# <2T>

# Uniform Cost Search Description

If all the edges in the search graph do not have the same cost then breadth-first search generalizes to uniform-cost search. Instead of expanding nodes in order of their depth from the root, uniform-cost search expands nodes in order of their cost from the root. At each step, the next step n to be expanded is one whose cost g(n) is lowest where g(n) is the sum of the edge costs from the root to node n. The nodes are stored in a priority queue. This algorithm is also known as Dijkstra’s single-source shortest algorithm.

Whenever a node is chosen for expansion by uniform cost search, a lowest-cost path to that node has been found. The worst case time complexity of uniform-cost search is O(bc/m), where c is the cost of an optimal solution and m is the minimum edge cost. Unfortunately, it also suggests the same memory limitation as [breadth-first search](http://intelligence.worldofcomputing.net/ai-search/breadth-first-search.html).

# Uniform Cost Search Pseudocode

Uniform-Cost-Search(problem) returns a solution, or failure

1    node <- a node with STATE = problem.Initial-State, Path-Cost = 0

2    frontier <- a priority queue ordered by Path-Cost, with node as the only element

3    explored <- an empty set

4    loop do

5        if Empty(frontier) return failure

6            node <- POP(Frontier) #Chooses the lowest-cost node in frontier

7        if problem.Goal-Test(node.State) then return Solution(node)

8        add node.State to explored

9        for each action in problem.Actions(node.State) do

10            child <- Child-Node(problem, node, action)

11            if child.State is not in explored or frontier then

12                frontier <- Insert(child, frontier)

13            else if child.State is in frontier with higher Path-Cost then

14                replace that frontier node with child

# Uniform Cost Search Code

import queue as Q

def search(graph, start, end):

    if start not in graph:

        raise TypeError(str(start) + ' not found in graph !')

        return

    if end not in graph:

        raise TypeError(str(end) + ' not found in graph !')

        return

    queue = Q.PriorityQueue()

    queue.put((0, [start]))

    while not queue.empty():

        node = queue.get()

        current = node[1][len(node[1]) - 1]

        if end in node[1]:

            print("Path found: " + str(node[1]) + ", Cost = " + str(node[0]))

            break

        cost = node[0]

        for neighbor in graph[current]:

            temp = node[1][:]

            temp.append(neighbor)

            queue.put((cost + graph[current][neighbor], temp))

def makeGraph(s):

    graph = {}

    for line in s.splitlines():

        tokens = line.split()

        node = tokens[0]

        graph[node] = {}

        for i in range(1, len(tokens) - 1, 2):

            # print(node, tokens[i], tokens[i + 1])

            # graph.addEdge(node, tokens[i], int(tokens[i + 1]))

            graph[node][tokens[i]] = int(tokens[i + 1])

    return graph

if \_\_name\_\_ == "\_\_main\_\_":

    s = """Arad Zerind 75 Timisoara 118 Sibiu 140

    Zerind Oradea 71 Arad 75

    Timisoara Arad 118 Lugoj 111

    Sibiu Arad 140 Oradea 151 Fagaras 99 RimnicuVilcea 80

    Oradea Zerind 71 Sibiu 151

    Lugoj Timisoara 111 Mehadia 70

    RimnicuVilcea Sibiu 80 Pitesti 97 Craiova 146

    Mehadia Lugoj 70 Dobreta 75

    Craiova Dobreta 120 RimnicuVilcea 146 Pitesti 138

    Pitesti RimnicuVilcea 97 Craiova 138 Bucharest 101

    Fagaras Sibiu 99 Bucharest 211

    Dobreta Mehadia 75 Craiova 120

    Bucharest Fagaras 211 Pitesti 101 Giurgiu 90

    Giurgiu Bucharest 90"""

    graph = makeGraph(s)

    search(graph, 'Arad', 'Bucharest')