Search—
Uninformed Search

KUO-SHIH TSENG (曾國師)
DEPARTMENT OF MATHEMATICS
NATIONAL CENTRAL UNIVERSITY, TAIWAN

2021/03/03

Course Announcement

- This is a “Modern” AI course. You will receive a modern training.
- If you are an undergraduate student, you should not take more than 3 courses this semester.
- If you are a 2nd year M.S. student, you will have the final project and your thesis defense simultaneously.
- Think about what’s a good decision for you.
- Change your eeclass email. Then, you can receive course announcement from your email.
- Start to think about your final project topic.

Course Announcement

- If you enrolled in this course, you can access to Robotics Lab (M213) via your NCU ID since this week.
- Robotics Lab has
  - Minibot X 10
  - Turtlebot3 X10
  - Bebop X8
  - PC X1
  - Notebook X2
  - Monitor X4

Course Announcement

- HW1 was released on 3/02. Download it from eeclass.
- 1 programming problem and 2 theory problems.
- Deadline: 3/24(Wed.) 00:00am
- Delivery: Please update your HW to eecalss using electrical format. Compress your HW into a zip file including PDF and code.
- **Late policy**: If your HW is late for 1 day, the discount rate is 0.8. For 2 days, the discount rate is 0.8^2. and so on.
- Start to work on it this week. You have 3 weeks to work.

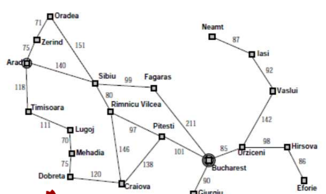
Outline

- Problem solving
- Formulate problems
- Uninformed search and informed search
- Uninformed search
  - BFS
  - DFS
  - Uniform-cost search
  - Depth-limited search
  - Iterative deepening depth-first search
  - Bi-directional search

Outline

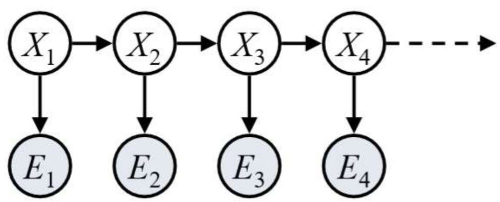
[Problem solving]

- Search problems
- Adversarial Search



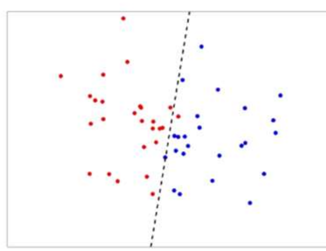
[Perception and Uncertainty]

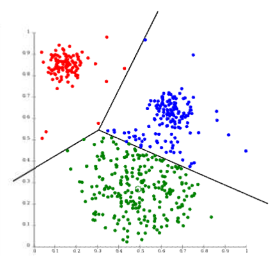
- Bayes Theorem
- Bayes Filter and Smoothing

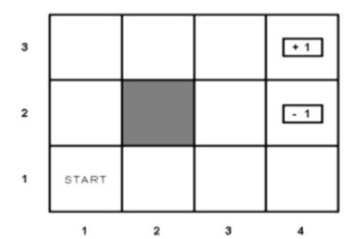


[Learning and Decision-making]

- Supervised learning
- Unsupervised learning
- Reinforcement learning





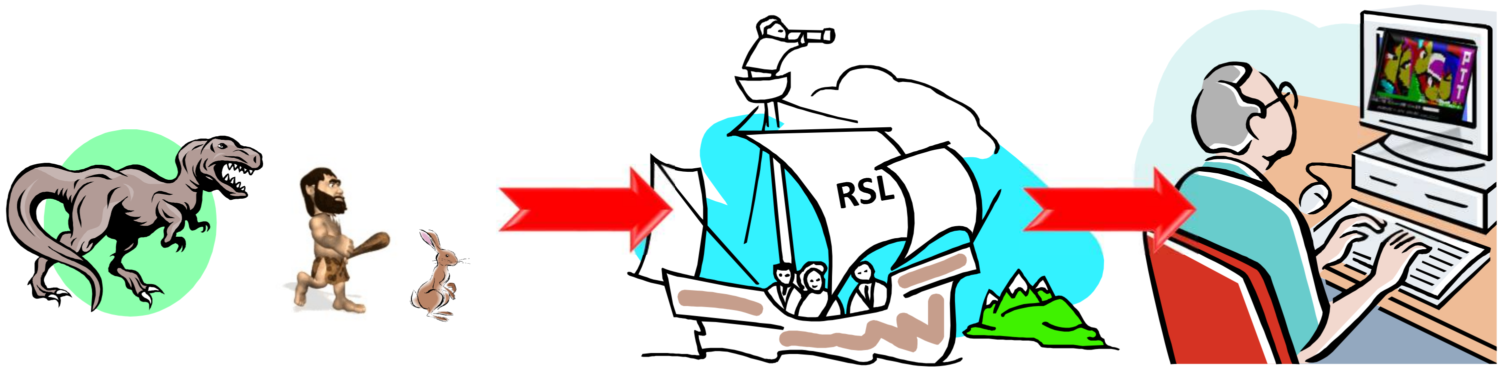


*Human have been involved in **search activities***

from the beginning of their history: Searching for

- ***food** and **shelter** in prehistoric times*
- *new **continent** countries in the Middle Ages*
- ***information** in the digital age.*

- Probabilistic Search for Tracking Targets, 2013.



Problem solving

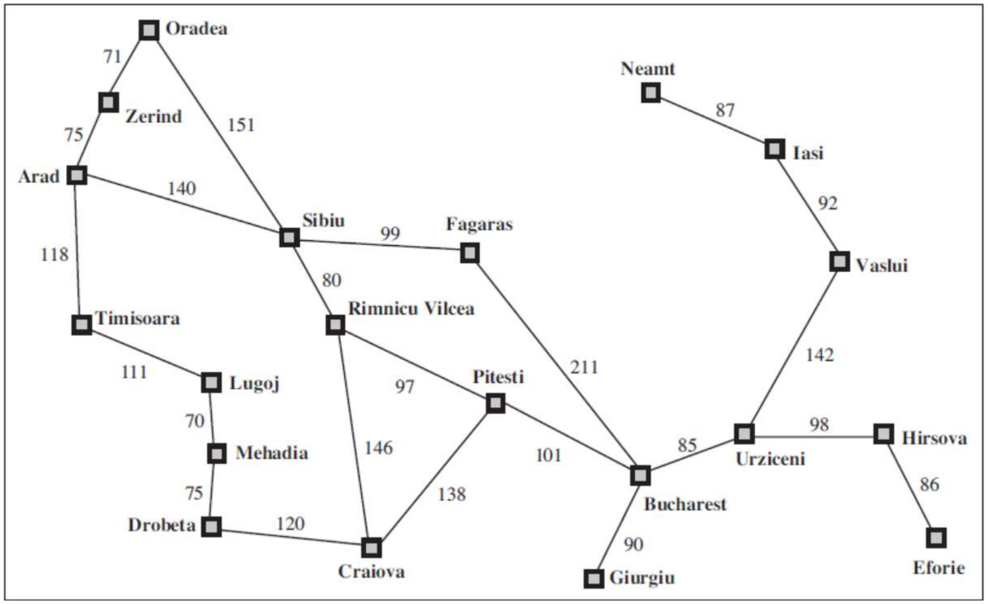
- In classical AI, researchers tried to solve problems via programs. These problems are like puzzles instead of real problems.
- These problems are deterministic **without any probability**.
- Hence, if these problems can be expanded to a search tree, searching for a solution from this tree is an intelligent behavior!
- There are a lot of examples. One of them is you used everyday – Google Map!

Problem solving

- Google Map
- Given:
  - Current position
  - Destination
  - Path cost of between nodes
- Find the shortest path

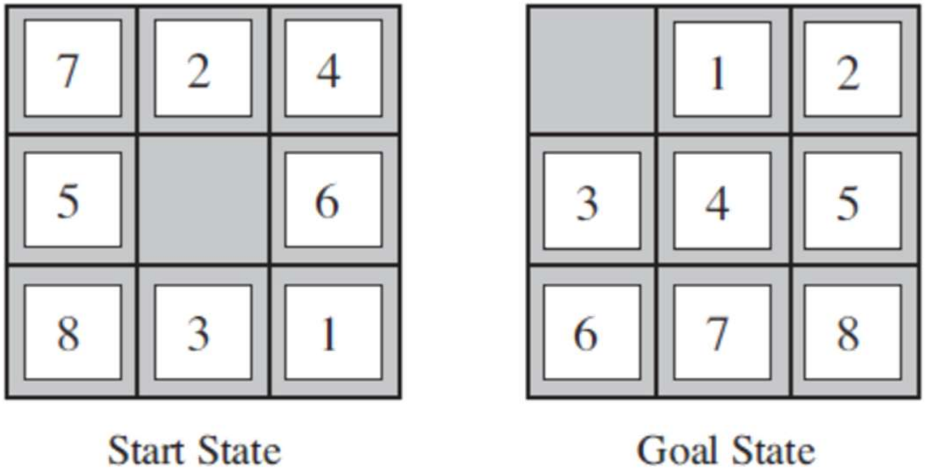
Problem solving

- Romania-distance
- Given:
  - Current position
  - Destination
  - Path cost of between nodes
- Find the shortest path



Problem solving

- 8-puzzle
- Given:
  - Start state, state and action
  - Goal state
- Find:
  - Optimal actions

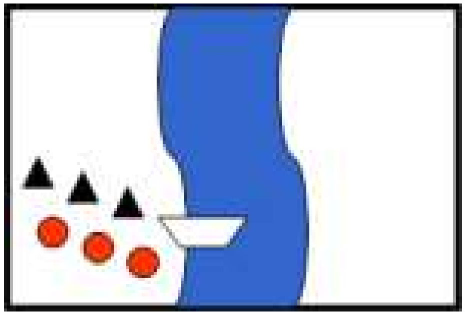


Problem solving

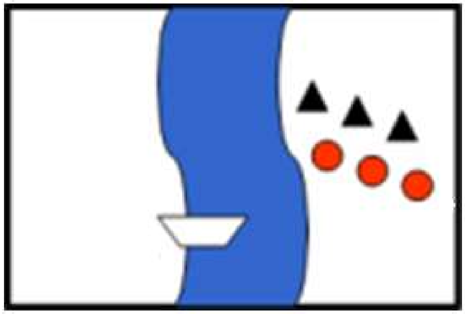
- Missionaries and Cannibals
- Given:
  - Start state, state and action
  - Goal state
- Find:
  - Optimal actions

On one bank of a river are three missionaries and three cannibals. There is one boat available that can hold up to two people and that they would like to use to cross the river. If the cannibals ever outnumber the missionaries on either of the river's banks, the missionaries will get eaten. How can the boat be used to safely carry all the missionaries and cannibals across the river?

http://www.aiai.ed.ac.uk/~gwickler/missionaries.html
https://www.novelgames.com/en/missionaries/



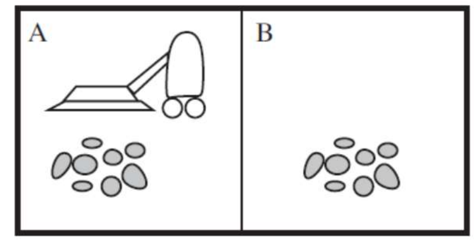
Start state



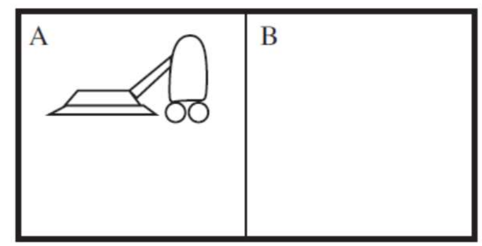
Goal state

Problem solving

- Vacuum world
- Given:
  - Start state, state and action
  - Goal state
- Find:
  - Optimal actions



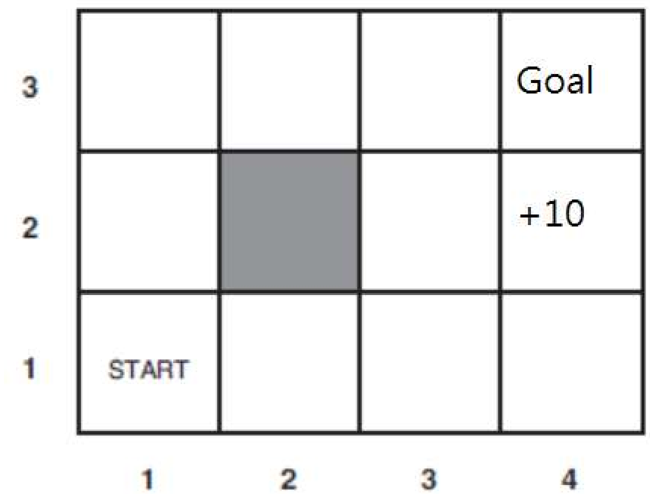
Start state



Goal state

Problem solving

- 4X3 world
- Given:
  - Start state, state and action
  - Goal state
- Find:
  - Optimal actions



Formulate problems

- States (S): the states of the problem
- Initial states (S0): the beginning state
- Actions (A): the action space of the agent
- Transition model: given the action, the output state ← **deterministic**
  (s,a→s')
- Goal test: test if the current state is goal
- Path cost: the cost of each action

Uninformed search and informed search

- Uninformed search:
  - There is NO additional information (only known path cost) beyond the given problem definition. It's also called blind search. The agent only can check if the current state is the goal state after actions.
- Informed search (Heuristic search):
  - There is additional information beyond the given problem definition. The agent can use this information to search more efficiently than blind search.

$$f(i) = \underbrace{g(i)}_{past} + \underbrace{h(i)}_{future}$$

$$\begin{cases} \text{Uninformed search}: f(i) = g(i) \\ \text{Heuristic search}: f(i) = g(i) + h(i) \end{cases}$$

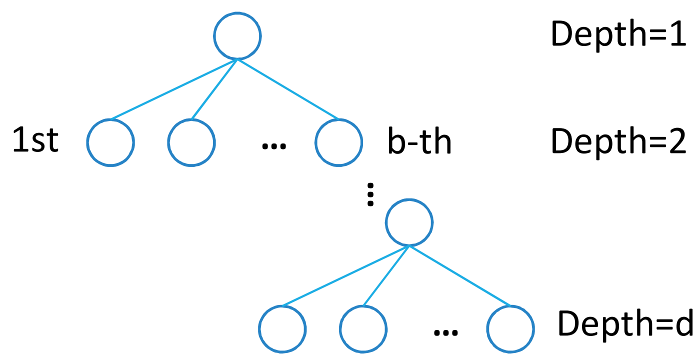
f : evaluation function

g : cost function

h : heuristic function

Uninformed search and informed search

- Search performance can be measured by the following ways:
  - Completeness: Is the algorithm guaranteed to find a solution when there is one?
  - Optimality: Does the strategy find the optimal solution?
  - Time complexity: How long does it take to find a solution?
  - Space complexity: How much memory is need to perform the search?



$b : branch\ \ factor$

$d : depth$

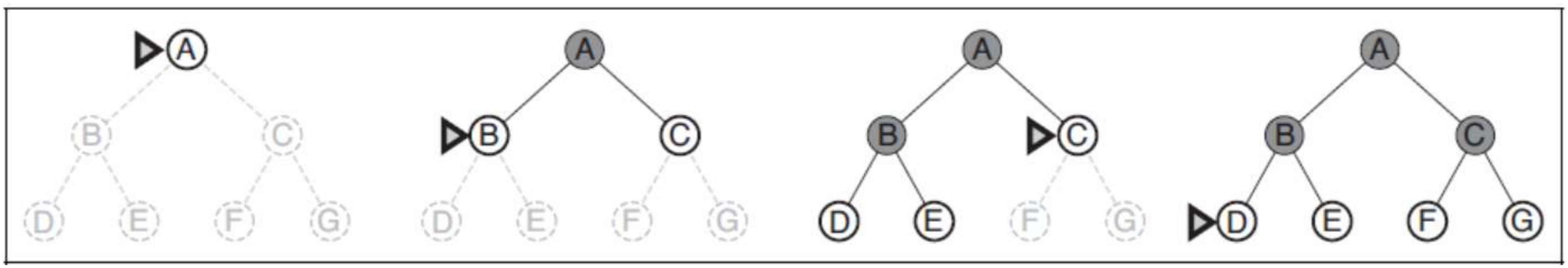
Breadth-first search

- Breadth-first search (BFS): Expand all nodes at the same level and then expand the next level.

$$Time: O(b^d)$$

$$Space: O(b^d)$$

Let's review stack and queue first!



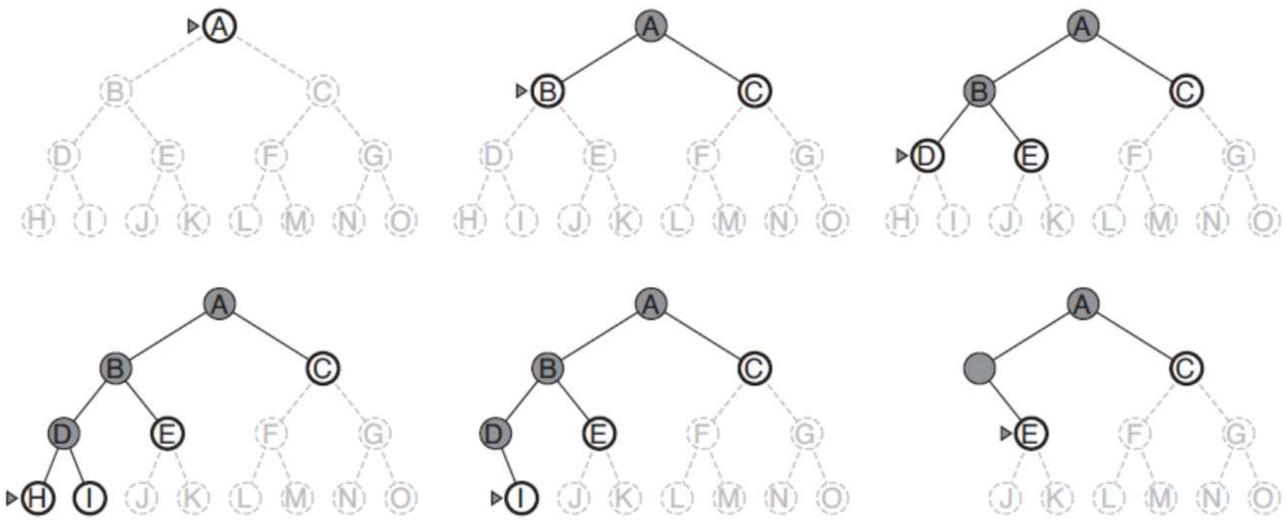
Stack or queue?

Breadth-first search

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
   node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
   if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
   frontier ← a FIFO queue with node as the only element
   explored ← an empty set
   loop do
       if EMPTY?(frontier) then return failure
       node ← POP(frontier)   /* chooses the shallowest node in frontier */
       add node.STATE to explored
       for each action in problem.ACTIONS(node.STATE) do
           child ← CHILD-NODE(problem, node, action)
           if child.STATE is not in explored or frontier then
               if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
               frontier ← INSERT(child, frontier)
```

Depth-first search

- Depth-first search (DFS): Expand the deepest node in the current frontier.



Stack or queue?

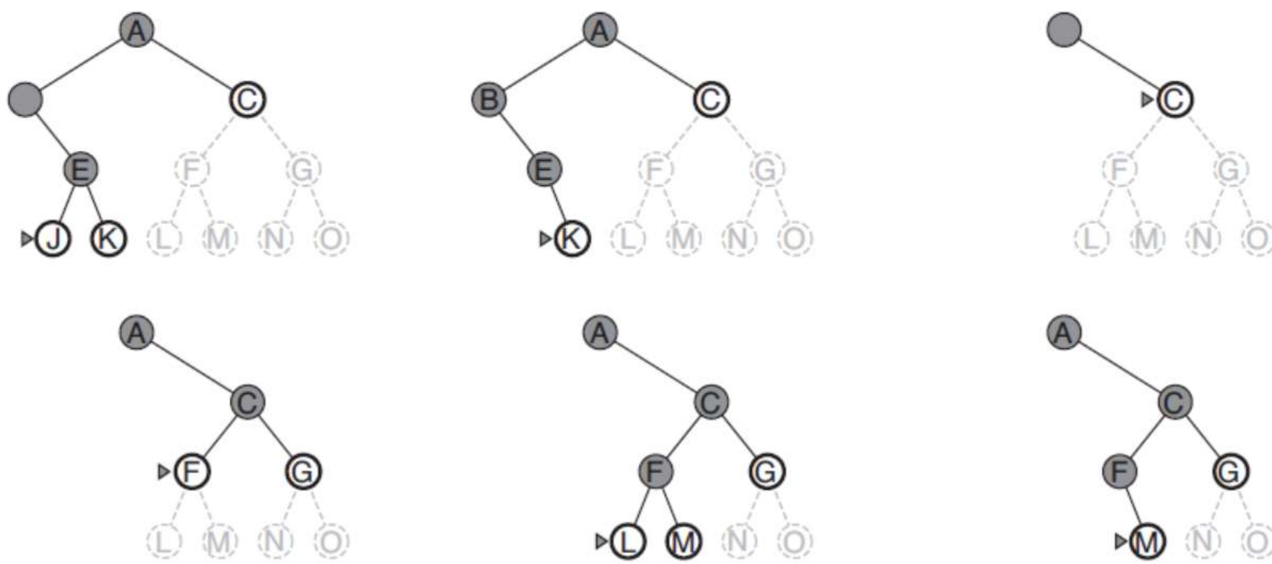
Depth-first search

- Depth-first search (DFS): Expand the deepest node in the current frontier.

$$Time : O(b^m)$$

$$Space : \cancel{O(b^m)}$$ O(bm)

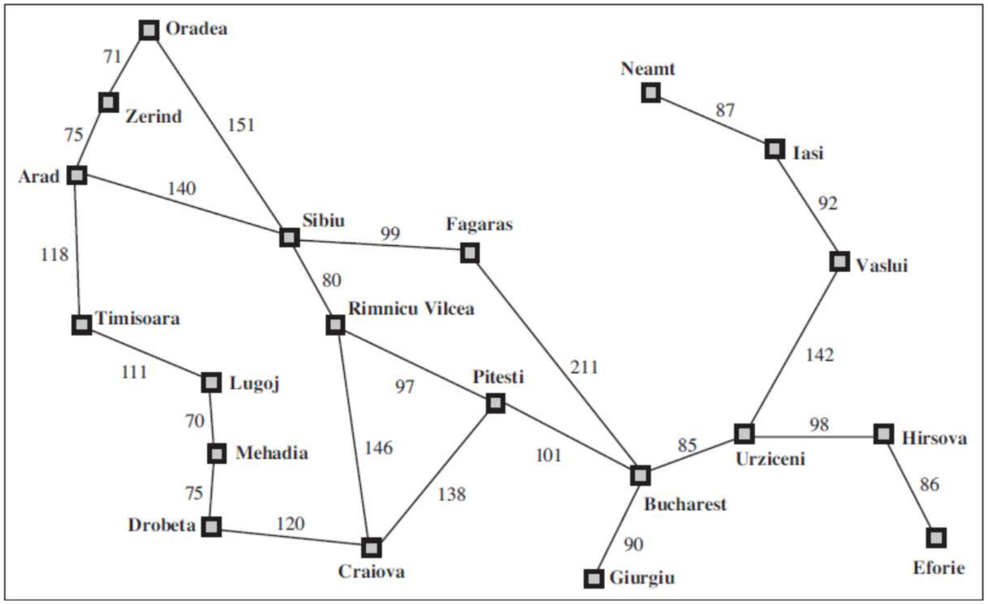
$$m : Maximal\ depth$$



Stack or queue?

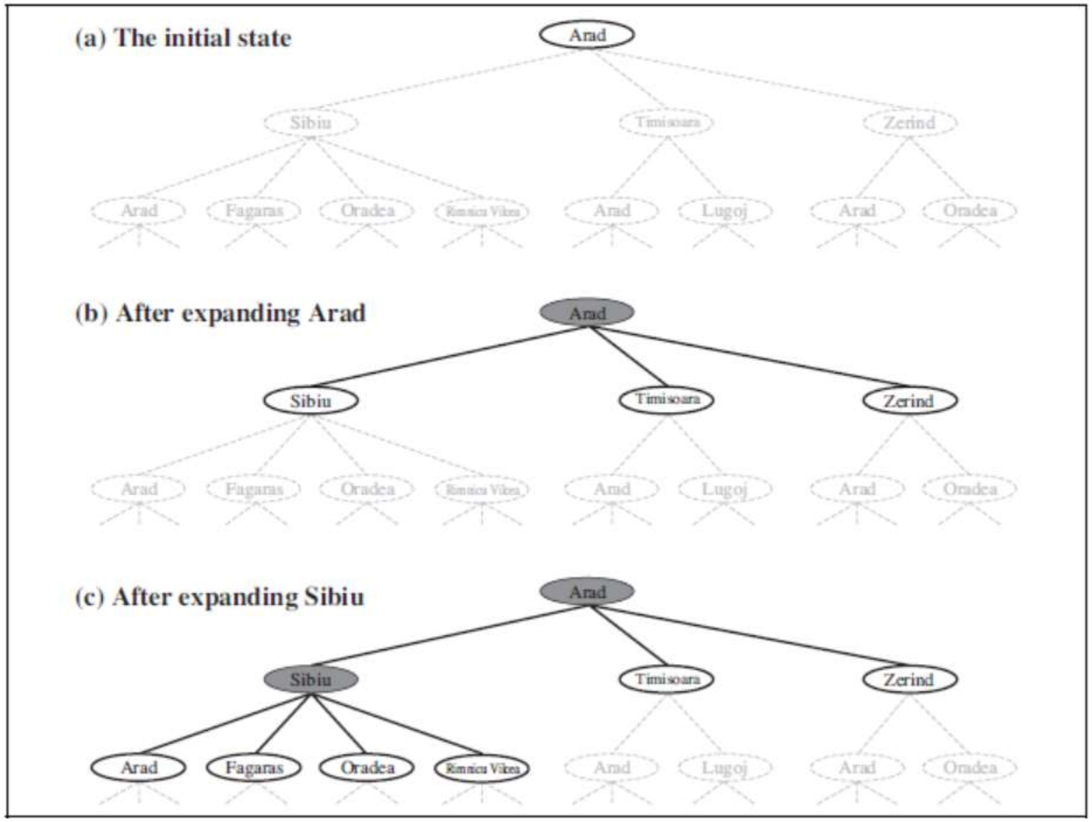
Depth-first search

- BFS and DFS don’t utilize the path cost for searching.
- In Romania-distance, there is path cost information, which is useful to search for solutions.



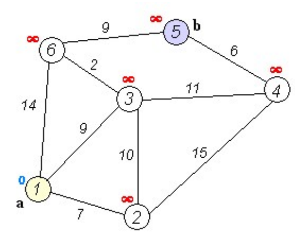
Depth-first search

- BFS and DFS don't utilize the path cost for searching for solutions.



Uniform-cost search

- Uniform-cost search: Expand the node with the optimal (lowest) path cost.
- Uniform-cost search is similar to BFS but it adopts **priority queue** instead of queue. When the cost of all path is the same, BFS is uniform-cost search.
- In data structure, algorithm, and computer network, Uniform-cost search is also called ***Dijkstra algorithm***.
- It is applied to find the shortest path for path planning and networking.

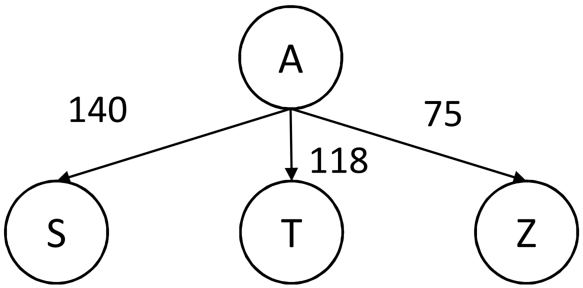


https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

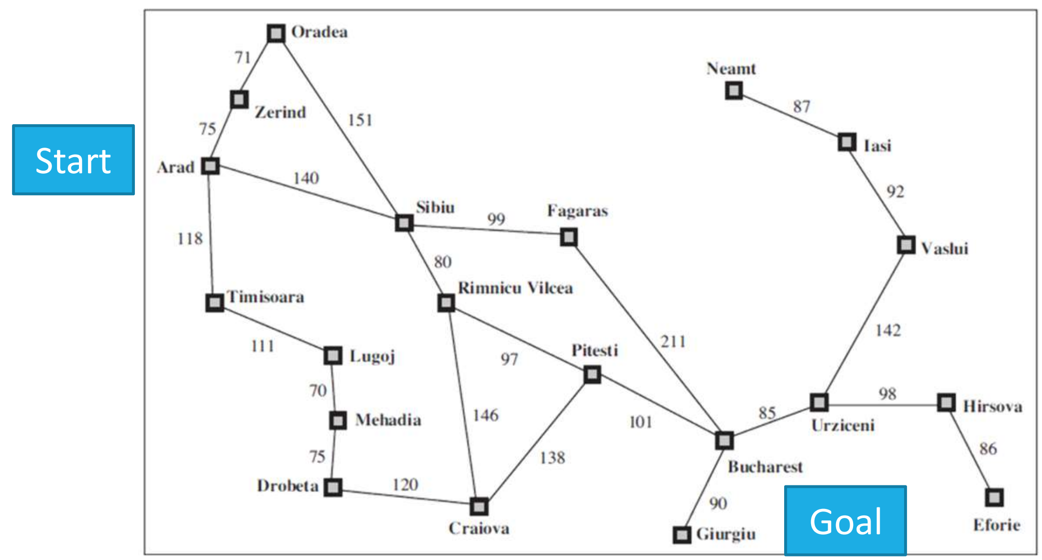
Uniform-cost search

```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  frontier ← a priority queue ordered by PATH-COST, with node as the only element
  explored ← an empty set
  loop do
      if EMPTY?(frontier) then return failure
      node ← POP(frontier)   /* chooses the lowest-cost node in frontier */
      if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
      add node.STATE to explored
      for each action in problem.ACTIONS(node.STATE) do
         child ← CHILD-NODE(problem, node, action)
         if child.STATE is not in explored or frontier then
            frontier ← INSERT(child, frontier)
         else if child.STATE is in frontier with higher PATH-COST then
            replace that frontier node with child
```

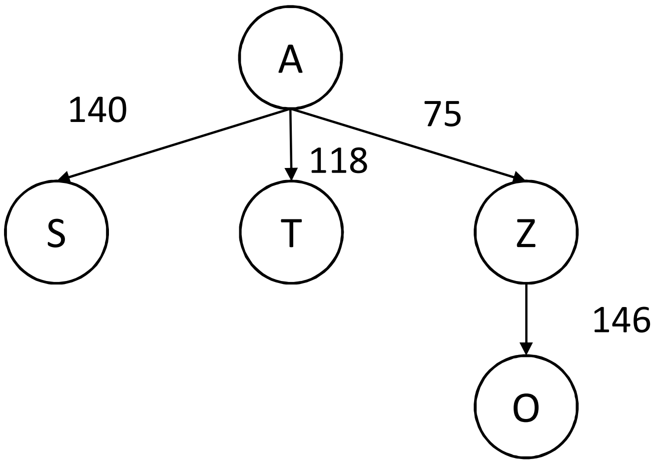
Uniform-cost search



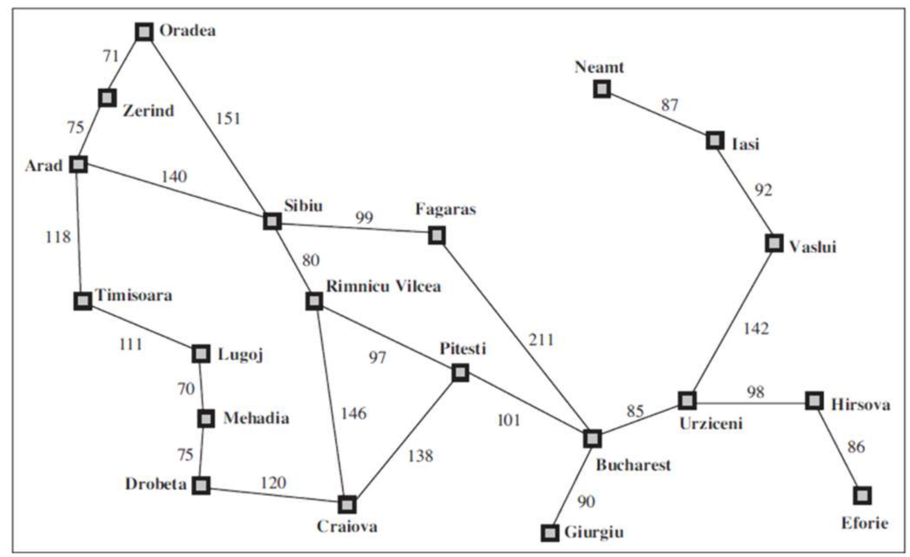
Z 75	T 118	S 140	



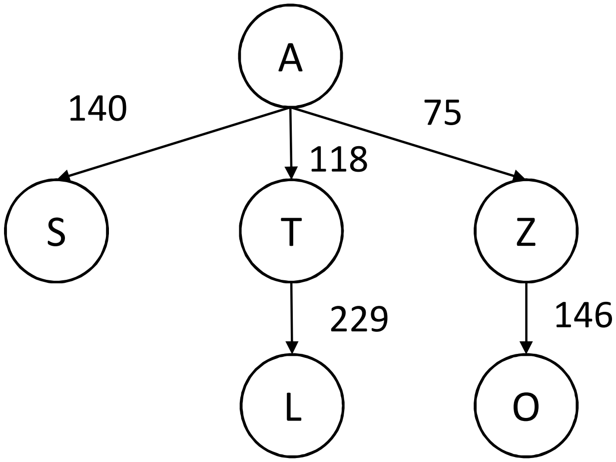
Uniform-cost search



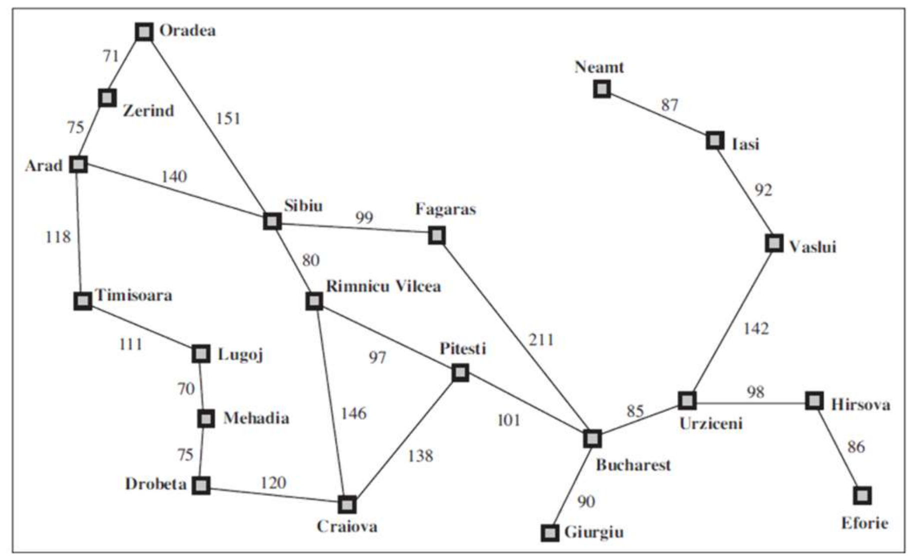
T **118**	S 140	O 146	



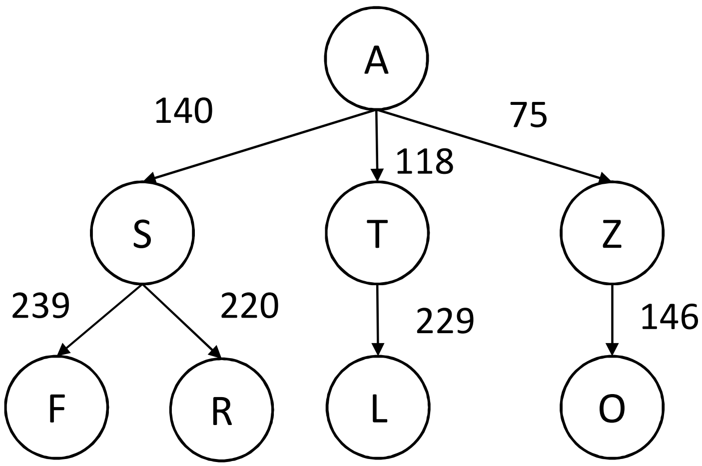
Uniform-cost search



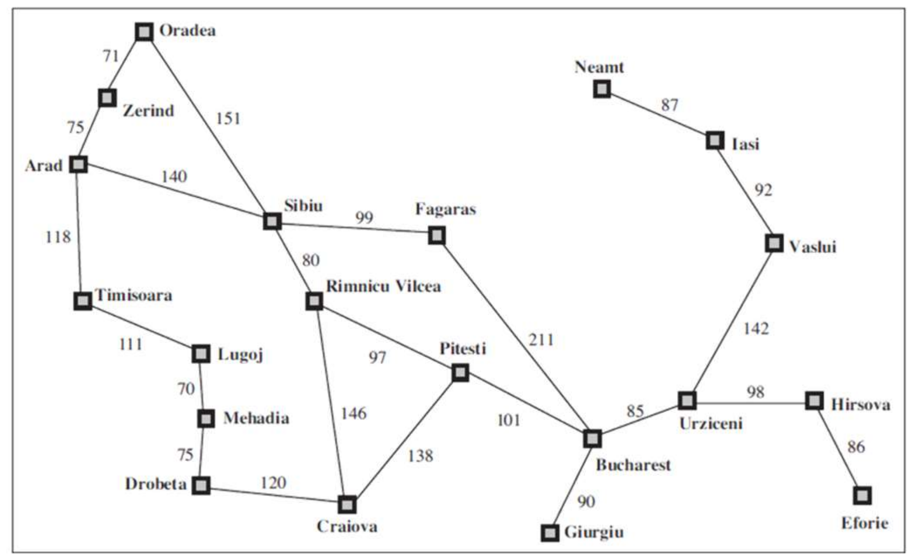
S 140	O 146	L 229	



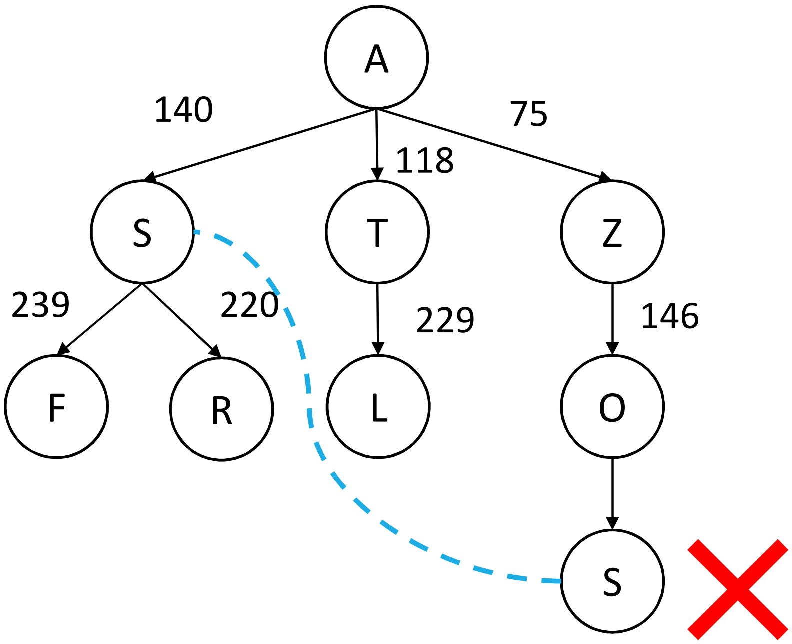
Uniform-cost search



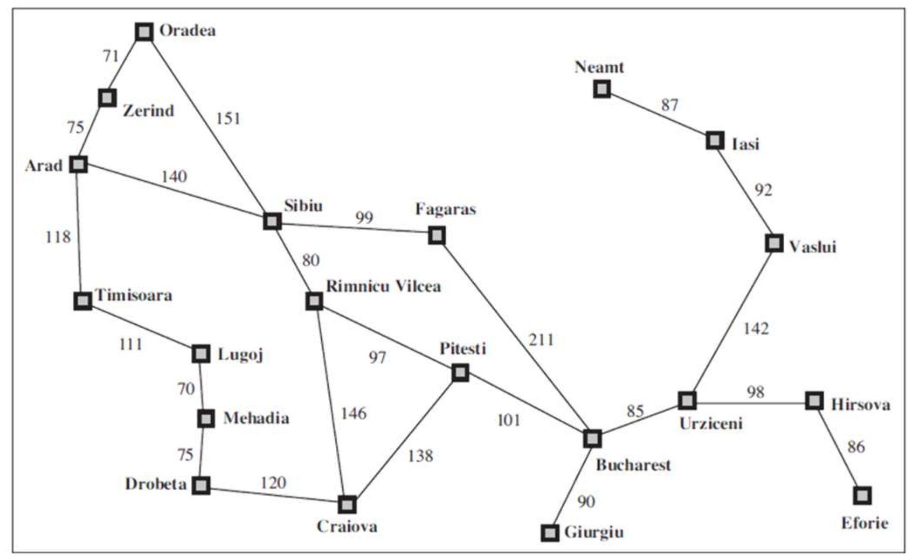
O 146	R 220	L 229	F 239	



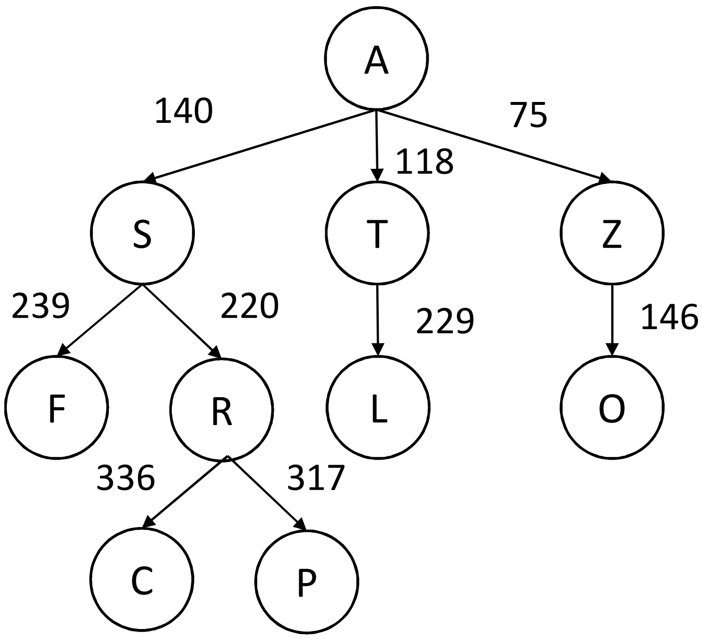
Uniform-cost search



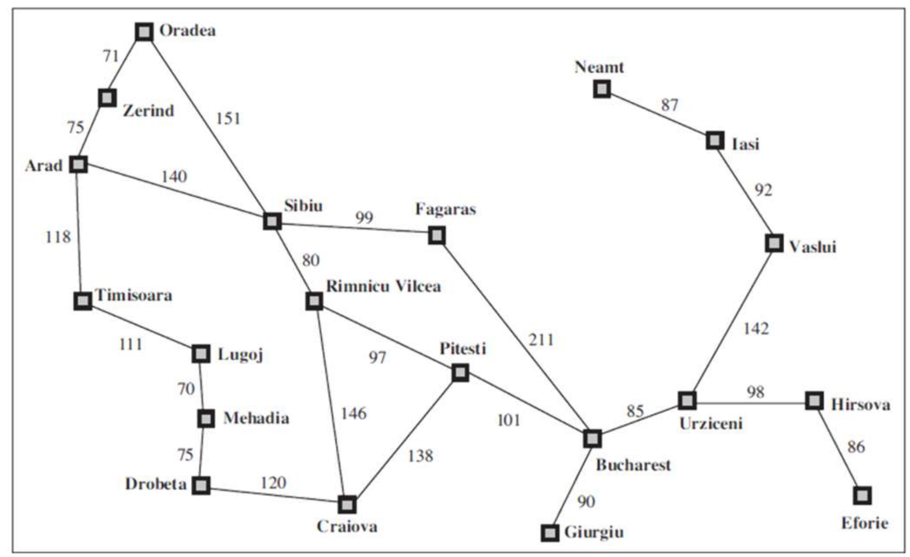
~~O 140~~	**R 220**	L 229	F 239	



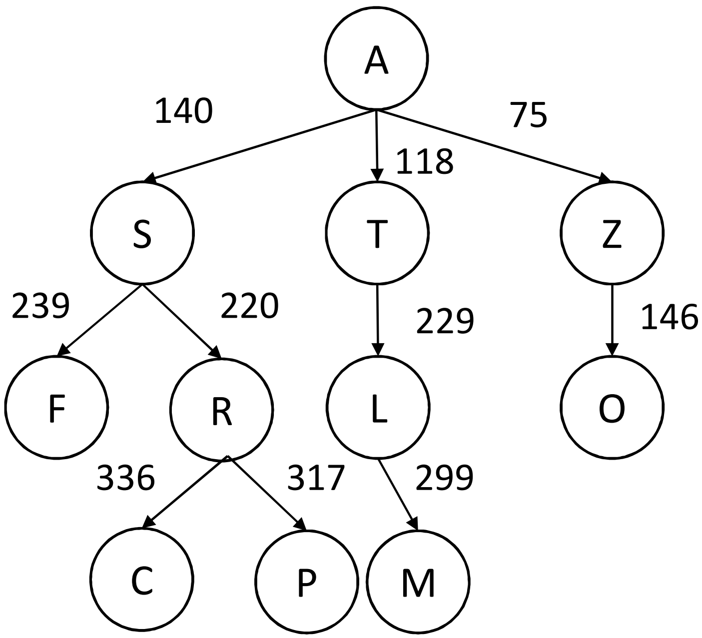
Uniform-cost search



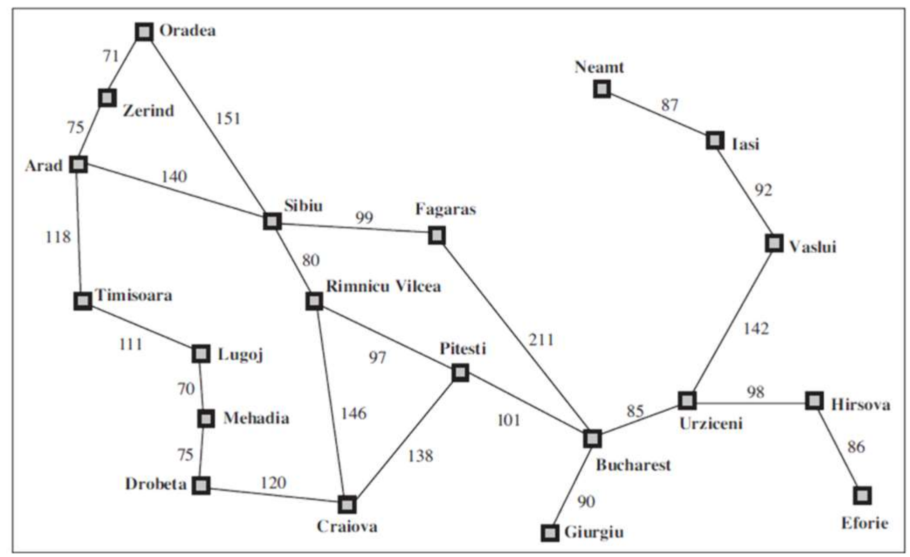
L 229	F 239	P 317	C 336	



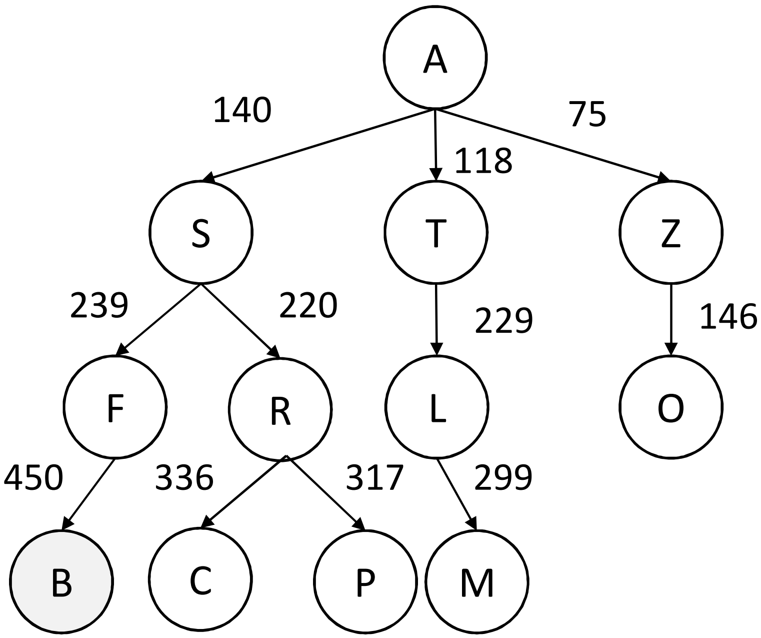
Uniform-cost search



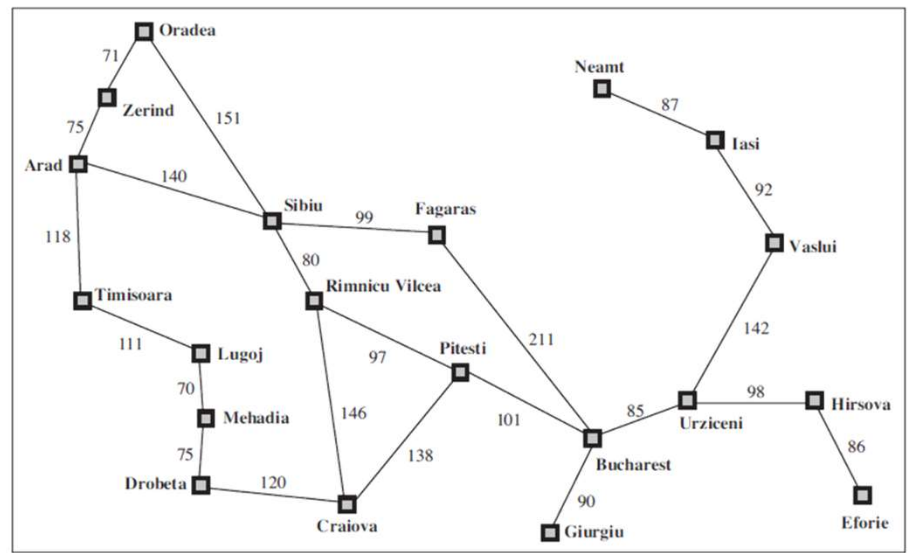
F 239	M 299	P 317	C 336	



Uniform-cost search

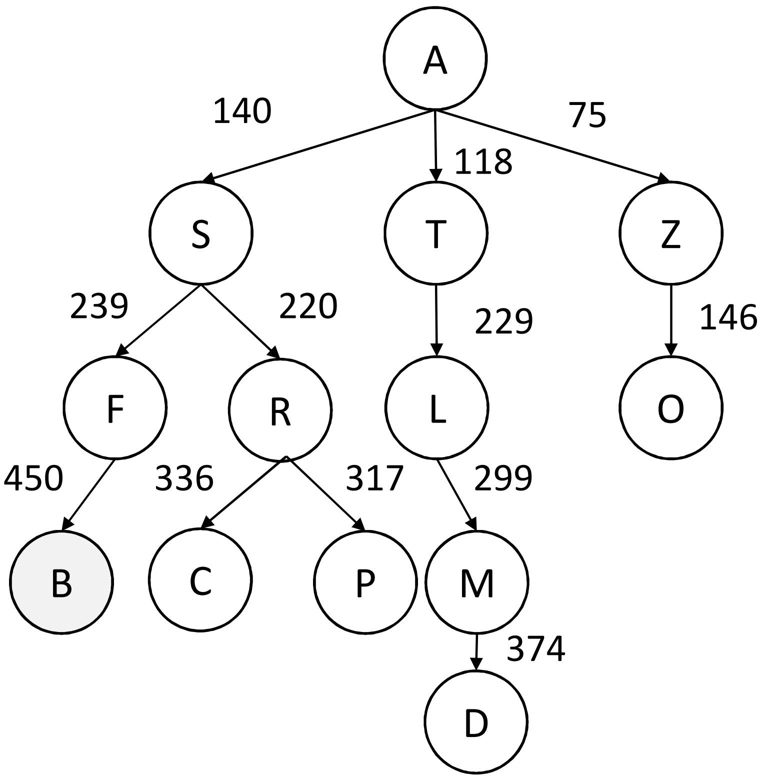


M 299	P 317	C 336	B 450	

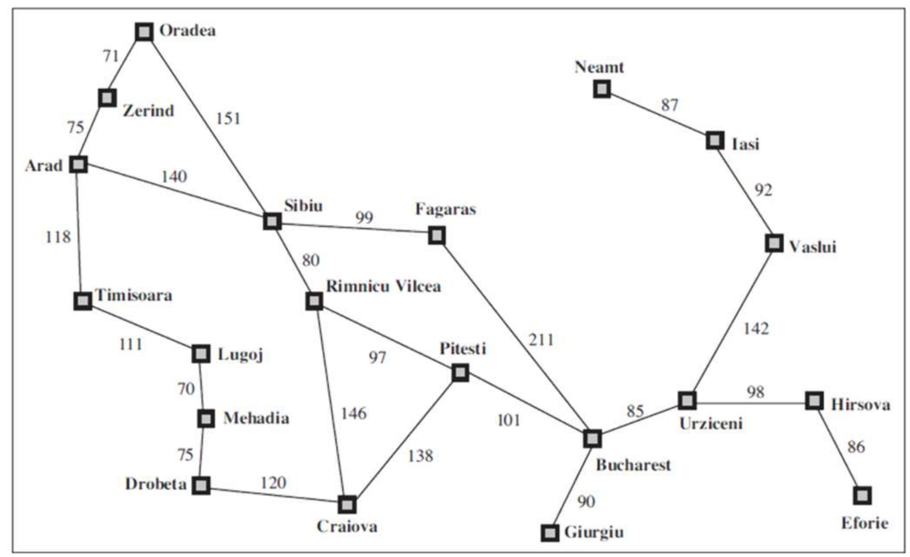


UCS will keep searching since this path could be not the lowest cost path!

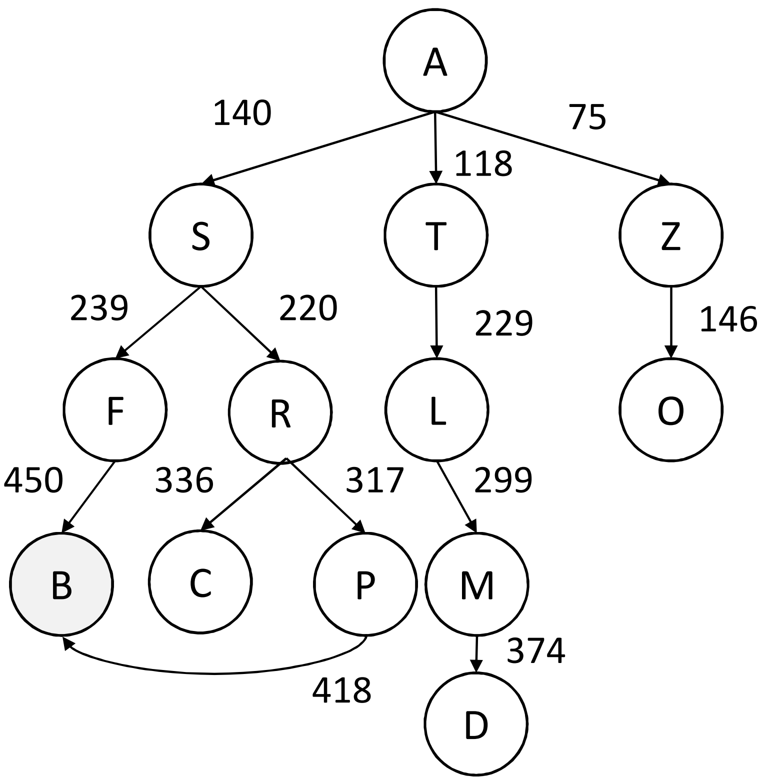
Uniform-cost search



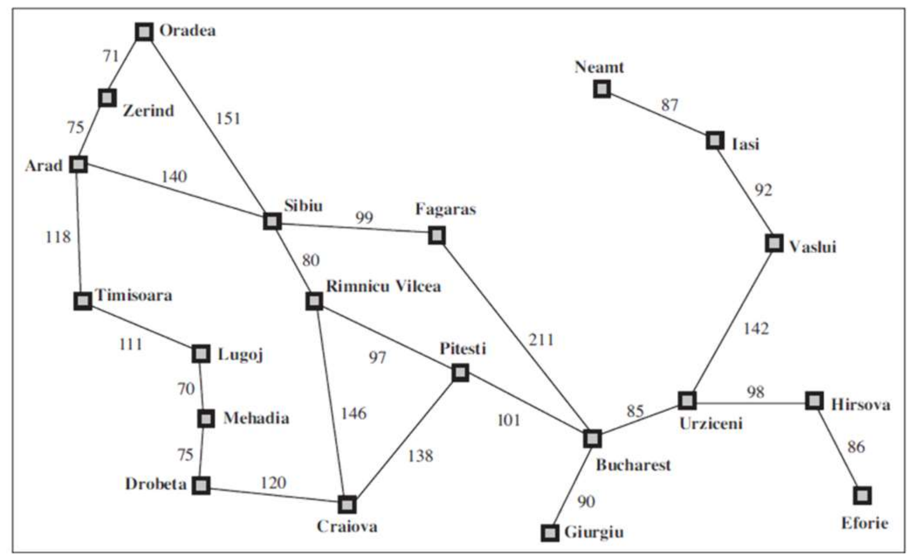
P **317**	C 336	D 374	B 450	



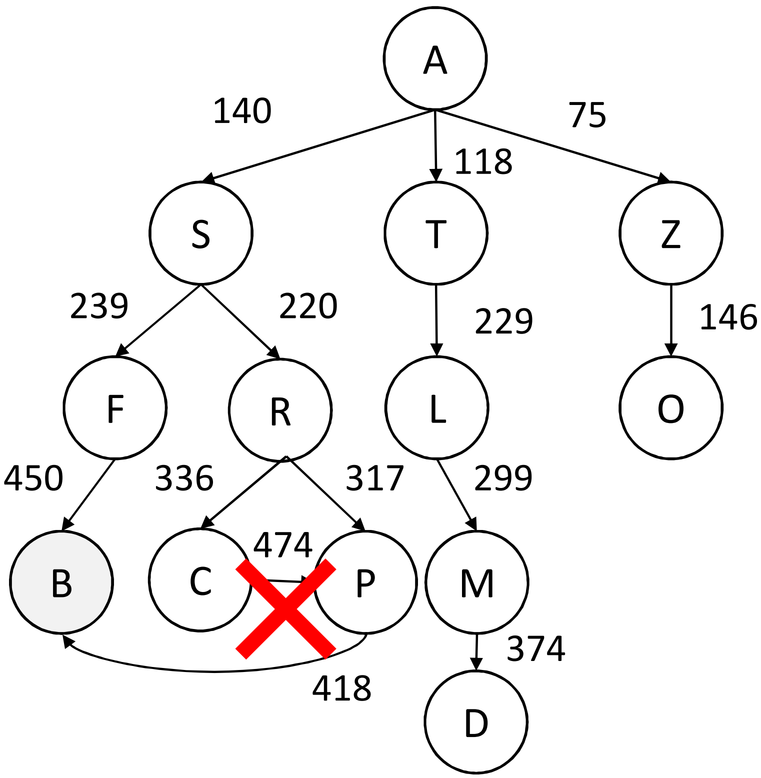
Uniform-cost search



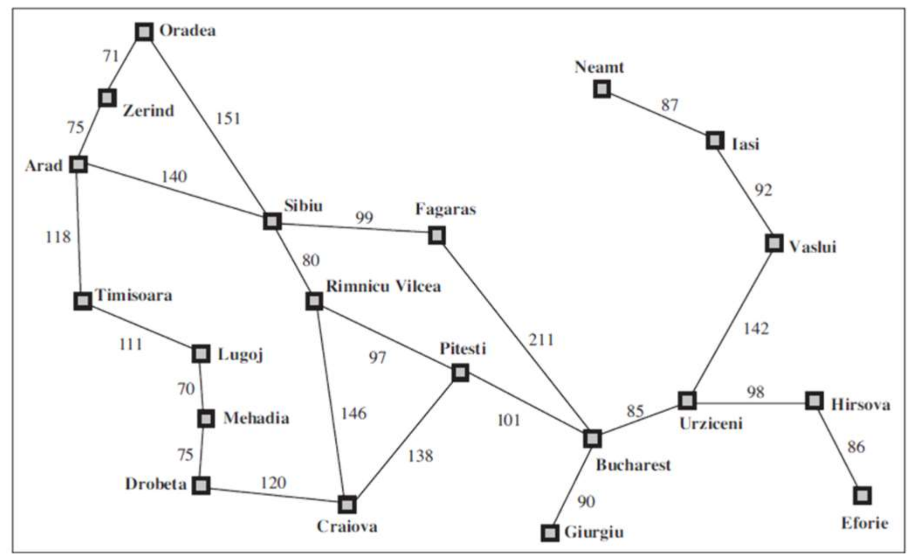
C **336**	D 374	B 418	



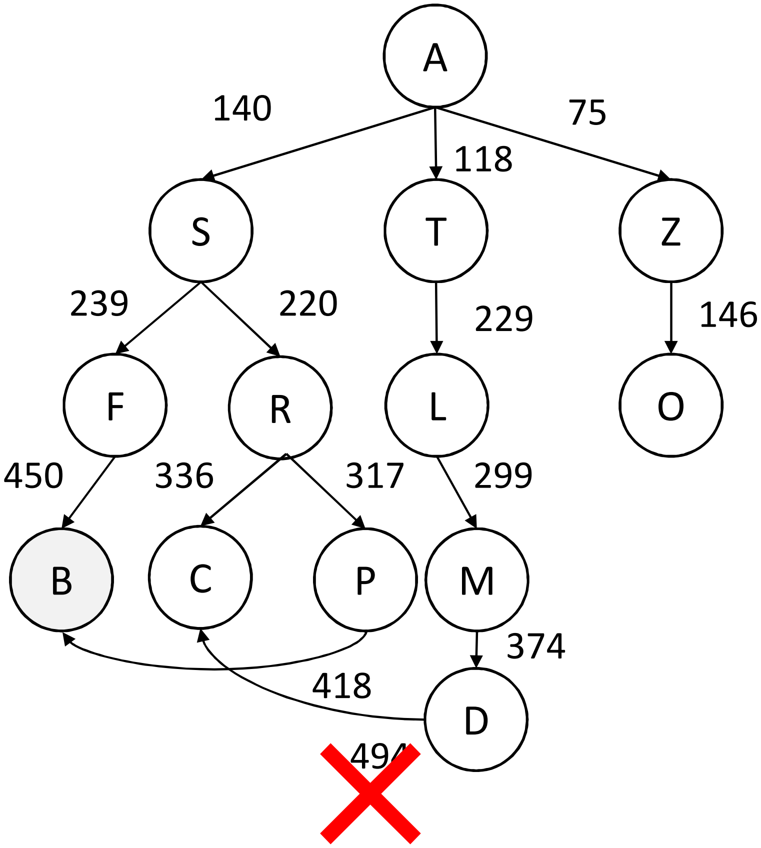
Uniform-cost search

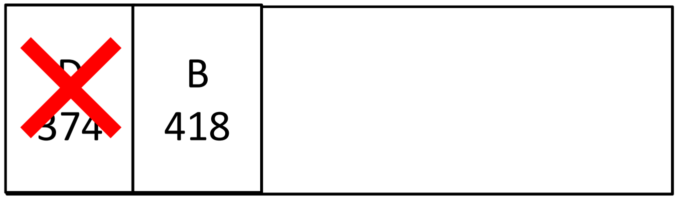


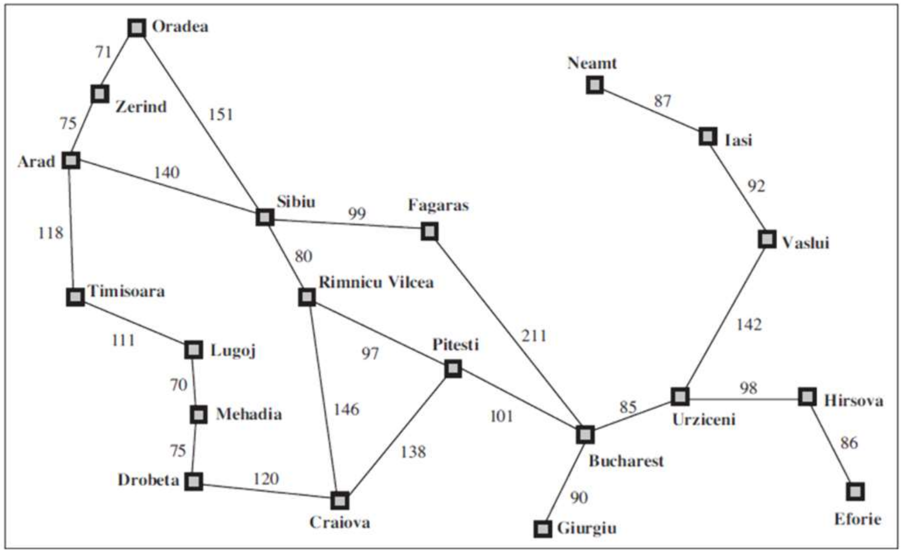
~~C 336~~	**D 374**	B 418	



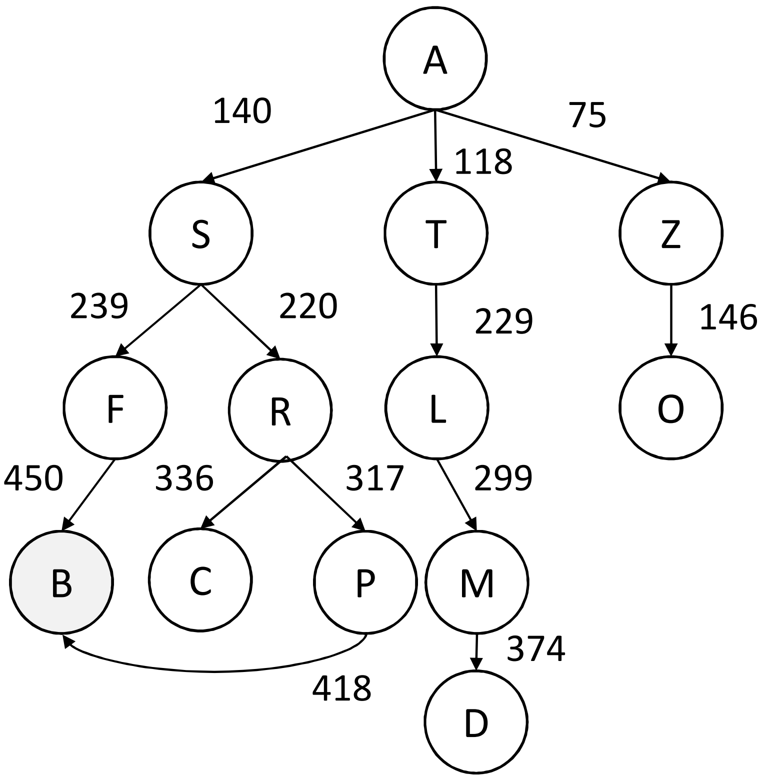
Uniform-cost search



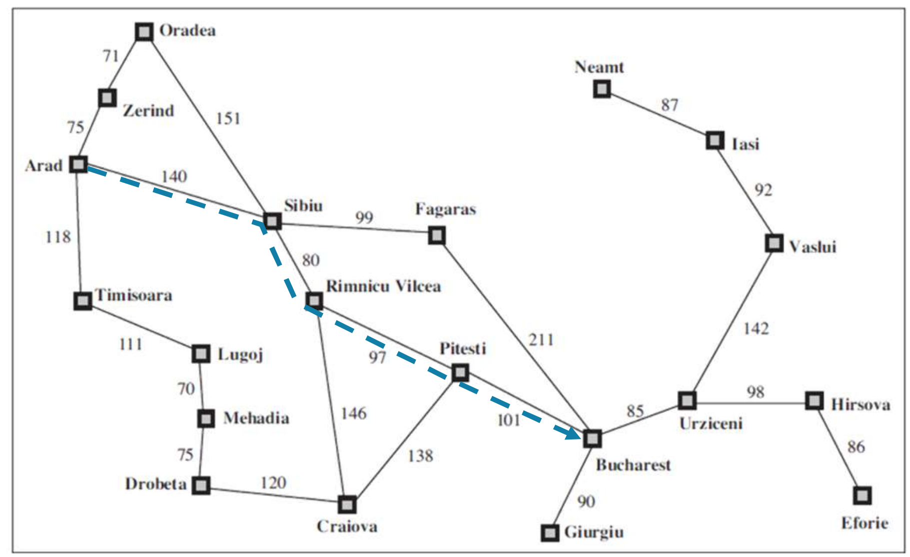




Uniform-cost search

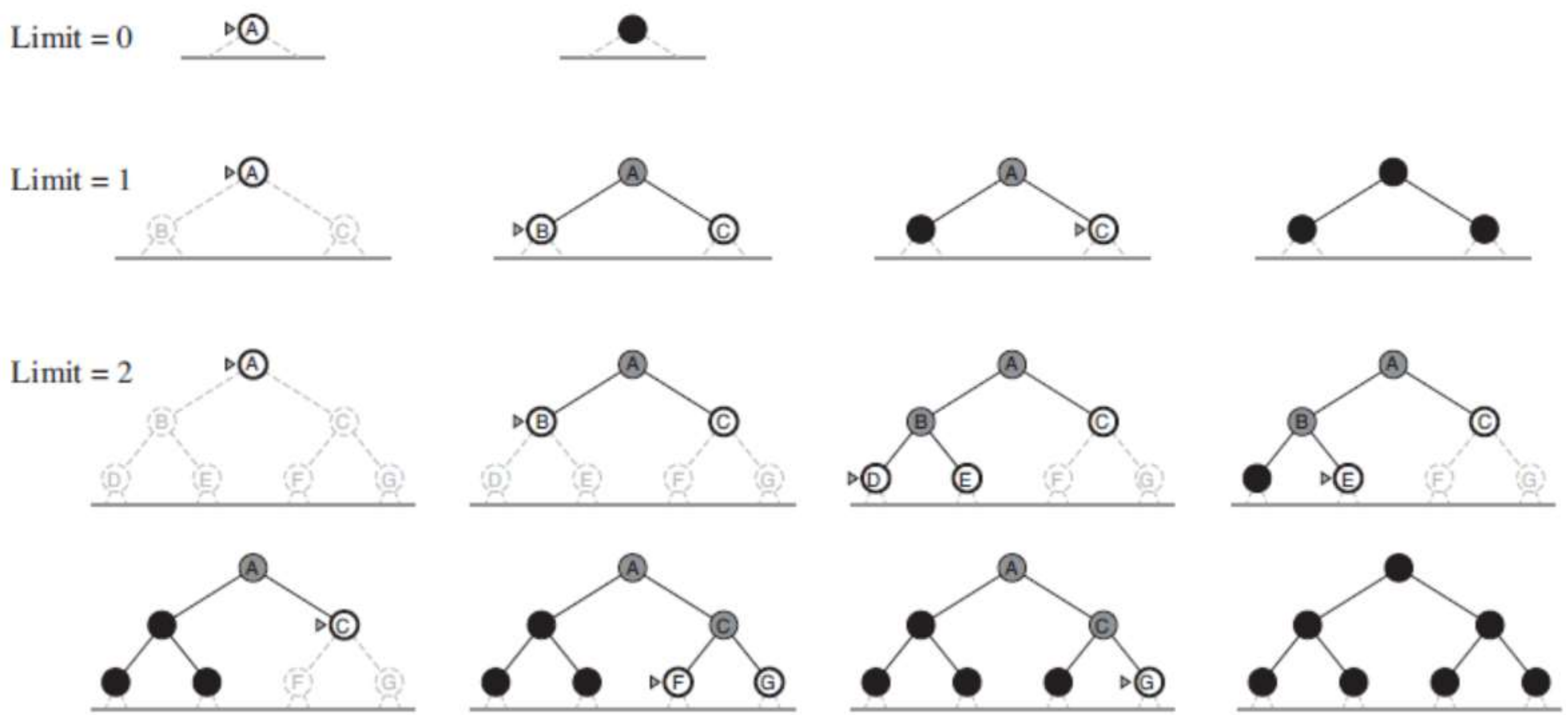


B 418	



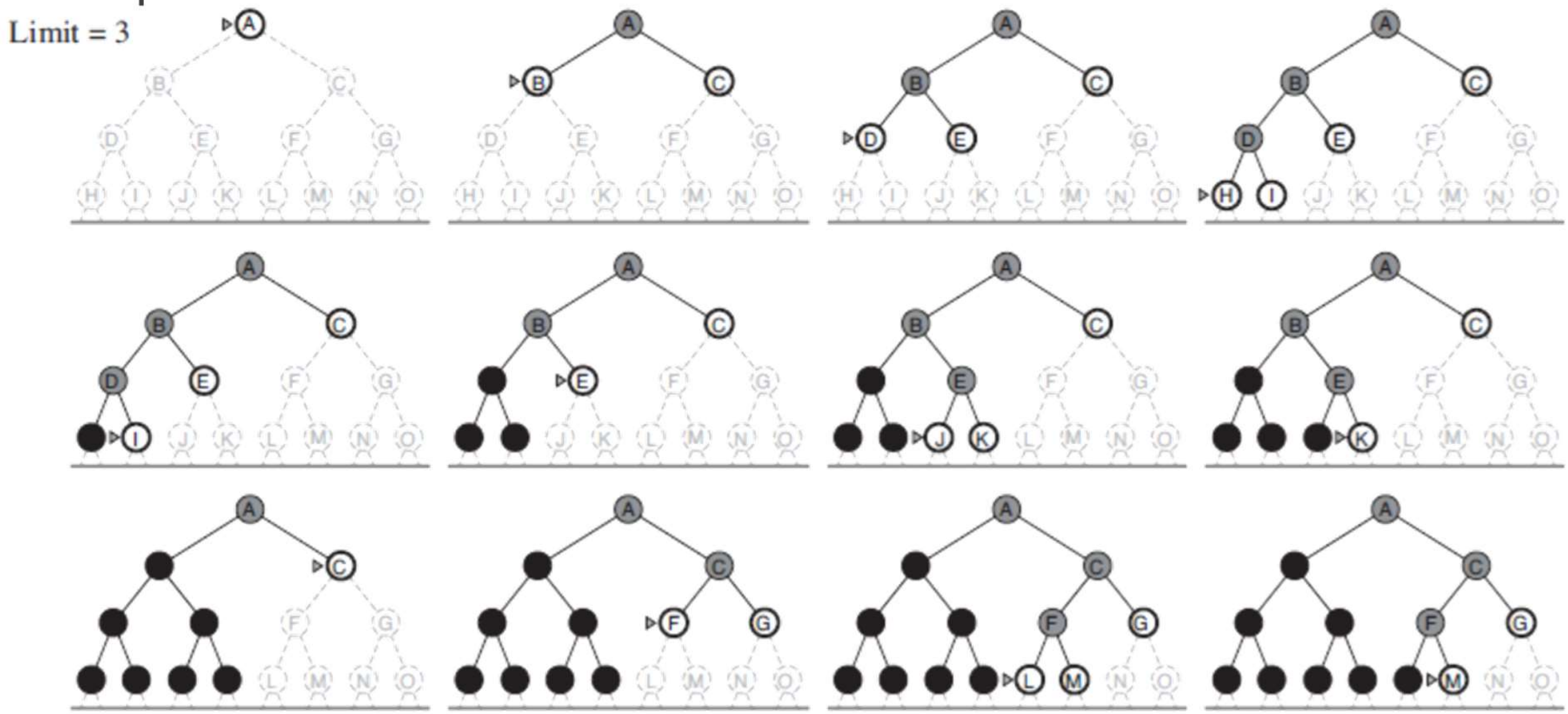
Depth-limited search

- Depth-limited search: Expand the deepest node with limited depth in the current frontier.



Depth-limited search

- Depth-limited search: Expand the deepest node with limited depth in the current frontier.



Depth-limited search

```
function DEPTH-LIMITED-SEARCH(problem, limit) returns a solution, or failure/cutoff
  return RECURSIVE-DLS(MAKE-NODE(problem.INITIAL-STATE), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns a solution, or failure/cutoff
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  else if limit = 0 then return cutoff
  else
     cutoff_occurred? ← false
     for each action in problem.ACTIONS(node.STATE) do
        child ← CHILD-NODE(problem, node, action)
        result ← RECURSIVE-DLS(child, problem, limit − 1)
        if result = cutoff then cutoff_occurred? ← true
        else if result ≠ failure then return result
     if cutoff_occurred? then return cutoff else return failure
```

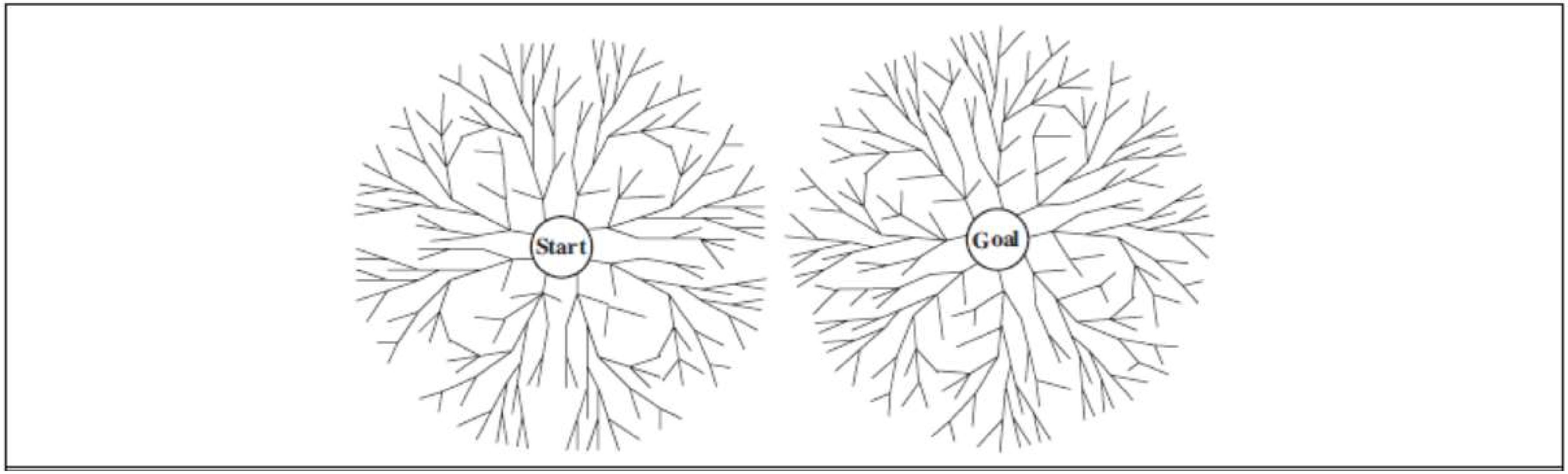
Iterative deepening depth-first search

- Iterative deepening depth-first search: Expand the deepest node with limited depth in the current frontier iteratively.

function ITERATIVE-DEEPENING-SEARCH(*problem*) **returns** a solution, or failure
  for $depth = 0$ **to** ∞ **do**
    $result \leftarrow$ DEPTH-LIMITED-SEARCH(*problem*, *depth*)
    if $result \neq$ cutoff **then return** *result*

Bi-directional search

- Bi-directional search: Expand the node from start and goal simultaneously.



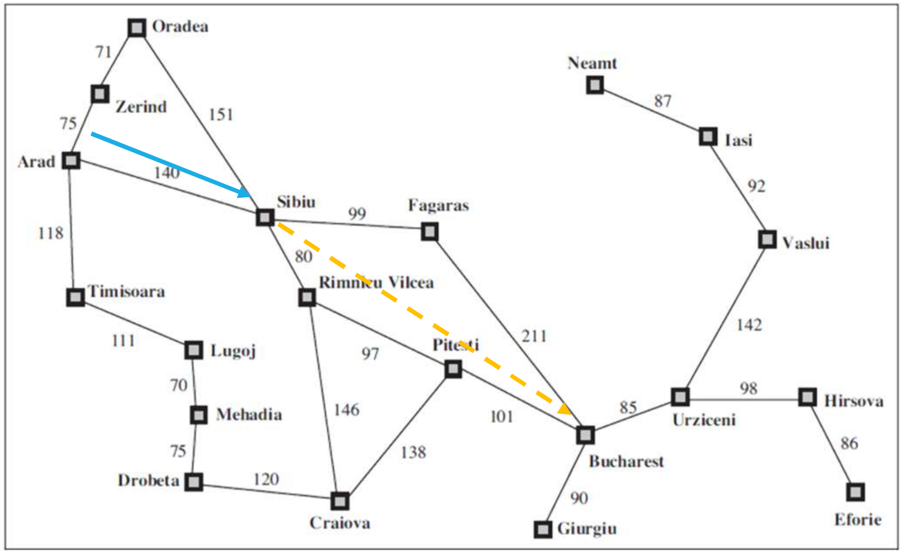
Uninformed search

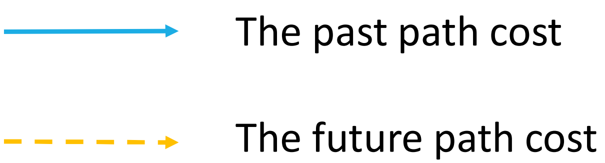
Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes^a	Yes^{a,b}	No	No	Yes^a	Yes^{a,d}
Time	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$
Optimal?	Yes^c	Yes	No	No	Yes^c	Yes^{c,d}

Figure 3.21 Evaluation of tree-search strategies. b is the branching factor; d is the depth of the shallowest solution; m is the maximum depth of the search tree; l is the depth limit. Superscript caveats are as follows: ^{a} complete if b is finite; ^{b} complete if step costs $\geq \epsilon$ for positive ϵ; ^{c} optimal if step costs are all identical; ^{d} if both directions use breadth-first search.

Conclusions

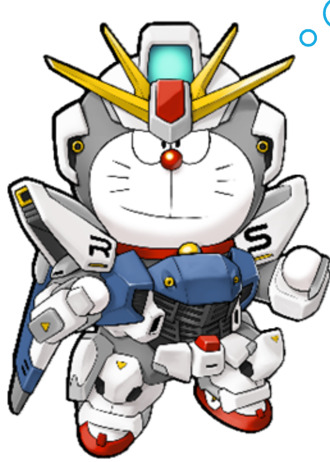
- Uninformed search only utilizes the path cost. If we have more information, the search will be more efficient.





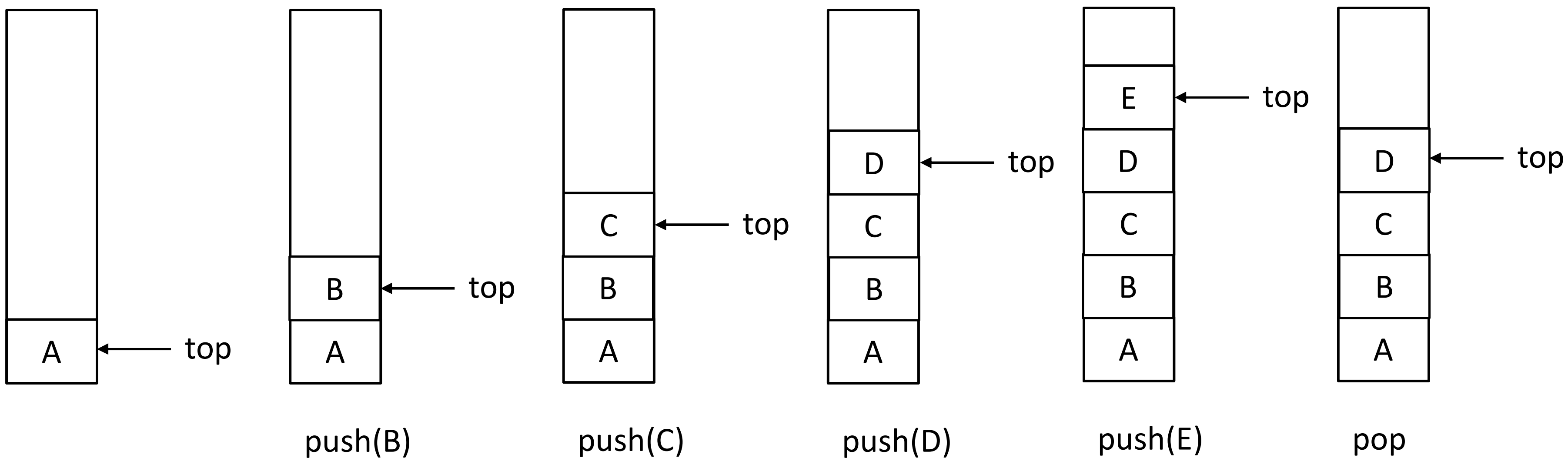
How could we know the future path cost, which we didn't expand?
ANS: Heuristic function

Q&A



Appendix– Stack

- Stack
- Last-In-First-Out (LIFO)



Sartaj Sahni, Ellis Horowitz, and Susan Anderson-Freed, "Fundamentals of Data Structure in C," Silicon Press, 2008.

Appendix– Stack

```
template<class T>
class Stack
{
   public:
    Stack(int stackCapacity = 10);
    ~Stack() {delete [] stack;}
    bool IsEmpty() const;
    T& Top() const;
    void Push(const T& item);
    void Pop();
   private:
    T *stack;      // array for stack elements
    int top;       // position of top element
    int capacity; // capacity of stack array
};
```

Appendix– Stack

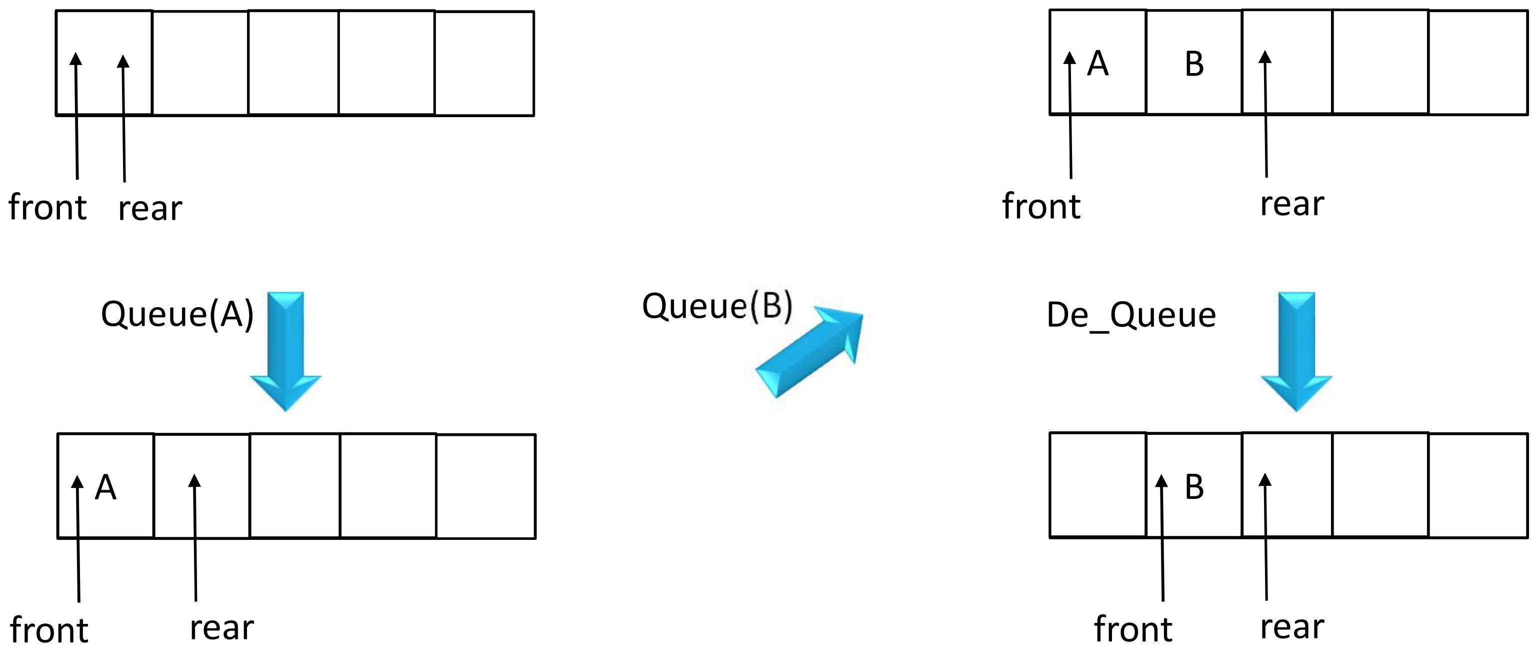
```
template<class T>
Stack<T>::Stack(int stackCapacity)
               :capacity(stackCapacity)
{
   if (capacity < 1)
     throw "Stack capacity must be > 0";
   stack = new T[capacity];
   top = -1;
}

template<class T>
inline bool Stack<T>::IsEmpty() const
   {return top == -1}
```

```
template<class T>
inline T& Stack<T>::Top() const
{
   if (IsEmpty())
       throw "Stack is empty";
   return stack[top];
}
```

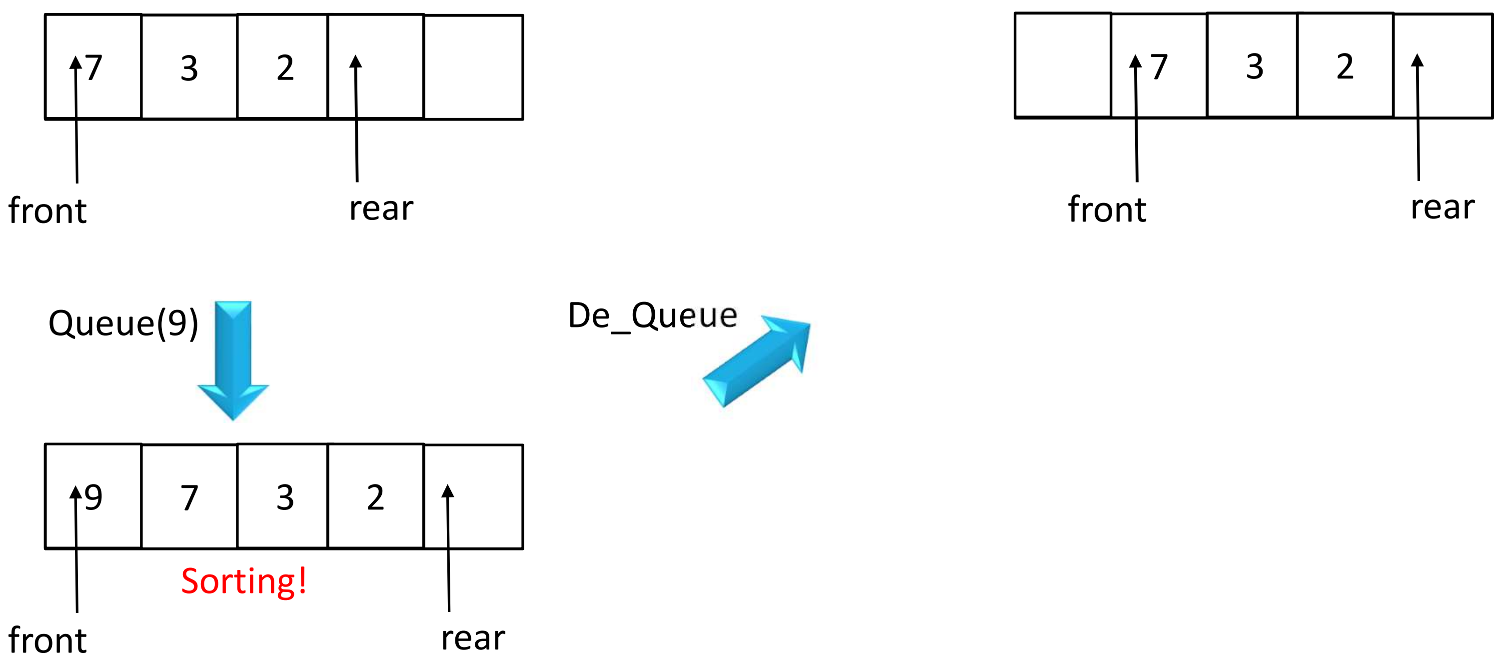
Appendix– Queue

- Queue
- First-In-First-Out (FIFO)



Appendix – Priority Queue

- Priority Queue with sorting
- The first out could be max or min.



Appendix – Priority Queue

- Priority Queue with sorting
- The first out could be max or min.

Operation	Binary[5]	Leftist	Binomial[5]	Fibonacci[5][6]	Pairing[7]	Brodal[8][a]	Rank-pairing[10]	Strict Fibonacci[11]	2-3 heap
find-min	$\Theta(1)$	$\Theta(1)$	$\Theta(\log n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	?
delete-min	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$O(\log n)$[b]	$O(\log n)$[b]	$O(\log n)$	$O(\log n)$[b]	$O(\log n)$	$O(\log n)$[b]
insert	$O(\log n)$	$\Theta(\log n)$	$\Theta(1)$[b]	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$O(\log n)$[b]
decrease-key	$O(\log n)$	$\Theta(n)$	$\Theta(\log n)$	$\Theta(1)$[b]	$o(\log n)$[b][c]	$\Theta(1)$	$\Theta(1)$[b]	$\Theta(1)$	$\Theta(1)$
merge	$\Theta(n)$	$\Theta(\log n)$	$O(\log n)$[d]	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	?

https://en.wikipedia.org/wiki/Priority_queue

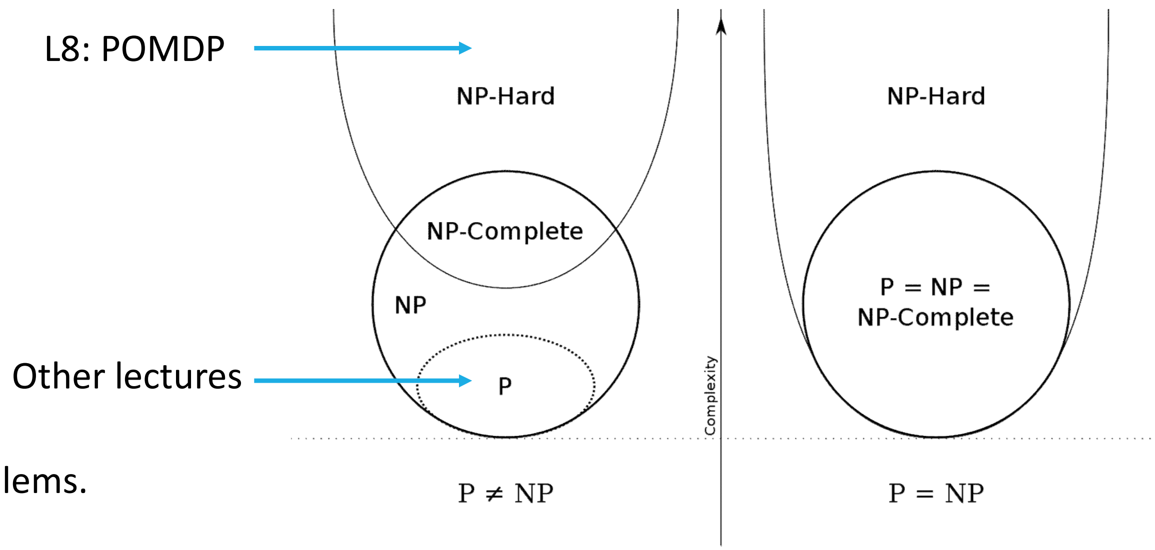
Appendix – Complexity

- In computer science, the computational complexity, or simply complexity of an algorithm is the amount of resources required for running it.
- O(n^2), O(n^3) or O(nlogn)
- Example, O(n^2) → O(nlogn)
- Discrete Fourier Transform → Fast Fourier Transform (FFT)

https://en.wikipedia.org/wiki/Computational_complexity
https://en.wikipedia.org/wiki/Fast_Fourier_transform

Appendix – NP-hard problems

- NP-hardness (non-deterministic polynomial-time hardness)
- NP-complete: Class of decision problems which contains the hardest problems in NP. Each NP-complete problem has to be in NP.



In MAI, we will try to solve P problems.
In L8, we will face POMDP, one of NP-hard problems.

https://en.wikipedia.org/wiki/NP-hardness

Appendix – NP-hard problems

- NP-complete problems: the list below contains some well-known problems that are NP-complete when expressed as decision problems.
  - Boolean satisfiability problem (SAT)
  - Knapsack problem
  - Travelling salesman problem (decision version)
  - Vertex cover problem
  - Independent set problem
  - Graph coloring problem

Appendix – NP-hard problems

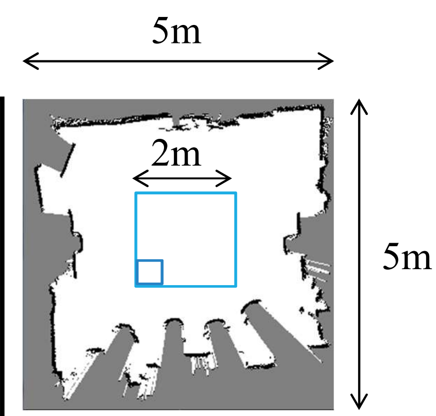
$\boxed{\text{C : Put 6 Kinect sensors to Max coverage ?}}$

[Maximum coverage]

Given a subgoal ground set $S = \{1,2,..,N\}$,

Find K subgoals s.t. sensors' coverage is maximal

$\max_{|S_g| \le K} f_C(S_g)$, where f_C is coverage function, $S_g \subseteq S$.



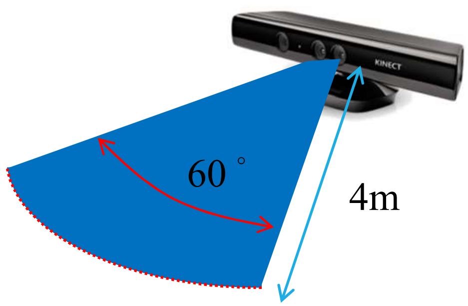
□ 10 cm × 10 cm × 45 °
N=20 × 20 × 8 = 3200

To find the optimal S_g, we need to try N^K sets.

Assume $K = 6$. A query of $f_C(S_g)$ takes 10^{-3} sec.

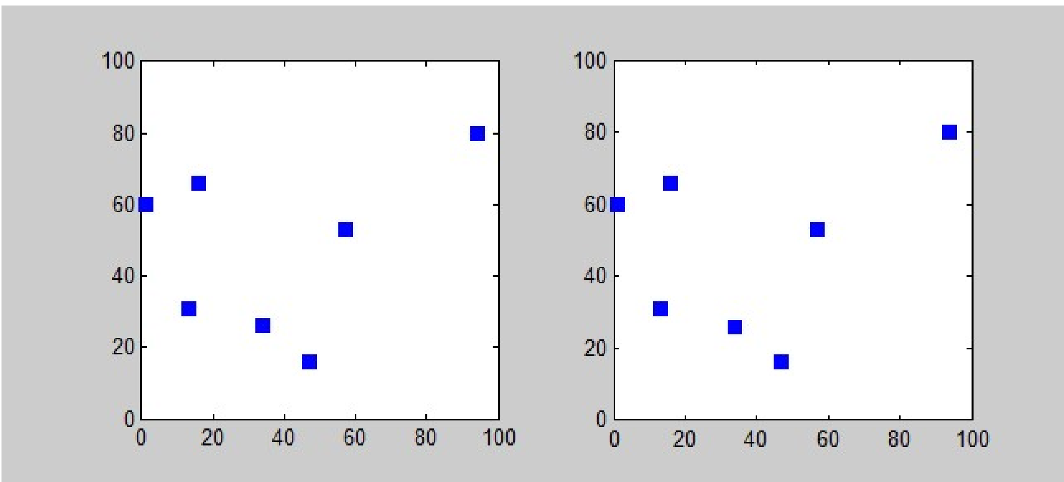
It takes $3200^6 \times 10^{-3}$ sec = 34,048,129,000 years

It's NP-Complete!



Appendix – NP-hard problems

- Travelling salesman problem (TSP) : Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city and returns to the origin city?



https://en.wikipedia.org/wiki/Travelling_salesman_problem