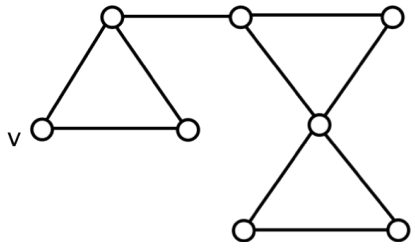
## Algorithmics I - Tutorial Sheet 2

## Graphs and graph algorithms

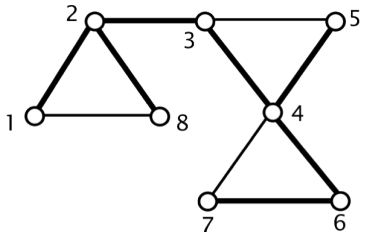
---

1. Construct ($a$) a depth-first spanning tree and ($b$) a breadth-first spanning tree of the graph shown below.

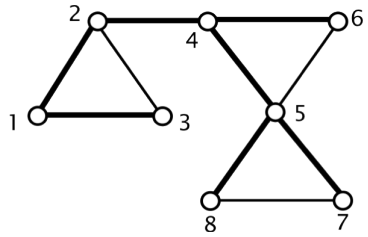



In each case, start the search/travel at vertex $v$.

**Solution:** A depth-first spanning tree:




A breadth-first spanning tree:




In each case, the numbers beside each vertex indicate the order in which the vertices are visited. Note that in both cases other solutions are possible depending on the order vertices are visited.

2. Recall, a *Eulerian cycle* in an undirected graph is a cycle that includes every edge of the graph exactly once. It can be shown that a graph $G$ contains an Eulerian cycle if and only if $G$ is connected and every vertex has even degree (i.e. is adjacent to an even number of vertices). Such a graph is called an *Eulerian graph*.

   Describe how depth-first search can be adapted to find an Eulerian cycle in an Eulerian graph.

(A *Hamiltonian cycle* in an undirected graph is a cycle that includes every vertex of the graph exactly once. A graph containing a Hamiltonian cycle is called a Hamiltonian graph. Despite the superficial similarity of this concept to that of an Eulerian graph, there is no known efficient algorithm to determine whether a given graph is Hamiltonian.)

**Solution:** Conduct a type of depth first search, by following successive unused edges from an arbitrary starting vertex $v$, until this cannot be continued. The path must have returned to the starting vertex $v$, since:

- there are an even number of edges adjacent to each vertex;
- each time we pass through any other vertex we remove two of the edges adjacent to the vertex (one when reaching the vertex and one when leaving the vertex);

imply that whenever we enter any other vertex there is always 'free' edge to leave on.

If this has not used up all the edges, then consider the first vertex $w$ on the path with an unused edge. Start from $w$ and traverse another path in the same way (removing more edges from the graph), then splice these paths into one to create a longer path starting and ending at $v$. Continue in this way until there are no edges remaining.

3. Describe in detail how depth-first search can be used to check for deadlock, i.e. to determine whether a given directed graph, represented by adjacency lists, contains a cycle. What is the complexity of your algorithm?

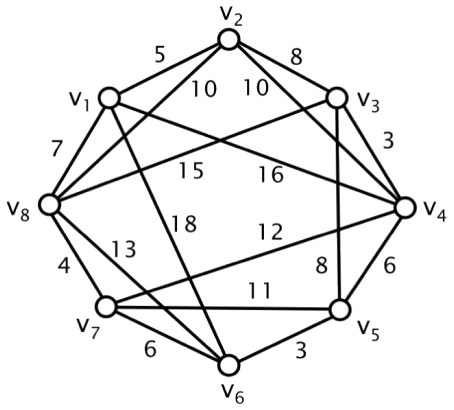
**Solution:** Care is needed in applying a depth-first search approach in a directed graph. As the search proceeds, it is essential to keep track, perhaps through a Boolean attribute in the Vertex class, of exactly which vertices are on the *current path* from the starting vertex. When a visited vertex is encountered on the adjacency list of the current vertex and it is on the current path, then there is a cycle in the graph. (It is certainly not sufficient that the vertex should have been visited already.)

The complexity is the same as for depth first search.

4. How would you extend Dijkstra's shortest paths algorithm so that, in addition to determining the length of the shortest path from a vertex u to each of the other vertices, it also collects enough information to allow actual the shortest paths to be constructed?

**Solution:** All that is required is the inclusion of an array *pred* where $pred(w)$ records the predecessor of vertex $w$ on the shortest path from $u$. Given this array, it is straightforward to construct all shortest paths. To build the array, whenever we change the value of $\mathtt{d}(w)$ using the vertex $v$, we change the value of $pred(w)$ to $v$ (see pseudocode in lecture handout).

5. Find the shortest paths, and their lengths, from vertex $v_1$ to each of the other vertices in the graph shown below.



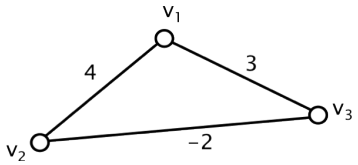

**Solution:** These following distances are found, by Dijkstra's algorithm, in increasing order of distance from $v_1$, namely $v_2$, $v_8$, $v_7$, $v_3$, $v_4$, $v_6$ and $v_5$.

| vertex | shortest path | length |
|---|---|---|
| $v_2$ | $v_1 \to v_2$ | 5 |
| $v_3$ | $v_1 \to v_2 \to v_3$ | 13 |
| $v_4$ | $v_1 \to v_2 \to v_4$ | 15 |
| $v_5$ | $v_1 \to v_8 \to v_7 \to v_6 \to v_5$ | 20 |
| $v_6$ | $v_1 \to v_8 \to v_7 \to v_6$ | 17 |
| $v_7$ | $v_1 \to v_8 \to v_7$ | 11 |
| $v_8$ | $v_1 \to v_8$ | 7 |

6. In some applications, it is appropriate to allow the weight of an edge in a graph to be negative. However, a cycle of negative weight rarely makes sense, because a path of arbitrarily large negative weight can be created by repeatedly traversing the cycle.

   Suppose $G$ is a weighted graph with arbitrary (positive or negative) integer weights, but with no negative weight cycles. Can Dijkstra's algorithm be used to find shortest paths in such a graph? If so give a proof, if not give an illustrative example.

**Solution:** Consider the graph $G$ with a negative edge weight as shown below:




If we assume that the source vertex is $v_1$, Dijkstra's algorithm will set $\mathtt{d}(v_2)$ to be 4 and $\mathtt{d}(v_3)$ to be 3 initially. This implies that $v_3$ is then added to $S$, so that $\mathtt{d}(v_3)$ does not subsequently change. However the correct value for $\mathtt{d}(v_3)$ is clearly 2, which results from the path $v_1 \to v_2 \to v_3$.

7. Suppose that the cost of a path in a weighted undirected graph is defined to be the largest weight of an edge in the path. Design an algorithm, similar to Dijkstra's algorithm, to find the path of least cost from one fixed vertex to all of the other vertices in a graph.

**Solution:** The algorithm is similar to Dijkstra's algorithm for the shortest paths. But during the updating phase, the current minimum cost value $c(w)$ for for each unsolved vertex w is given by:

$$\mathtt{c}(w) = \min\{\mathtt{c}(w), \max\{\mathtt{c}(v), \mathtt{wt}(v,w)\}\}$$

In other words, $\mathtt{c}(w)$ can be reduced if there is a better path that uses the vertex $v$ just solved and this will be true if the larger of the quantities $\mathtt{c}(v)$ and $\mathtt{wt}(v,w)$ is smaller than the current value of $\mathtt{c}(w)$. See the tutorial notes for further details.

8. Suppose you are given an arbitrary directed acyclic graph $G = (V, E)$. By using a topological ordering of $G$, describe an $\mathcal{O}(m+n)$ algorithm for finding the length of a longest path in $G$, where $n = |V|$ and $m = |E|$.

   **Hint:** use the Topological ordering to construct a recurrence relation for the longest path to each vertex. For example, in the base case, consider the longest path to each source vertex (i.e. vertices with indegree 0).

**Solution:** Construct a topological ordering $v_1, v_2, \ldots, v_n$ of $G$, which as we have seen takes $\mathcal{O}(m+n)$ time. For $1 \leq j \leq n$, let $lp(v_j)$ denote the length of a longest path terminating at vertex $v_j$. Given that $v_1, v_2, \ldots, v_n$ is a topological ordering, the following recurrence relation may be used in order to calculate $lp(v_i)$ for $1 \leq j \leq n$ in turn. More precisely we set:

$$lp(v_i) = \begin{cases} 0 & \text{if } v_i \text{ is a source vertex} \\ 1 + \max\{lp(v_j) \,|\, (v_j, v_i) \in E\} & \text{otherwise} \end{cases}$$

where a source vertex is a vertex with indegree 0.

It follows from the definition of a topological ordering that we compute these values when follow the topological ordering to 'processing' vertices, i.e. we compute the values in the order $lp(v_1)$, $lp(v_2)$, $\ldots$, $lp(v_n)$.

The length of a longest path in $G$ is then given by:

$$\max\{lp(v_j) \,|\, 1 \leq j \leq n\}\,.$$

Since the computation of

- the maximum has complexity $\mathcal{O}(n)$;
- computing $lp(v_j)$ for all vertices has complexity $\mathcal{O}(m+n)$;
- computing the topological ordering has complexity $\mathcal{O}(m+n)$;

the overall algorithm has complexity $\mathcal{O}(m+n)$ as required.