**CCT College Dublin**

**Assessment Cover Page**

| | |
|---|---|
| **Module Title:** | Artificial Intelligence |
| **Assessment Title:** | Individual |
| **Lecturer Name:** | David McQuaid |
| **Student Full Name:** | Shivam Kumar Mehta |
| **Student Number:** | 2020359 |
| **Assessment Due Date:** | **Firday 17th November @ 23:59** |
| **Date of Submission:** | **Firday 17th November @ 23:59** |

**Declaration**

By submitting this assessment, I confirm that I have read the CCT policy on Academic Misconduct and understand the implications of submitting work that is not my own or does not appropriately reference material taken from a third party or other source. I declare it to be my own work and that all material from third parties has been appropriately referenced. I further confirm that this work has not previously been submitted for assessment by myself or someone else in CCT College Dublin or any other higher education institution.

**Github Link:**

**https://github.com/Shivam12-wq/AI_Lv8_CA2_v8**

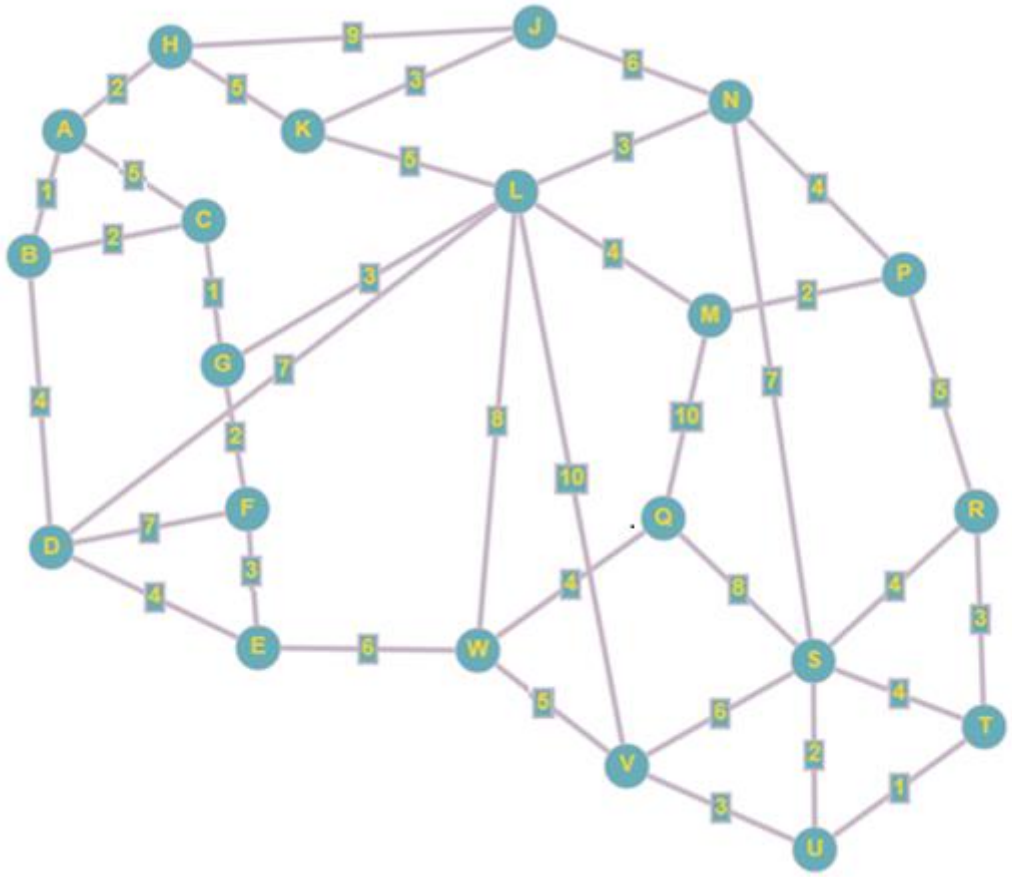



**Fig. 1**

(a) **The graph in Fig. 1 is a visualisation of the problem.**

(i) **Identify the differences between a graph and a tree. [0-5]**

Answer:

Difference between Graph and Tree:

| No. | Graph | Tree |
|---|---|---|
| 1 | It is a non-linear data structure. | It is also a non-linear data structure. |
| 2 | A graph is a set of vertices/nodes and edges. | A tree is a set of nodes and edges. |
| 3 | In the graph, there is no unique node | In a tree, there is a unique node |

| | | |
|---|---|---|
| | which is known as root. | which is known as root. |
| 4 | Each node can have several edges. | Usually, a tree can have several child nodes, and in the case of binary trees, each node consists of two child nodes. |
| 5 | Graphs can form cycles. | Trees cannot form a cycle. |

## (ii) Explain in detail how the graph is an abstraction of the problem. [0-5]

Answer:

Infinite or very large state spaces often prohibit the successful verification of graph transformation systems. Abstract graph transformation is an approach that tackles this problem by abstracting graphs to abstract graphs of bounded size and by lifting application of productions to abstract graphs. In this work, we present a new framework of abstractions unifying and generalising existing takes on abstract graph transformation. The precision of the abstraction can be adjusted according to the properties to be verified facilitating abstraction refinement.We present a modal logic defined on graphs, which is preserved and reflected by our abstractions. Finally, we demonstrate the usability of the framework by verifying a graph transformation model of a firewall.

Introduction

Formal verification of graph transformation systems aims at statically proving or inferring properties of a graph transformation system, where such properties are typically given in some form of temporal logic. It is crucial to distinguish verification and simulation, the latter being very useful only for debugging, whereas verification establishes a property for all computations of a graph transformation system. Problems do arise when approaching this task. One such problem is the possibly infinite behaviour of a system which in most cases makes it impossible to study the whole behaviour of the system. Another problem is space: even for a finite state space, each state can be quite big to represent.They can be characterised as to which approach to graph transformation is used for modelling, which verification technique is applied, and which applications are tackled. The technique presented in  feeds finite-state graph transformation systems, given as a double pushout system, to an offthe-shelf model checker to verify reactive systems. However, we face the more general problem of unbounded systems. The approaches presented in both  use

backwards reachability analysis for hyperedge replacement grammars trying to reach an initial graph by backwards search from a forbidden configuration.The technique is applied to mechatronic systems and ad-hoc network routing,respectively, but, unfortunately, is not guaranteed to terminate. An approximation of the behaviour of a graph transformation system in terms of Petri net unfoldings was used in  to verify properties of data structures residing in the  run-time heap of programs with dynamically allocated heap memory. Abstract graph transformation relies on abstract interpretation  of graph transformation systems, that is, given some equivalence relation, graphs are quotiented into abstract graphs of bounded, finite size. Application of productions is then lifted to work on abstract graphs. The abstraction first introduced in  summarises nodes with similar kind and number of incident edges, while the abstraction of considers similar adjacent nodes. These two abstractions are generalised in this work and put into a unifying framework. To this end, we introduce the notion of neighbourhood abstraction as a part of a general abstraction framework. For this abstraction, nodes are summarized  if they have similar neighbourhood up to some radius i, parameter of the abstraction. This enables abstractions with different precisions. Additionally,the number of possible abstract graphs obtained by neighbourhood abstraction  is bounded.We introduce a logic accompanying our abstractions: given a formula our abstraction guarantees that a) if the formula holds for the original graph, then it holds for the abstracted graph (preservation); and b) if the formula holds for the abstracted graph, then it holds for the original one too (reflection).

(iii) **Identify the advantages of using a visualisation such as the one shown in Fig. 1. [0-5]**

Answer:

**Introduction to Graphs**

Graphs are data structures used to represent "connections" between pairs of elements.

- These elements are called nodes. They represent real-life objects, persons, or entities.
- The connections between nodes are called edges.

This is a graphical representation of a graph:

Nodes are represented with colored circles and edges are represented with lines that connect these circles.

**Applications**

Graphs are directly applicable to real-world scenarios. For example, we could use graphs to model a transportation network where nodes would represent facilities that send or receive products and edges would represent roads or paths that connect them (see below).

Network represented with a graph

**Types of Graphs**

Graphs can be:

- **Undirected:** if for every pair of connected nodes, you can go from one node to the other in both directions.
- **Directed:** if for every pair of connected nodes, you can only go from one node to another in a specific direction. We use arrows instead of simple lines to represent directed edges.

**Weighted Graphs**

A **weight graph** is a graph whose edges have a "weight" or "cost". The weight of an edge can represent distance, time, or anything that models the "connection" between the pair of nodes it connects.
For example, in the weighted graph below you can see a blue number next to each edge. This number is used to represent the weight of the corresponding edge.

**Purpose and Use Cases**

With Dijkstra's Algorithm, you can find the shortest path between nodes in a graph. Particularly, you can find the shortest path from a node (called the "source node") to all other nodes in the graph, producing a shortest-path tree.
This algorithm is used in GPS devices to find the shortest path between the current location and the destination. It has broad applications in industry, specially in domains that require modeling networks.

**Basics of Dijkstra's Algorithm**

- Dijkstra's Algorithm basically starts at the node that you choose (the source node) and it analyzes the graph to find the shortest path between that node and all the other nodes in the graph.
- The algorithm keeps track of the currently known shortest distance from each node to the source node and it updates these values if it finds a shorter path.
- Once the algorithm has found the shortest path between the source node and another node, that node is marked as "visited" and added to the path.

- The process continues until all the nodes in the graph have been added to the path. This way, we have a path that connects the source node to all other nodes following the shortest path possible to reach each node.

**Requirements**

Dijkstra's Algorithm can only work with graphs that have **positive** weights. This is because, during the process, the weights of the edges have to be added to find the shortest path. If there is a negative weight in the graph, then the algorithm will not work properly. Once a node has been marked as "visited", the current path to that node is marked as the shortest path to reach that node. And negative weights can alter this if the total weight can be decremented after this step has occurred.

## (b) Demonstrate how Dijkstra's algorithm would find the shortest path to the solution in Fig.1 through diagrams and written explanation of each stage. [0-25]

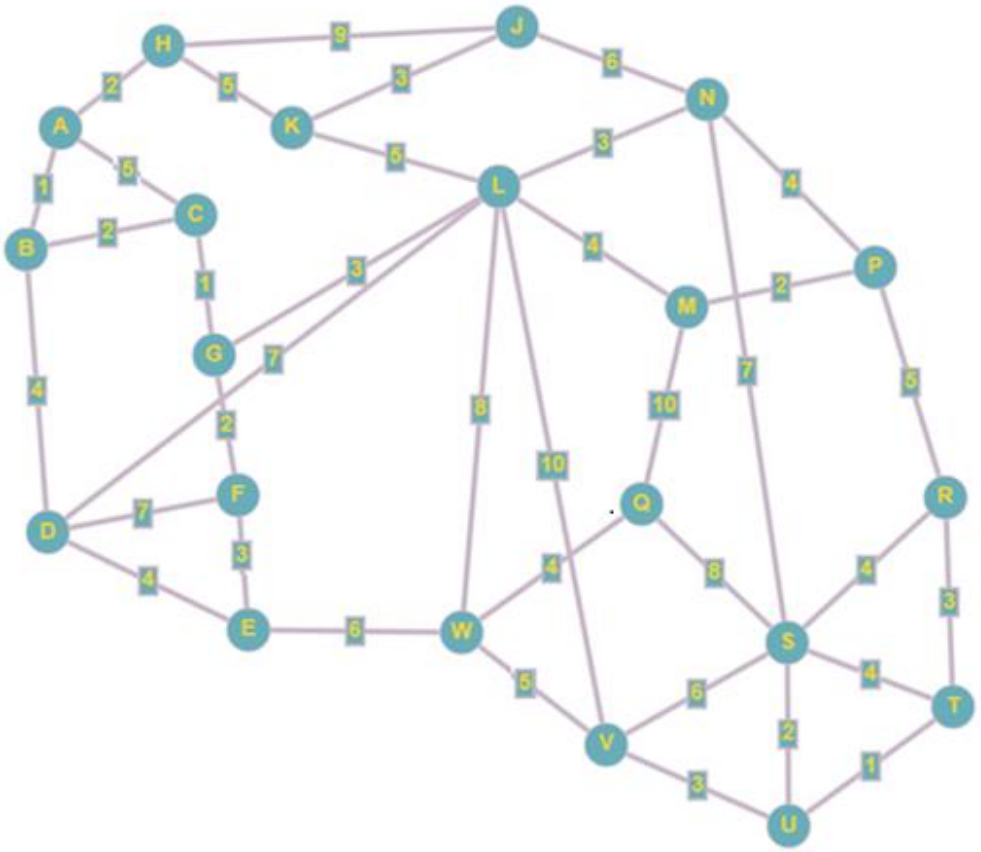
Answer:

**Basics of Dijkstra's Algorithm**

- Dijkstra's Algorithm basically starts at the node that you choose (the source node) and it analyzes the graph to find the shortest path between that node and all the other nodes in the graph.
- The algorithm keeps track of the currently known shortest distance from each node to the source node and it updates these values if it finds a shorter path.
- Once the algorithm has found the shortest path between the source node and another node, that node is marked as "visited" and added to the path.
- The process continues until all the nodes in the graph have been added to the path. This way, we have a path that connects the source node to all other nodes following the shortest path possible to reach each node.

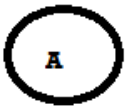
**Requirements:**

Dijkstra's Algorithm can only work with graphs that have positive weights. This is because, during the process, the weights of the edges have to be added to find the shortest path. If there is a negative weight in the graph, then the algorithm will not work properly. Once a node has been marked as "visited", the current path to that node is marked as the shortest path to reach that node. And negative weights can alter this if the total weight can be decremented after this step has occurred.




After updating the distances of the adjacent nodes, we need to:

**A-A:**

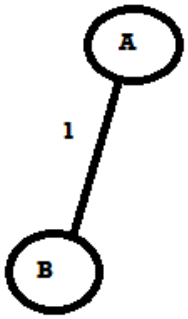



**Distance : a-a->0**

**Unvisited Nodes:[ A,B,C,D,E,F,G,H,J,K,L,M,N,P,Q,R,S,T,U,V,W ]**

- Select the node that is closest to the source node based on the current known distances.
- Mark it as visited.
- Add it to the path.

**A-B:**

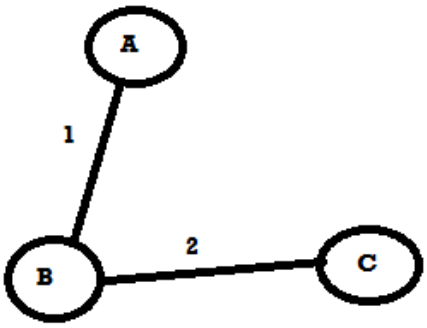



**Distance : a-b->1**

**Unvisited Nodes:[ ~~A~~,~~B~~,C,D,E,F,G,H,J,K,L,M,N,P,Q,R,S,T,U,V,W ]**

- Select the node that is closest to the source node based on the current known distances.
- Mark it as visited.
- Add it to the path.

**A-C:**

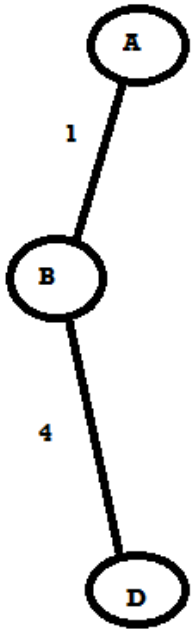



**Distance : a-c->(a+b)-(1+2)->3**

**Unvisited Nodes:[ ~~A~~,~~B~~,~~C~~,D,E,F,G,H,J,K,L,M,N,P,Q,R,S,T,U,V,W ]**

- Select the node that is closest to the source node based on the current known distances.
- Mark it as visited.
- Add it to the path.

**A-D:**

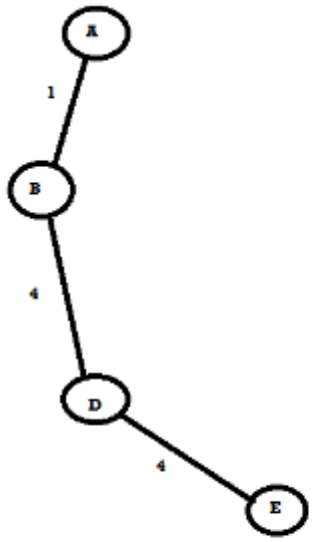



**Distance : a-d->(a-b) ->(1+4)->5**

**Unvisited Nodes:[ ~~A~~,~~B~~,~~C~~,~~D~~,E,F,G,H,J,K,L,M,N,P,Q,R,S,T,U,V,W ]**

- Select the node that is closest to the source node based on the current known distances.
- Mark it as visited.
- Add it to the path.

**A-E:**

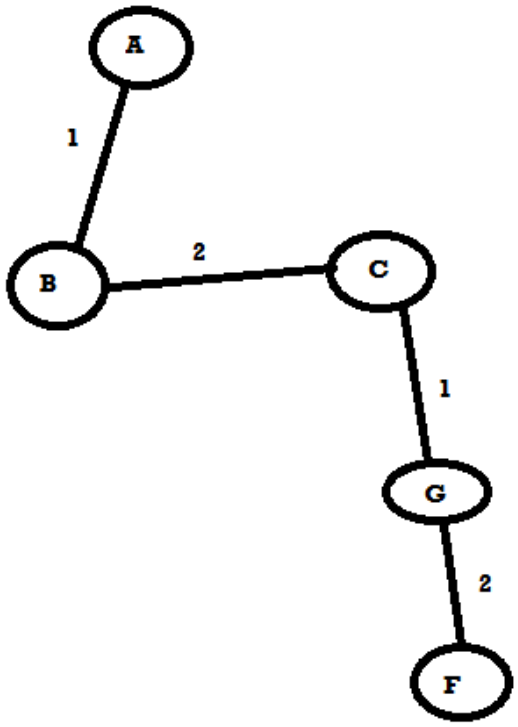



**Distance : a-e->(a-b-d->e):->(1+4+4)->9**

**Unvisited Nodes:[ A,B,C,D,E,F,G,H,J,K,L,M,N,P,Q,R,S,T,U,V,W ]**

- Select the node that is closest to the source node based on the current known distances.
- Mark it as visited.
- Add it to the path.

- **A-F:**

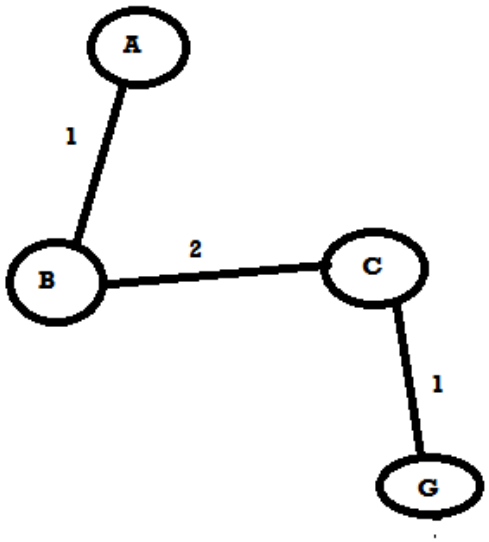



**Distance : a-f->(a-b-c-g->):(1+2+1+2)->6**

**Unvisited Nodes:[ A,B,C,D,E,F,G,H,J,K,L,M,N,P,Q,R,S,T,U,V,W ]**

- Select the node that is closest to the source node based on the current known distances.
- Mark it as visited.
- Add it to the path.

**A-G:**

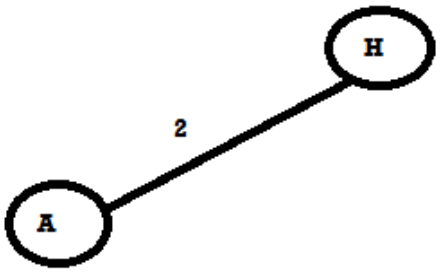



**Distance : a-g->(a-b-c->g):(1+2+1)->4**

**Unvisited Nodes:[ A,B,C,D,E,F,G,H,J,K,L,M,N,P,Q,R,S,T,U,V,W ]**

- Select the node that is closest to the source node based on the current known distances.
- Mark it as visited.
- Add it to the path.

**A-H:**

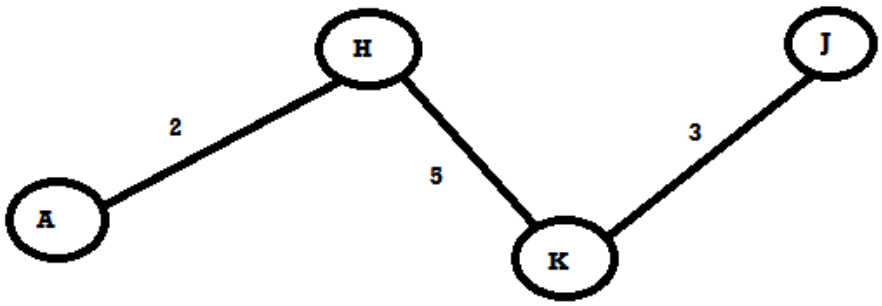



**Distance : a-h->2**

**Unvisited Nodes:[ A,B,C,D,E,F,G,H,J,K,L,M,N,P,Q,R,S,T,U,V,W ]**

- Select the node that is closest to the source node based on the current known distances.
- Mark it as visited.
- Add it to the path.

**A-J:**



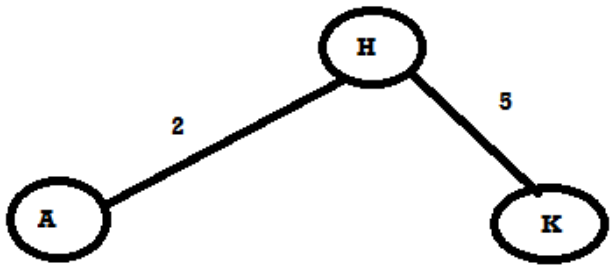

**Distance : a-j->(a-h-k->j): (2+5+3)-->10**

**Unvisited Nodes:[ A,B,C,D,E,F,G,H,J,K,L,M,N,P,Q,R,S,T,U,V,W ]**

- Select the node that is closest to the source node based on the current known distances.

- Mark it as visited.
- Add it to the path.

- **A-K:**

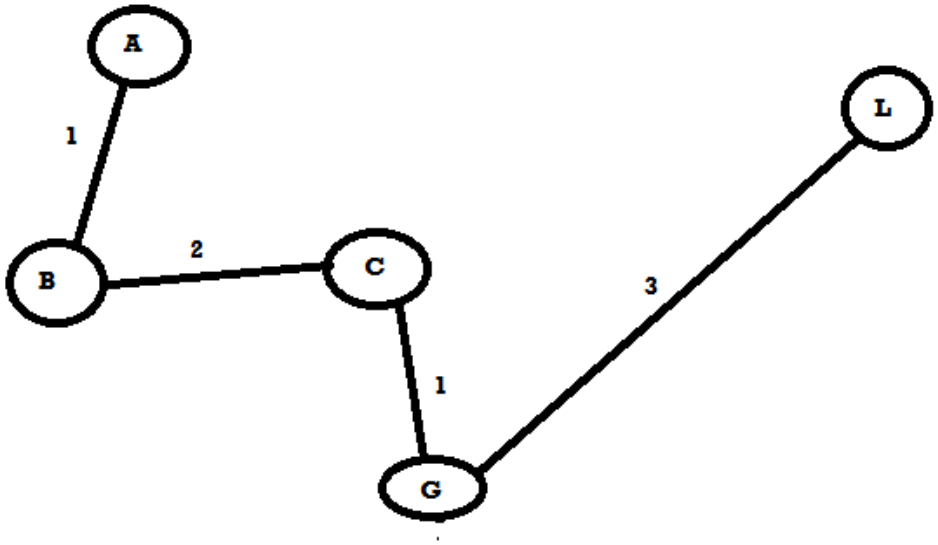



**Distance : a-k->(a-h->k):(2+5)-> 7**

**Unvisited Nodes:[ A,B,C,D,E,F,G,H,J,K,L,M,N,P,Q,R,S,T,U,V,W ]**

- Select the node that is closest to the source node based on the current known distances.
- Mark it as visited.
- Add it to the path.

**A-L:**

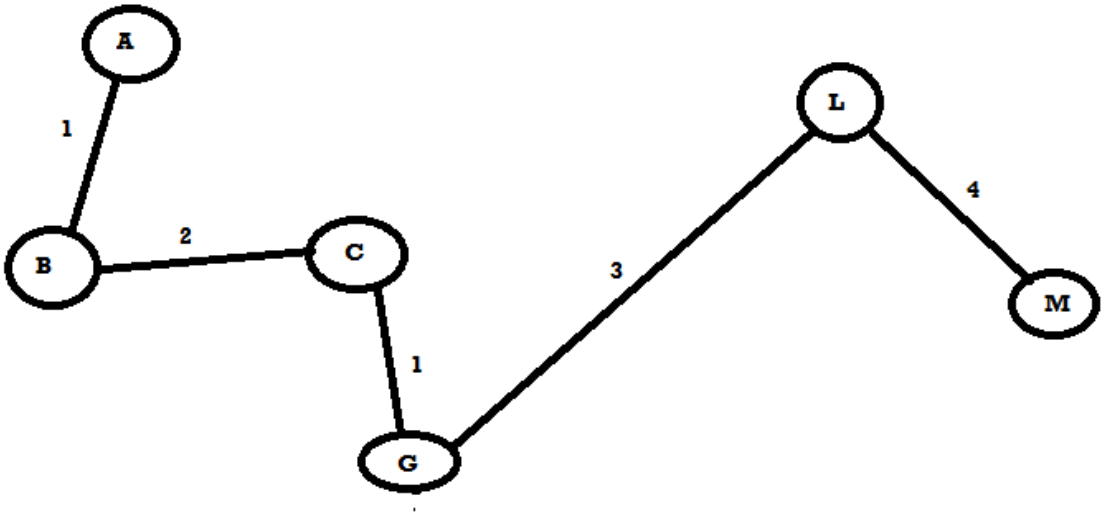



**Distance : a-l->(a-b-c-g->L):(1+2+1+3)->7**

**Unvisited Nodes: [ A B C D E F G H I K L M N P O R S T U V W ]**

- Select the node that is closest to the source node based on the current known distances.
- Mark it as visited.
- Add it to the path.

**A-M:**

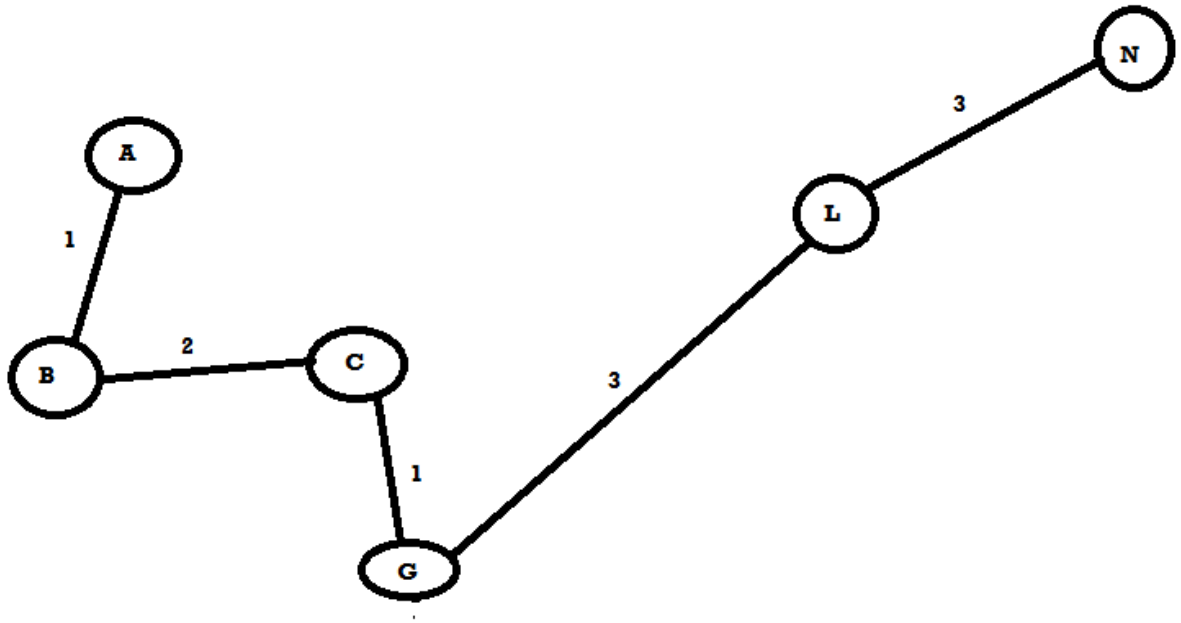



**Distance : a-m->(a-b-c-g-l->m):(5+1+3+4)->11**

**Unvisited Nodes:[A,B,C,D,E,F,G,H,I,J,K,L,M,N,P,Q,R,S,T,U,V,W ]**

- Select the node that is closest to the source node based on the current known distances.
- Mark it as visited.
- Add it to the path.

**A-N:**

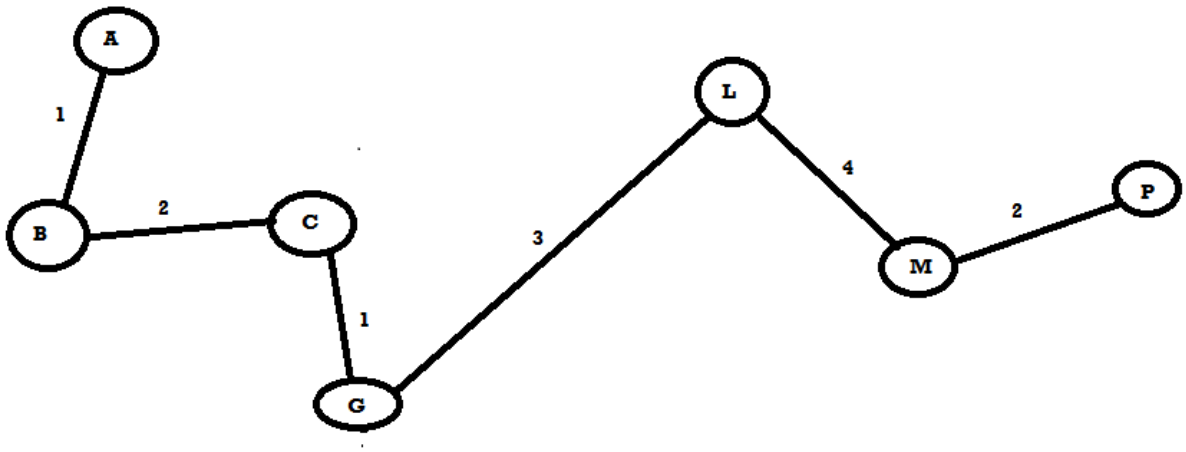



**Distance : a-n->(a-b-c-g-l->n):(5+1+3+3)->10**

**Unvisited Nodes:[ A,B,C,D,E,F,G,H,J,K,L,M,N,P,Q,R,S,T,U,V,W ]**

- Select the node that is closest to the source node based on the current known distances.
- Mark it as visited.
- Add it to the path.

**A-P:**

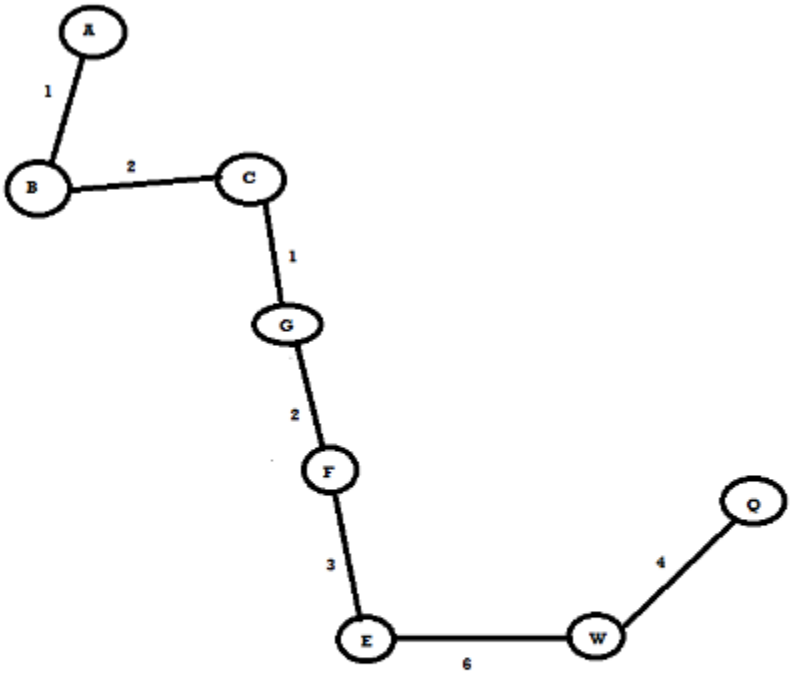



**Distance : a-p->(a-b-c-g-l-m->p):(5+1+3+4+2)->13**

**Unvisited Nodes:[ A,B,C,D,E,F,G,H,I,K,L,M,N,P,Q,R,S,T,U,V,W ]**

- Select the node that is closest to the source node based on the current known distances.
- Mark it as visited.
- Add it to the path.

**A-Q:**

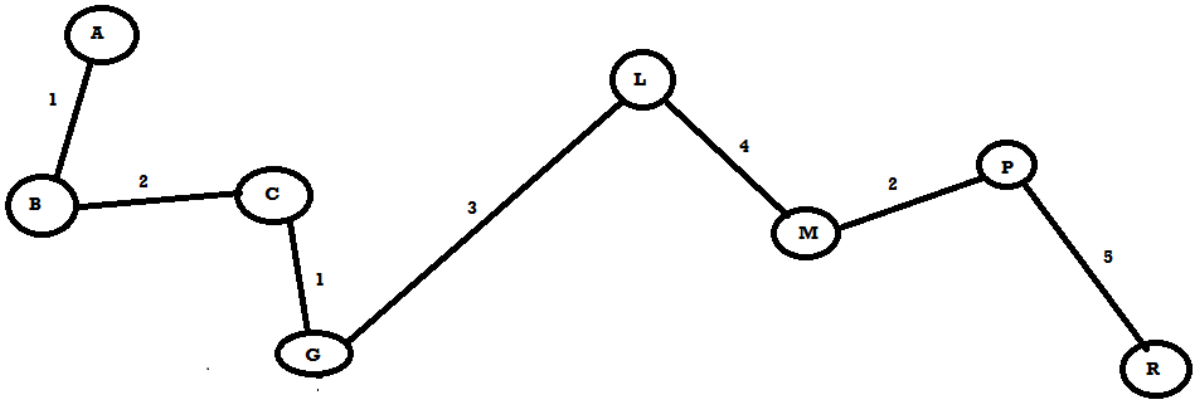



**Distance : a-q->(a-b-c-g-f-e-w->q:(5+1+2+3+6+4)->19**

**Unvisited Nodes:[ A,B,C,G,F,E,W,Q,H,J,K,L,M,N,P,R,S,T,U,V,W ]**

- Select the node that is closest to the source node based on the current known distances.
- Mark it as visited.
- Add it to the path.

**A-R:**

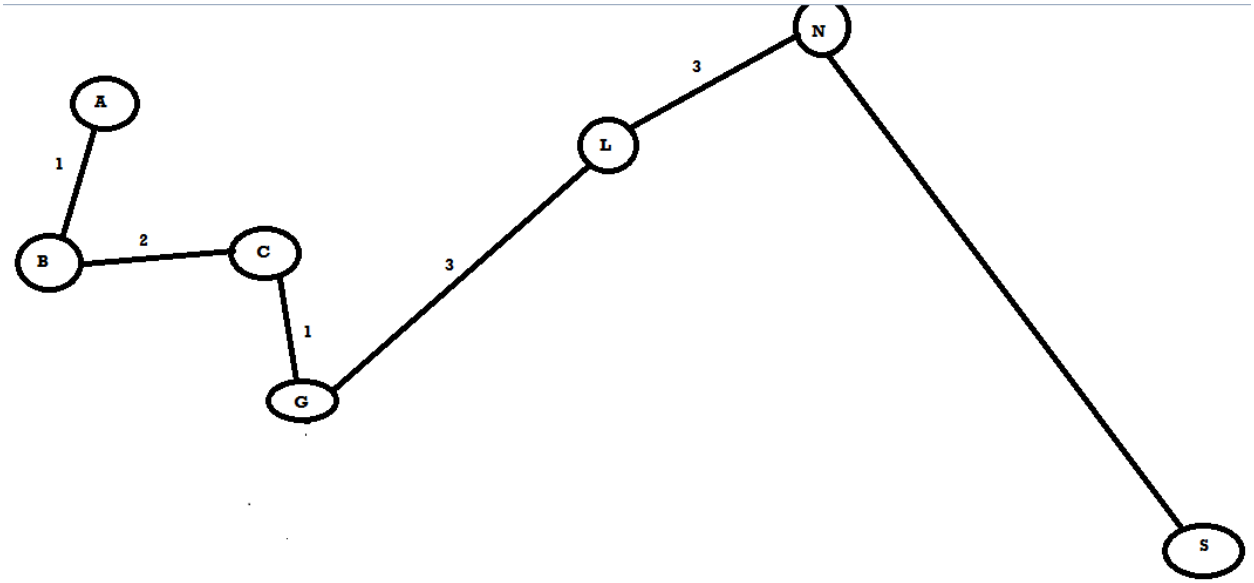



**Distance : a-r->(a-b-c-g-l-m-p->r): (1+2+1+3+4+2+5)->18**

**Unvisited Nodes:[ ~~A,B,C,D,E,F,G,H,I,K,L,M,N,P,Q,R~~,S,T,U,V,W ]**

- Select the node that is closest to the source node based on the current known distances.
- Mark it as visited.
- Add it to the path.

**A-S:**

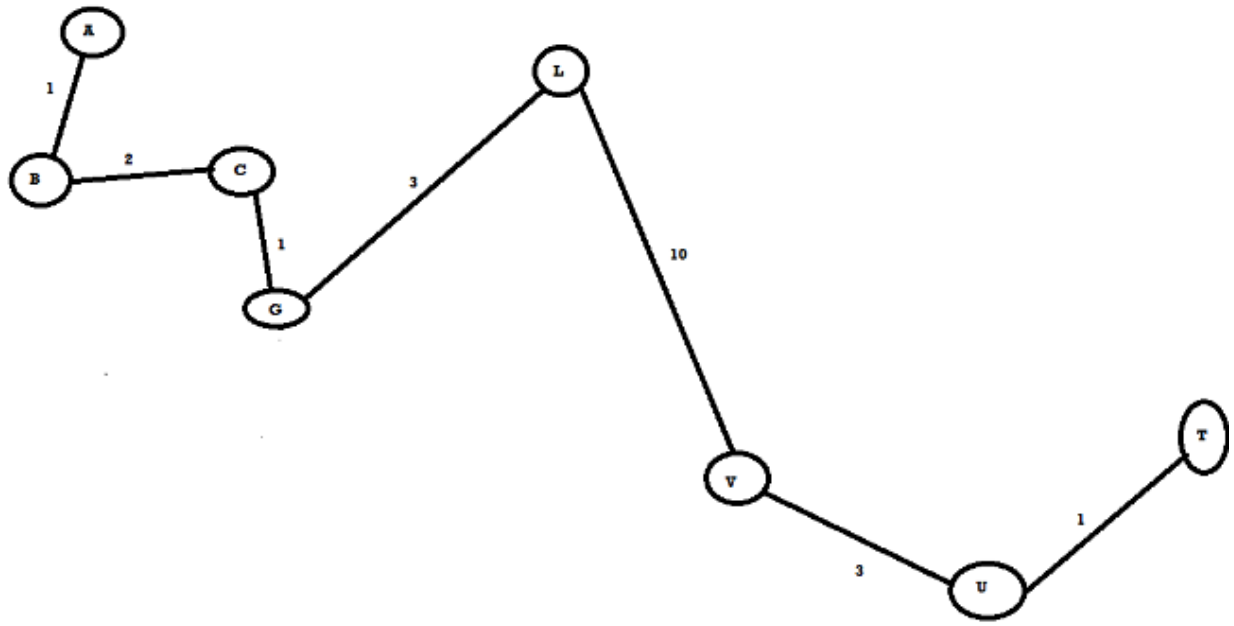



**Distance : a-s->(a-b-c-g-l-m-n->s)->(1+2+1+3+3+7)->17**

**Unvisited Nodes:[ ~~A,B,C,D,E,F,G,H,I,K,L,M,N,P,Q,R,S~~,T,U,V,W ]**

- Select the node that is closest to the source node based on the current known distances.
- Mark it as visited.
- Add it to the path.

**A-T:**

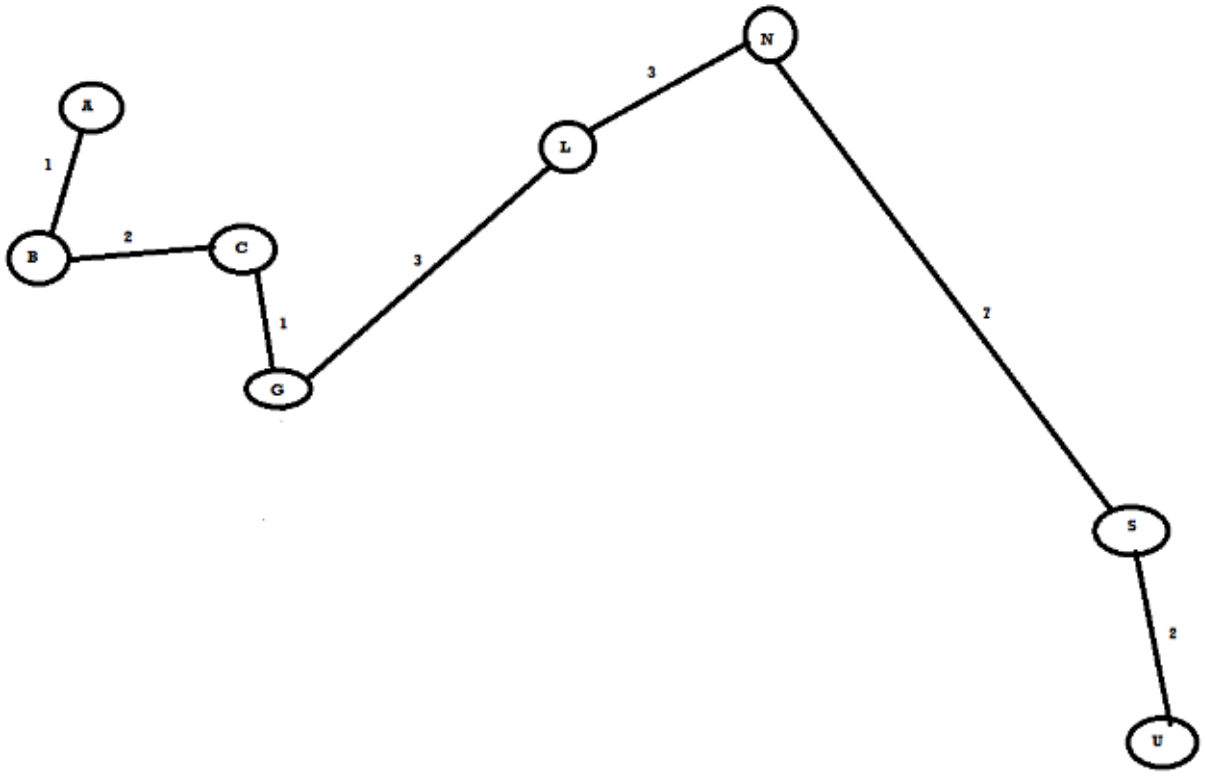



**Distance : a-t->(a-b-c-g-l-v-u->t):->(1+2+1+3+10+3+1)->21**

**Unvisited Nodes:[ A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S,T,U,V ]**

- Select the node that is closest to the source node based on the current known distances.
- Mark it as visited.
- Add it to the path.

**A-U:**

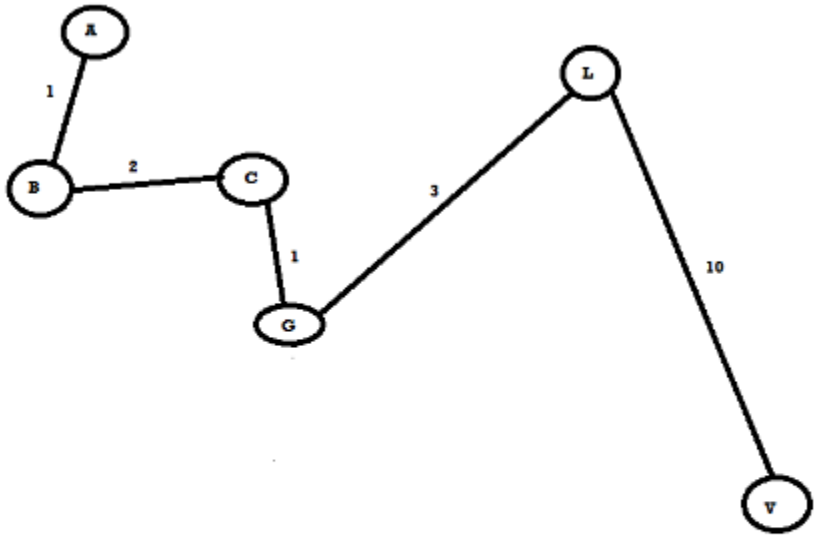



**Distance : a-u->(a-b-c-g-l-n-s->u):->(1+2+1+3+3+7+2)->19**

**Unvisited Nodes:[ ~~A,B,C,D,E,F,G,H,I,K,L,M,N,P,Q,R,S,T,U,V~~ ]**

- Select the node that is closest to the source node based on the current known distances.
- Mark it as visited.
- Add it to the path.

**A-V:**

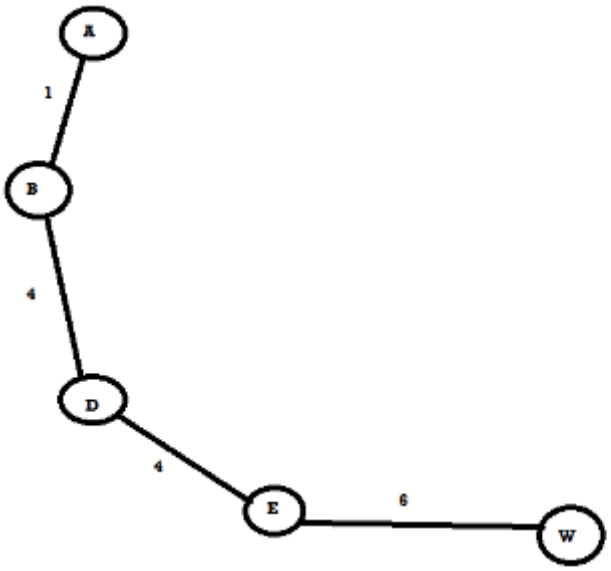



**Distance : a-v->(a-b-c-g-l->v):->(1+2+1+3+10)->17**

**Unvisited Nodes: [ ~~A,B,C,D,E,F,G,H,I,K,L,M,N,P,Q,R,S,T,U,V,W~~ ]**

- Select the node that is closest to the source node based on the current known distances.
- Mark it as visited.
- Add it to the path.

**A-W:**




**Distance : a-w->(a-b-d-e->w):->(1+4+4+6)->15**

**Unvisited Nodes:[A,B,C,D,E,F,G,H,I,K,L,M,N,P,Q,R,S,T,U,V,W ]**

- Select the node that is closest to the source node based on the current known distances.
- Mark it as visited.
- Add it to the path.

CONCLUSION:

With help of dijkstras algorithm find shortest path from source node

Distance of a from source vertex: 0
Distance of b from source vertex: 1
Distance of c from source vertex: 3
Distance of d from source vertex: 5
Distance of e from source vertex: 9
Distance of f from source vertex: 6
Distance of g from source vertex: 4
Distance of h from source vertex: 2
Distance of j from source vertex: 10
Distance of k from source vertex: 7
Distance of l from source vertex: 7

Distance of m from source vertex: 11
Distance of n from source vertex: 10
Distance of p from source vertex: 13
Distance of q from source vertex: 19
Distance of r from source vertex: 18
Distance of s from source vertex: 17
Distance of t from source vertex: 21
Distance of u from source vertex: 19
Distance of v from source vertex: 17
Distance of w from source vertex: 15

2 **The creator of the puzzle has been told that the A* algorithm is more efficient at finding the shortest path because it uses heuristics. Compare the performance of Dijkstra's algorithm and the A* search algorithm, referring to heuristics, to find the shortest path to the problem by implementing both algorithms programmatically and comparing the solutions generated in Mark-down. Refer to the complexity of the algorithms and compare the actual time it takes for the solutions to be processed.[0-60]**

Answer:

**Dijkstra-vs-a*-pathfinding**

# 1. Introduction

Dijkstra's Algorithm and A* are well-known techniques to search for the optimal paths in graphs. In this tutorial, we'll discuss their similarities and differences.

# 2. Finding the Optimal Path

In AI search problems, we have a graph whose nodes are an AI agent's states, and the edges correspond to the actions the agent has to take to go from one state to another. **The task is to find the optimal path that leads from the start state to a state that passes the goal test.**

For example, the start state can be the initial placement of the pieces on the chessboard, and any state in which white wins is a goal state for its search–the same holds for the black's goal states.

**When the edges have costs, and the total cost of a path is a sum of its constituent edges' costs, then the optimal path between the start and goal states is the least expensive one.** We can find it using Dijkstra's algorithm, Uniform-Cost Search, and A*, among other algorithms.

# 3. Dijkstra's Algorithm

The input for Dijkstra's Algorithm contains the graph $G = (V, E, c)$, the source vertex $s \in V$, and the target node $t \in V$. $V$ is the set of vertices, $E$ is the set of edges, and $c(u, v)$ is the cost of the edge $(u, v) \in E$. The connection to AI search problems is straightforward. $V$ corresponds to the states, $s$ to the start state, and the goal test is checking for equality to $t$.

Dijkstra splits $V$, the set of vertices in the graph, in two disjoint and complementary subsets: $S$ and $Q$. **$S$ contains the vertices whose optimal paths from $s$ we've found. In contrast, $Q$ contains the nodes whose optimal paths we currently don't know but have the upper bounds $g$ of their actual costs.** Initially, Dijkstra places all the vertices in $Q$ and sets the upper bound $g(u)$ to $+\infty$ for every $u \in V$.

Dijkstra moves a vertex from $Q$ to $S$ at each step until it moves $t$ to $S$. It chooses for removal the node in $Q$ with the minimal value of $g$. That's why $Q$ is usually a priority queue.

When removing $u$ from $Q$, the algorithm inspects all the outward edges $(u, v) \in E$ and checks if $g(v)$ if $g(u) + c(u, v) < g(v)$. If so, Dijkstra has found a tighter upper bound, so it sets $g(v)$ to $g(u) + c(u, v)$. This step is called the relax operation.

**The algorithm's invariant is that whenever it chooses $u \in Q$ to relax its edges and remove it to $S$, $g(u)$ is equal to the cost of the optimal path from $s$ to $u$.**

# 4. A*

In AI, many problems have state graphs so large that they can't fit the main memory or are even infinite. So, we can't use Dijkstra to find the optimal paths. Instead, we use UCS. It's logically equivalent to Dijkstra but can handle infinite graphs.

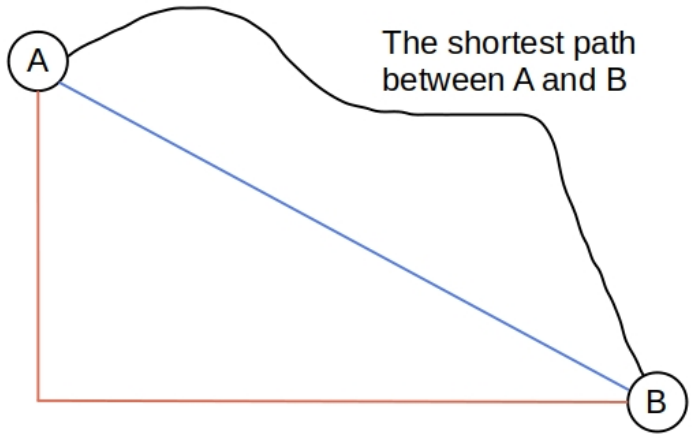
However, because UCS explores the states systematically in the waves of uniform cost, it may expand more states than necessary. For example, let's imagine that UCS has reached a state whose optimal path costs $C'$ and whose immediate successor is the sought goal state (with the optimal cost $C^*$). By the nature of its search, UCS would get to the goal only after visiting all the states whose path costs between $C'$ and $C^*$. The problem is that there may be many such states.

The A* algorithm intends to avoid such scenarios. **It does so by considering not only how far a state is from the start but also how close it is to the goal.** The latter cost is unknown, for to know it with certainty, we'd have to find the lowest-cost paths between all the nodes and the closest goal, and that's a problem as hard as the one we're currently solving! So, we can only use the more or less precise estimates of the costs to the closest goal.

**Those estimates are available through functions we call heuristics.** That's why we say that A* is informed while we call UCS an uninformed search algorithm. The latter uses only the problem definition to solve it, whereas the former uses additional knowledge the heuristics provide.

## 4.1. Heuristics

**A heuristic $h$ is a function that receives a state $u$ and outputs an estimate of the optimal path's cost from $u$ to the closest goal.** For example, if the states represent points on a 2D map, the Euclidean and Manhattan distances are good candidates for heuristics:




The **Euclidean** and **Manhattan** distances estimate **the shortest path's actual length.**

However, the use of heuristics is not the only difference between Dijkstra and A*.

### 4.2. The Frontier

A* differs from UCS only in the ordering of its frontier. UCS orders the nodes in the frontier by their $g$ values. They represent the costs of the paths from the start state to the frontier nodes' states. **In A*, we order the frontier by $g + h$.** If $u$ is an arbitrary node in the frontier, we interpret the value of $g(u) + h(u)$ as the estimated cost of the optimal path that connects the start and the goal and passes through the $u$'s state.

Now, confusion may arise because we defined $h$ as taking a state as its input but now give it a search node. The difference is only technical, because we can define $h(u) = h(u.state)$.

## 5. The Difference in Assumptions

**Dijkstra has two assumptions:**

- **the graph is finite**
- **the edge costs are non-negative**

The first assumption is necessary because Dijkstra places all the nodes into $Q$ at the beginning. If the second assumption doesn't hold, we should use the Bellman-Ford algorithm. The two assumptions ensure that Dijkstra always terminates and returns either the optimal path or a notification that no goal is reachable from the start state.

**It's different for A*. The following two assumptions are necessary for the algorithm to terminate always:**

- **the edges have strictly positive costs $\geq \varepsilon > 0$**
- **the state graph is finite, or a goal state is reachable from the start**

So, A* can handle an infinite graph if all the graph's edges are positive and there's a path from the start to a goal. But, there may be no zero-cost edges even in the finite graphs. Further, there are additional requirements that A* should fulfill so that it returns only the optimal paths. We call such algorithms optimal.

### 5.1. Optimality of A*

First, we should note that A* comes in two flavors: the Tree-Like Search (TLS) and the Graph-Search (GS). **TLS doesn't check for repeated states, whereas GS does.**

**If we don't check for repeated states and use the TLS A*, then $h$ should be admissible for the algorithm to be optimal.** We say that a heuristic is admissible if it never overestimates the cost to reach the goal. So, if $c^*(u)$ is the actual cost of the optimal path from $u.state$ to the goal, then $h$ is admissible if $h(u) \leq c^*(u)$ for any $u$. Even inadmissible heuristics may lead us to the optimal path or even give us an optimal algorithm, but there are no guarantees for that.

**If we want to avoid repeated states, we use the GS A*, and to make it optimal, we have to use consistent heuristics.** A heuristic $h$ is consistent if it fulfills the following condition for any node $u$ and its child $v$:

$$h(u) \leq c(u, v) + h(v)$$

where $c(u, v)$ is the cost of the edge $u.state \rightarrow v.state$ in the state graph. A consistent heuristic is always admissible, but the converse doesn't necessarily hold.

# 6. Search Trees and Search Contours

**Both Dijkstra's Algorithm and A* superimpose a search tree over the graph.** Since paths in trees are unique, **each node in the search tree represents both a state and a path to the state**. So, we can say that both algorithms maintain a tree of paths from the start state at each point in their execution. However, Dijkstra and A* differ by the order in which they include the nodes in their trees.

**We can visualize the difference by drawing the search wavefronts over the graph**. We define each wave as the set of nodes having the path cost in the range $[C', C'']$ when added to the tree, where we choose the border values $C'$ and $C''$ in advance. If we suitably select the incremental values $C_1, C_2, \ldots$ to define the waves, they'll show us how the search progresses through the graph.

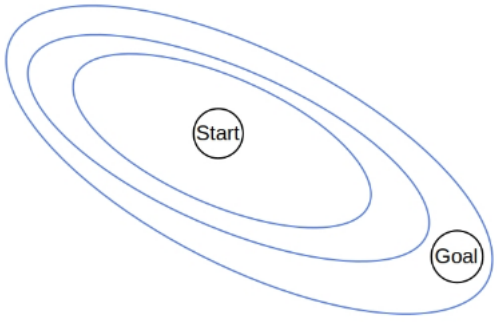
## 6.1. Dijkstra

The search tree's root is $s$, and its nodes are the vertices in $S$. Because of the invariant, we know that the nodes represent the optimal paths. The edge non-negativity assumption, together with the policy of choosing the lowest-cost node to move from $Q$ to $S$, ensures another property of Dijkstra. It can move to $S$ a state whose optimal path costs more than $C$ only after moving all the states whose optimal paths cost $\leq C$, for any $C$.

**So, the contours which separate the waves in Dijkstra look more or less like uniform circles:**

## 6.2. A*

Whereas UCS and Dijskra spread through the state graph in all directions and have uniform contours, A* favors some directions over the others. Between two nodes with the same $g$ values, A* prefers the one with a better $h$ value. So, **the heuristic stretches the contours in the direction of the goal state(s)**:



# 7. Differences in Complexity

Here, we'll compare the space and time complexities of Dijkstra's Algorithm and A*

## 7.1. Dijkstra

The worst-case time complexity depends on the graph's sparsity and the data structure to implement $Q$. For example, in dense graphs, $|E| = O(|V|^2)$, and since Dijkstra checks each edge twice, its worst-case time complexity is also $O(|V|^2)$.

However, if the graph is sparse, $|E|$ is not comparable to $|V|^2$. With a Fibonacci heap as $Q$, the time complexity becomes $O(|E| + |V| \log |V|)$.

Since $Q \cup S = V$, the space complexity is $O(|V|)$.

## 7.2. A*

The time and space complexities of GS A* are bounded from above by the state graphs' size. In the worst-case, A* will require $O(|V|)$ memory and have $O(|V| + |E|)$ time complexity, similar to Dijkstra and UCS. However, that's the worst-case scenario. In practice, the search trees of A* are smaller than those of Dijkstra and UCS. That's because **the heuristics usually prune large portions of the tree that Dijkstra would grow on the same problem**. For those reasons, A* focuses on the promising nodes in the frontier and finds the optimal path faster than Dijkstra or UCS.

The worst-case time of TLS A* is $O\left(b^{\left\lfloor \frac{C^*}{\varepsilon} \right\rfloor + 1}\right)$, where $b$ is the upper bound of the branching factor, $C^*$ is the optimal path's cost, and $\varepsilon > 0$ is the minimal edge cost. However, its effective complexity isn't as bad in practice because A* reaches fewer nodes. The same goes for the space complexity of TLS A*.

# 8. The Hierarchy of Algorithms

In a way, Dijkstra is an instance of A*. If we use a trivial heuristic that always returns 0, A* reduces to UCS. But, UCS is equivalent to Dijkstra in the sense that they have the same search trees. So, we can simulate Dijkstra with an A* that uses $h(\cdot) = 0$ as the heuristic.

## 10. Summary

So, here's the summary of Dijkstra vs. A*.

| | Dijkstra | A* |
|---|---|---|
| Graph | finite | both finite and infinite graphs |
| Edge costs | non-negative | strictly positive |
| Time complexity | $O(\|V\|^2)$ or $O(\|E\| + \|V\| \log \|V\|)$ | $O(\|E\| + \|V\|)$ or $O\left(b^{\left\lfloor \frac{C^*}{\varepsilon} \right\rfloor + 1}\right)$ |
| Space complexity | $O(\|V\|)$ | $O(\|V\|)$ or $O\left(b^{\left\lfloor \frac{C^*}{\varepsilon} \right\rfloor + 1}\right)$ |
| Search contours | uniform | stretched toward the goal(s) |
| In practice | slower than A* | fast if the heuristic is good |
| Computing | inside the algorithm | heuristics incur computational overhead |
| Input | require only the problem definition | designing a good heuristic requires time |

As we see, the efficiency of A* stems from the properties of the heuristic function used. So, implementations of A* require an additional step of designing a quality heuristic. **It has to guide the search efficiently through the graph but also be computationally lightweight.**

## 11. Conclusion

In this article, we talked about Dijkstra's Algorithm and A*. We presented their differences and explained why the latter is faster in practice.

# SOURCE CODE

# File_name: Shivam_DV_CA2.ipynb

```
# Dijkstra's Algorithm in Python


import sys

# Providing the graph
#declare vertices as given in graph

vertices = [
    [0, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
```

```
    [1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 1, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1],
    [0, 0, 0, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 1, 1],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 1, 1, 1, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1],
    [0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0]
]

#declate weight between vertices
edges = [
    [0, 1, 5, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [1, 0, 2, 4, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [5, 2, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 4, 0, 0, 4, 7, 0, 0, 0, 0, 7, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 4, 0, 3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 6],
    [0, 0, 0, 7, 3, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 1, 0, 0, 2, 0, 0, 0, 0, 3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [2, 0, 0, 0, 0, 0, 0, 0, 0, 9, 5, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 9, 0, 3, 0, 0, 6, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 5, 3, 0, 5, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 7, 0, 0, 3, 0, 0, 5, 0, 4, 3, 0, 0, 0, 0, 0, 0, 10, 8],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 4, 0, 0, 2, 10, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 6, 0, 3, 0, 0, 4, 0, 0, 7, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 4, 0, 0, 5, 0, 0, 0, 0, 0],
```

```
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 10, 0, 0, 0, 0, 8, 0, 0, 0, 4],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 5, 0, 0, 4, 3, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 7, 0, 8, 4, 0, 4, 2, 6, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 3, 4, 0, 1, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 4, 0, 3, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 10, 0, 0, 0, 0, 0, 0, 6, 0, 2, 0, 5],
    [0, 0, 0, 0, 6, 0, 0, 0, 0, 0, 0, 8, 0, 0, 0, 4, 0, 0, 0, 0, 5, 0]

]

#define a function to find the node with the smallest distance that has not been visited yet

# Find which vertex is to be visited next
def to_be_visited():
    global visited_and_distance
    v = -10
    #loop  through all nodes to find minimum distance
    for index in range(num_of_vertices):
        if visited_and_distance[index][0] == 0 and (v < 0 or visited_and_distance[index][1] <=
visited_and_distance[v][1]):
            v = index
    return v


#Implement Dijkstta's  algorithm
#get total number of vertices in graph
num_of_vertices = len(vertices[0])

#initialize distance and visited arrays
visited_and_distance = [[0, 0]]

#loop through all nodes to  find shortest path to each node
for i in range(num_of_vertices - 1):
    visited_and_distance.append([0, sys.maxsize])
#Loop  through all neighboring nodes of current_node
for vertex in range(num_of_vertices):

    # Find next vertex to be visited
```

```
    to_visit = to_be_visited()
    for neighbor_index in range(num_of_vertices):

        # Updating new distances
        if vertices[to_visit][neighbor_index] == 1 and \
                visited_and_distance[neighbor_index][0] == 0:
            new_distance = visited_and_distance[to_visit][1] \
                        + edges[to_visit][neighbor_index]
            if visited_and_distance[neighbor_index][1] > new_distance:
                visited_and_distance[neighbor_index][1] = new_distance

        visited_and_distance[to_visit][0] = 1
#display shortest path values from source vertex
i = 0

for distance in visited_and_distance:
    if(  (chr(ord('a') + i) in 'abcdefghjklmnpqrstuvw' )):
        print("Distance of ", chr(ord('a') + i)," from source vertex: ", distance[1])
        i=i+1
    else:
        i=i+1
        print("Distance of ", chr(ord('a') + i)," from source vertex: ", distance[1])
        #print('Mid:',i)
        i=i+1
```

OUTPUT:

```
Distance of  a  from source vertex:  0
Distance of  b  from source vertex:  1
Distance of  c  from source vertex:  3
Distance of  d  from source vertex:  5
Distance of  e  from source vertex:  9
Distance of  f  from source vertex:  6
Distance of  g  from source vertex:  4
Distance of  h  from source vertex:  2
Distance of  j  from source vertex:  10
Distance of  k  from source vertex:  7
Distance of  l  from source vertex:  7
Distance of  m  from source vertex:  11
Distance of  n  from source vertex:  10
Distance of  p  from source vertex:  13
Distance of  q  from source vertex:  19
```

```
Distance of  r  from source vertex:  18
Distance of  s  from source vertex:  17
Distance of  t  from source vertex:  21
Distance of  u  from source vertex:  19
Distance of  v  from source vertex:  17
Distance of  w  from source vertex:  15
```