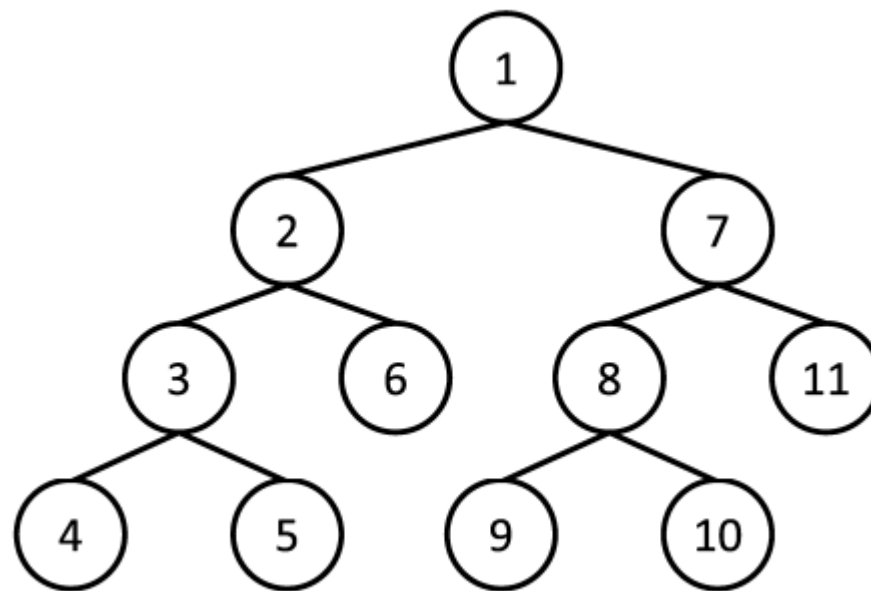


# 目录

- ▶ 深度优先搜索和栈
- ▶ 广度优先搜索和队列
- ▶ Uniform Cost Search和优先队列
- ▶ A star

# 深度优先搜索

- ▶ 深度优先搜索算法从某个状态开始，不断地转移状态直到无法转移，然后退回前一步的状态，继续转移到其他状态，如此不断重复。



状态转移的顺序

**function** Depth-FIRST-SEARCH(*problem*) **returns** a solution, or failure

*node*  $\leftarrow$  a node with STATE = *problem*.INITIAL-STATE, PATH-COST = 0

→ **if** *problem*.GOAL-TEST(*node*.STATE) **then return** SOLUTION(*node*)

→ *frontier*  $\leftarrow$  a LIFO stack with *node* as the only element

*explored*  $\leftarrow$  an empty set

**loop do**

**if** EMPTY?(*frontier*) **then return** failure

→ *node*  $\leftarrow$  POP(*frontier*) /\* choose the deepest node in frontier\*/

add *node*.STATE to *explored*

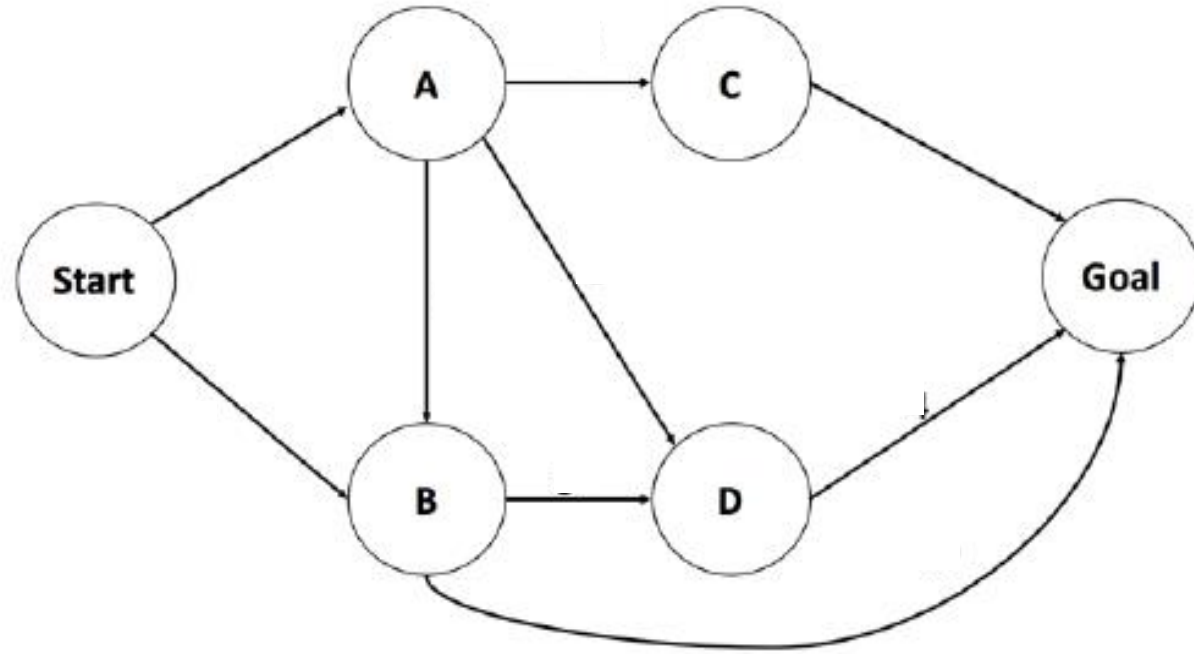
**for each** *action* **in** *problem*.ACTIONS(*node*.STATE) **do**

→ *child*  $\leftarrow$  CHILD-NODE(*problem*, *node*, *action*)

**if** *child*.STATE is not in *explored* or *frontier* **then**

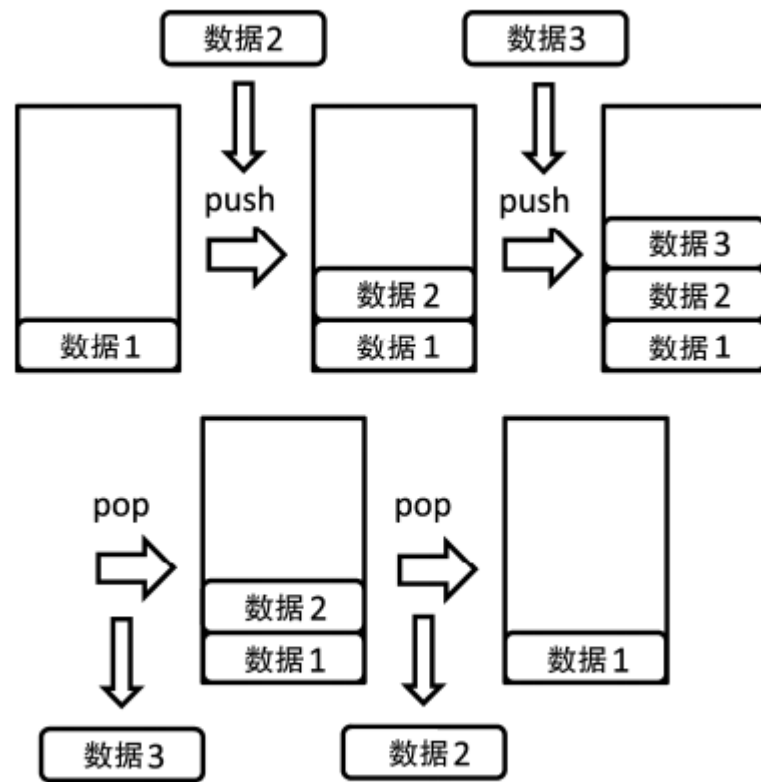
→ **if** *problem*.GOAL-TEST(*child*.STATE) **then return** SOLUTION(*child*)

*frontier*  $\leftarrow$  INSERT(*child*, *frontier*)



# LIFO- 栈

- ▶ 栈 (stack) 又名堆栈，它是一种运算受限的线性表。
- ▶ 其限制是仅允许在表的一端进行插入和删除运算。这一端被称为栈顶，相对地，把另一端称为栈底。
- ▶ 向一个栈插入新元素又称作进栈、入栈或压栈，它是把新元素放到栈顶元素的上面，使之成为新的栈顶元素。
- ▶ 从一个栈删除元素又称作出栈或退栈，它是把栈顶元素删除掉，使其相邻的元素成为新的栈顶元素。



栈的操作

# 伪代码

STACK-EMPTY(S)

- ▶ if  $S.top == 0$ :
  - ▶ return True
- ▶ else return False

PUSH(S, x)

- ▶  $S.top = S.top + 1$
- ▶  $S[S.top] = x$

POP(S)

- ▶ if STACK-EMPTY(S)
  - ▶ error 'underflow'
- ▶ else:
  - ▶  $S.top = S.top - 1$
  - ▶ return  $S[S.top+1]$

```
class Stack(object):
    def __init__(self):
        self._elements = []
        self._size = 0

    def stack_empty(self):
        if self._size <= 0:
            return True
        else:
            return False

    def insert(self, e):
        self._elements = self._elements + [e]
        # self._elements.append(e)
        self._size += 1

    def pop(self):
        if self.stack_empty():
            raise Exception('Stack underflow!')
        e = self._elements[-1]
        self._size -= 1
        self._elements = self._elements[:self._size]
        # del self._elements[-1]
        return e

    def get_size(self):
        return self._size
```

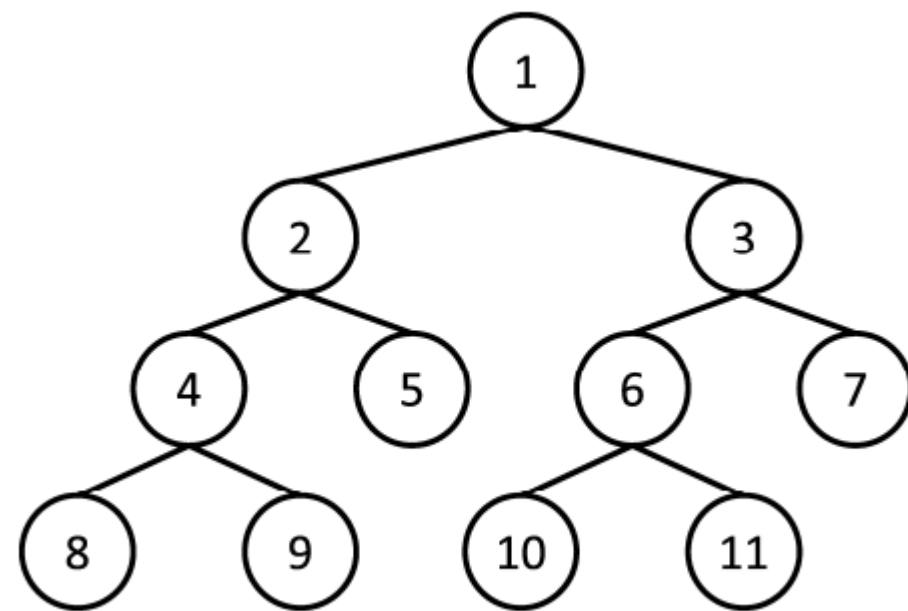
# 目录

- ▶ 深度优先搜索和栈
- ▶ 广度优先搜索和队列
- ▶ Uniform Cost Search和优先队列
- ▶ A star



# 广度优先搜索BFS

- ▶ BFS总是先搜索距离初始状态近的状态。
- ▶ 也就是说，开始状态→1次转移可以到达的所有状态→2次转移可以到达的所有状态→.....



状态转移的顺序

**function** BREADTH-FIRST-SEARCH(*problem*) **returns** a solution, or failure

*node*  $\leftarrow$  a node with STATE = *problem*.INITIAL-STATE, PATH-COST = 0

→ **if** *problem*.GOAL-TEST(*node*.STATE) **then return** SOLUTION(*node*)

→ *frontier*  $\leftarrow$  a FIFO queue with *node* as the only element

*explored*  $\leftarrow$  an empty set

**loop do**

**if** EMPTY?(*frontier*) **then return** failure

→ *node*  $\leftarrow$  POP(*frontier*) /\* chooses the shallowest node in *frontier* \*/

add *node*.STATE to *explored*

**for each** *action* **in** *problem*.ACTIONS(*node*.STATE) **do**

→ *child*  $\leftarrow$  CHILD-NODE(*problem*, *node*, *action*)

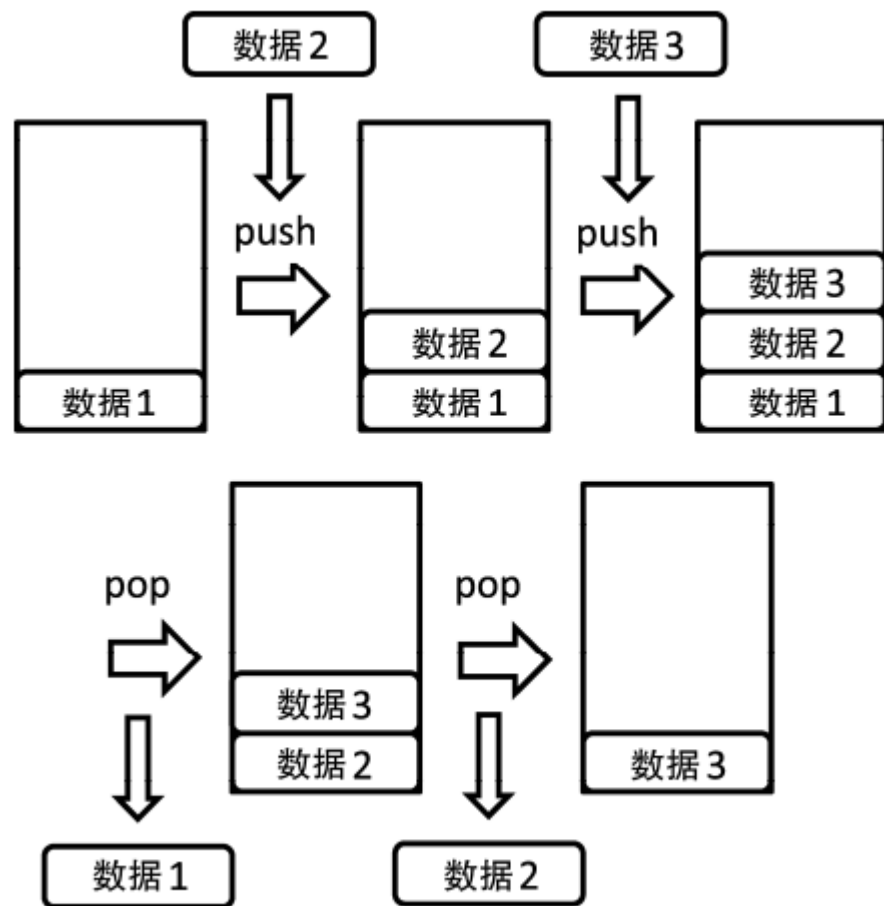
**if** *child*.STATE is not in *explored* or *frontier* **then**

→ **if** *problem*.GOAL-TEST(*child*.STATE) **then return** SOLUTION(*child*)

*frontier*  $\leftarrow$  INSERT(*child*, *frontier*)

# FIFO- 队列

- ▶ 队列也是一种特殊的线性表
- ▶ 特殊之处在于它只允许在表的前端 (front) 进行删除操作，而在表的后端 (rear) 进行插入操作，和栈一样，队列是一种操作受限制的线性表。
- ▶ 进行插入操作的端称为队尾，进行删除操作的端称为队头。



队列的操作

# 伪代码

STACK-EMPTY(S)

- ▶ if  $S.size == 0$ :
  - ▶ return True
- ▶ else return False

INSERT(S, x)

- ▶  $S.size = S.size + 1$
- ▶  $S[S.size] = x$

POP(S)

- ▶ if STACK-EMPTY(S)
  - ▶ error 'underflow'
- ▶ else:
  - ▶  $e = S[0]$
  - ▶  $S = S[1:S.size]$
  - ▶  $S.size = S.size - 1$
  - ▶ return e

```
class Queue(object):
    def __init__(self):
        self._elements = []
        self._size = 0

    def queue_empty(self):
        if self._size <= 0:
            return True
        else:
            return False

    def insert(self, e):
        self._elements = self._elements + [e]
        # self._elements.append(e)
        self._size += 1

    def pop(self):
        if self.queue_empty():
            raise Exception('Queue underflow!')
        e = self._elements[0]
        self._elements = self._elements[1:]
        self._size -= 1
        return e

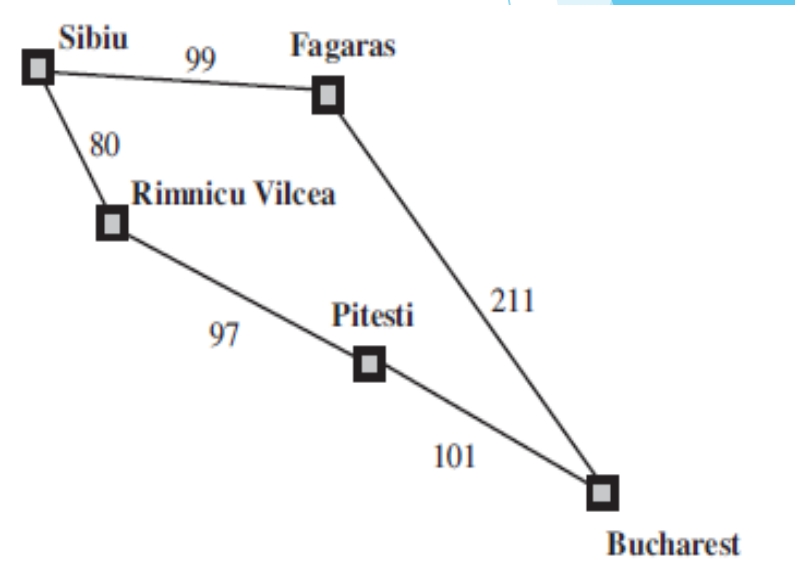
    def get_size(self):
        return self._size
```

# 目录

- ▶ 深度优先搜索和栈
- ▶ 广度优先搜索和队列
- ▶ **Uniform Cost Search**和优先队列
- ▶ A star

# Uniform Cost Search

- ▶ Uniform Cost Search 和 BFS 一样是先搜索距离初始节点较近的节点，但是二者对于距离的定义方式不同，一般来说 Uniform Cost Search 当中每一条边的权重是不太一样的。如果 Uniform Cost Search 的每一条边上定义的距离都是1，那么 Uniform Cost Search 等价于 BFS。



**function** UNIFORM-COST-SEARCH(*problem*) **returns** a solution, or failure

*node*  $\leftarrow$  a node with STATE = *problem*.INITIAL-STATE, PATH-COST = 0

→ *frontier*  $\leftarrow$  a priority queue ordered by PATH-COST, with *node* as the only element

→ *explored*  $\leftarrow$  an empty set

**loop do**

**if** EMPTY?(*frontier*) **then return** failure

*node*  $\leftarrow$  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*  $\leftarrow$  CHILD-NODE(*problem*, *node*, *action*)

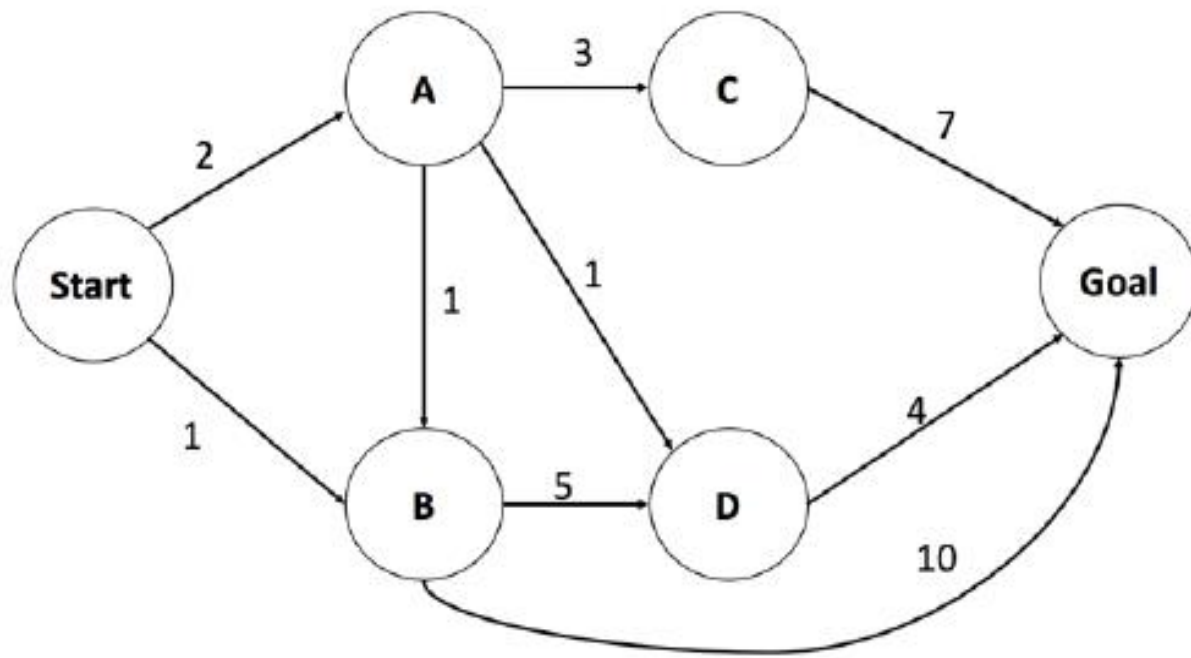
**if** *child*.STATE is not in *explored* or *frontier* **then**

→ *frontier*  $\leftarrow$  INSERT(*child*, *frontier*)

**else if** *child*.STATE is in *frontier* with higher PATH-COST **then**

→ replace that *frontier* node with *child*





# Priority Queue- 优先队列？

- ▶ 入队之后的排序不再由入队的时间顺序决定，转而由数据的值的顺序来决定。
- ▶ 进行取出操作的时候，拿出在在优先队列中排名最好的那一个数据。

# 用二叉堆实现优先队列

- ▶ 优先队列的插入更多使用的是堆排序算法。
- ▶ 可以直接调用heapq
- ▶ 更多的内容可以参考算法导论第六章堆排序

# 目录

- ▶ 深度优先搜索和栈
- ▶ 广度优先搜索和队列
- ▶ Uniform Cost Search和优先队列
- ▶ A star

# A star

- ▶ A star 在 Uniform Cost Search 的基础上利用  $h(n)$  估计从当前节点  $n$  到最终节点需要走的实际距离  $f(n)$ ，当  $h(n)$  估计满足下面的条件的时候，A star 可以保证选出来的路径是最优的。

$$f(n) \geq h(n)$$

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

$c(n, a, n')$  为从  $n$  到  $n'$  所走的实际距离

假设  $g(n)$  是初始节点到  $n$  的最优路径的值，我们将  $g(n) + h(n)$  作为节点  $n$  的值放入优先队列进行排序。

- ▶ Lab : Python3
- ▶ Project : Python2
- ▶ Pycharm(建议)
- ▶ 浏览器输入 10.88.3.61/JudgeOnline  
学号作为User ID申请账号


# Online Judgement

问题：求和

输入 1+1 ( $i+j$ ,  $i, j$  在 0~9 之间)

输出 2

```
▶ def easy_sum(s):  
▶     raise exception('完成str转int, 并进行加操作')  
  
▶ def main():  
▶     s = input().strip()  
▶     print(easy(s))  
  
▶ if __name__ == '__main__':  
▶     main()
```



```
▶ def easy_sum(s):  
▶     a1, a2 = int(s[0]), int(s[-1])  
▶     return a1 + a2  
  
▶ def main():  
▶     s = input().strip()  
▶     print(easy_sum(s))  
  
▶ if __name__ == '__main__':  
▶     main()
```



# Lab-1

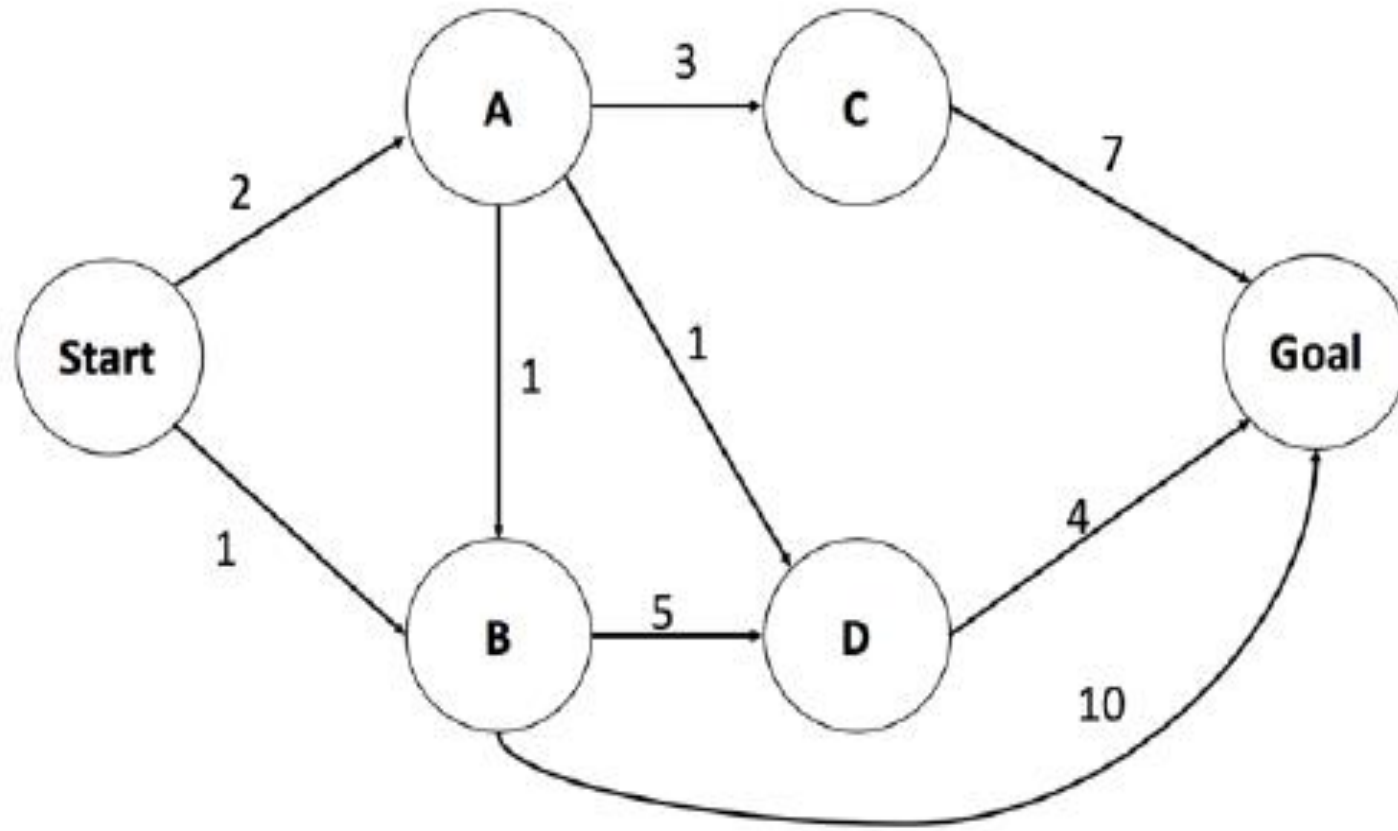
## ► 任务:

1. 实现 Uniform Cost Search 算法
2. 找到并输出从 'Start' 节点到 'Goal' 节点的最优搜索路径
3. 若无可行路径, 输出 'Unreachable'
4. Time Limits 2000ms
5. 先在自己的电脑上跑通, 不要直接在OJ上写!

## ► 注意:

1. 在OJ上提交并通过测试
2. OJ成绩计入平时分, 通过即满分

# Lab-1



# Lab-1

## ▶ Sample Input

Start A 2

Start B 1

A B 1

A C 3

A D 1

B D 5

B Goal 10

C Goal 7

D Goal 4

END

## ▶ Sample Output

Start->A->D->Goal

## Input

Each line presents an edge consisting of a tuple of start node, end node and cost. Input ends with 'END'.

## Output

One line for the optimal path in visit order, join by '->'. If there is no solution, print 'Unreachable'.

# Online Judgement

```
def main():
    actions = []
    while True:
        a = input().strip()
        if a != 'END':
            a = a.split()
            actions += a
        else:
            break
    graph_problem = problem('Start', actions)
    answer = UCS(graph_problem)
    s = "->"
    if answer == 'Unreachable':
        print(answer)
    else:
        path = s.join(answer)
        print(path)

if __name__ == '__main__':
    main()
```

处理之后：

a = ['start node',  
 'end node',  
 'distance']

例如：

Start A 2

['Start', 'A', '2']

```

import heapq
class PriorityQueue(object):
    def __init__(self):
        self.heap = []
        self.count = 0

    def push(self, item, priority):
        entry = (priority, self.count, item)
        heapq.heappush(self.heap, entry)
        self.count += 1

    def pop(self):
        (_, _, item) = heapq.heappop(self.heap)
        return item

    def isEmpty(self):
        return len(self.heap) == 0

    def update(self, item, priority):
        for index, (p, c, i) in enumerate(self.heap):
            if i == item:
                if p <= priority:
                    break
                del self.heap[index]
                self.heap.append((priority, c, item))
                heapq.heapify(self.heap)
                break
        else:
            self.push(item, priority)

class node:
    """define node"""
    def __init__(self, state, parent, path_cost, action):
        self.state = state
        self.parent = parent
        self.path_cost = path_cost
        self.action = action

```

```

class problem:
    """searching problem"""
    def __init__(self, initial_state, actions):
        self.initial_state = initial_state
        self.actions = actions
        # 可以在这里随意添加代码或者不加

    def search_actions(self, state):
        raise Exception('获取state的所有克星的动作')

    def solution(self, node):
        raise Exception('获取从初始节点到node的路径')

    def transition(self, state, action):
        raise Exception('节点的状态(名字)经过action转移之后的状态(名字)')

    def goal_test(self, state):
        raise Exception('判断state是不是终止节点')

    def step_cost(self, state1, action, state2):
        raise Exception('获得从state1到通过action到达state2的cost')

    def child_node(self, node_begin, action):
        raise Exception('获取从起始节点node_begin经过action到达的node')

def UCS(problem):
    node_test = node(problem.initial_state, '', 0, '')
    frontier = PriorityQueue()
    frontier.push(node_test, node_test.path_cost)
    explored = []
    raise Exception('进行循环')

```

# Homework-1

- ▶ Pacman.
- ▶ Different search methods.
- ▶ You need to fulfill the blank in the code following our requirements.
- ▶ Python only.
- ▶ 18pts + 2pts (bonus)
- ▶ This project accounts for 10% in the overall grading of the course.

# Homework-1

- ▶ DDL: 11:59 pm, April 2nd, Tuesday, 2019.
- ▶ Submit your code (only two files: `search.py` and `searchAgents.py`) and a short report (English, MAX 4 pages, 12pt font), through E-learning system in a zip file with the name “StudentID\_YourName\_PJ1.zip”.