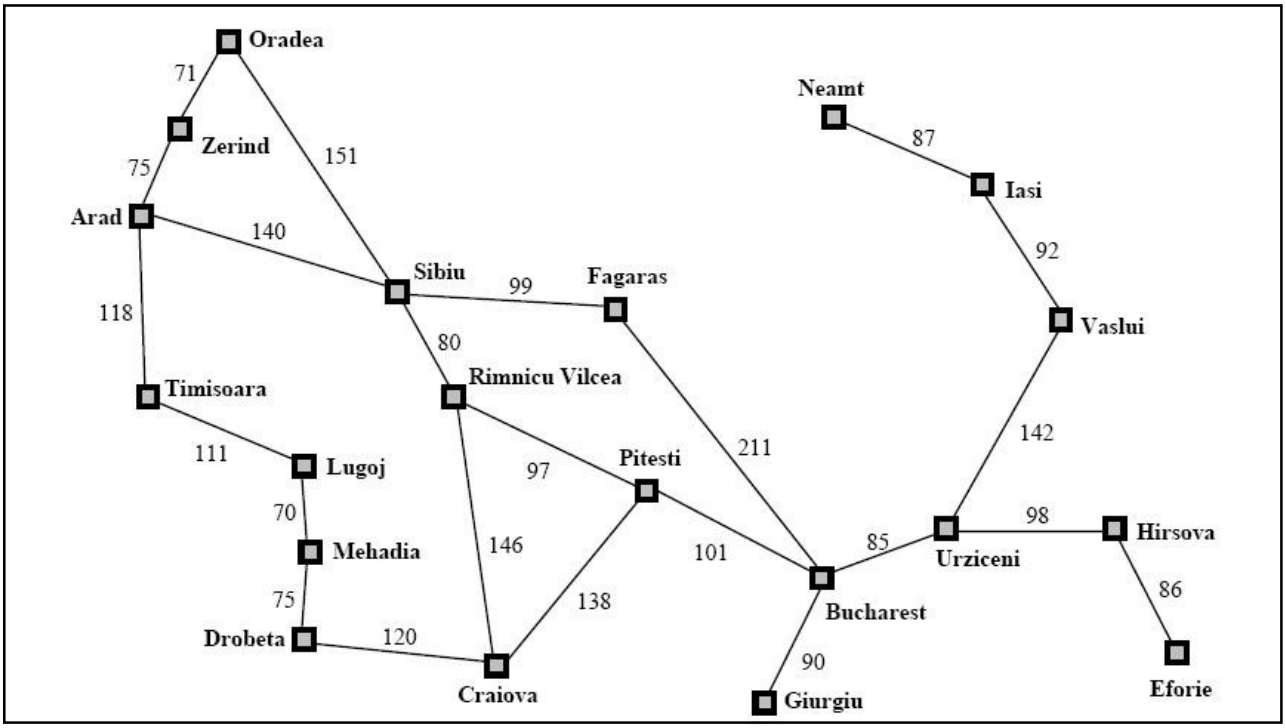
Caveat: Power of Naming

- "If you name something you get control over it"
  - Patrick Henry Winston (Artificial Intelligence)
  - Patrick Rothfuss (The Kingkiller Chronicle)

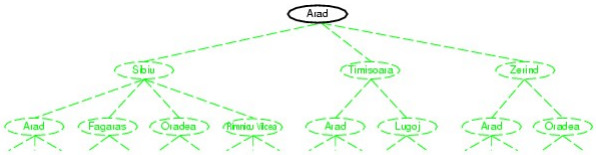


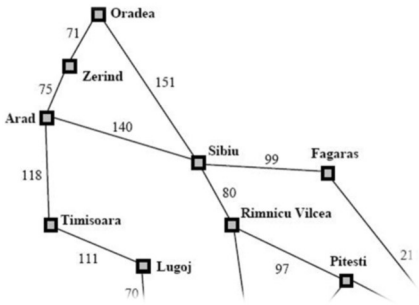
Problem Solving: Generate n' Test

- **Heuristic** search technique which **guarantees** to find a solution **if done systematically** (and there exists a solution)
- Based on *depth-first search with backtracking*
- **Trial n' error**
- Good generators
  - Exhaustive (systematic)
  - Not redundant (should not generate the same solution twice)
  - Informable (absorb information)

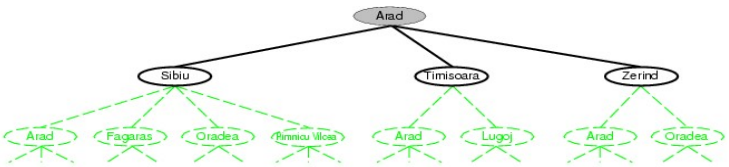


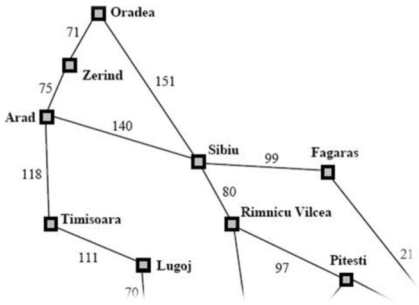
Find a Path (Breadth First)



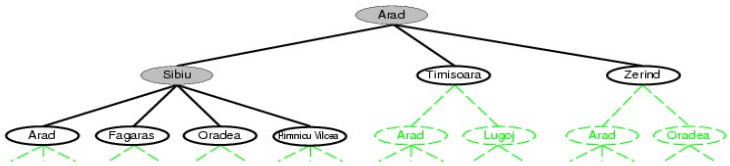


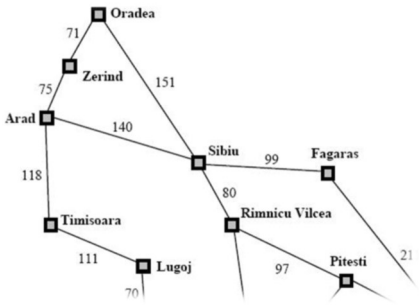
Find a Path (Breadth First)



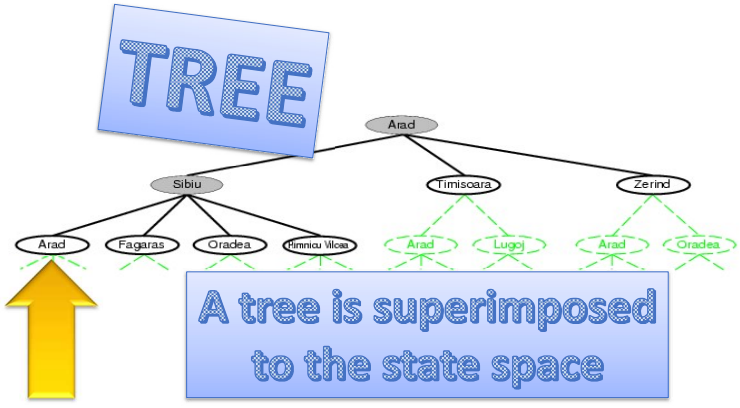


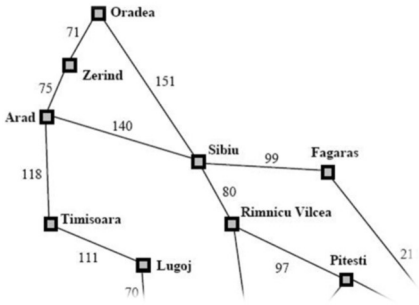
Find a Path (Breadth First)





Find a Path (Breadth First)



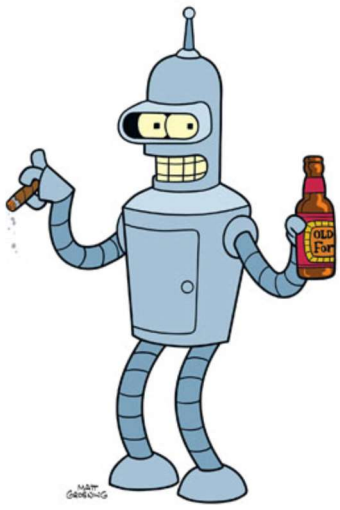


Tree vs. Graph Search

- **Tree Search**
  - Don't remember visited nodes
  - Exponentially slower, but memory efficient
- **Graph Search**
  - Do remember visited nodes
  - Exponentially faster, but memory blow-up
- I.e., the classic *Time vs. Space trade-off*
- Not related with "searching trees" vs. "searching graphs"

Solve by Searching

- **The Agent**



Solve by Searching

- **State space**
  - A set of states that describe all the relevant details of all the possible situations
  - The initial state
  - The "goal test": a function $G(s)$ that checks if the goal has been reached in state s
    … or a set of one or more goal states
  - Terminology: State vs. Node
  - Type: explicit vs. implicit; finite vs. infinite

Solve by Searching

- **Actions**
  - A function $A(s)$ yields the set $\{a_0^s, a_1^s, \ldots, a_n^s\}$ of the possible actions that the agent can perform in state s
  - A function $R(s,a)$ calculate the result (i.e., the next state s') of performing action a in state s
  - One action corresponds to one "step" over the state space, a "path" is a sequence of steps
  - A function $C(P)$ yields the cost paid by the agent to cover the path P, that is, to perform the sequence of actions $(a_0, a_1, \ldots, a_n)$
    NB: It may be as simple as the sum of the cost of the steps

Problem: *n* friends

- n students of Mimetic Learning (half from computer engineering, half from data science), are in a pizzeria to celebrate, after the meal they decided to drink a beer in good Irish pub
- Unfortunately, the pub is not at walking distance. But one of the student has a tandem bike that can transport two people

Problem: *n* friends

- Find the fastest way to move all 6 students from the pizzeria (S) to the pub (D), given that
  - The tandem bike can transport 1 or 2 students (not 0!)
  - If, at any moment, either in the pizzeria or in the pub, the group of students contains more computer engineers (**C**) than data scientists (**D**), the computer engineers start to talk about video games, and the data scientists will run away bored to death — ruining the celebration
  - I.e., `count(C) <= count(D) or count(D) == 0`

Problem: *n* friends

- Representation:
  - Possible scenario:
    (friends in pizzeria, friends in pub)
  - Some scenarios are impossible (invalid), for instance with $n = 6$:
    (CCCDDD, ·) ✓
    (CC, CDDD) ✓
    (DD, CCCD) ✗
- A good representation exposes the constrains
  - I.e., make possible to find a solution

Problem: *n* friends

- Representation:
  - Possible scenario: **(friends in pizzeria, friends in pub)**
  - Graph of possible scenarios (nodes)
  - An edge if it is possible to move from one scenario to the other with the bicycle. E.g. (with $n = 4$), **(CCDD, -) → (CD, CD)**
- Algorithm:
  - Find a path between **(C…CD…D, -)** and **(-, C…CD…D)**

(CCDD, -) → (CD, CD) → (-, CCDD)

- Representation:
  - Possible scenario: **(friends in pizzeria, friends i**
  - Graph of possible scen odes)
  - An ed s possible to move from on er with the bicycle. E CD
- Al hm
  - Find a pat

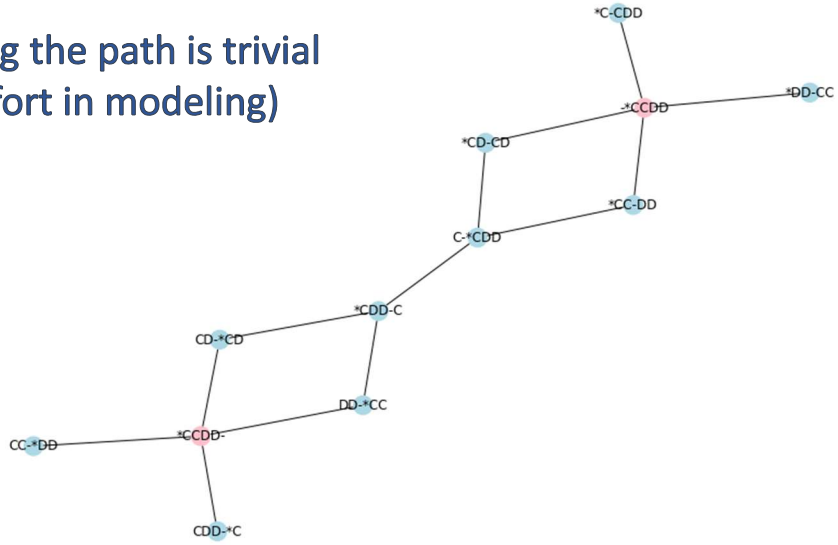
A good representation exposes the constrains (makes possible to find a solution)

Problem: *n* friends

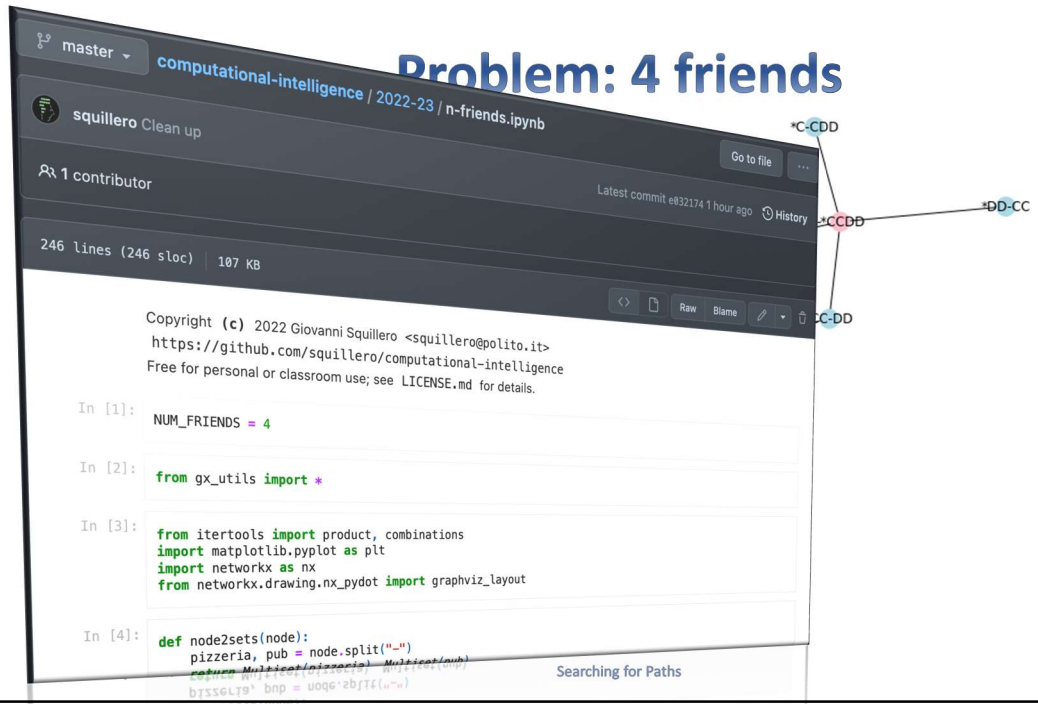
- Representation:
  - Possible scenario: **(friends in pizzeria, friends in pub)** and bike position *
  - Graph of possible scenarios (nodes)
  - An edge if it is possible to move from one scenario to the other with the bicycle. E.g., **(*CCDD, -) → (CD, *CD)** but not **(CD, *CD) → (-, *CCDD)**
- Solution:
  - Find a path between **(*C…CD…D, -)** and **(-, *C…CD…D)**

Problem: 4 friends

- Finding the path is trivial (all effort in modeling)



Problem: 4 friends



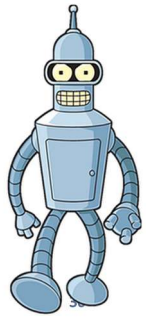
Generic Search

```
def search(state):
   frontier = PriorityQueue()
   while state is not None and not goal_test(state):
      for a in possible_actions(state):
         new_state = result(state, a)
         if new_state not in state_cost and new_state not in frontier:
            # Add node to frontier and update solution
            cost[new_node] = path_cost(new_state)
            frontier.push(new_state, p=priority_function(new_state))
         elif new_state in frontier and cost[new_node] > path_cost(new_state) :
            # Update node and solution
            cost[new_node] = path_cost(new_state)
      state = frontier.pop()

   return solution
```


Search

- Not *one* algorithm, but a family of algorithms
  - Same structure
  - Key element: the priority queue used to store the frontier
  - The goal is tested only when a state is extracted from the frontier
- A data structure is superimposed on the search space
  - States vs. Nodes
  - Some states may appear more than once (a.k.a., duplicates)

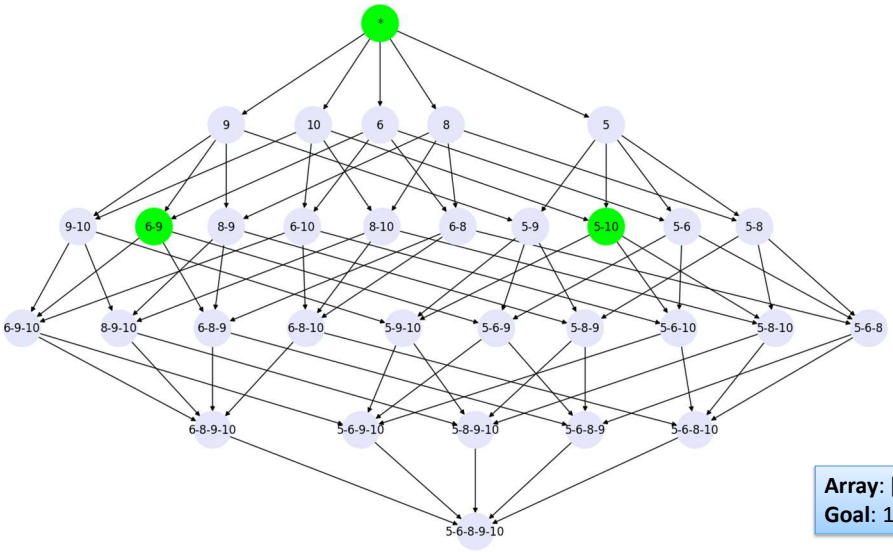


Problem: Min Sum

- Given an array of n positive integers
- Find the minimum number of elements that sum to G

- E.g.,
  - Array: [9, 10, 6, 8, 5]
  - Goal: 15
  - Solution: [6, 9] or [5, 15]

Solution Space



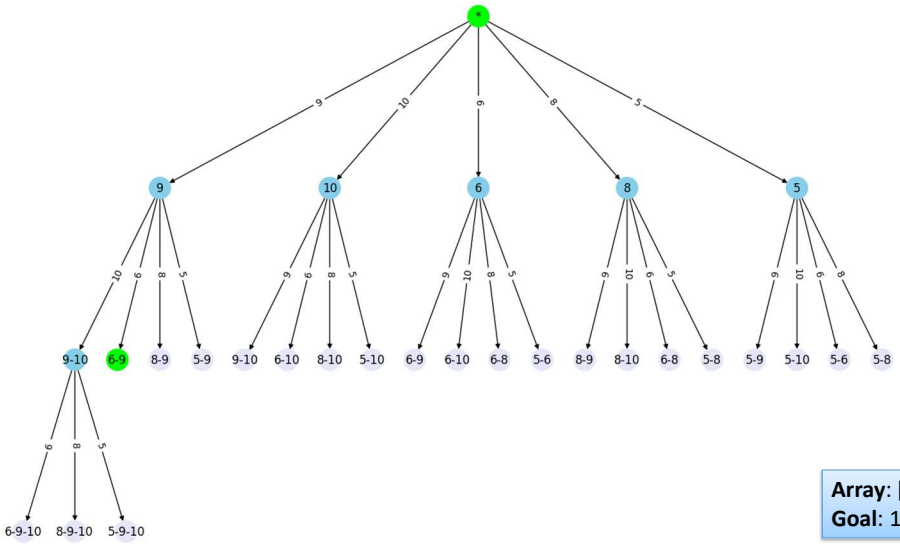
Array: [9, 10, 6, 8, 5]
Goal: 15

Uninformed strategies

- **Breadth-first** search
  - The root node is expanded, then its successors, then their successors, and so on…
    (i.e., children are added to the frontier with **`append`**)
  - Systematic
  - Always complete (if branching factor < ∞)
  - Optimal when all costs are equal

```
priority_function=lambda s: len(state_cost)
```


Breadth First

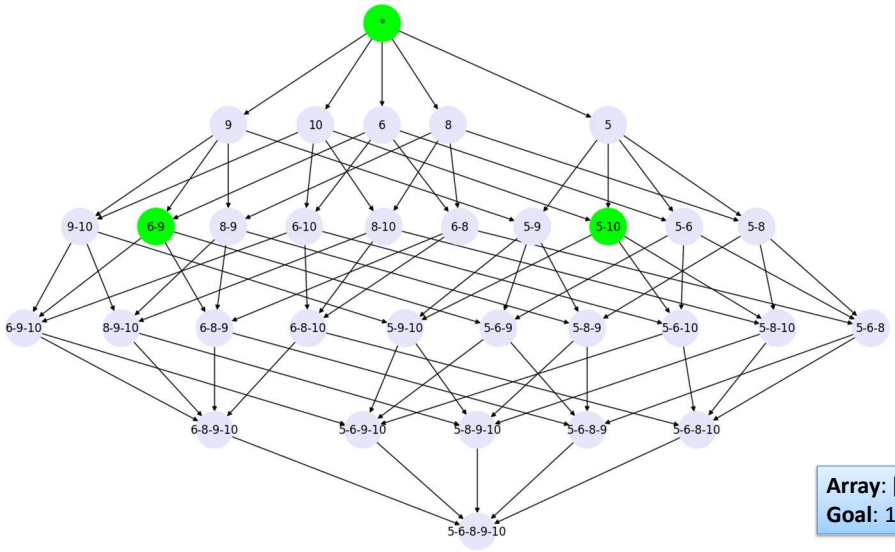


Array: [9, 10, 6, 8, 5]
Goal: 15

Uninformed strategies

- **Uniform-cost search**
  - Like Breadth-first, but the node with the lowest path cost from the root is expanded
    (i.e., the frontier is a priority queue)
  - Also called **Dijkstra's algorithm**
  - Breadth-first can be seen as Dijkstra with unit cost

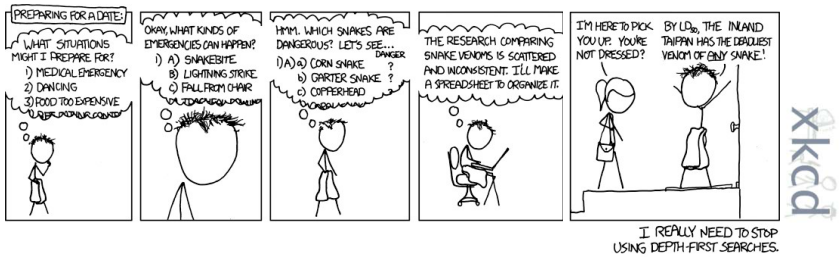
Solution Space



Array: [9, 10, 6, 8, 5]
Goal: 15

Uninformed strategies

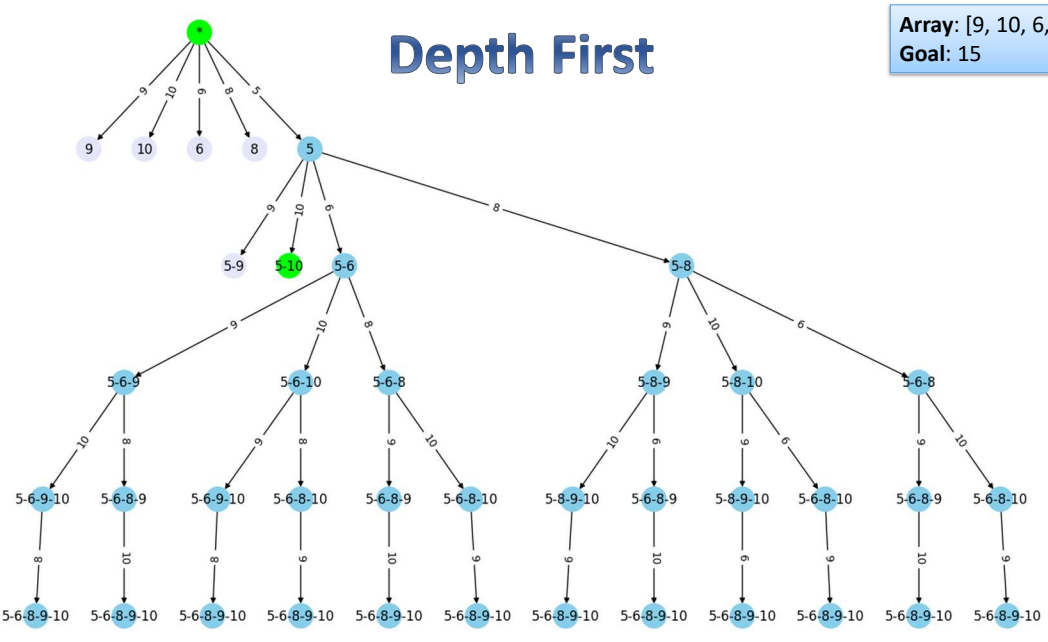
- **Depth-first** search
  - The deepest node in the frontier is expanded
    Problems of cycles




```
priority_function=lambda s: -len(state_cost)
```

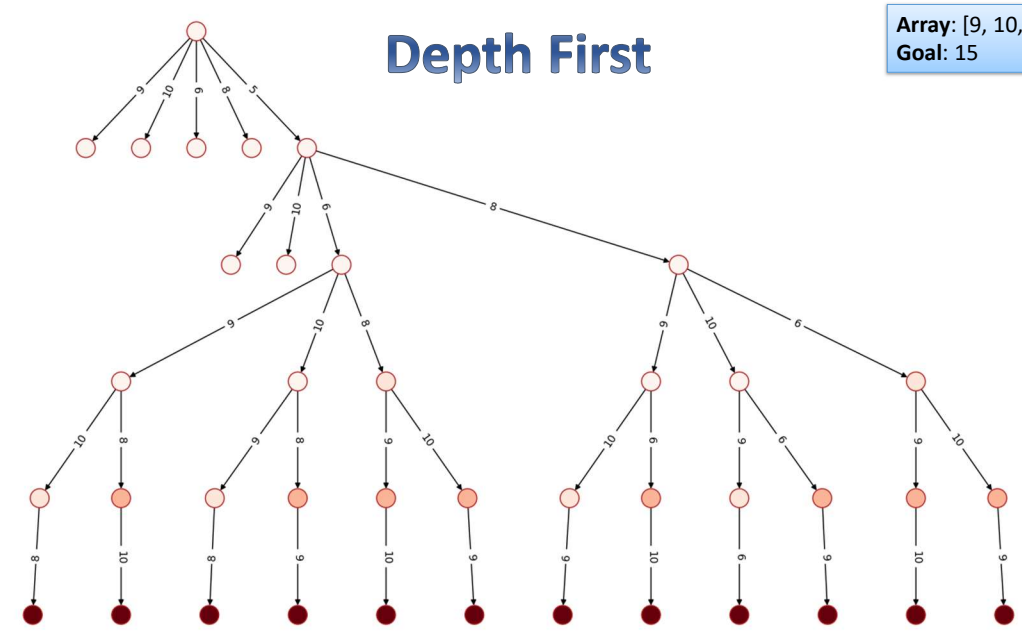

Depth First

Array: [9, 10, 6, 8, 5]
Goal: 15



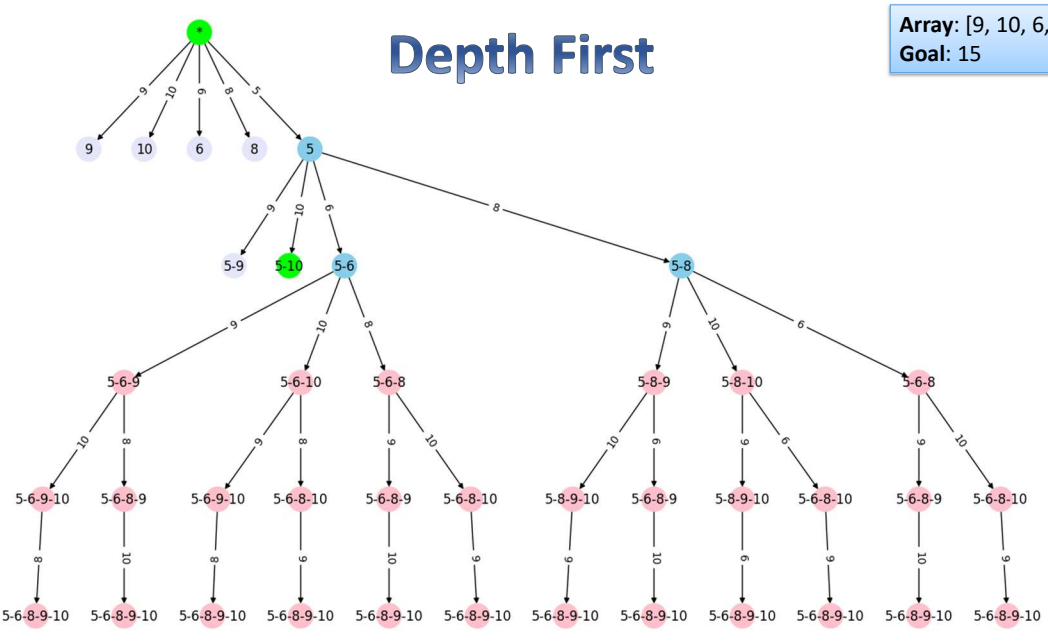
Depth First

Array: [9, 10, 6, 8, 5]
Goal: 15

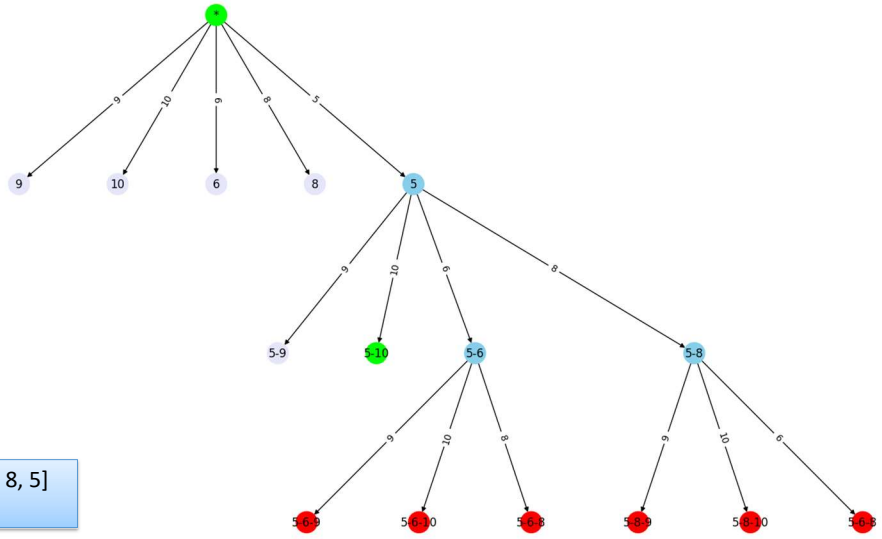


Depth First

Array: [9, 10, 6, 8, 5]
Goal: 15



Depth First + Bound



Array: [9, 10, 6, 8, 5]
Goal: 15

Uninformed strategies

- **Depth-limited** search
  - Depth-first + limited *diameter*
    (i.e., children are added to the frontier, but deeper nodes are discarded)
  - Not complete unless **iterative deepening** is also used

Uninformed strategies

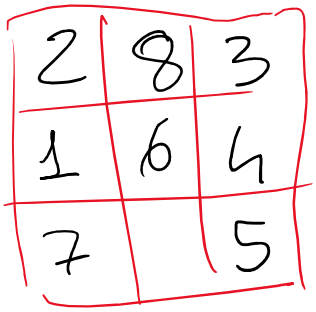
- **Beam** search
  - Uniform-cost search + limited number of nodes in each level (i.e., children are added to the frontier, then a certain number of nodes are trimmed from the frontier)
  - Not complete

Uninformed strategies

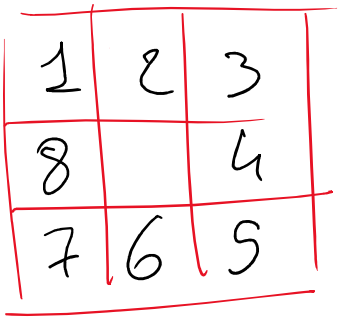
- **Bi-directional** search
  - Search forward from initial state; backward, from goal

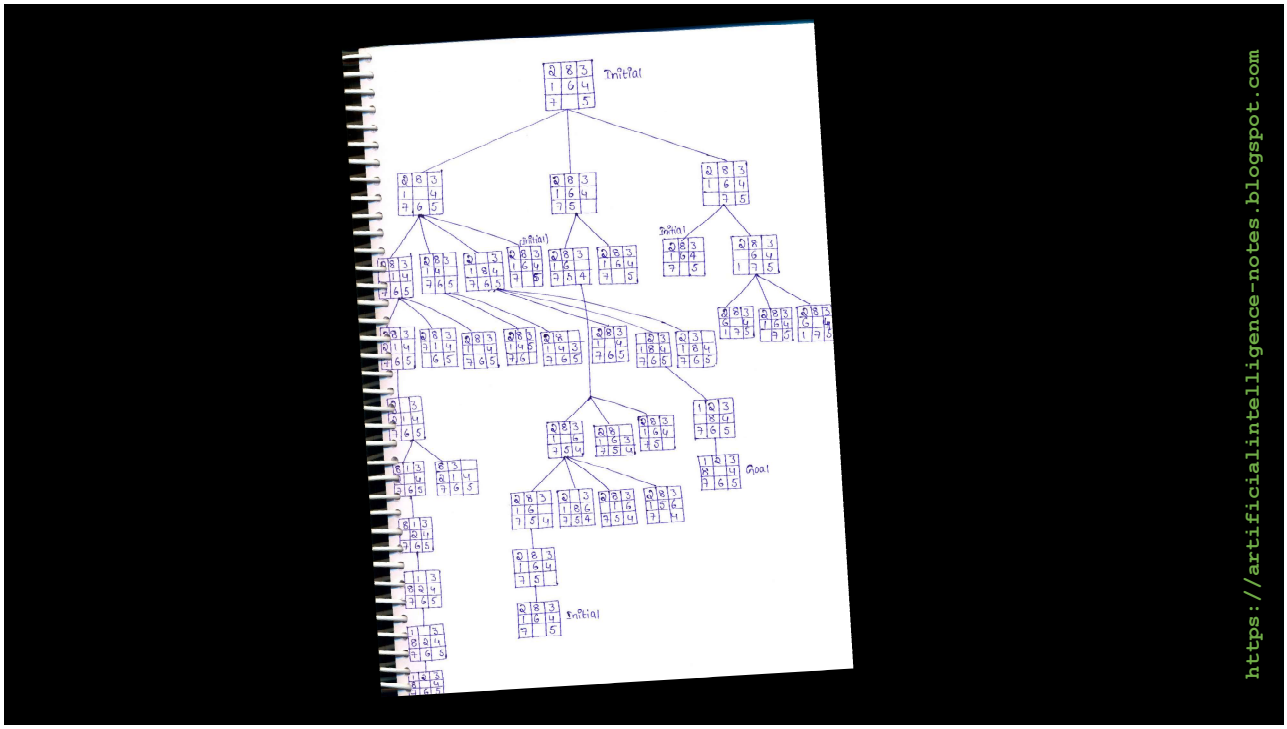
Problem: 8 Puzzle

- Start state



- Goal state





Informed strategies

- **Greedy best-first**
  - Expand the node with the maximum expected value
  - Not optimal
  - Incomplete (unless backtracking)

```
def h(state):
    return …

priority_function=lambda s: h(s)
```


Informed strategies

- **A* search**
  - Best-first using function:

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

  - g is the actual cost
  - h is the estimated cost (heuristic)
- **Complete** and **Optimally efficient**
  - It always computes the path with minimum cost by expanding a minimum number of nodes
  - NB: Under *reasonable* assumptions

$h(\cdot)$ is admissible

Branching factor is finite

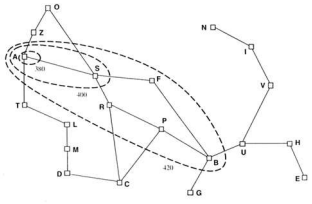
Informed strategies: A*

- $h(\cdot)$ is **admissible** if it never overestimates the cost to reach the destination node (in a *minimization* problem)
  - If the search space is a **tree**, admissible heuristics guarantee optimal solutions

Some textbooks use the term
A* Search
only if $h(\cdot)$ is admissible

Informed strategies: A*

- $h(\cdot)$ is **consistent** if for every node n and for every successor node n_s of n: $h(n) \leq cost(n, n_s) + h(n_s)$
  - I.e., $h(\cdot)$ is monotonic
  - If $h(\cdot)$ is consistent then it is also admissible
  - Consistent always heuristics guarantee optimal solutions (once a node has been expanded, it won't be reached again at a lower cost)
  - A* searches through contour



Informed strategies: A*

- Caveats:
  - "Usually" when `h(·)` is admissible it is also consistent
  - Any optimistic heuristic is admissible (e.g., *Euclidean distance to destination*, or simply *zero*)
- How to Create Admissible Heuristics?
  - Simplify the problem by relaxing constraints

Informed strategies: A*

- Caveats:
  - How to handle infinite branching factor?

Informed strategies: A*

- Space/Time complexity: $O(b^d)$
  (branching factor, depth of optimal solution)