Student: Bui Tu Ha
Student ID: 081751020

Application of Graph Theory in Route Search Algorithm for Route Guidance System in Automobiles

1 Route Guidance System

Route Guidance System is one of the major elements of travel and transportation management, which contributes to Intelligent Transportation System (ITS). Using maps, arrows, and/or a voice interface, Route Guidance System provide users with turn-by-turn guidance to a destination. Typical applications of Route Guidance Systems are car navigation system, delivery truck route planning system, fright route finder,...

Route Guidance Systems process route search based on Route Search Algorithm. After origin point, destination point and departure time are set, routes are searched in a road map (road network) according to predefined criteria (e.g. shortest travel time, shortest distance, minimal CO_2 emission, ...). The most common criterion is often the shortest travel time.

Route search problem is treated using the knowledge of Graph Theory. Road network is considered as plane graphs. It is expressed by nodes (vertices) and links (edges/arcs). Nodes indicate road intersections while links indicate road segment between two adjacent intersections. Another element is weight of link. It can also be regarded as impedance or cost of link, for example, travel time or travel distance. Weight of link is determined according to observation by road side infrastructure (e.g. loop-coil/ultrasonic vehicle detector, radio wave beacon), observation by probe vehicle system (e.g. taxi fleets), estimation based on traffic volume, as well as some other information. In addition, if links in road network have no restriction for moving direction, we use undirected graph. Then in many cases, weight on a link is the same in both direction. By contrast, if links have restriction for moving direction (e.g. vehicles must keep left), a directed graph is used. In this case, it is possible that weight becomes different when we move in opposite direction between two adjacent nodes.

2 Route Search Algorithm

Route Search Algorithm plays a key role in the application of Route Guidance System. Why is it necessary? When the size of road network increases, the number of routes also increases. More specifically, in case of a grid network having x links along horizontal side and y links along vertical sides (x and y are positive integer) , the number of paths between the origin and the destination as shown in the Figure 1 becomes $n = (x + y)!/(x!y!)$. This number can be large, for instance, $n = 70$ if $(x, y) = (4, 4)$; $n = 126$ if $(x, y) = (4, 5)$. Thus, an efficient Route Search Algorithm is necessary.

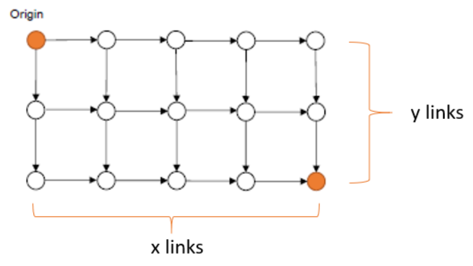


Figure 1: A grid network

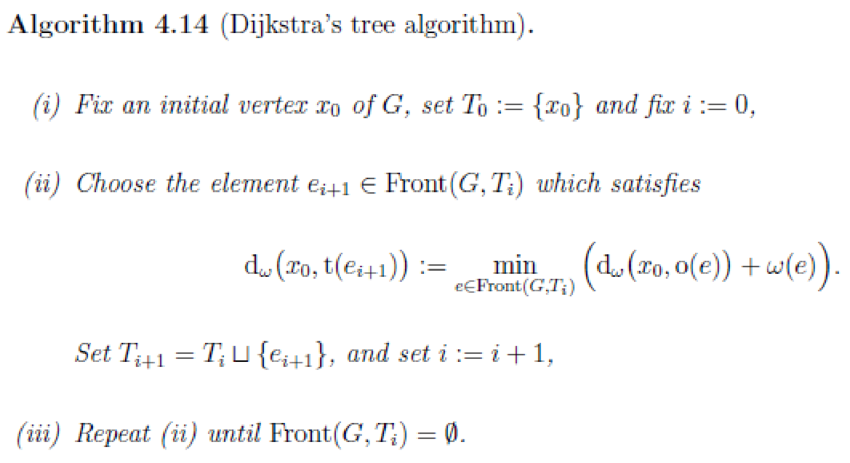
Route Search Algorithm is often used to solve shortest path problem, i.e. to find a route (path) between to locations with minimized total weight. There are three main types of shortest path problem:

- Single-Source Shortest Path problem: to find the shortest routes from a source node to all other nodes in the network (typical algorithm: Dijkstra algorithm, Bellman-Ford algorithm)
- Single-Pair Shortest Path problem: to find the shortest route between two predetermined locations (typical algorithm: A* algorithm). Every known algorithm takes just as long as solving Single-Source.
- All-Pairs Shortest Path problem: to find the shortest routes between every pair of nodes in the network (typical algorithm: Warshall-Floyd algorithm).

The use of Warshall-Floyd algorithm involves matrix calculation and it is difficult to apply this to a large road network. Thus, we do not usually consider this algorithm. In the next part, we shall recall the Dijkstra algorithm (see Section **??**), then discuss about A* algorithm, which is a revised method of Dijkstra's algorithm. After that, Heap Data Structure will be introduced as a method to mitigate a weakness of Dijkstra algorithm and A* algorithm. In the end, we will study Bellman-Ford Algorithm and compare it with Dijkstra algorithm.

2.1 Dijkstra's Algorithm

Dijkstra's algorithm can be perceived as the most popular and frequently used route search algorithm for Route Guidance System. To implement this algorithm, we need to assume non-negative edge weight. Recall Algorithm **??** in Section **??**:



Algorithm 4.14 (Dijkstra's tree algorithm).

(i) Fix an initial vertex x_0 *of* G*, set* $T_0 := \{x_0\}$ *and fix* $i := 0$,

(ii) Choose the element $e_{i+1} \in \mathrm{Front}(G, T_i)$ *which satisfies*

$$\mathrm{d}_\omega(x_0, \mathrm{t}(e_{i+1})) := \min_{e \in \mathrm{Front}(G,T_i)} \Big(\mathrm{d}_\omega(x_0, \mathrm{o}(e)) + \omega(e)\Big).$$

Set $T_{i+1} = T_i \sqcup \{e_{i+1}\}$*, and set* $i := i + 1$,

(iii) Repeat (ii) until $\mathrm{Front}(G, T_i) = \emptyset$.

Figure 2: Algorithm **??**

With this expression, for each discovery number i ($i \geq 0$), we need to update the set of frontier edges, $Front(G, T_i)$. From the viewpoint of programming, we shall use an one dimensional array to store the set of non-tree endpoints of the frontier edges. Choosing an element in this node set can be an alternative of "Choose the element $e_{i+1} \in Front(G, T_i)$" in Algorithm **??**.

Overall, for shortest path problem using Dijkstra algorithm, it is more convenient to classify nodes into three node sets: V(Visited), T (Tentative) and U (Unvisited).

Nodes in set V are the nodes that have been visited with fixed distance and route from origin. The set T is comprised of nodes that have been visited but distance and route from origin have not been fixed. Accordingly, this set provides the information about frontier edges. The last node set is the set U, which contains nodes that have not been visited and distance and route from origin are unknown.

* Remark: The term "distance" mentioned above implies the value of function d_w in Algorithm 4.14. In other cases, d_w can be regarded as a function that indicates travel time or travel cost from origin.

Based on the definition of set V, T and U, we can express the Dijkstra's algorithm in another way which have underlying meaning similar to the Algorithm **??**.

Step 0:

- Assign a beginning value of distance (infinity) to all nodes
- All nodes are in set U

Step 1:

- Set the distance of the origin node to zero
- Origin node is moved from set U to set V
- Set the origin node as "base node"

Step 2:

- For the based node, consider all of its unvisited or tentative neighbors and calculate their tentative distance via the current base node.
- For each of these neighbors, compare the tentative distance newly calculated to the tentative distance calculated before.
- If the newly calculated distance is smaller, the tentative distance is updated and the upstream node is changed to the current base node.
- Newly calculated neighbors are moved from set U to set T

Step 3:

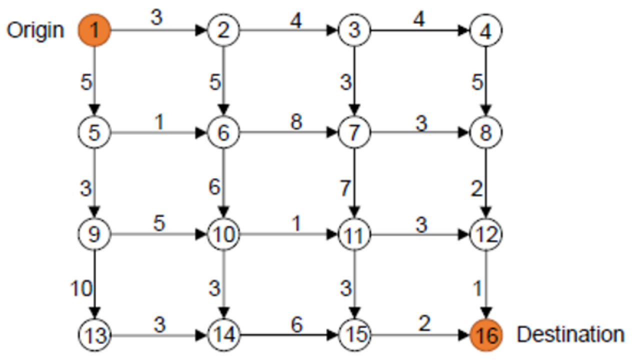
- From set T, the node with the smallest tentative distance is extracted and set as new base node.
- This new base node is moved from set T to set V.

Step 4:

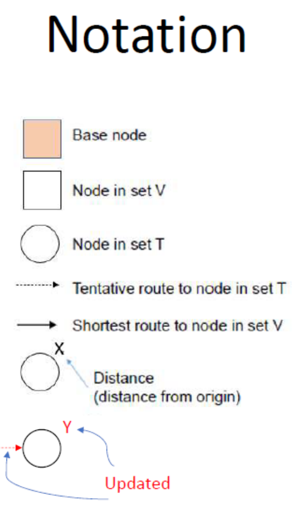
- If the base node is the destination node, then stop. The algorithm has finished.
- Otherwise, go to step 2.

Next, let's practice with an example of finding shortest path using Dijkstra's algorithm.

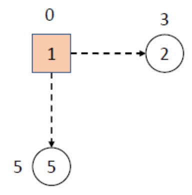
Exercise: Search the shortest path from the origin to the destination using Dijkstra's algorithm.



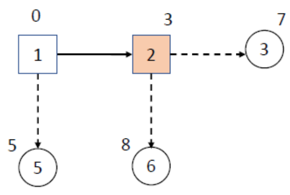
The step-by-step solution is described below:



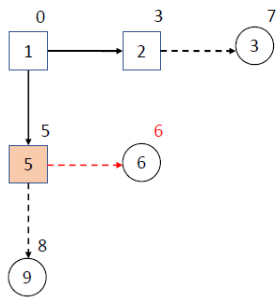
1.



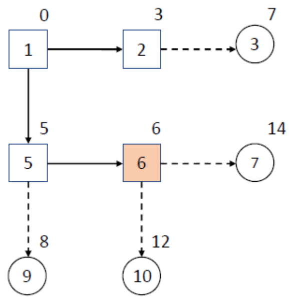
2.



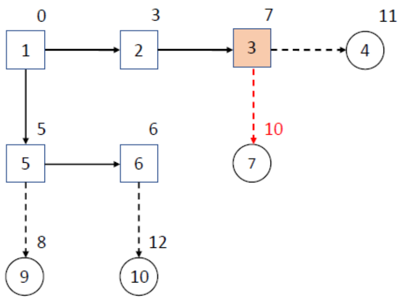
3.



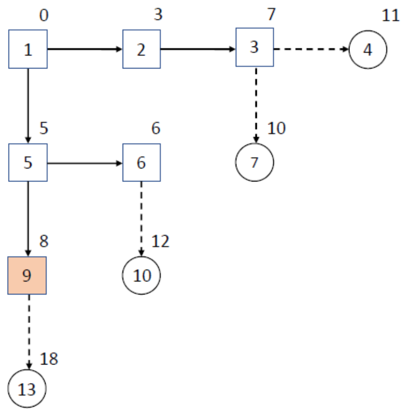
4.



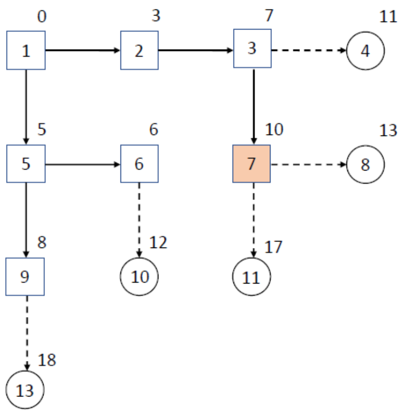
5.



6.



7.



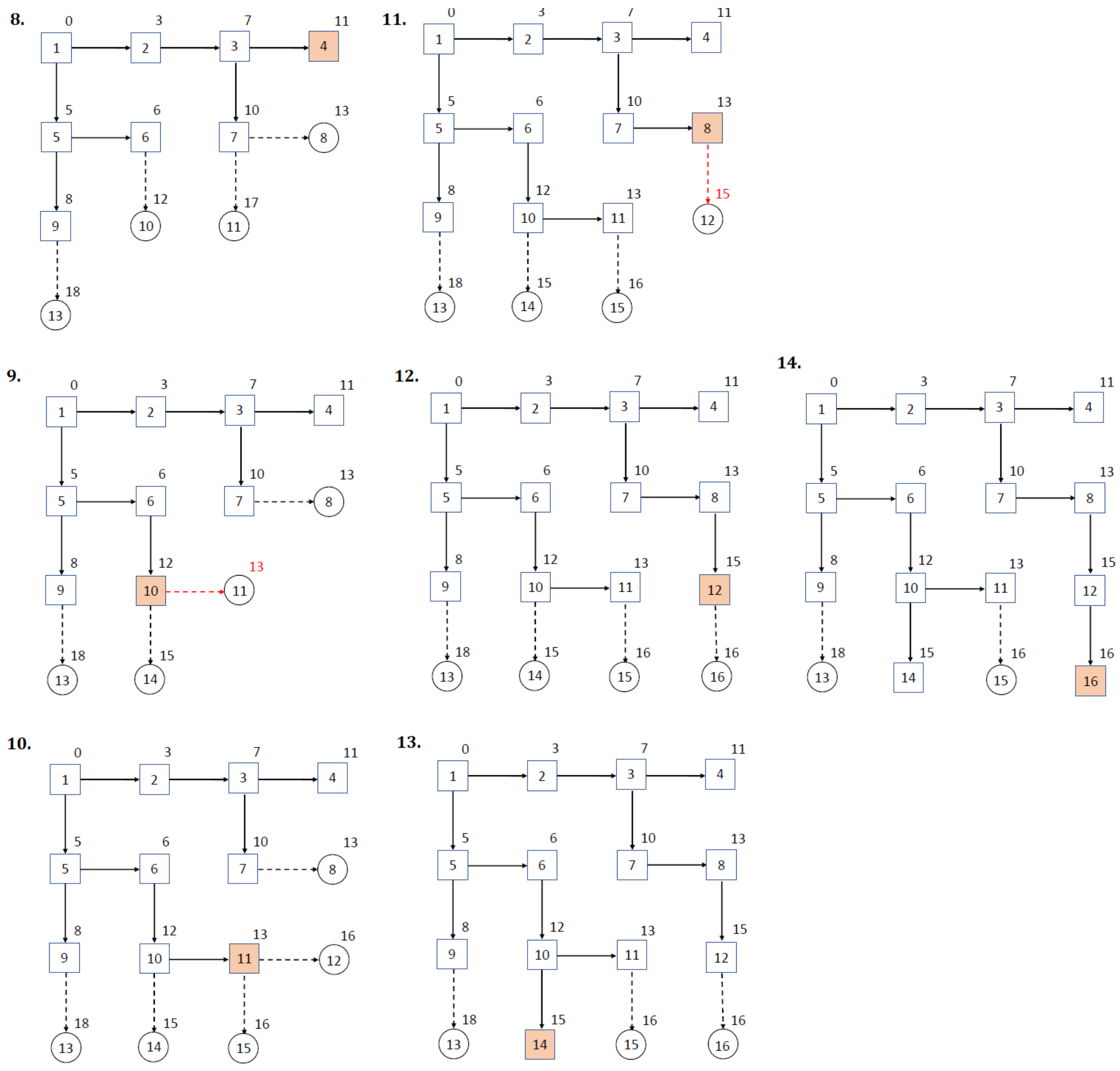


Figure 3: An example of using Dijkstra's algorithm to solve the shortest path problem

2.2 A* Algorithm

A* algorithm is proposed in 1968 as an improved algorithm of Dijkstra algorithm. [1] It is an improved method based on Dijkstra algorithm, with modification only in the step 2 of Dijkstra algorithm.

Let n be a node in set T, or equivalently, the non-tree endpoint of a frontier edge:
Dijkstra's algorithm:

$$\text{Tentative distance } D(n) = \text{Tentative distance from origin node}$$

A* algorithm:

$$\text{Tentative distance } D'(n) = D(n) + \text{Heuristic } h(n)$$

$h(n)$ is the estimation of the minimum distance from node n to destination node. It must be positive and smaller than the true minimum distance to the destination node. Practically, $h(n)$ is the length of the straight line from node n to destination node using X-Y coordinate.

2.3 Heap Data Structure

In the practical case of large road network, the number of nodes in set T can reach several thousands or more. However, in Dijkstra algorithm and A* algorithm, we only need to find one node with the smallest tentative distance (step 3) and the process to search this node consumes the most amount of processing time. To tackle this weakness of Dijkstra algorithm and A* algorithm, we shall use an implementation of priority queue - Heap Data Structure (Binary Heap). In fact, Heap Data Structure can be perceived as a priority tree (see Definition **??**) of total ordered set with priorities (elements which label vertices) that are numeric values . Heap Data Structure efficiently searches the node with the minimal tentative distance. Based on the tentative distance values, nodes in set T are ordered in a specific way.

In computer science, a **heap** is a specialized tree-based data structure which is essentially an almost complete binary tree that satisfies the heap property:

- In a max heap, for any given node C (child node), if P is a parent node of C, then the key (the value) of P is greater than or equal to the key of C.
- In a min heap, the key of P is less than or equal to the key of C. The node at the "top" of the heap (with no parents) is the root node.

For Dijkstra's algorithm and A* algorithm, we shall use the property of min heap as heap condition.

Heap condition: Tentative distance of parent node is less than or equal to tentative distance of each child node.

When a node is added to set T (and distance of some tentative nodes is updated), heap is updated as follows:
(1) Insertion of a new node

- (1)-1: Add the new node to the last position of the bottom level

[1]Hart P.E., Nilson N.J. and Raphael B. (July 1968), "A Formal Basis for the Heuristic Determination of Minimal Cost Paths". IEEE Transaction on Systems Science and Cybernetics 4 (2): 100-107

- (1)-2: Compare the added element with its parent. If they are in the correct order (according the heap condition), then stop. If not, swap the node with its parent. (Note that the number of swap is at most the height of the tree ($log_2|V|$) and this takes $O(log(|V|))$ time.

  Step (1)-2 is repeated until the structure satisfies the heap condition.

(2) Deletion of root node, which is the node with the smallest tentative distance. In addition, the removed node will be the next base node.

(3) Bubble-down operation is performed to restore the structure

- (3)-1: Move that last node on the bottom level to the root of the heap
- (3)-2: Compare the new root with its children. If they are in the correct order, stop. If not, swap it with its smaller child. Note that the number of swap is at most the height of the tree ($log_2|V|$) and this takes $O(log(|V|))$ time.

  Step (3)-2 is repeated until the structure satisfies the heap condition.

* Remark: How to find the target node for comparison (i.e to find the parent node or a child node) ?

Binary heap can be memorized by using a single dimensional array. The tree nodes have a natural ordering: row by row (starting from the root node) and moving left to right within each row. If there are n nodes, this ordering specifies their positions $1, 2, ..., n$ in the array. Moving up an down the tree is easily simulated on the array, using that node number j has parent $\lfloor j/2 \rfloor$, and children $2j$ and $2j + 1$.

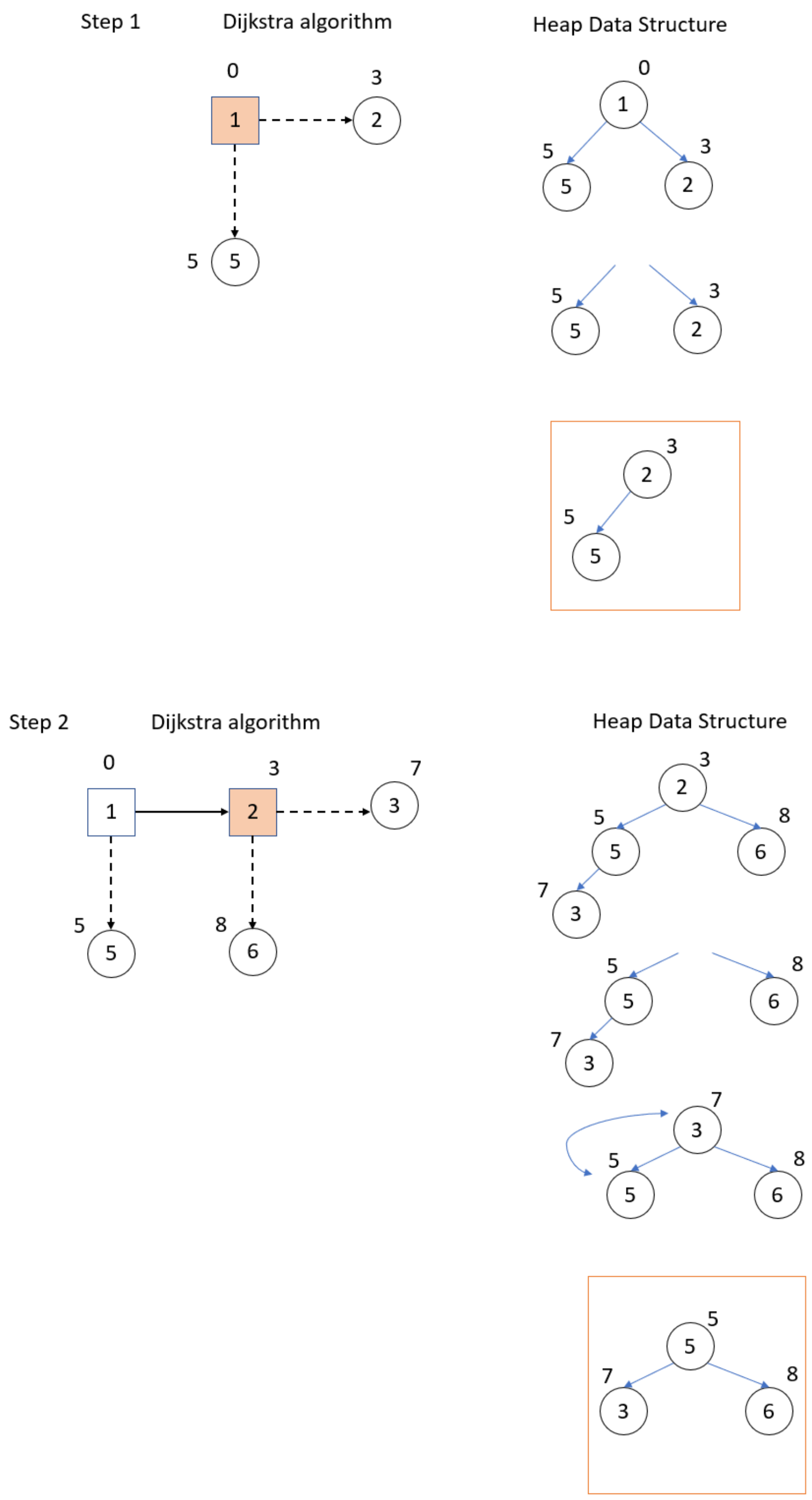


Figure 4: Examples of binary heap which corresponds to calculation step 1 and step 2 in the mentioned example of finding the shortest path using Dijkstra algorithm.

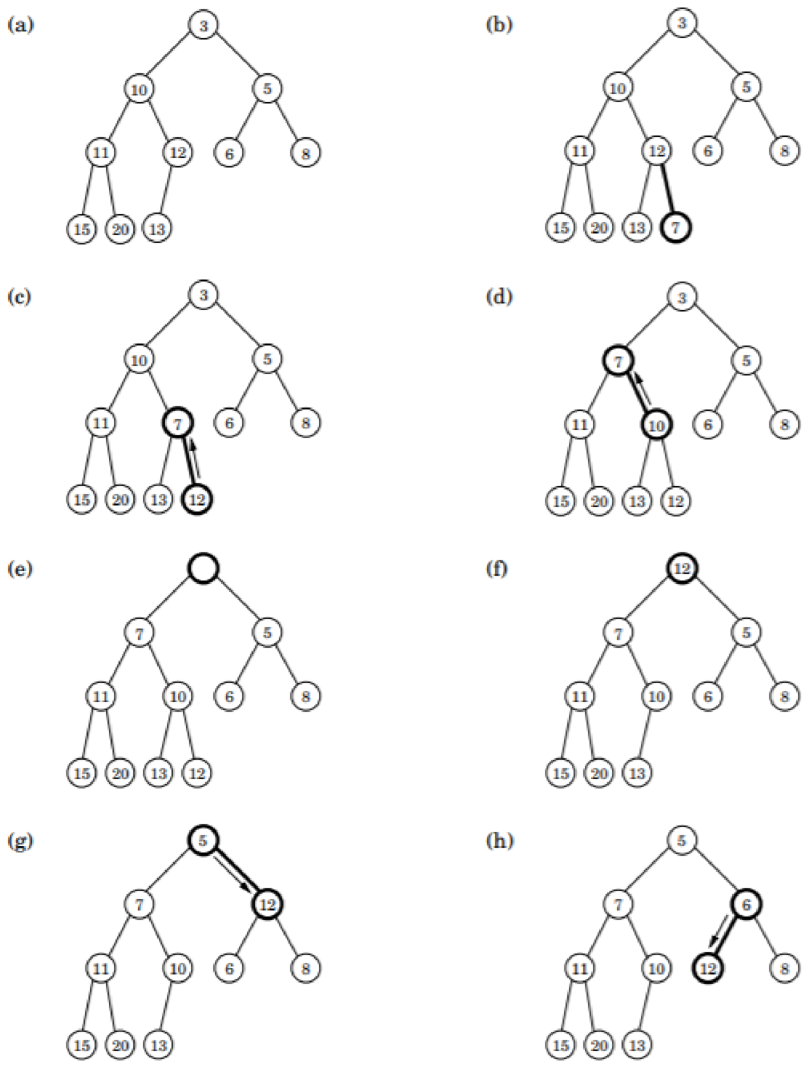


Figure 5: (a) A binary heap with 10 elements. Only the key values are shown. (b)-(d) The intermediate "bubble-up" steps in inserting an element with key 7. (e)-(g) The "sift-down" steps in a delete-min operation. [5]

*

2.4 Bellman-Ford Algorithm

Beside Dijkstra's algorithm, Bellman-Ford (BF) algorithm is another Single-Source Shortest Path (SSSP) algorithm. However, BF algorithm is not ideal for most SSSP problems because of its high time complexity, which is $O(|V|.|E|)$. Meanwhile, Dijkstra's algorithm using binary heap is much faster with time complexity $O((|V| + |E|)log|V|)$.

Nonetheless, Dijkstra's algorithm has one disadvantage: it can fail when negative link weights exist. This problem can be mitigated by the use of BF algorithm, which can treat negative value of link weight. However, we still have to assume no cycle of negative length for BF algorithm. After understanding how BF algorithm works, you may find out that it is because a cycle of negative length possibly leads to infinite reduction of distance (to $-\infty$) and consequently, some vertices can not be visited.

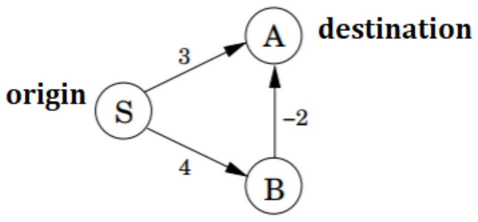
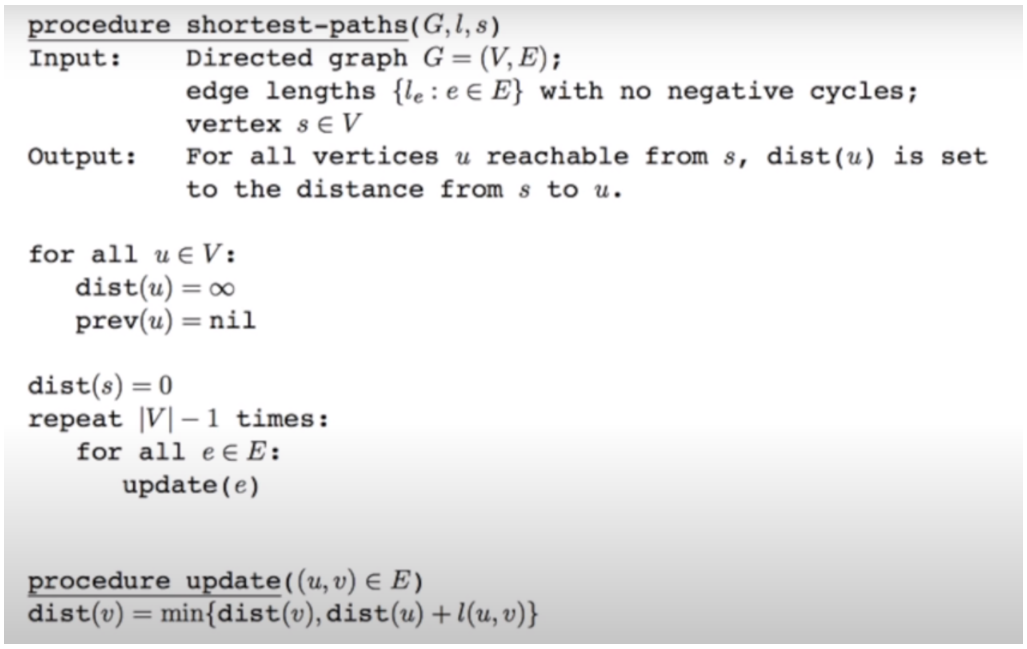


Figure 6: One example that Dijkstra's algorithm fails when negative link weights exist.

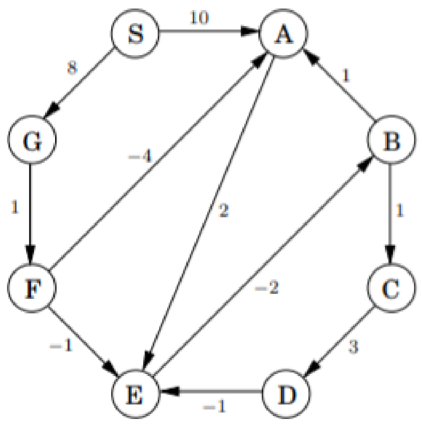


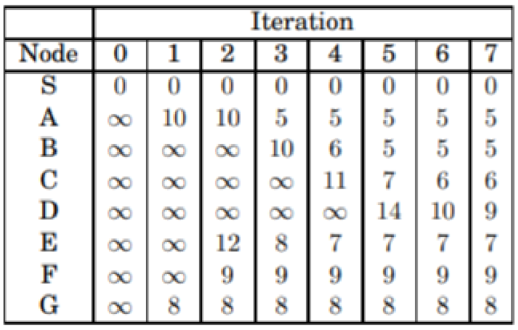
```
procedure shortest-paths($G, l, s$)
Input:     Directed graph $G = (V, E)$;
           edge lengths $\{l_e : e \in E\}$ with no negative cycles;
           vertex $s \in V$
Output:    For all vertices $u$ reachable from $s$, dist($u$) is set
           to the distance from $s$ to $u$.

for all $u \in V$:
    dist($u$) $= \infty$
    prev($u$) $=$ nil

dist($s$) $= 0$
repeat $|V| - 1$ times:
    for all $e \in E$:
        update($e$)


procedure update($(u, v) \in E$)
dist($v$) $= \min\{$dist($v$), dist($u$) $+ l(u, v)\}$
```

Figure 7: The Bellman-Ford Algorithm for Single-Source Shortest Path [5]

Firstly, distance from s to each vertex u is set to infinity. When the total number of vertices in G is $|V|$, we do (at most [2]) $|V| - 1$ iterations and each iteration will update along all edges. An example of Bellman-Ford algorithm is shown in Figure 8.

[2]We sometimes make the algorithm stop earlier if nothing improves with more iterations.





	Iteration							
Node	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
S	0	0	0	0	0	0	0	0
A	∞	10	10	5	5	5	5	5
B	∞	∞	∞	10	6	5	5	5
C	∞	∞	∞	∞	11	7	6	6
D	∞	∞	∞	∞	∞	14	10	9
E	∞	∞	12	8	7	7	7	7
F	∞	∞	9	9	9	9	9	9
G	∞	8	8	8	8	8	8	8

Figure 8: The Bellman-Ford Algorithm illustrated on a sample graph [5]

References

[1] Tomio Miwa (2020), *"Route Search Method"*, Intelligent Transportation System Lecture, Nagoya University.

[2] CORMEN, THOMAS H. (2009). INTRODUCTION TO ALGORITHMS. United States of America: The MIT Press Cambridge, Massachusetts London, England. pp. 151–152. ISBN 978-0-262-03384-8.

[3] Black (ed.), Paul E. (2004-12-14). Entry for heap in Dictionary of Algorithms and Data Structures. Online version. U.S. National Institute of Standards and Technology, 14 December 2004. Retrieved on 2017-10-08 from https://xlinux.nist.gov/dads/HTML/heap.html

[4] David Walden (January 2008), THE BELLMAN-FORD ALGORITHM AND "DISTRIBUTED BELLMAN-FORD

https://www.researchgate.net/publication/250014977_THE_BELLMAN-FORD_ALGORITHM_AND_DISTRIBUTED_BELLMAN-FORD

[5] https://code.google.com/archive/p/eclipselu/downloads