Homework 2

Problem 5.3:

5.3. Design a linear-time algorithm for the following task.

Input: A connected, undirected graph G.
Question: Is there an edge you can remove from G while still leaving G connected?

Can you reduce the running time of your algorithm to $O(|V|)$?

Yes, there is an edge you can remove from G while still leaving G connected. To determine if there's a removable edge, we need to check if the graph is not a tree. A connected, undirected graph G with |V| vertices is a tree if and only if it has exactly |V| - 1 edges. Thus,

- If |E| > |V| - 1, then G contains at least one cycle and you can remove an edge from that cycle and still keep the graph connected
- If |E| = |V| - 1, the graph is a tree and removing any edge disconnects the graph.

This basic check can be done in O(|V| + |E|) time by just counting the edges as you read the input and comparing |E| to |V| - 1.

Linear-Time Algorithm:

- Read the graph's input (as either an adjacency list or edge list)
- Count the total number of vertices |V| and edges |E|
- Compare |E| to |V| - 1. If |E| > |V| - 1, then yes there is a removable edge. Otherwise, there isn't

Reading the entire input takes O(|V| + |E|) time.

To reduce this algorithm to O(|V|), let's perform DFS as we read the edges. Let's set up a visitation procedure from any vertex. As we read a new edge, if it connects to an unvisited vertex, we can add that vertex to a spanning tree. When we've visited all vertices, we have found exactly |V| -1 edges in our spanning tree. So, if after visiting all vertices and we see that there are more edges left in the input, we know there is at least one cycle and thus we can remove an edge and keep the graph connected .

Problem 5.7:

5.7. Show how to find the *maximum* spanning tree of a graph, that is, the spanning tree of largest total weight.

To find the Maximum Spanning Tree, we can use Kruskal's or Prim's algorithm just by reversing the edge-comparison i.e. instead of choosing the smallest-weight edges first, let's choose the largest-weight edges first.

Prim's Algorithm for Maximum Spanning Tree:

1. Stat from any vertex
2. At each step, add the largest-weight edges first that connects the current tree to a new vertex outside the tree.

We can use Prim's Algorithm with a binary heap tree and an adjacency list representation. With this, the time complexity would be O(|E|log|V|).

Problem 5.21:

5.21. In this problem, we will develop a new algorithm for finding minimum spanning trees. It is based upon the following property:

Pick any cycle in the graph, and let e be the heaviest edge in that cycle. Then there is a minimum spanning tree that does not contain e.

(a) Prove this property carefully.

(b) Here is the new MST algorithm. The input is some undirected graph $G = (V, E)$ (in adjacency list format) with edge weights $\{w_e\}$.

```
sort the edges according to their weights
for each edge e ∈ E, in decreasing order of w_e:
    if e is part of a cycle of G:
        G = G − e (that is, remove e from G)
return G
```

Prove that this algorithm is correct.

(c) On each iteration, the algorithm must check whether there is a cycle containing a specific edge e. Give a linear-time algorithm for this task, and justify its correctness.

(d) What is the overall time taken by this algorithm, in terms of $|E|$? Explain your answer.

A) Given any cycle in the graph, the heaviest edge on that cycle can be safely removed without losing the ability to form some minimum spanning tree.

Suppose we have a minimum spanning tree T. If an edge e is the heaviest edge in some cycle C, then we can replace e with any other edge of C not in T to potentially form another spanning tree that is no heavier than T.

Proof of Correctness: Consider any cycle C in the original graph G. Let e_max be the heaviest edge in C. Suppose we have a minimum spanning tree T of G.

1. If e_max is not in T, then we are done, this MST already does not contain e_max.
2. If e_max is in T, consider the cycle C. Since T is a spanning tree, it has no cycles. Add T to all edges of C except e_max and if this forms a cycle in T ∪ (C - {e_max}), then the cycle is formed by all edges of C plus e_max. Now remove e_max from this new structure. We get another spanning tree T' that is different from T by replacing e_max with one of other edges in C. Since e_max was the heaviest edge in the cycle C, replacing it with a lighter (or equally weighted) edge cannot increase the total weight. Thus $w(T') \leq w(t)$. Because T is assumed to be a MST and T' has a weight not greater than T, it follows that T' is also a MST. Thus, we have constructed a MST T' that does not contain e_max.

Problem 5.21 cont…

B)

Algorithm:

1. Sort edges in decreasing order by weight
2. For each edge e (from heaviest to lightest): If e is part of a cycle in the current graph, remove e
3. Return resulting graph

Proof of Correctness:

1. Start with G, which is a connected weighted graph. At the end of the procedure, if we remove no edges, G is a spanning tree itself. If G is initially not a tree, we will remove edges until it is.
2. For any cycle that forms in the current graph, we can remove the heaviest edge and still be able to form and MST from the remaining edges
3. Invariant: At every step we consider an edge e:

- If e is not part of a cycle, it may still be potentially needed for the MST so we keep it
- If e is part of a cycle, we remove it and it will not destroy any MSTs

4. Since we have removed only edges that were the heaviest in their respective cycles, we never removed an edge that might be crucial for minimizing the total weight. By always removing the heaviest edge from cycles, the final spanning tree is minimal in weight.

Thus, the set of edges we end up with is a MST>

Problem 5.21 cont…

C) Let's use Union-Find.

1. Maintain a Disjoint Set Union (DSU) data structure over the vertices. Initially, each vertex is in its own set.
2. Process each edge in decreasing order. Consdier an edge = (u,v). Check the DSU sets of u and v. If they are in the same set, then adding e would form a cycle. If they are in different sets, joining them does not form a cycle, thus union their sets.

Union-Find operations run in almost constant time O(|V|). Over the entire run of the algorithm, performing union-find operations for all edges is O(|E||V|) which is pretty much linear for most practical purposes.

Proof of Correctness:

- Union-Find captures the exact connectivity structure of the graph as edges are read
- If two vertices are already in the same set, adding them creates a cycle
- DSU operations ensure we detect this condition in almost constant time

D) We have |E| edges. Sorting them by weight in decreasing order takes O(|E| log|E|) time. Each union-find operation takes O(|V|) time. So for each |E| operations, the total complexity is O(|E||V|) which is close to linear. The largest cost is sorting the edges O(|E|log|E|) and O(|V|) is a nearly constant time. So the time complexity is O(|E|log|E|) which matches up with Kruskal's algorithm.

Problem 7.4:

7.4. Moe is deciding how much Regular Duff beer and how much Duff Strong beer to order each week. Regular Duff costs Moe $1 per pint and he sells it at $2 per pint; Duff Strong costs Moe $1.50 per pint and he sells it at $3 per pint. However, as part of a complicated marketing scam, the Duff company will only sell a pint of Duff Strong for each two pints or more of Regular Duff that Moe buys. Furthermore, due to past events that are better left untold, Duff will not sell Moe more than 3,000 pints per week. Moe knows that he can sell however much beer he has. Formulate a linear program for deciding how much Regular Duff and how much Duff Strong to buy, so as to maximize Moe's profit. Solve the program geometrically.

Let:

- R = number of pints of Regular Duff to order per week
- S = number of pints of Duff Strong to order per week S

Moe's profit on each pint of Regular Duff:

- Cost: $1 per pint
- Selling Price: $2 per pint
- Profit per pint of Regular Duff: $1

Moe's profit on each pint of Duff Strong:

- Cost: $1.50 per pint
- Selling Price: $3 per pint
- Profit per point of Duff Strong: $1.50

Objective Function: Maximize $Z = R + 1.5S$

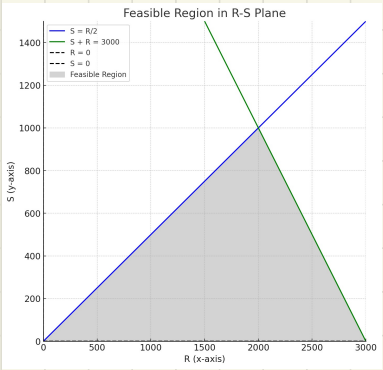
Constraints:

- Marketing Constraint: The Duff company will only sell one pint of Duff Strong for each two or more pints of Regular Duff. This means the number of Duff Strong pints cannot exceed half the number of Regular Duff pints. Therefore, $S \leq R/2$.
- Duff will not sell more than 3000 pints. Therefore $R + S \leq 3000$.
- No negative number of pints. Therefore, $R \geq 0$ and $S \geq 0$.

So, Maximize $Z = R + 1.5S$ subject to $S \leq R/2$, $R + S \leq 3000$, $R \geq 0$, and $S \geq 0$

Problem 7.4 cont…

To solve geometrically, let's consider the feasible region in the R-S plane defined by the constraints.



Intersections:

- (0,0)
- (2000,1000)
- (3000,0)

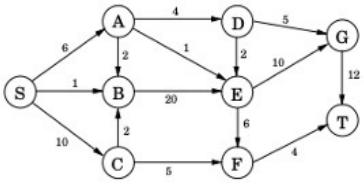
Evaluate the function at each extrema, the largest value is the optimal solution:

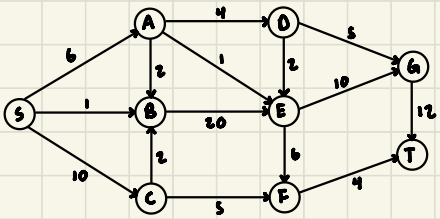
- At (0,0): Z = 0 + 1.5 * 0 = 0
- At (2000,1000): Z = 2000 + 1.5*1000 = 3500
- At (3000,0): Z = 3000 + 1.5*0 = 3000

The maximum Z value is 3500 at (R,S) = (2000,1000). Therefore the optimal solution is R = 2000 pints of Regular Duff and S = 1000 points of Duff Strong.

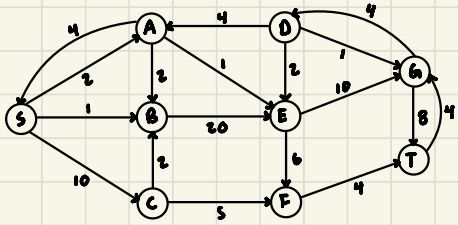
Problem 7.10:

7.10. For the following network, with edge capacities as shown, find the maximum flow from S to T, along with a matching cut.

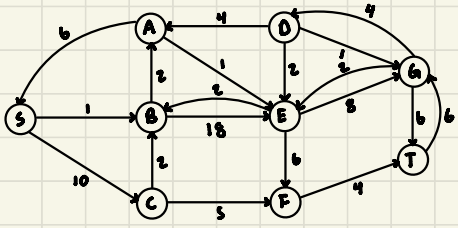




Augmenting Path: S → A → D → G → T, Bottleneck = 4

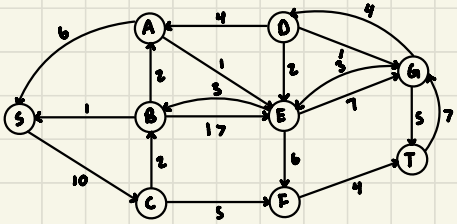


Augmenting Path: S → A → B → E → G → T, Bottleneck = 2

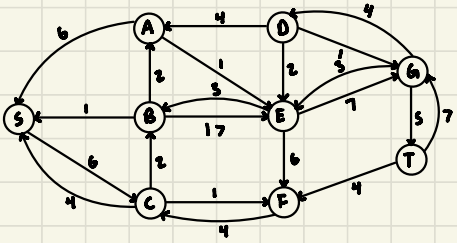


Problem 7.10 cont…

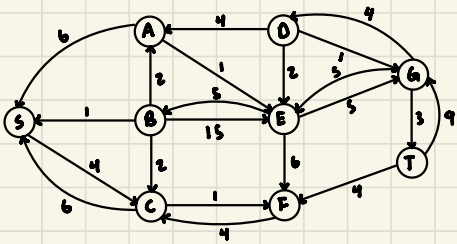
Augmenting Path : S → B → E → G → T , Bottleneck : 1



Augmenting Path : S → C → F → T , Bottleneck : 4



Augmenting Path : S → C → B → E → G → T , Bottleneck : 2

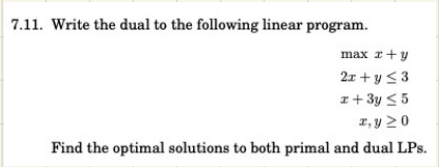


The maximum flow is the sum of all the bottlenecks. Maximum Flow = 4 + 2 + 1 + 4 + 2 = 13.

The matching cut where S should be in M and T should be in N:

- M = {S, C, F}
- N = {A, B, D, E, G, T}

Problem 7.11:



7.11. Write the dual to the following linear program.

$$\max\ x + y$$
$$2x + y \leq 3$$
$$x + 3y \leq 5$$
$$x, y \geq 0$$

Find the optimal solutions to both primal and dual LPs.

Primal Variables: $x, y \geq 0$
Primal Constraints:

- $2x + y \leq 3$
- $x + 3y \leq 5$

Let's denote the dual variables associated with these constraints as u and v where $u, v \geq 0$

The primal object is to max $x + y$. For the dual we have to minimize each primal constraint's RHS. Therefore the dual objective min $3u + 5v$

Each primal variable generates one dual constraint. The coefficients in these constraints come from the columns of the primary constraint matrix.

- For x: $2u + v \geq 1$
- For y: $u + 3v \geq 1$

And $u, v \geq 0$

Final Dual Problem:
min $3u + 5v$
$2u + v \geq 1$
$u + 3v \geq 1$
$u, v \geq 0$

Problem 7.11 cont…

Optimal Solution to the Primal:

max $x + y$

$2x + y \leq 3$

$x + 3y \leq 5$

$x, y \geq 0$

Intersections:

- (0,0)
- (1.5,0)
- (0, 5/3)
- (0.8, 1.4)

Calculations:

- At (0, 0): $0 + 0 = 0$
- At (1.5,0): $1.5 + 0 = 1.5$
- At (0, 5/3): $0 + 5/3 = 1.667$
- At (0.8, 1.4): $0.8 + 1.4 = 2.2$

The primal optimal solution is $x = 0.8$, $y = 1.4$, and the optimal value = 2.2.

Problem 7.11 cont…

Optimal Solution to the Dual:

min $3u + 5v$

$2u + v \geq 1$

$u + 3v \geq 1$

$u, v \geq 0$

(1) $2u + v = 1 \Rightarrow v = 1 - 2u$

(2) $u + 3(1 - 2u) = 1 \Rightarrow u + 3 - 6u = 1 \Rightarrow -5u = -2 \Rightarrow u = 2/5$

(3) $v = 1 - 2(0.4) = 1 - 0.8 = 0.2$

(4) $3u + 5u = 3(0.4) + 5(0.2) = 1.2 + 1 = 2.2$

Thus the dual optimal solution is $u = 0.4$, $v = 0.2$, and the optimal value = 2.2.

Problem 8.4:

8.4. Consider the CLIQUE problem restricted to graphs in which every vertex has degree at most 3. Call this problem CLIQUE-3.

(a) Prove that CLIQUE-3 is in **NP**.

(b) What is wrong with the following proof of **NP**-completeness for CLIQUE-3?
We know that the CLIQUE problem in general graphs is **NP**-complete, so it is enough to present a reduction from CLIQUE-3 to CLIQUE. Given a graph G with vertices of degree ≤ 3, and a parameter g, the reduction leaves the graph and the parameter unchanged: clearly the output of the reduction is a possible input for the CLIQUE problem. Furthermore, the answer to both problems is identical. This proves the correctness of the reduction and, therefore, the **NP**-completeness of CLIQUE-3.

(c) It is true that the VERTEX COVER problem remains **NP**-complete even when restricted to graphs in which every vertex has degree at most 3. Call this problem VC-3. What is wrong with the following proof of **NP**-completeness for CLIQUE-3?
We present a reduction from VC-3 to CLIQUE-3. Given a graph $G = (V, E)$ with node degrees bounded by 3, and a parameter b, we create an instance of CLIQUE-3 by leaving the graph unchanged and switching the parameter to $|V| - b$. Now, a subset $C \subseteq V$ is a vertex cover in G if and only if the complementary set $V - C$ is a clique in G. Therefore G has a vertex cover of size $\leq b$ if and only if it has a clique of size $\geq |V| - b$. This proves the correctness of the reduction and, consequently, the **NP**-completeness of CLIQUE-3.

(d) Describe an $O(|V|^4)$ algorithm for CLIQUE-3.

A) To show that CLIQUE-3 is in NP, we must show two things:

1. A canditation solution (a set of vertices) for clique of size g can be verified in polynomial time
2. The problem's solution is a yes/no decision problem

Given an instance of CLIQUE-3: A graph G = (V,E) with a max vertex degree of at most 3 and an integer g, we ask whether there exits a set of g vertices that form a complete subgraph (clique) of size g. A non-deterministic algorithm can guess a set of g vertices and then verify in time O(g^2) which is polynomial, whether all pairs of chosen vertices are connected by an edge. Since verify a solution is in polynomial time, CLIQUE-3 is in NP.

Problem 8.4 cont…

B) To prove that CLIQUE-3 is NP-complete, we must start from a known NP-complete problem and reduce that problem to CLIQUE-3, not the other way around. The direct of reduction is crucial. They provided a trivial upper bound on the complexity of CLIQUE-3 by embedding it in CLIQUE but they did not show that CLIQUE-3 is hard enough to encode other NP-complete problems.

C) The problem states that the vertex cover problem restricted to graphs of max degree 3, say VC-3) remains NP-complete. It then present a supposed reduced from VC-3 to CLIQUE-3. The fundamental mistake is the claim that the complement of a vertex cover is a clique. This is not true and the complement of a vertex cover is actually an independent set, not a clique.

D) Since CLIQUE-3 is the CLIQUE problem restricted to graphs where each vertex has degree at most 3, the max size of a clique in such a graph is 4. We cannot have a clique larger than 4 because a clique of size 5 requires each vertex to have a degree of at least 4 (connect to 4 other vertices in the clique) which contradicts the max-degree-3 constraint. Thus, any clique in a degree-3 bounded graph have size at most 4. Then, enumerating all subsets of up to 4 vertices out of |V| times takes O(|V|^4) time in worse case. For each subset of size g, verifying that it forms a clique takes O(g^2) time. Hence, the entire algorithm run in O(|V|^4) time.