**UNIFORM COST SEARCH**

**Introduction:**

[Uniform-Cost Search](https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm#Practical_optimizations_and_infinite_graphs) is a variant of [Dijikstra’s algorithm](https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/). Here, instead of inserting all vertices into a priority queue, we insert only the source, then one by one insert when needed. In every step, we check if the item is already in the priority queue (using the visited array). If yes, we perform the decrease key, else we insert it.

This variant of Dijkstra is useful for infinite graphs and that graph which are too large to represent in memory. Uniform-Cost Search is mainly used in Artificial Intelligence

**Working:**

Uniform-Cost Search is similar to Dijikstra’s algorithm. In this algorithm from the starting state, we will visit the adjacent states and will choose the least costly state then we will choose the next least costly state from the all un-visited and adjacent states of the visited states, in this way we will try to reach the goal state (note we won’t continue the path through a goal state), even if we reach the goal state we will continue searching for other possible paths ( if there are multiple goals). We will keep a priority queue that will give the least costly next state from all the adjacent states of visited states.

**Measuring problem-solving performance:s**

* Completeness - Yes
* Optimality - Yes
* Time Complexity - O(b^[1+c/a])
* Space Complexity - O(b^[1+c/a])

**Algorithm:**

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

**Advantages**

* It helps to find the path with the lowest cumulative cost inside a weighted graph having a different cost associated with each of its edge from the root node to the destination node.
* It is considered to be an optimal solution since, at each state, the least path is considered to be followed.

**Disadvantage**

* The open list is required to be kept sorted as priorities in priority queue needs to be maintained.
* The storage required is exponentially large.
* The algorithm may be stuck in an infinite loop as it considers every possible path going from the root node to the destination node.