Algorithms Example Sheet 5: Core Questions
Solution Notes

Minimum Spanning Trees

Exercise 5.C.1 [Basic tree properties] In this exercise, you will prove some of the basic properties about trees that are useful in analysing spanning trees:

(a) Show that adding an edge to a tree creates a unique cycle.

(b) Show that given a tree with a cycle, removing any edge from the cycle, leaves us with a tree.

(c) Given an undirected cycle C where each vertex is coloured either red or blue. Given that there is an edge (u, v) with $u \in R$ and $v \in B$, show that there must be another edge $e' = (u', v')$ on the cycle such that $u' \in R$ and $v' \in B$.

(d) Deduce that given a cycle C and a cut $(S, V \setminus S)$ in an undirected graph G, if there is one edge e crossing the cut, then there must be another e' that also crosses the cut.

Exercise 5.C.2 [Basic properties]

(a) Define the *minimum spanning tree* of a weighted undirected graph.

(b) **[Cut property]** Consider a weighted undirected graph G and a cut S. There exists an MST that contains the edge e with lightest weight in the cut.

(c) What happens if e has strictly the lightest weight in the cut? What can you say about the heaviest edge in the cut?

(d) **[Cycle property]** Consider a weighted undirected graph G and an edge e. If e is greater than all other edges in a cycle, then it cannot belong to any MST of G.

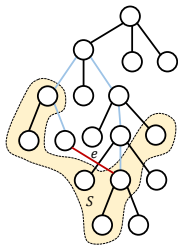
(e) What happens if there are multiple heaviest edges? What can you say about the lightest edge?

(f) In a connected undirected graph with edge weights ≥ 0, let $u \leftrightarrow v$ be a minimum-weight edge. Show that $u \leftrightarrow v$ belongs to a minimum spanning tree.

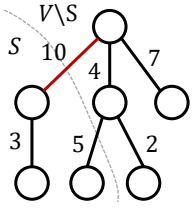
[Exercise 16 in Lecturer's handout]

(a) A spanning tree of a graph G any subgraph of G that is a tree and includes all V vertices of G. The weight of a tree is the sum of the edges in the tree. The MST is the spanning tree of minimum weight.

(b) Consider a cut $(S, V \setminus S)$ and let $e = (u, v)$ be the minimum weight edge in the cut. Then consider an MST T. If $e \in T$, then we are done. Otherwise, it does not contain e. Adding the edge e will produce a cycle in the MST (shown in blue in the figure). Since $u \in S$ and $v \notin T$, there must be at least one more edge e' on the cycle that crosses the cut. Hence, by replacing e' with e still results in a tree and its weight changes by $w(e) - w(e') \leq 0$. So this is also an MST.



(c) Continuing from part **??**, considering an MST T that does not contain e, adding e will change the weight by $w(e) - w(e') < 0$, i.e. the weight strictly decreases and so T cannot be the MST.
We cannot say anything about the heaviest edge. Consider the case where the input graph is a tree and choose any cut.

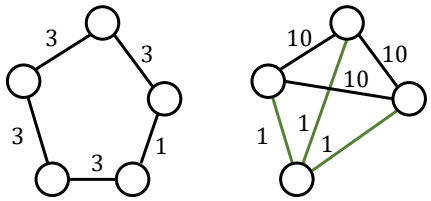


(d) Assume that the strictly heaviest edge e in a cycle belongs to the MST T. Since the MST is by definition a tree, some of the other edges, say e' does not belong to T. Since e was the heaviest edge we know that

$w(e) > w(e')$. By the properties of trees, when removing e from T we obtain two trees, which can be joined back to a tree T' by adding e'. But $w(T') = w(T) + w(e') - w(e) < w(T)$. Hence, T' is a spanning tree and has smaller weight than the MST (contradiction).

(e) If there are multiple heaviest edges, then it is possible that one of these is part of the MST. Consider the case where the graph is a cycle and all edge weights are the same.

The lightest edge is not guaranteed to be in an MST. Consider the graph on the right:



(f) **(Solution 1):** Since the graph is connected, Kruskal's algorithm computes the MST and will pick a minimum edge as the first edge to be included in the MST. So, there exists an MST that includes this edge.

(Solution 2): Assume there is an MST T without the minimal edge w^*. Then adding w^* to T will create a cycle which includes this edge (see properties of trees). Remove any other edge e' from the cycle and the resulting graph T' will be a tree (again by properties of trees). Since w^* is the minimal edge in the cycle, $\text{cost}(T') = \text{cost}(T) - w + w^* \leq \text{cost}(T)$ (since $-w + w^* \leq 0$). Hence, the resulting tree is also an MST.

Why does adding an edge to a tree create a cycle? Consider a tree T. When adding (u, v) to T, by definition T is connected so there is a path $u \rightsquigarrow v$, so in the new graph $u \rightsquigarrow v \to u$ is a cycle (and it involves (u, v)).

Why does removing an edge from the unique cycle of a graph leads to a tree? We will show that it is connected and there is no cycle. Another way is to start from first principles.

Let x and y be some vertices in G. Before removing the edge (u, v), x and y should be connected, so the path $p_{xy} = x \rightsquigarrow y$. If p_{xy} does not include (u, v), then the path will still be valid. If p_{xy} includes (u, v) (wlog $p = x \rightsquigarrow u \to v \rightsquigarrow y$), then because (u, v) belongs to a cycle there is a path $p_{uv} = u \rightsquigarrow v$ that does not include the edge (u, v). Hence, we can construct $p'_{xy} = x \rightsquigarrow up_{uv} \rightsquigarrow v$. Since the graph is connected and has $V - 1$ edges it is a tree.

Exercise 5.C.3 [Prim's algorithm]

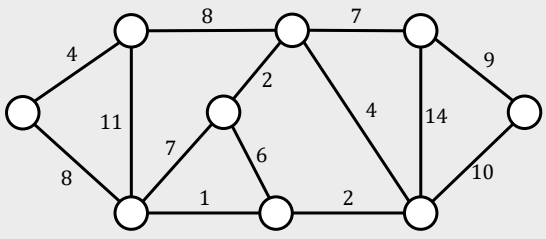
(a) Define Prim's algorithm.

(b) Write out a formal proof of correctness of Prim's algorithm. You may use without proof the theorem stated in lecture notes: "Suppose we have a forest F and a cut C such that (i) C contains no edges of F, and (ii) there exists a MST containing F. If we add to F a min-weight edge among those that cross C, then the result is still part of a MST." [*Note: the proof of this theorem is not examinable, but the application to Prim's algorithm is examinable. Also note that the term cut has two slightly different definitions, one for flow networks, one for spanning trees.*]

[Exercise 10 in Lecturer's handout]

(c) What is the time complexity of the implementation of Prim's algorithm?

(d) Describe how Prim operates on the following graph.



[Exercise 8 in Lecturer's handout]

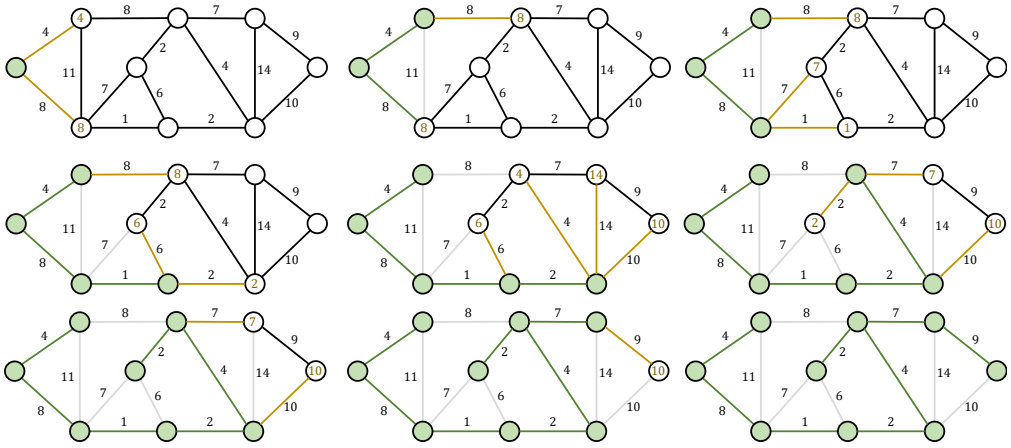
(e) Implement Prim's algorithm.

(a) Prim's algorithm starts from some vertex v and incrementally constructs a tree T by adding the edge e connecting $u \in T$ and some vertex in $w \in V \setminus T$. The candidate edges are maintained in a heap by the endpoint not in T and each time we pop a vertex w, we mark it as being in the tree and update the values for all its neighbours by the incident edges of w.

(b) Correctness follows because at each step we add the minimum edge in the cut $(T, V \setminus T)$. But we need a slightly stronger version than the cut theorem. We assume that T is part of an MST and we need to show that $T \cup \{v\}$ is also an MST. Assume there an MST T' extending T that did not include e. Then, consider the cut $(T, V \setminus T)$, then adding edge e would create a cycle. This cycle crosses the cut, hence one edge e' of the cut is also on the cycle. Removing e' leaves a tree with the weight decreasing by $w(e) - w(e')$. Since $w(e) \leq w(e')$, this is also an MST.

(c) There are $\mathcal{O}(E)$ decrease-key updates to the priority queue and V pop-min operations. If we use a binary heap as a priority queue, then we get a running time of $\mathcal{O}(V \log V + E \log V) = \mathcal{O}(E \log V)$. If we use a Fibonacci heap then we get $\mathcal{O}(E + V \log V)$, since the decrease-key operation takes $\mathcal{O}(1)$ amortised-time.

(d) Here is one possible execution starting from the left-most vertex. The vertices in the heap have their weight as a label. The yellow edges represent some edge that attains this minimum weight for each vertex. Grey edges we are sure that they won't be in the MST.



Note: We had a tie between two vertices with weight 8, but we can just choose any of these.

(e) The following code passes the testcases of **[SPOJ MST]**.

```
#include <algorithm>
#include <queue>
#include <stdio.h>
#include <vector>

/* Node and distance */
struct NodeAndDist {
  int u;
  int dist;

  bool operator<(const NodeAndDist& c) const {
    return dist > c.dist;
  }
};


int main() {
  int n, m;
  scanf("%d%d", &n, &m);
```

```
  // Adjacency list representation of the graph.
  std::vector<std::vector<NodeAndDist>> adj(n, std::vector<NodeAndDist>());
  std::priority_queue<NodeAndDist> pq;

  for (int i = 0; i < m; ++i) {
    int a, b, c;
    scanf("%d%d%d", &a, &b, &c);
    --a; --b; // Make the nodes zero based.
    adj[a].push_back({ b, c });
    adj[b].push_back({ a, c });
  }

  /* Main code for Prim's algorithm. */
  std::vector<bool> visited(n, false);
  long long total_cost = 0;
  pq.push({ 0, 0 });
  while (!pq.empty()) {
    auto top_candidate = pq.top();
    pq.pop();

    if (!visited[top_candidate.u]) {
      visited[top_candidate.u] = true;
      total_cost += top_candidate.dist;

      for (auto neigh : adj[top_candidate.u]) {
        pq.push({ neigh.u, neigh.dist });
      }
    }
  }
  printf("%lld\n", total_cost);

  return 0;
}
```

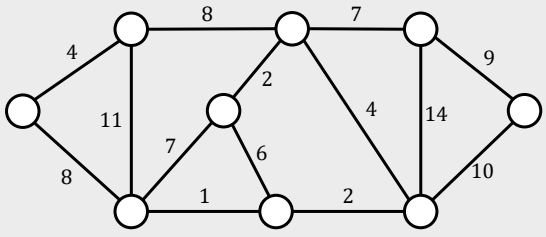
Exercise 5.C.4 [Kruskal's algorithm]

(a) Define Kruskal's algorithm.

(b) Write out a formal proof of correctness of Kruskal's algorithm.

(c) What is the time complexity of the implementation of Kruskal's algorithm?

(d) Describe how Kruskal operates on the following graph.



[Exercise 8 in Lecturer's handout]

(e) Implement Kruskal's algorithm.

(f) Are there any cases in which you would prefer Kruskal's algorithm?

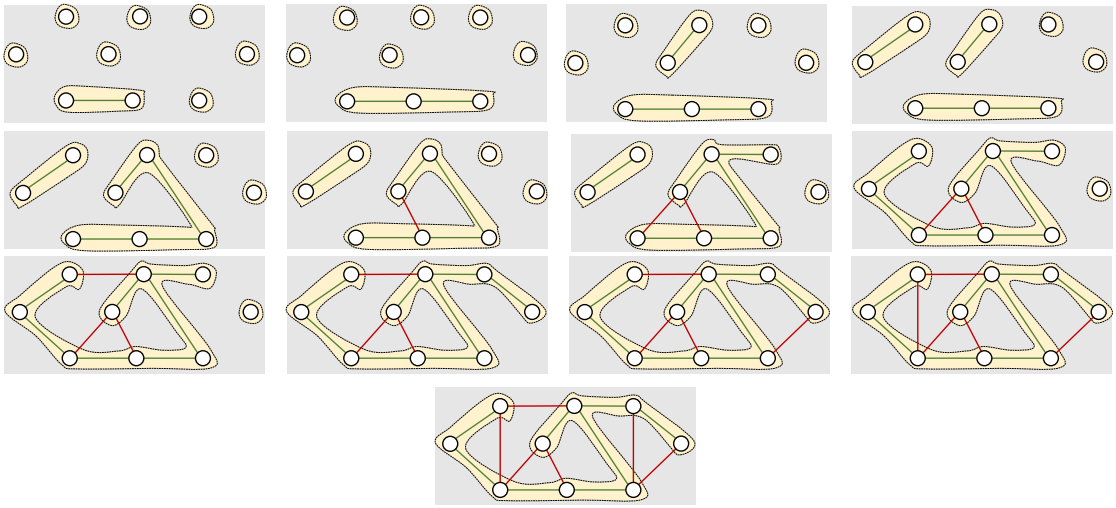
(a) Kruskal's algorithm sorts the edges by weight and initialises each vertex to be in a group of its own. Then it goes through the sorted edges in order and if the vertices of the current edge are not in the same group, then their groups are merged and the edge is added to the answer.

(b) We will prove by induction that if F is a forest that is the subgraph of an MST T, then if e is the smallest

edge not connecting two connected vertices, then $F \cup \{e\}$ is also the subset of an MST. Assume the edge $e = (u, v)$, and let C_u be the component of u. Then consider the MST T and assume that it does not contain e, then adding e will create a cycle. This cycle will intersect the cut $(C_u, V \setminus C_u)$ at another edge e'. By replacing e' with e, we obtain a tree T' with $w(T') \leq w(T)$, since $w(e) \leq w('e)$.

(c) Sorting takes $\mathcal{O}(E \log V)$ time and each disjoint set operation takes $\alpha(n)$ time. Hence, since for each edge we need to perform a search once this gives $\mathcal{O}(E\alpha(n) + E \log V) = \mathcal{O}(E \log V)$ time.

(d) Here is a possible execution, where green are the edges accepted in the MST, red are the edges not in the MST, the yellow groups indicate the disjoint sets during the operation.



(e) The following code passes the testcases of **[SPOJ MST]**.

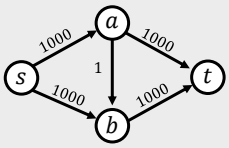
```
#include <algorithm>
#include <stdio.h>
#include <vector>

/* Undirected edge between u and v with weight w. */
struct Edge {
  int u, v, w;

  Edge(int u, int v, int w) : u(u), v(v), w(w) { }

  /* Comparison function to order in non-decreasing order of weights. */
  bool operator<(const Edge& e) const {
    return w < e.w;
  }
};


/* Implementation of the union-find data structure. */
struct UnionFind {
  /* The parent of the node. */
  std::vector<int> parent;
  /* The size of the component rooted at that node.
     (only valid if this is the root. */
  std::vector<int> sz;

  int findParent(int x) {
    if (parent[x] == x) return x;
    return parent[x] = findParent(parent[x]);
  }

  bool areConnected(int a, int b) {
    int pA = findParent(a), pB = findParent(b);
```

```
    return pA == pB;
  }

  void unite(int a, int b) {
    int pA = findParent(a), pB = findParent(b);
    if (pA == pB) return;

    if (sz[pA] < sz[pB]) {
      parent[pA] = pB;
    } else {
      parent[pB] = pA;
      if (sz[pA] == sz[pB]) ++sz[pA];
    }
  }

  UnionFind(int n) : parent(n, 0), sz(n, 0) {
    for (int i = 0; i < n; ++i) {
      parent[i] = i;
    }
  }
};

int main() {
  int n, m;
  scanf("%d%d", &n, &m);

  std::vector<Edge> edges;

  for (int i = 0; i < m; ++i) {
    int a, b, c;
    scanf("%d%d%d", &a, &b, &c);
    --a; --b; // Make the nodes zero based.
    edges.push_back(Edge(a, b, c));
  }

  /* Main Kruskal's implementation starts here. */
  std::sort(edges.begin(), edges.end());

  UnionFind union_find(n);

  long long total_cost = 0;
  for (const auto edge : edges) {
    if (!union_find.areConnected(edge.u, edge.v)) {
      union_find.unite(edge.u, edge.v);
      total_cost += edge.w;
    }
  }
  printf("%lld\n", total_cost);

  return 0;
}
```

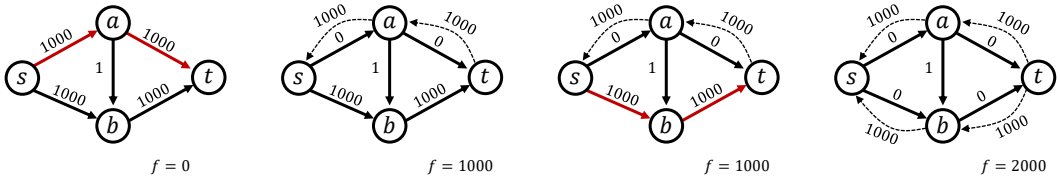
(f) Kruskal's algorithm has a worst-case $\mathcal{O}(E \log V)$ time, while Prim's algorithm has a worst-case $\mathcal{O}(E + V \log V)$. The constants are usually a bit worse in Prim's algorithm because of the Fibonacci heap. Also, in some cases the weights may be small, so we could use faster sorting methods, like counting sort and get a better asymptotic running time.

Maximum flow

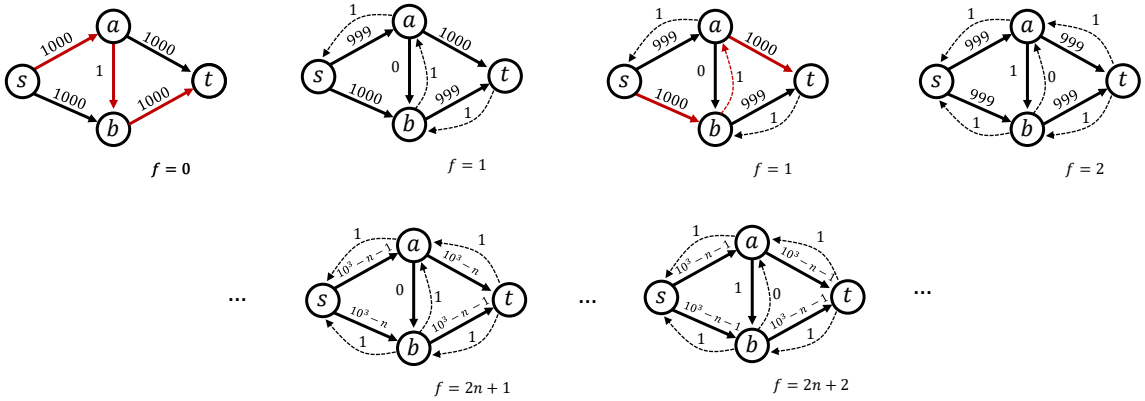
Exercise 5.C.5 Use the Ford-Fulkerson algorithm, by hand, to find the maximum flow from s to t in the following graph. How many iterations did you take? What is the largest number of iterations it might take, with unfortunate choice of augmenting path?



The best case is running two iterations and in the third one detecting that there is no augmenting path. For example,



The worst-case is running 2000 iterations and in the last one detecting that there is no augmenting path. For example,



where the general case is for after the respective iteration ($2n+1$ and $2n+2$) has executed.
Question: *This proves a lower bound that there are graphs where the Ford-Fulkerson algorithm needs $\Omega(f^*)$ iterations. Why does this imply $\Omega(f^* \cdot E)$ time complexity?* By adding some edges and vertices that do not lead to t, the algorithm will possibly need to traverse these in order to find an augmenting path, so each iteration could take $\Theta(E)$ time.

Exercise 5.C.6 [Non-Terminating example for Ford-Fulkerson] (optional) Look at the example graph presented here (and here or here).

(a) Prove that r (usually denoted as ψ, being related to the golden ratio ϕ) satisfies $r^2 = 1 - r$.

(b) What is the maximum flow for the given graph?

(c) Define formally the pathological sequence of augmented paths.

(d) Does this sequence converge to the maximum flow? What does this mean?

Exercise 5.C.7 Argue why the Ford-Fulkerson algorithm terminates when the edge capacities are rational (in a finite graph).

Transform all edge capacities so that they all have the same denominator (in the worst-case $\prod_{i=1}^{m} b_i$ where $c(e_i) = a_i/b_i$ for $a_i, b_i \in \mathbb{N}^+$). Then every operation of the Ford-Fulkerson algorithm will change the numerator.

Hence, there is a constant additive improvement to the flow (namely $1/denom$) in each iteration. Hence, after $f^* \cdot denom$ the algorithm should terminate with the correct flow.
Note: These might be a lot of iterations since *denom* could be large.

Exercise 5.C.8 Consider a flow f on a directed graph with source vertex s and sink vertex t. Let $f(u \to v)$ be the flow on edge $u \to v$, and set $f(u \to v) = 0$ if there is no such edge.

(a) Show that

$$\sum_{v \neq s,t} \left(\sum_w f(v \to w) - \sum_u f(u \to v) \right) = 0.$$

(b) The value of the flow is defined to be the net flow out of s,

$$\text{value}(f) = \sum_w f(s \to w) - \sum_u f(u \to s).$$

Prove that this is equal to the net flow into t. [*Hint:* Add the LHS of the equation from part **??**]

(a) The flow conservation equation states that for every vertex $v \neq s, t$,

$$\sum_w f(v \to w) - \sum_u f(u \to v) = 0.$$

Hence, by summing this over all $v \neq s, t$, we get

$$\sum_{v \neq s,t} \left(\sum_w f(v \to w) - \sum_u f(u \to v) \right) = 0.$$

(b) Note that the following sum is equal to 0,

$$\sum_v \left(\sum_w f(v \to w) - \sum_u f(u \to v) \right) = \sum_v \sum_w f(v \to w) - \sum_v \sum_u f(u \to v) = \sum_v \sum_w f(v \to w) - \sum_v \sum_w f(v \to w) = 0,$$

since we are summing the flow assigned to each edge once with $+$ and one with $-$. Now, taking the difference of the two sums, the only terms that remain are the edges in and out of s and t, i.e.

$$\begin{aligned}
&\sum_v \left(\sum_w f(v \to w) - \sum_u f(u \to v) \right) - \sum_{v \neq s,t} \left(\sum_w f(v \to w) - \sum_u f(u \to v) \right) \\
&= (\sum_w f(s \to w) - \sum_u f(u \to s)) + (\sum_w f(t \to w) - \sum_u f(u \to t)) \Rightarrow \\
0 &= (\sum_w f(s \to w) - \sum_u f(u \to s)) + (\sum_w f(t \to w) - \sum_u f(u \to t)) \Rightarrow \\
(\sum_w f(s \to w) - \sum_u f(u \to s)) &= -(\sum_w f(t \to w) - \sum_u f(u \to t)) \Rightarrow \\
\text{value}(f) &= \sum_w f(s \to w) - \sum_u f(u \to s).
\end{aligned}$$

Exercise 5.C.9 The code for `ford_fulkerson` as given in the handout has a bug: lines 27-39, which augment the flow, rely on an unstated assumption about the augmenting path. Give an example which makes the code fail. State the required assumption, and prove that the assertion on line 39 is correct, i.e. that after augmenting we still have a valid flow.

The implementation of method `find_augmenting_path()` is underspecified. In particular, in the following lines:

```
if h has a path from s to t:
  return some such path, together with the labels of its edges
```

there is no restriction for the path to be simple. Hence, the returned augmenting path could contain a cycle. This means that when augmentation takes place the value maybe be removed twice from some of the edges (even though they do not have the capacity). This may result into using more flow than available in subsequent iterations. For example, the maximum flow in the following graph is 4, but augmenting a path with a cycle, makes the second augmentation wrong leading to a flow of 6.