Intro to Artificial Intelligence

Lecture 2: Problem Solving

Ahmed Sallam {http://sallam.cf / http://sallamah.weebly.com}

Goal

- The theory and technology of building agents that can plan ahead to solve problems
- These problems characterized by having many states.
- Example: navigation problem where there are many states and you need to pick the right choice now and in all the following states streaming together a sequence of actions that guarantee to reach the goal.

Not covered

- Navigation in fog is a different problem where the environment is partially observable and the possible paths are unknown .

Navigation Problem

Problem

- Find a route from Arad to Bucharest

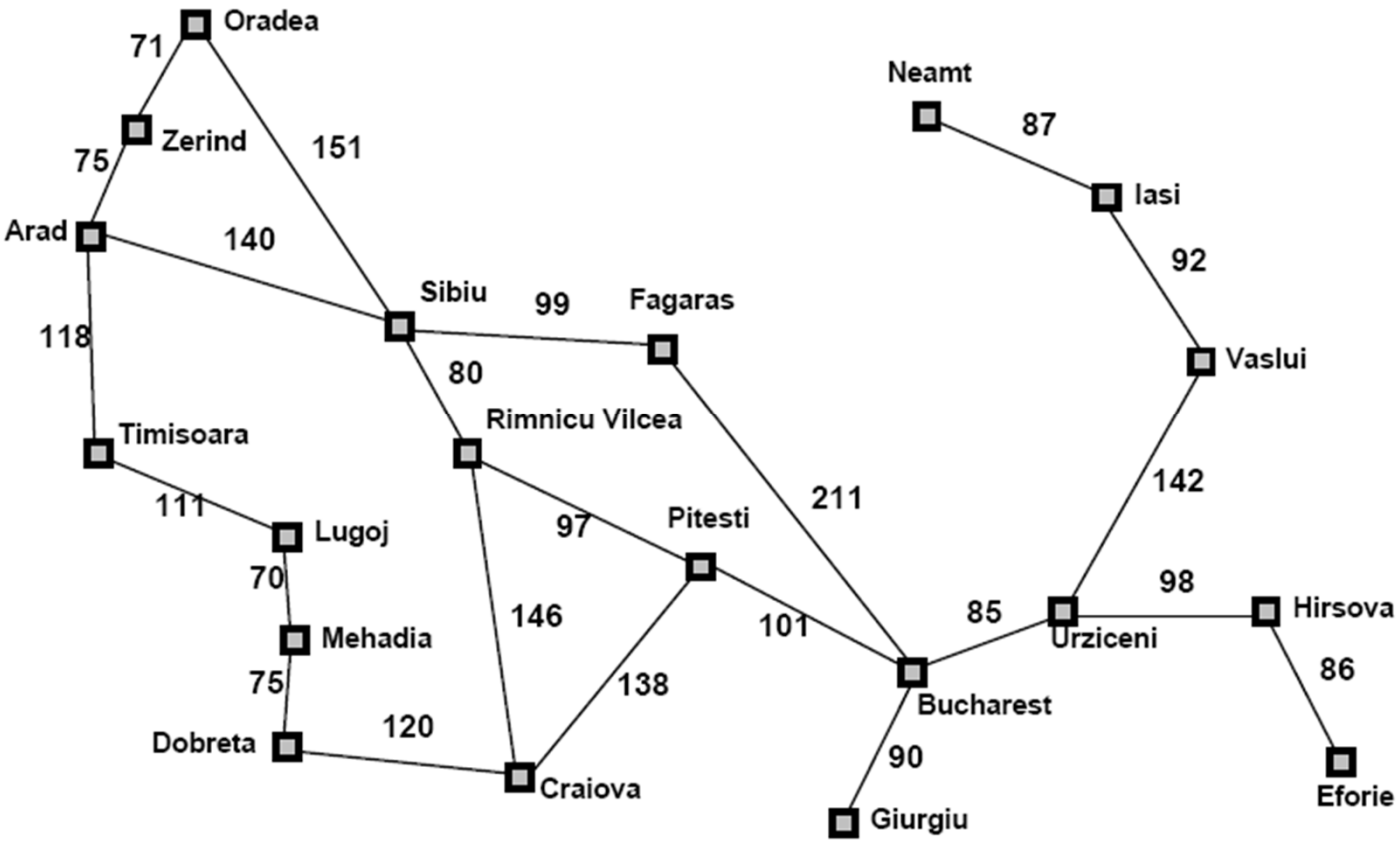
Problem definition

1. Initial state
2. Actions (state) -> {$action_1$,$action_2$,$action_{3, ...}$ }
3. Result (state, action) -> new state
4. GoalTeast (state) -> T / F
5. PathCost ($s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} \dots \xrightarrow{a_n} s_n$) -> cost

In this problem, the cost of the path is sum of the costs of Individual steps. So we need step cost function.

StepCost(s, a, s') -> cost #it may be number of KM or number of minuts

Problem definition



- After every action we separate states out into 3 parts:
  - End of the paths (Frontier states)
  - Explored states
  - Unexplored states

Solution

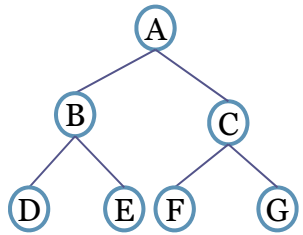
1. Initial state
2. Actions (state) -> {$action_1$,$action_2$,$action_3$, ...}
3. Result (state, action) -> new state
4. GoalTeast (state) -> T / F
5. PathCost ($s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} \ldots \xrightarrow{a_n} s_n$) -> cost

```
TreeSearch(problem)

        frontiers={initial}

        loop:
                if frontier is empty: return Fail

                path = frontiers.remove()          //choose frontier
                s=path.end

                if s is a goal: return path        // test goal

                for a in actions(s):               //expand the path
                        path.end（result(s,a) )
                        frontiers.add(path)
```

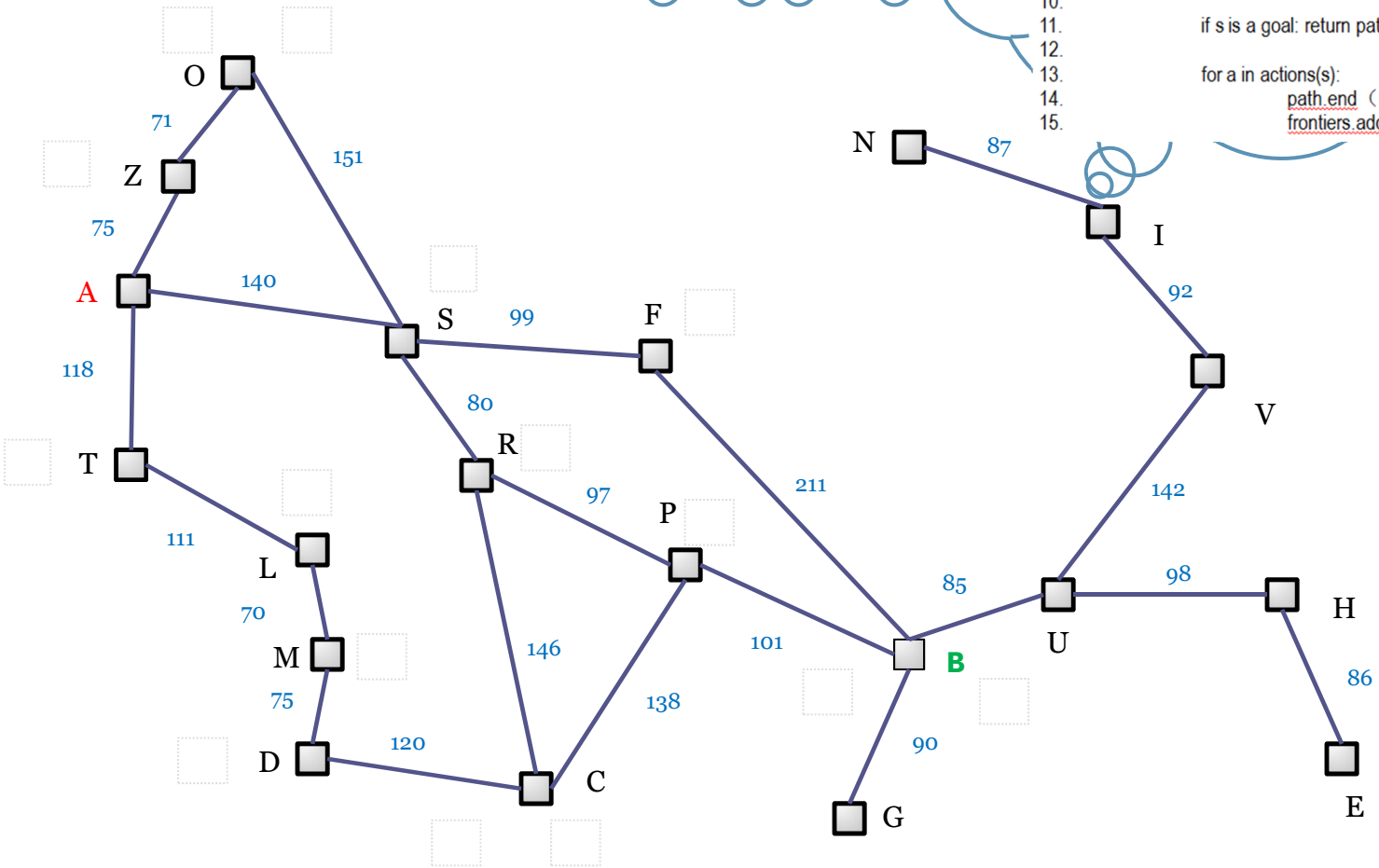
Searching Techniques

Breadth First Search

- Shortest path first search




```
TreeSearch(problem)

        frontiers={initial}

        loop:
                if frontier is empty: return Fail

                path = frontiers.remove()          //choose frontier
                s=path.end

                if s is a goal: return path        // test goal

                for a in actions(s):               //expand the path
                        path.end（result(s,a) )
                        frontiers.add(path)
```



Enhance Solution

To avoid repeated paths we replace the tree search by a graph search

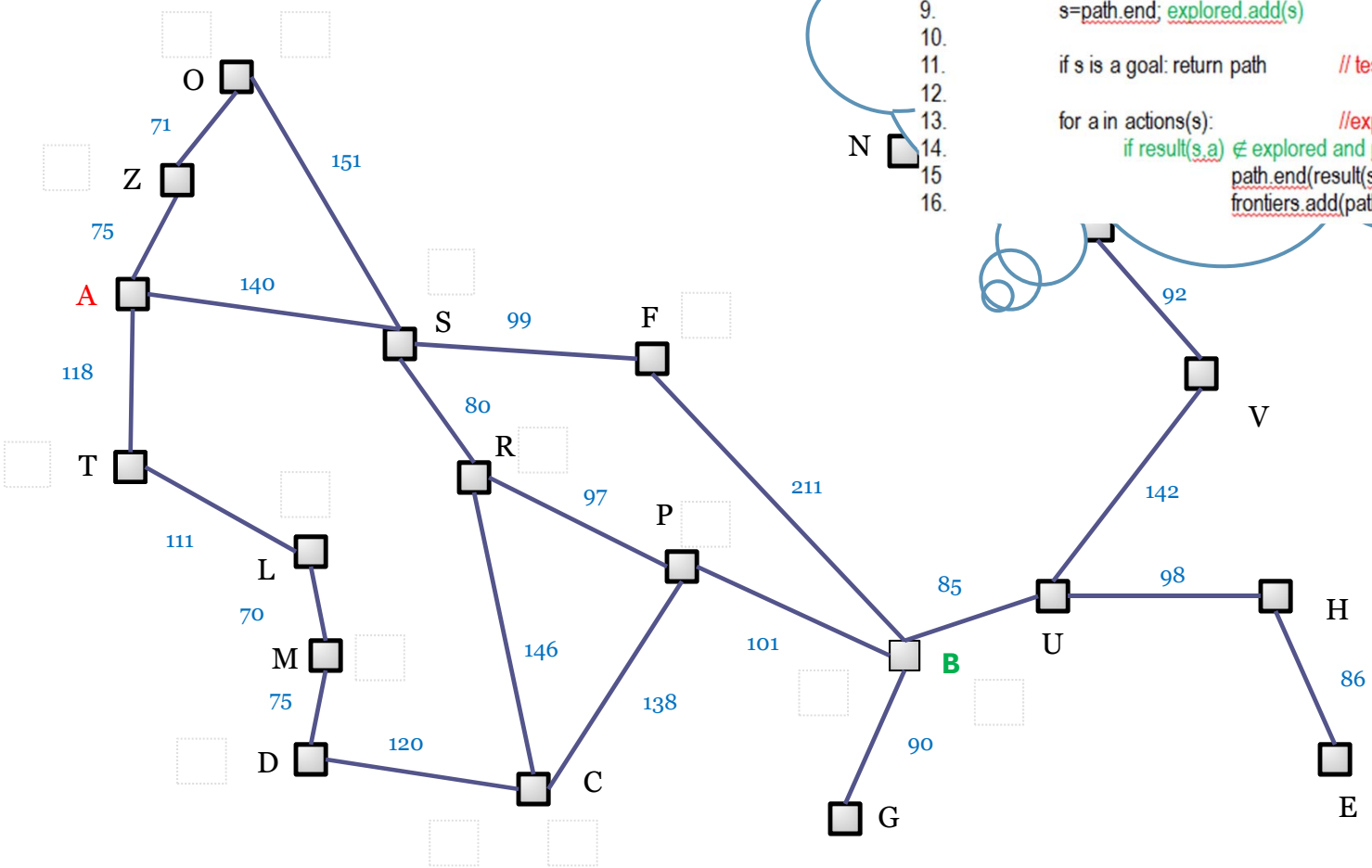
1. Initial state
2. Actions (state) -> {$action_1$, $action_2$, $action_3$, ...}
3. Result (state, action) -> new state
4. GoalTeast (state) -> T / F
5. PathCost ($s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} \ldots \xrightarrow{a_n} s_n$) -> cost

```
GraphSearch(problem)

        frontiers={initial}; explored={}

        loop:
            if frontier is empty: return Fail

            path = frontiers.remove()          //choose frontier
            s=path.end; explored.add(s)

            if s is a goal: return path        // test goal

            for a in actions(s):               //expand the path
                if result(s,a) ∉ explored and path.end(result(s,a)) ∉  frontiers
                        path.end(result(s,a)) //add s′ to the end of the path
                        frontiers.add(path)
```

Breadth First Search cont.

- Shortest path first search

```
GraphSearch(problem)

        frontiers={initial}; explored={}

        loop:
            if frontier is empty: return Fail

            path = frontiers.remove()          //choose frontier
            s=path.end; explored.add(s)

            if s is a goal: return path        // test goal

            for a in actions(s):               //expand the path
                if result(s,a) ∉ explored and path.end(result(s,a)) ∉ frontiers
                        path.end(result(s,a))
                        frontiers.add(path)
```



BFS and Brute-force

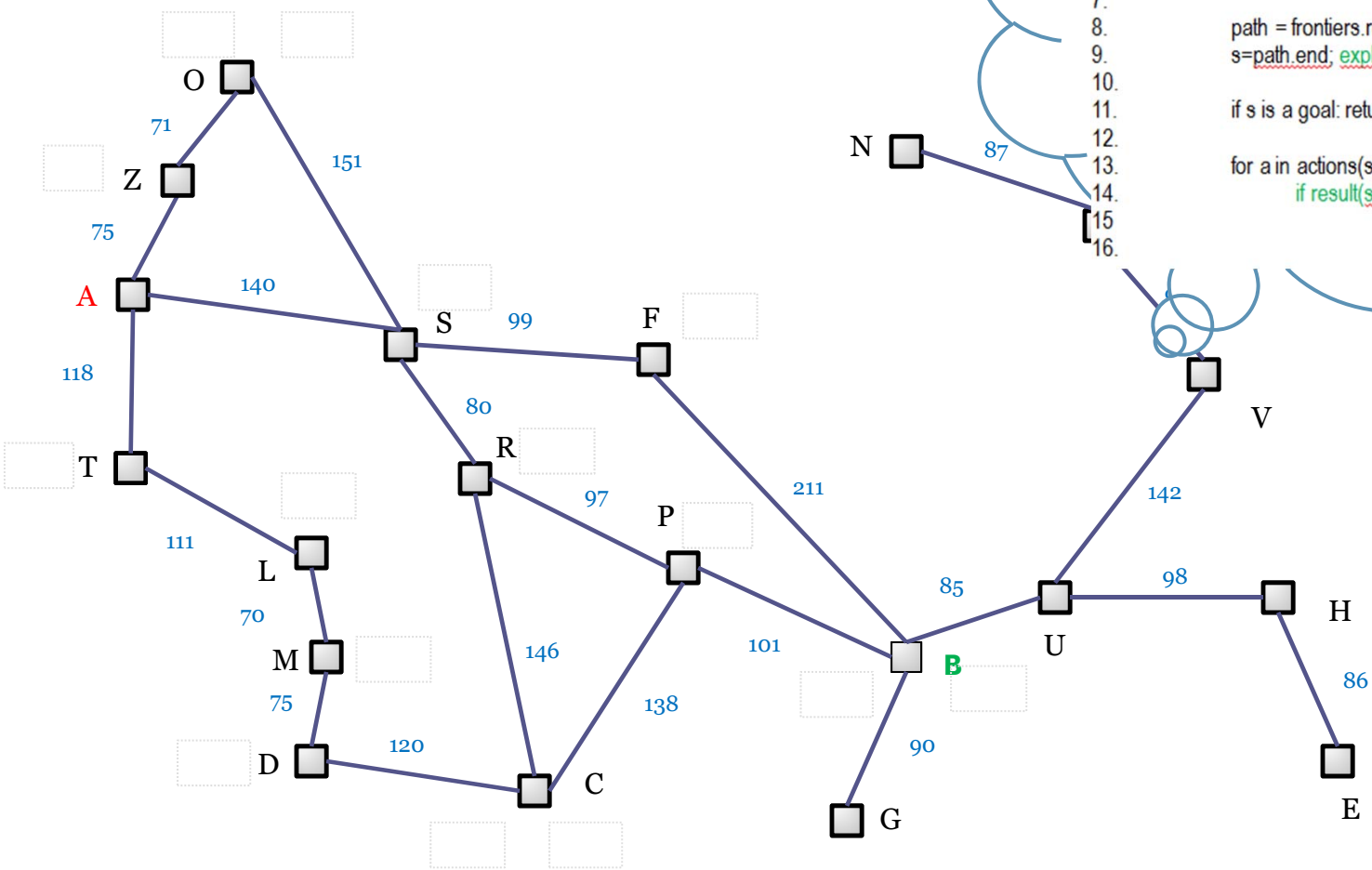
```
1. GraphSearch(problem)
2.
3.          frontiers={initial}; explored={}
4.
5.          loop:
6.              if frontier is empty: return Fail
7.
8.              path = frontiers.remove()    //choose frontier
9.              s=path.end; explored.add(s)
10.
11.             if s is a goal: return path    // test goal
12.
13.             for a in actions(s):                    //expand the path
14.                 if result(s,a) ∉ explored and path.end(result(s，a)) ∉ frontiers
15.                     path.end(result(s,a))
16.                     frontiers.add(path)
```

- Why we test the goal (line 11) after choosing the frontier , not after expanding the path???
- Does Breadth first search require this order?
  - It depends on what “shortest” means, if it means length we can enhance this order to terminate after expanding. However, if it means the distance, this order becomes important.
  - So the answer here is No, BFS don’t need this order and we can test the goal directly after expanding.

Uniform-Cost Search

- Cheapest first search

```
GraphSearch(problem)

        frontiers={initial}; explored={}

        loop:
            if frontier is empty: return Fail

            path = frontiers.remove()          //choose frontier
            s=path.end; explored.add(s)

            if s is a goal: return path        // test goal

            for a in actions(s):               //expand the path
                if result(s,a) ∉ explored and path.end(result(s,a)
                    path.end(result(s,a))
                    frontiers.add(path)
```



UCS and Brute-force

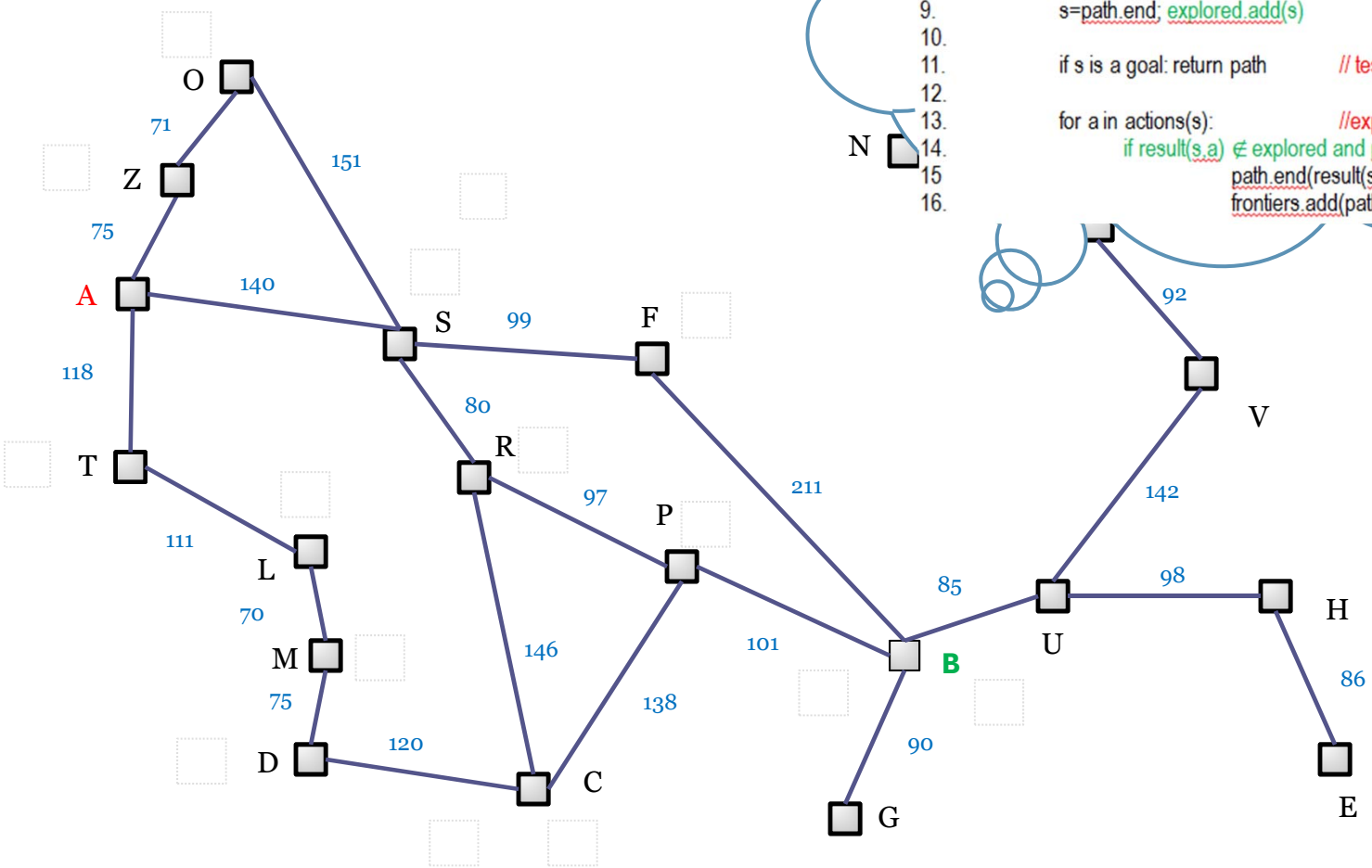
```
1. GraphSearch(problem)
2.
3.          frontiers={initial}; explored={}
4.
5.          loop:
6.              if frontier is empty: return Fail
7.
8.              path = frontiers.remove()    //choose frontier
9.              s=path.end; explored.add(s)
10.
11.             if s is a goal: return path    // test goal
12.
13.             for a in actions(s):                    //expand the path
14.                 if result(s,a) ∉ explored and path.end(result(s,a)) ∉ frontiers
15.                     path.end(result(s,a))
16.                     frontiers.add(path)
```

- Again: Why we test the goal (line 11) after choosing the frontier , not after expanding the path???
- Does Uniform-Cost search require this order?
  - Yes, this order is important here.

Depth First Search

- longest path first search

```
GraphSearch(problem)

        frontiers={initial}; explored={}

        loop:
            if frontier is empty: return Fail

            path = frontiers.remove()          //choose frontier
            s=path.end; explored.add(s)

            if s is a goal: return path        // test goal

            for a in actions(s):               //expand the path
                if result(s,a) ∉ explored and path.end(result(s,a)) ∉ frontiers
                        path.end(result(s,a))
                        frontiers.add(path)
```

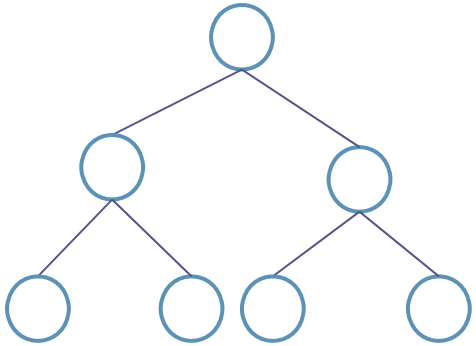


Search Comparison

Optimality

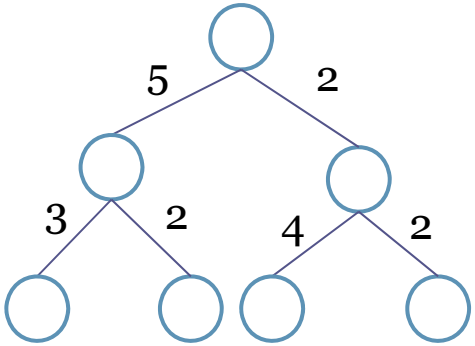
- Which algorithm is optimal with <u>respect to the frontiers Selection Criteria</u> ?

Breadth first



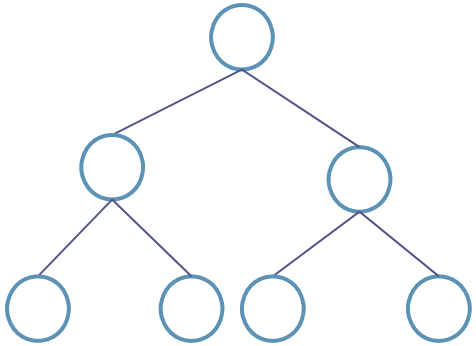
✓

Cheapest first



✓

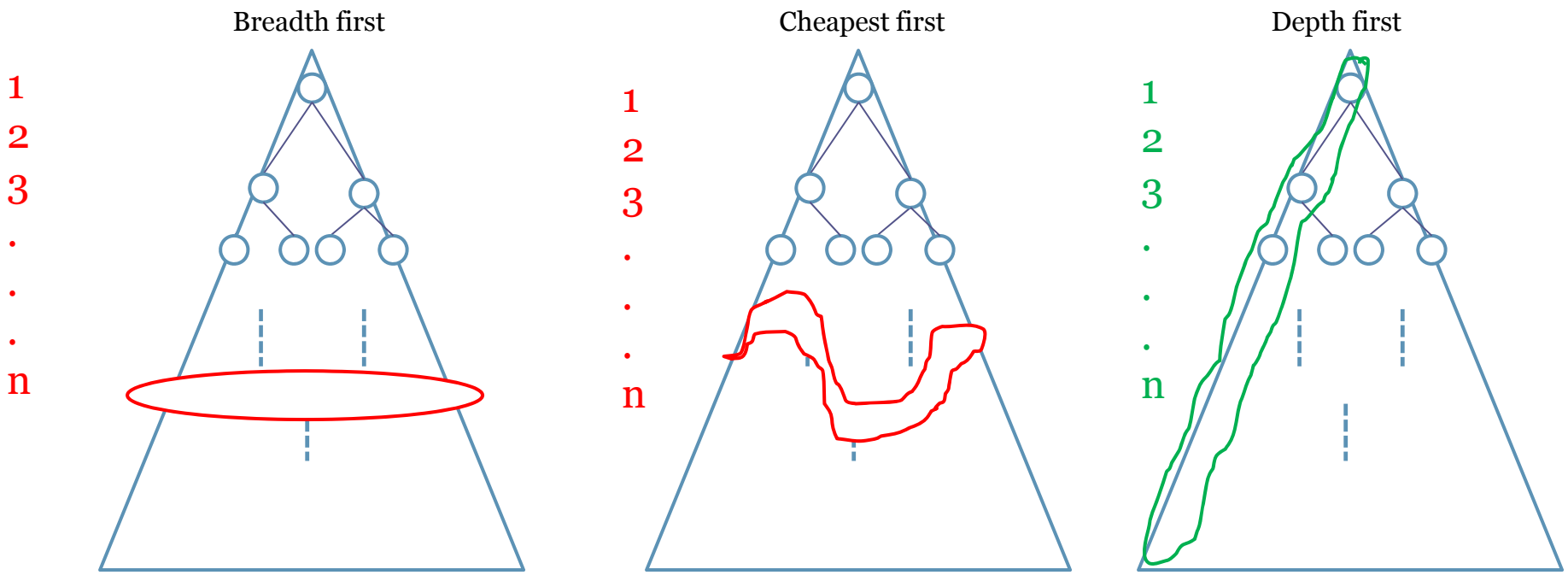
Depth first



✗

Memory

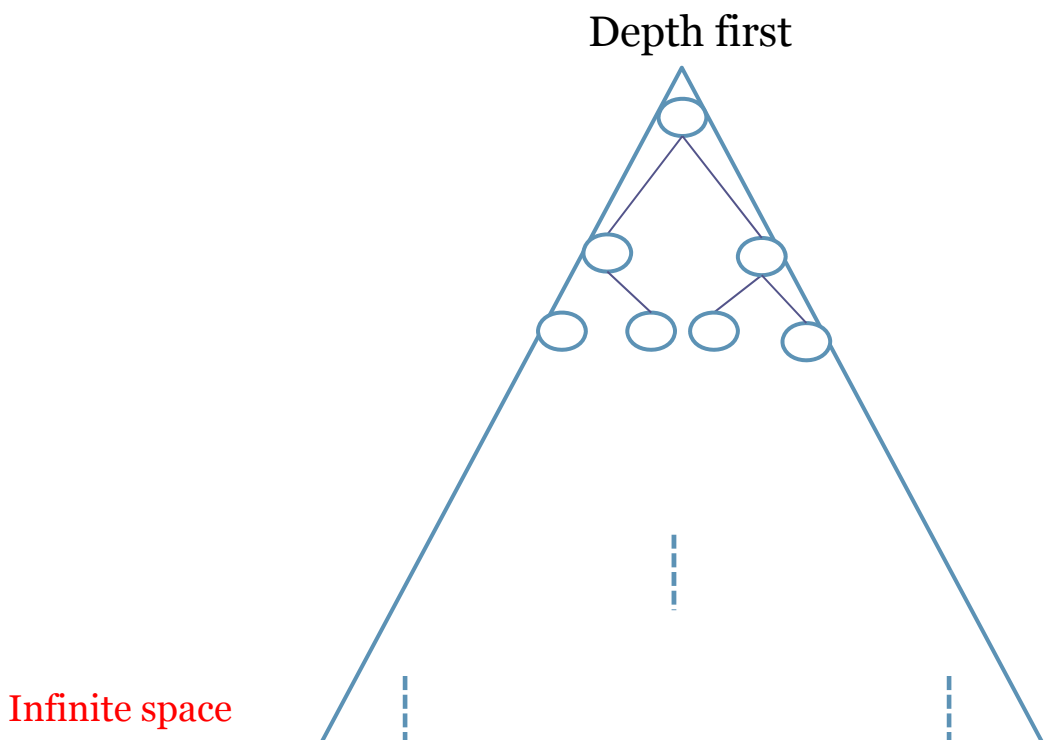
- Depth First search is not optimal, why we still using it?
  - Because it saves the memory when we have a large tree.



2^n $\approx 2^n$ n

Completeness

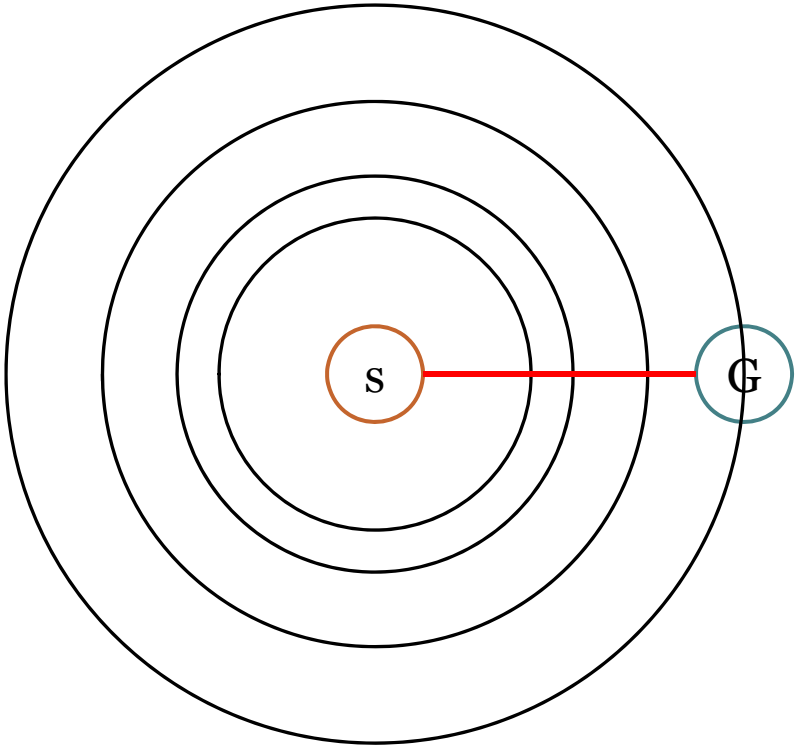
- Does Depth First guaranty to find a solution? (Completeness)
  - No, because the solution depends on the problem space. In other words, if the problem has an infinite space then Depth First could miss the right path forever.



Combine the benefits

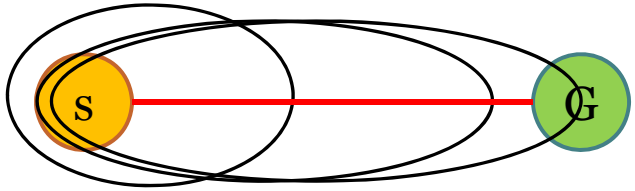
Refereeing to the Uniform-Cost Search

- In uniform-cost search the searching contour with respect to the cost (distance) looks as following:



The Idea

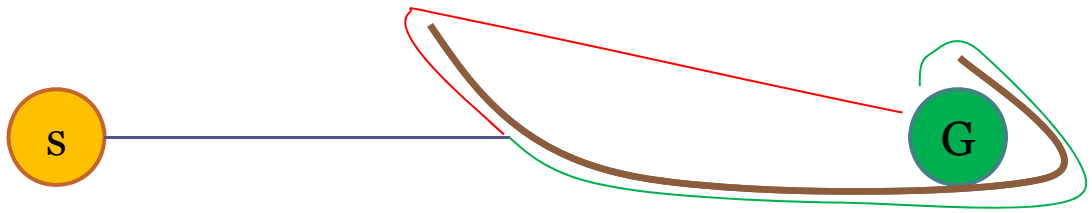
- If we have some more information about the distance between S and G we can direct the search toward the goal.



- This known as: Greedy best-first search

The Idea cont.

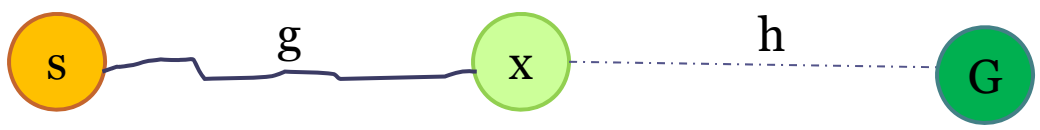
- Greedy best-first search depends on estimated distance between S and the goal.



- This would result to accept a longer path under some conditions.
- Can we can combine the benefits of Uniform-Cost search and Greedy best first search ?

A* search (best estimated total path cost first)

- Combines the benefits of Uniform-Cost search and Greedy best first search with a new search function f=g+h.
  - g(path)=path cost
  - h(path)=estimated distance to the goal.

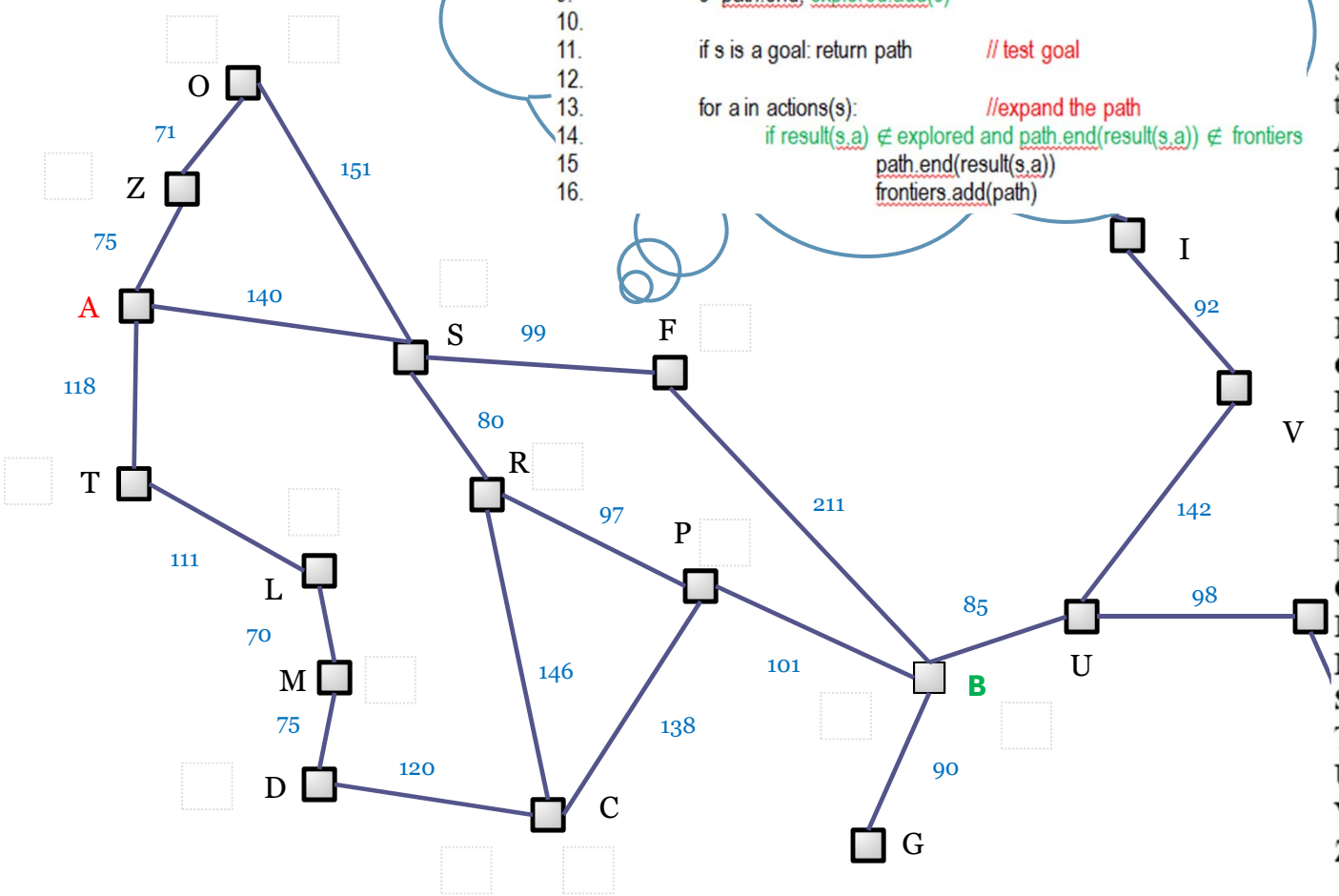


- Benefits:
  - Minimizing ‘g’ keep the path cheap.
  - Minimizing ‘h’ keep focus on finding the goal.

A* search

- best estimated total –

```
GraphSearch(problem)

    frontiers={initial}; explored={}

    loop:
        if frontier is empty: return Fail

        path = frontiers.remove()          //choose frontier
        s=path.end; explored.add(s)

        if s is a goal: return path        // test goal

        for a in actions(s):               //expand the path
            if result(s,a) ∉ explored and path.end(result(s,a)) ∉ frontiers
                path.end(result(s,a))
                frontiers.add(path)
```



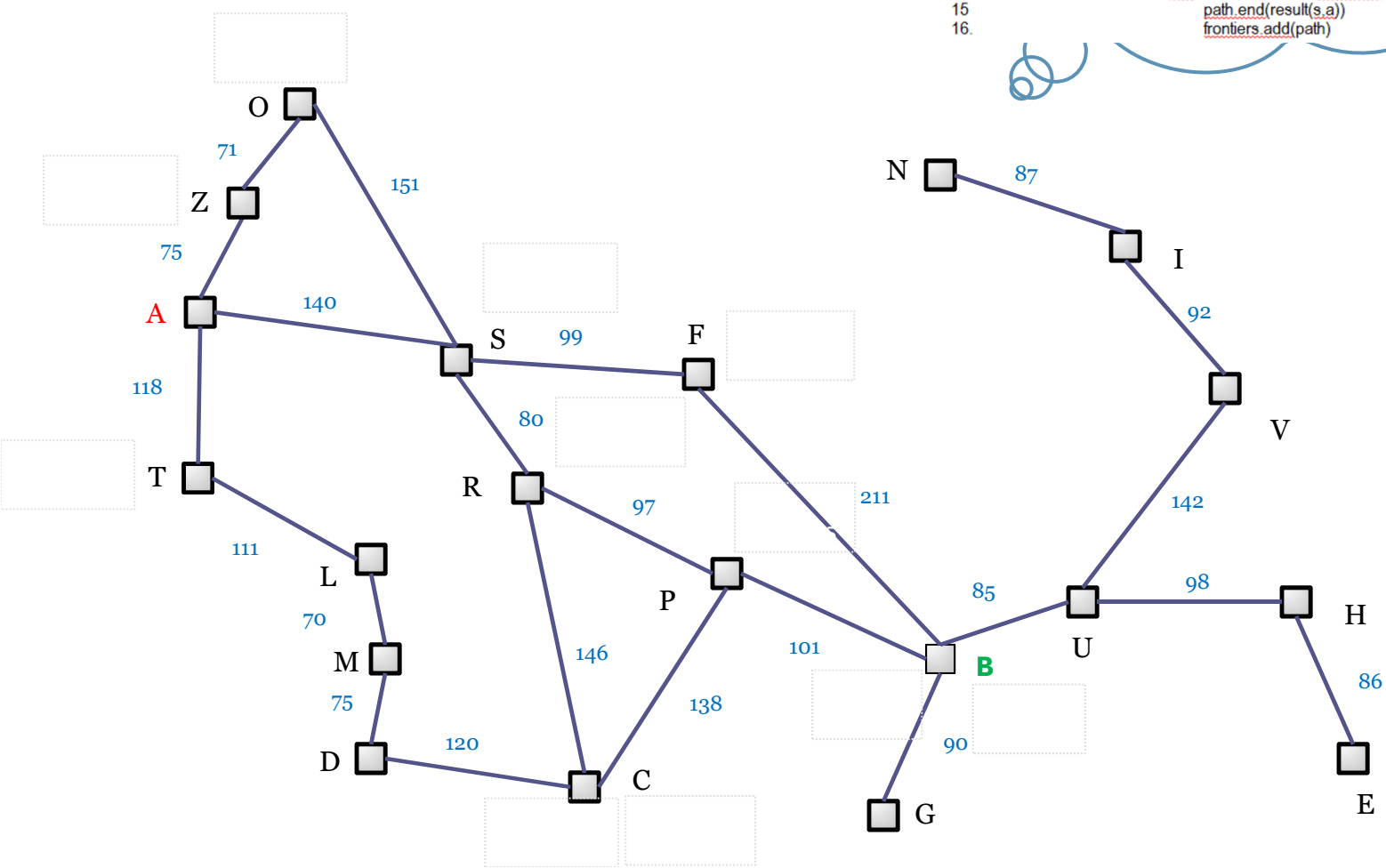
h

Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

A* search

- Best estimated total path cost first)
- F=g+h

```
GraphSearch(problem)

    frontiers={initial}; explored={}

    loop:
        if frontier is empty: return Fail

        path = frontiers.remove()          //choose frontier
        s=path.end; explored.add(s)

        if s is a goal: return path        // test goal

        for a in actions(s):               //expand the path
            if result(s,a) ∉ explored and path.end(result(s,a)) ∉ frontiers
                path.end(result(s,a))
                frontiers.add(path)
```



h

Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

A* analysis

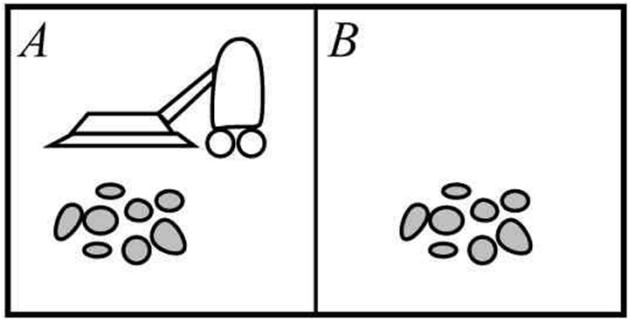
- Does A* always find the lost cost path?

  ❑Yes
  ❑No, depends on the problem
  ☑No, depends on h

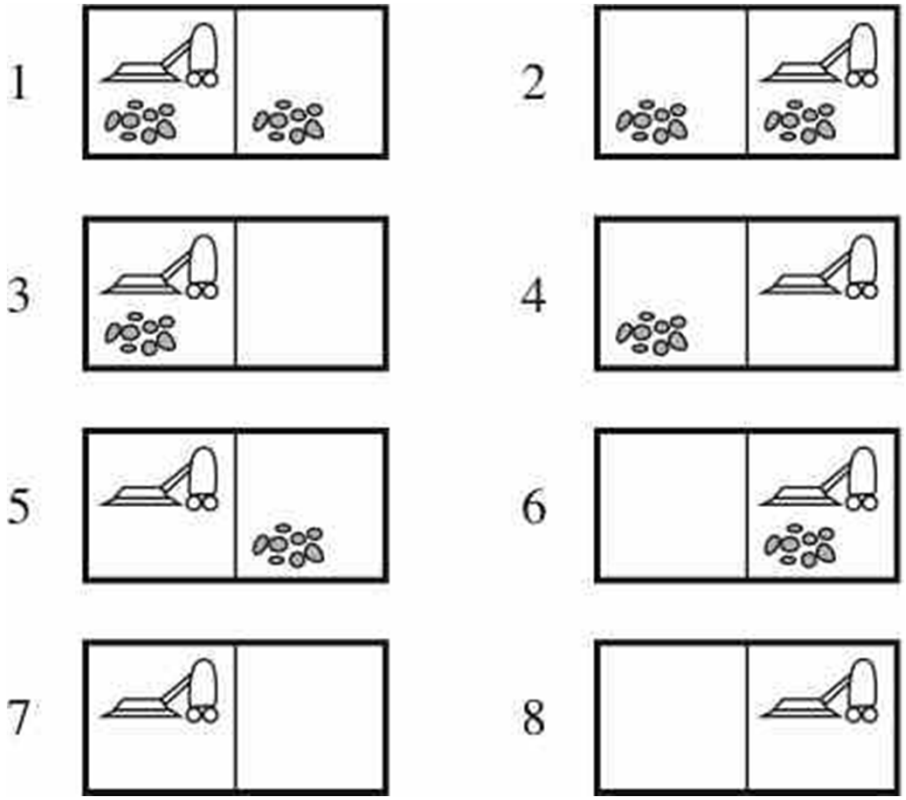
- Lowest cost only if h(s)< true cost (s->g)
  - This means 'h' never overestimate distance to the goal
  - 'h' in this called <u>optimistic</u>
  - Also 'h' is <u>admissible</u> to find the lowest cost path.

State Space

Vacuum cleaner world

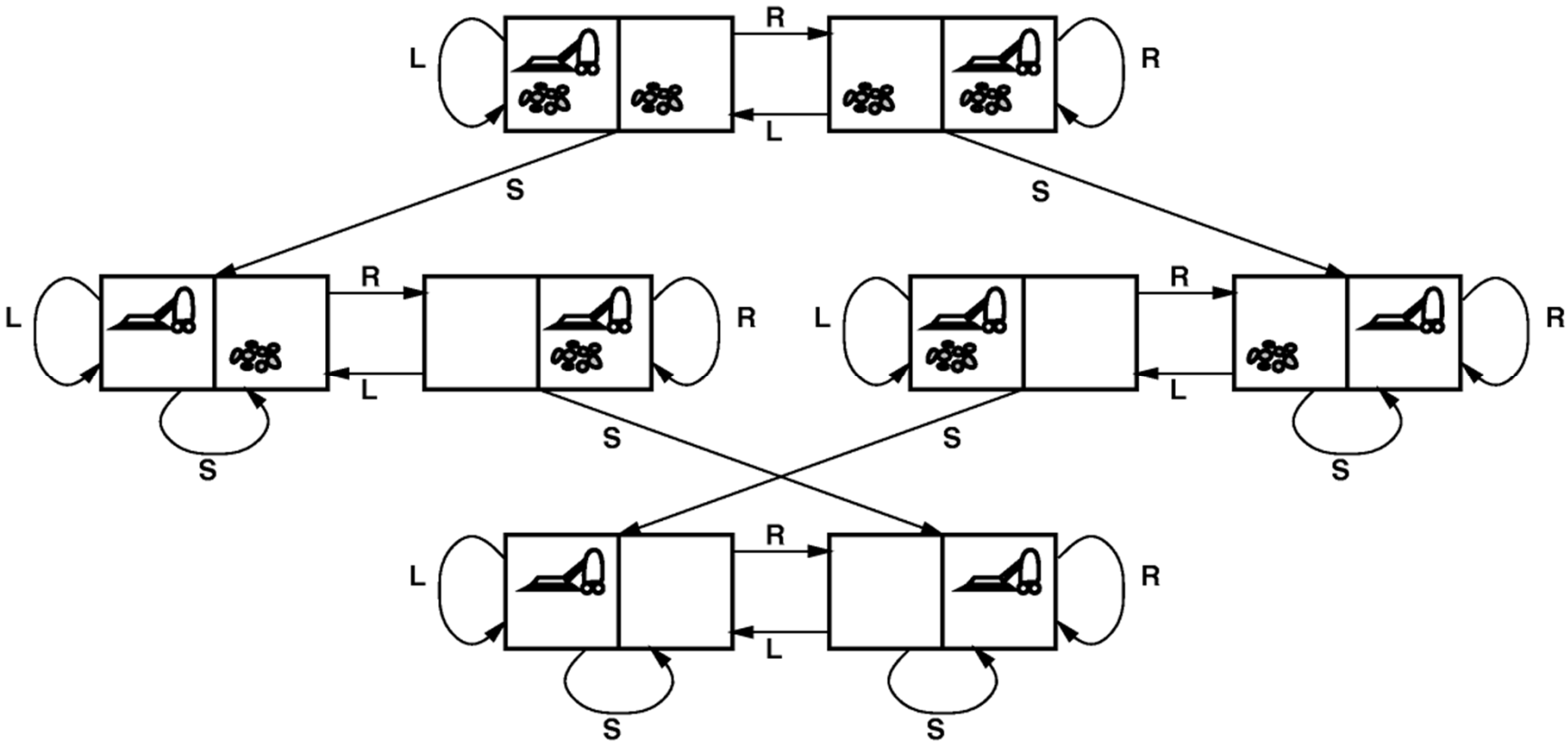


$2 * 2^8$

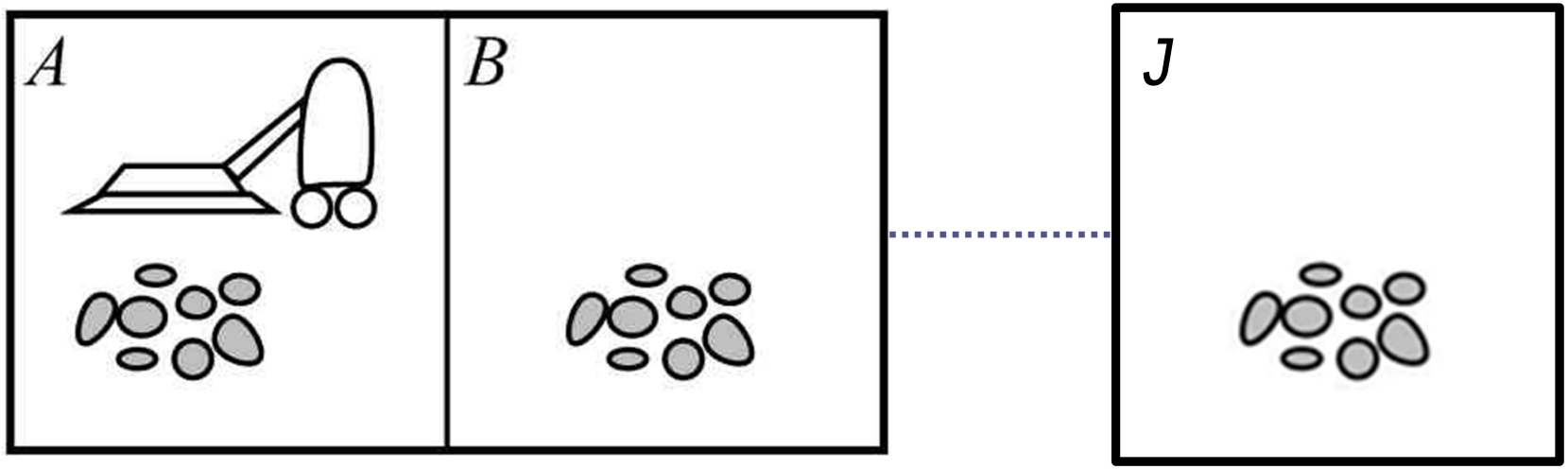


Vacuum cleaner state space

- Goal formulation: intuitively, we want all the dirt cleaned up. Formally, the goal is { state 7, state 8 }.
- Problem formulation: we already know what the set of all possible states is. The actions/operators are "move left", "move right", and "vacuum".



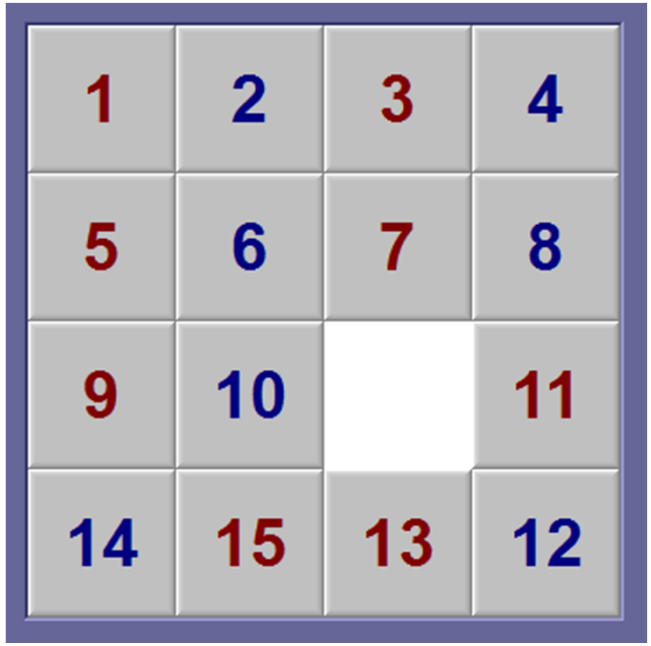
Why we need search techniques?



- Power (On / Off / Sleep)
- Camera (On /Of)
- Brush Height (1, 2, 3, 4, 5)

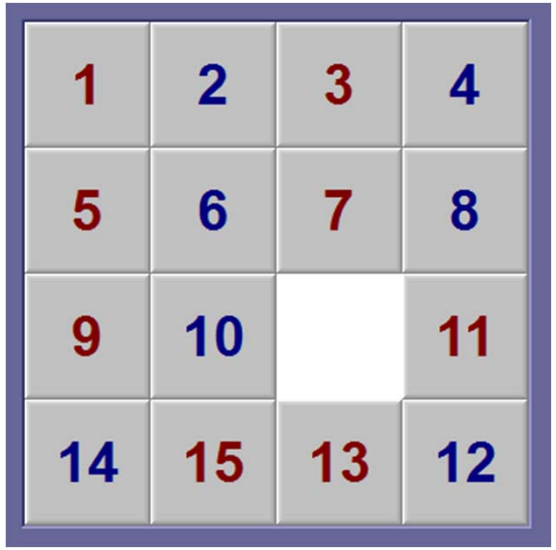
- States=10 * 2^{10} * 3 * 2 * 5 (Large state space)

Sliding blocks puzzle



- h_1= no. of misplaced blocks
- $h_2 = \sum_{i=1}^{15} (no. distances\ for\ i\ to\ the\ correct\ place)$

Sliding blocks puzzle



Which one is admissible ?

- ☑ h_1= no. of misplaced blocks
- ☑ $h_2 = \sum_{i=0}^{15}(no.\,distances\ for\ i\ to\ the\ correct\ place)$

$h_1 < h_2$, thus h_2 expand fewer pathes

Where is the Intelligence ?

8	10	15	1
11		5	14
12	6	3	9
4	7	2	13

- h_1= no. of misplaced blocks
- $h_2=\sum_{i=0}^{15}(no.\,distances\ for\ i\ to\ the\ correct\ place)$

- The intelligence if the agent was able to automatically find a good heuristic function.
- e.g. the agent know that: a block can move from A to B
  - ~~If (A adjacent to B)~~ h_1
  - ~~If (B is blank)~~ h_2

Where is the Intelligence ? cont.

8	10	15	1
11		5	14
12	6	3	9
4	7	2	13

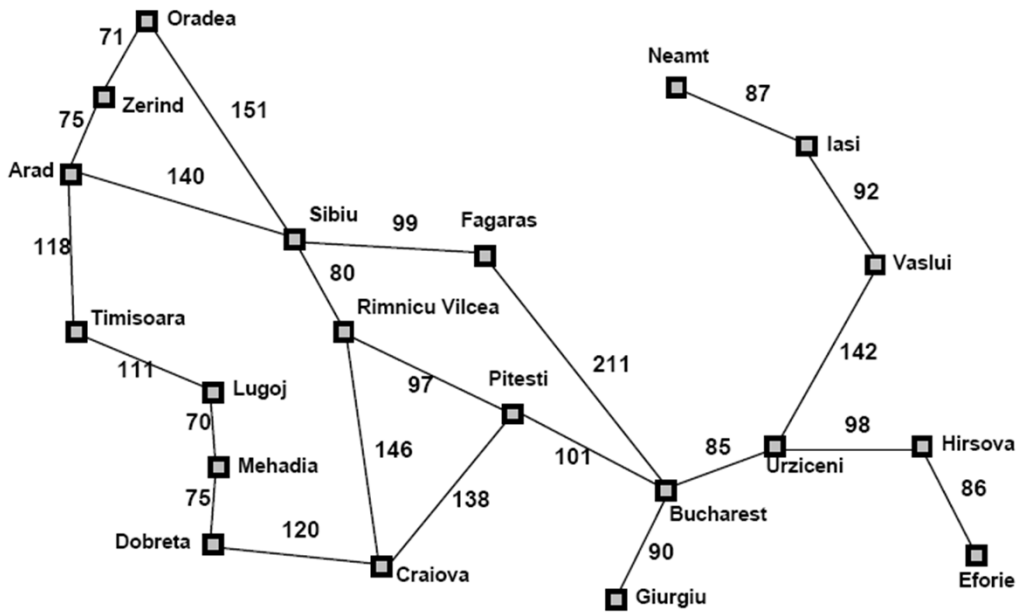
- h_1= no. of misplaced blocks
- $h_2=\sum_{i=0}^{15}(no.\,distances\ for\ i\ to\ the\ correct\ place)$

- So heuristic rules can be derived automatically from the formal description of the problem.
- A good heuristic would be
  - $h = \max(h_1, h_2)$

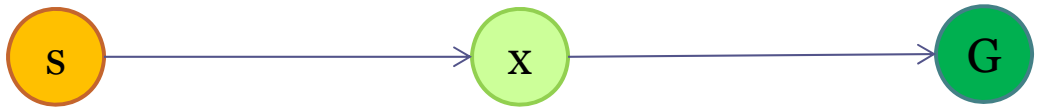
Notes

Problem-solving works when:

- Fully Observable
- Discrete
- Deterministic
- Static



Implementation



- Path can be represented by data structure "Node"