Exercise IX, Algorithms 2024-2025

These exercises are for your own benefit. Feel free to collaborate and share your answers with other students. There are many problems on this set, solve as many as you can and ask for help if you get stuck for too long. Problems marked * are more difficult but also more fun :).

More on maxflows

1 ($\star$, Radar evasion) The People's Republic of Isolatia is a perfectly rectangular country. It shares its entire west border with the Republic of Westilia and its entire east border with the Republic of Eastilia. Westilia would like to send aircraft to provide emergency aid to Eastilia. The only way of reaching Eastilia is by flying through Isolatia's airspace (dangerous uncharted territory borders Isolatia on the south and north!), but Isolatia prohibits other countries from flying in its airspace. Therefore, Isolatia has deployed radar installations throughout the country. Each of the n radar sites is specified by its coordinates and the radius of its range. Westilia plans to send saboteurs to Isolatia to disable some radar stations in order to allow it to fly planes from the west border to the east border of Isolatia without being detected by radar.

A flight path is a continuous curve entirely contained in the airspace of Isolatia that connects a point on the west border to a point on the east border. You can use the fact that there exists a flight path if and only if there is no sequence of radars $r_1, r_2, \ldots, r_k$ such that the range of r_1 overlaps the north border, the range of r_i overlaps the range of r_{i+1} for all $i = 1 \ldots k-1$ and the range of r_k overlaps the south border.

Give an efficient algorithm for determining the smallest number of radar sites to be disabled to establish a flight path from Westilia to Eastilia that is outside the range of any radar station.

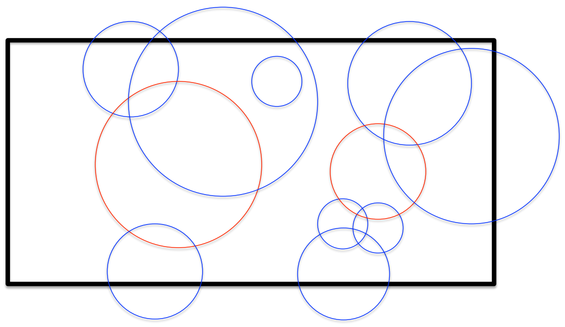


Figure 1. An example of radars in a rectangular region. Removal of the two radars indicated in red provides a valid flight path between the east and west boundaries.

Solution: We first reduce the problem to finding a minimum **vertex** s-t cut in an undirected graph. We then show how to find the minimum vertex cut using a maximum flow algorithm on a modified graph. The value of the maximum flow in the graph obtained in our reduction is no more than n, and the number of edges in the graph is $O(n^2)$. Thus, the runtime is $O(n^3)$ using the Ford-Fulkerson algorithm for maximum flow.

We first construct a graph $G = (V, E)$ such that our problem reduces to the minimum **vertex** $s-t$ cut problem in G. We then reduce the vertex $s-t$ cut problem to the edge $s-t$ cut problem in a graph G' derived from G, and solve the minimum $s-t$ cut problem in G' using maximum flow algorithms.

The graph G is constructed as follows. The vertex set V contains a vertex c_i for each circle $C_i, i = 1, \ldots, N$, as well as a vertex s (source) corresponding to the north border, and a vertex t (sink) corresponding to the south border. Connect c_i to c_j iff C_i overlaps with C_j, $i, j = 1, \ldots, N$. Connect s with $C_i, i = 1, \ldots, N$ iff C_i intersects the north border, connect t with $C_i, i = 1, \ldots, N$ iff C_i intersects the south border.

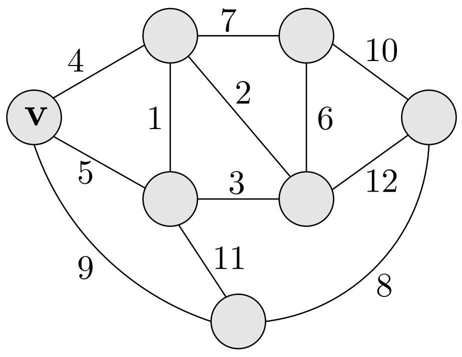
We will use the following fact given in the statement of the problem: there exists a flight path if and only if there is not sequence of circles $C_1, C_2, \ldots, C_k$ such that C_1 overlaps the north border, C_i overlaps C_{i+1} for all $i = 1 \ldots k-1$ and C_k overlaps the south border. This means that we need to find the smallest set of circles S such that removal of these circles disconnects the source from the sink, i.e. we are interested in the minimum $s-t$ vertex cut in the graph G.

We can solve the minimum vertex cut problem in G by solving the minimum edge cut problem in the graph G' obtained from G as follows. First replace each node u except for the source and the sink by a pair of nodes $\{u^{in}, u^{out}\}$, and add an edge (u^{in}, u^{out}) of capacity 1. For each edge $\{u, v\} \in E$ of the graph G add directed edges $(u^{out}, v^{in}), (v^{out}, u^{in})$ of capacity $+\infty$. For each edge $\{s, u\} \in E$ in G add an edge (s, u^{in}) of capacity $+\infty$, and for each edge $\{u, t\} \in E$ in G add an edge (u^{out}, t) of capacity $+\infty$.

Now note that there is a one to one correspondence between $s-t$ paths in G and $s-t$ paths in G', and vertices u of G are mapped to edges (u^{in}, u^{out}) in G'. Thus, the minimum number of vertices that one needs to delete to disconnect s from t in G is equal to the minimum weight of edges that one needs to delete to disconnect s from t in G'. The latter quantity is the minimum $s-t$ cut in G', so it can be computed using max-flow covered in class. The value of the maximum flow is at most n, and the number of edges in the graph is $O(n^2)$. Thus, the runtime is $O(n^3)$ using the Ford-Fulkerson algorithm for maximum flow.

Minimum spanning trees

2 (Exam question 2012 worth 20 pts) Consider the following minimum spanning tree instance, i.e., an undirected connected graph with weights on the edges.



2a (5 pts) Write the weights of the edges of the minimum spanning tree in the order they are added by Prim's algorithm starting from vertex v.

Solution: 4,1,2,6,9,8.

2b (5 pts) Write the weights of the edges of the minimum spanning tree in the order they are added by Kruskal's algorithm.

Solution: 1,2,4,6,8,9.

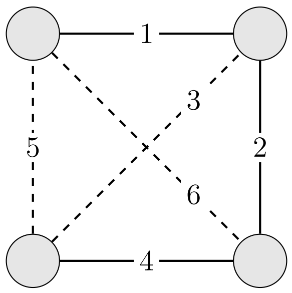
2c (*,10 pts) A bottleneck edge is an edge of highest weight in a spanning tree. A spanning tree is a minimum bottleneck spanning tree if the graph does not contain a spanning tree with a smaller bottleneck edge weight. *Show that a minimum spanning tree is also a minimum bottleneck spanning tree.*

Solution: Suppose that there exists a minimum spanning tree T that is not a minimum bottleneck spanning tree. Identify the edge e in T of maximum weight and remove it. This leads to a graph with two components and there must exist an edge e' that connects them with $w(e') < w(e)$ (take an edge of a minimum bottleneck spanning tree crossing the cut defined by the two components).
Adding e' yields a new spanning tree of lower cost which contradicts that we started with a minimum spanning tree T .

3 *(*, Exam question 2013 worth 10 pts)* Consider three undirected edge-weighted connected graphs $G_1 = (V, E_1), G_2 = (V, E_2)$, and $H = (V, E_1 \cup E_2)$ with non-negative weights $w : E_1 \cup E_2 \to \mathbb{R}_+$ on the edges. Note that they are all graphs on the same vertex set but their edges differ: G_1 has only the edges in E_1, G_2 has only the edges in E_2, and H has all the edges $(E_1 \cup E_2)$.

Let T, T_1, T_2 be minimum spanning trees of H, G_1, and G_2, respectively. Assuming that the weights of the edges are unique, i.e., no two edges have the same weight, *prove that* $T \subseteq T_1 \cup T_2$.

For an example of the statement see the figure below. The solid edges are E_1 and the dashed edges are E_2. Note that the minimum spanning tree of G_1 is $T_1 = \{1, 2, 4\}$, the minimum spanning tree of G_2 is $T_2 = \{3, 5, 6\}$, and the minimum spanning tree of H is $T = \{1, 2, 3\}$. We have thus that $T \subseteq T_1 \cup T_2$ in this case. You should prove that it holds in general.



Solution:

Let e be an arbitrary edge in T. Consider the cut defined by the two connected components of $T \setminus \{e\}$. Then e is the minimum-cost edge in this cut. (*Proof.* Suppose $e' \neq e$ is the minimum-cost edge in the cut (A, B). Then $T \setminus \{e\} \cup \{e'\}$ is a spanning tree of strictly better cost, contradicting the assumption that T is a minimum spanning tree.)

Since $e \in E_1 \cup E_2$, $e \in E_i$ for some $i \in \{1, 2\}$, and e is the minimum-cost edge in (A, B) in G_i; thus, $e \in T_i$ from the cut property.

Alternative proof: Suppose that we run Kruskal's algorithm on H, G_1, G_2 to find T, T_1, T_2. Let $m = |E_1 \cup E_2|$.

Suppose toward contradiction that there exists an edge that Kruskal's algorithm adds to T but not to T_1 or T_2. Let $e = \{u, v\}$ be the first such edge. Suppose that $e \in E_1$ (the case $e \in E_2$ is symmetric). If Kruskal's algorithm don't add e to T_1, T_1 already contains a path between u and v that consists of edges of strictly smaller weight than e. However, as Kruskal's algorithm greedily adds edges of smallest edge first, we have that there must be a path between u and v in T as well, which contradicts that e is added to T.

Alternative proof: Consider the edges in $E_1 \cup E_2$. Let $m = |E_1 \cup E_2|$ and suppose that they are ordered $e_1, e_2, \ldots, e_m$ such that $w(e_1) < w(e_2) < \cdots < w(e_m)$. Now note that if we run Kruskal's algorithm on H then an edge $e_i = \{u, v\}$ is added to the tree T if and only if the vertices u and v are in different components in the graph with edges $E_{<i} = \{e_1, e_2, \ldots, e_{i-1}\}$. Therefore if e_i is added to the tree T of H it is also clearly added by Kruskal's algorithm to the tree T_1 of G_1 if $e_i \in E_1$ or to the tree T_2 of G_2 if $e_i \in E_2$. To see this note that since u and v are in different components in the graph with edges $E_{<i}$, they are also in different components in the graph with edges $E_{<i} \cap E_1$ and in the graph with edges $E_{<i} \cap E_2$.

4 (Exercise 23.2-1) Let (u, v) be a minimum-weight edge in a connected graph G. Show that (u, v) belongs to some minimum spanning tree of G.

Solution: We give a constructive proof. Suppose that there is some minimum spanning tree T that does not contain edge (u, v). If we add edge (u, v) to the tree T, we obtain a connected graph with n edges that contains a single cycle, which includes edge (u, v). Note that if we remove any edge from this cycle, the resulting graph is still connected and contains $n - 1$ edges, and is therefore a spanning tree. Since we assumed that edge (u, v) is a minimum weight edge, removing any edge (not equal to (u, v)) from this cycle will result in a graph with weight no more than the weight of T. Thus, we have constructed a minimum spanning tree containing edge (u, v).

5 (Exercise 23.2-8) Professor Borden proposes a new divide-and-conquer algorithm for computing minimum spanning trees, which goes as follows. Given a graph $G = (V, E)$, partition the set V of vertices V_1 and V_2 such that $|V_1|$ and $|V_2|$ differ by at most 1. Let E_1 be the set of edges that are incident only on vertices in V_1, and let E_2 be the set of edges that are incident only on vertices V_2. Recursively solve a minimum-spanning-tree problem on each of the two subgraphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$. Finally, select the minimum-weight edge in E that crosses the cut (V_1, V_2), and use this edge to unite the resulting two minimum spanning trees into a single spanning tree.

Either argue that the algorithm correctly computes a minimum spanning tree of G, or provide an example for which the algorithm fails.

Solution: The algorithm fails on the following example: Consider a triangle $\{(u, v), (v, w), (w, u)\}$ in which each edge has a unique, positive weight. Suppose that edge (u, v) has the largest edge weight. If Professor Borden partitions the three vertices as follows, $V_1 = \{u, v\}, V_2 = \{w\}$, then his algorithm will choose edge (u, v) as the minimum spanning tree of V_1. Then he picks the minimum weight edge crossing the cut (V_1, V_2)—either (v, w) or (u, w)—to merge the minimum spanning trees on V_1 and V_2, the latter of which contains no edges. Note that, out of the three original edges, the final spanning tree will contain the minimum weight edge and the maximum weight edge (which is edge (u, v) by assumption). The actual minimum spanning tree will, however, contain the two edges out of the original three that have the smallest weight.

6 (part of Problem 23-3) A **bottleneck spanning tree** T of an undirected graph G is a spanning tree of G whose largest edge weight is minimum over all spanning trees of G. We say that the value of the bottleneck spanning tree is the weight of the maximum-weight edge in T.

Give a linear-time algorithm that given a graph G and an integer b, determines whether the value of the bottleneck spanning tree is at most b.

Solution: We can run DFS ignoring all edges that have weight greater than b. We can use the modification of DFS (a problem from a previous Exercise session) that keeps track of the number of connected components. If there is only one connected component, then the graph containing only edges with weight at most b is connected and therefore contains some spanning tree whose maximum weight edge is at most b. If there are two or more components, then edges with weight greater than b are necessary to construct any connected subgraph, e.g. a spanning tree. This algorithm runs in the same asymptotic time as DFS: $O(|E| + |V|)$.