CSE202
4 – Network Flow

TA: Joseph L.

I. "The 15-Part Play" of Network Flow

1. $G = (V, E)$ is a directed graph with source vertex s, sink vertex t. There will always be exactly one source and one sink. *As an analogy, one can think of this graph as a section of a pipeline network from s to t, wherein edges are pipes.*

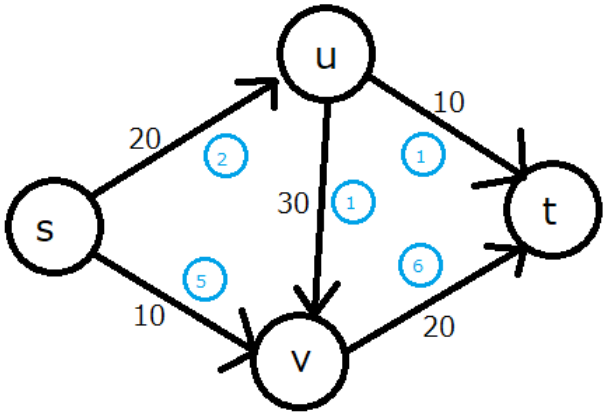

Figure 1. A sample flow network. The black numbers represent *capacities* and the light blue, circled numbers represent the *flows*. The *value* of the flow is 7. The following 3 sections will explain each of these concepts.

2. **Capacity**: Akin to…the capacity of a pipe. Assume (for now) that capacities will always be integers.
$$c: E \to Z_{\geq 0}$$
3. **Flow**: Akin to how much liquid is actually flowing through a given pipe.
$$f: E \to Z_{\geq 0}$$
   a. For any edge e: $0 \leq f(e) \leq c(e)$
   b. Conservation of flow: For any non-source/non-sink node u, the amount of flow entering u is equal to the amount of flow leaving u
$$\forall u \neq s, u \neq t: \sum_{vu \in E} f(vu) = \sum_{uv \in E} f(uv)$$

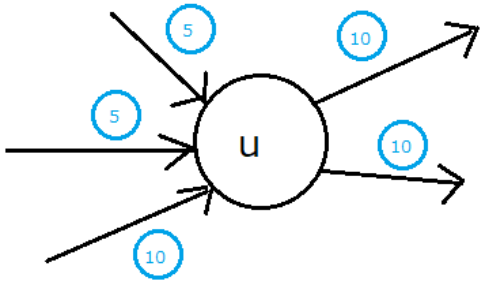

Figure 2. Conservation of flow for an arbitrary node u, where $u \neq s, u \neq t$.

4. **Value**: The total flow leaving s, which is equal to the total flow entering t.
$$Value(f) = \sum_{su \in E} f(su) = \sum_{vt \in E} f(vt)$$

5. **Max Flow**: The flow that maximizes $Value(f)$.

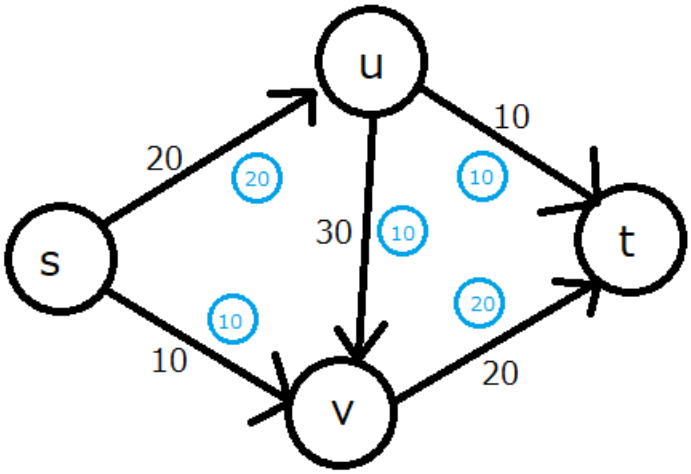

Figure 3. A flow network with max flow. In this example, we can easily see that the flow is maxed since all edges leaving s are **saturated** (as are the ones entering t, for that matter).
A **saturated edge** is an edge e wherein $f(e) = c(e)$.

6. **Residual Graph**: A graph with some flow.
, and differs from the original in the following ways:
   a. For each edge $e \in E$ with $f(e) > 0$, there is a "backwards" edge going in the opposite direction with capacity equal to $f(e)$.
   b. Each edge $e \in E$ has had its capacity reduced by $f(e)$ – its new capacity is called a **residual capacity**. Accordingly, saturated edges have residual capacity 0.

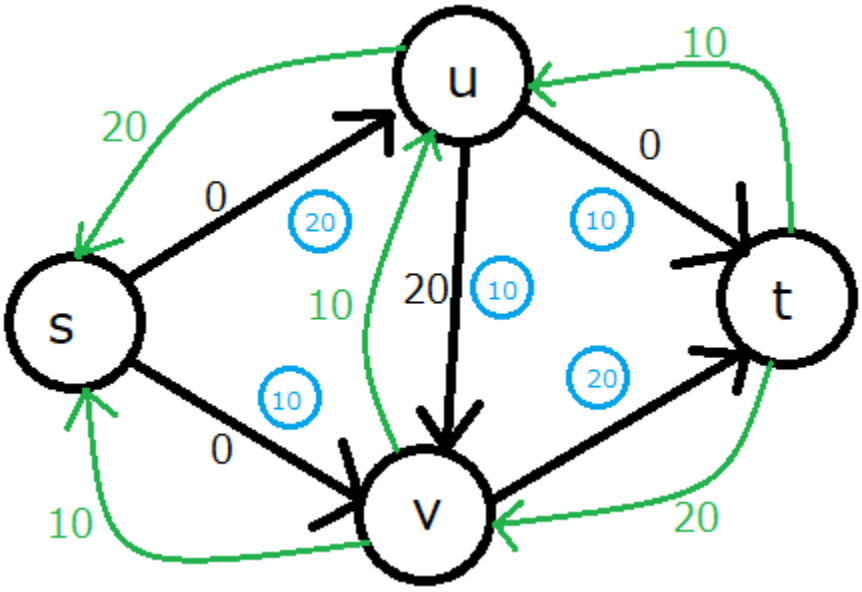

Figure 4. A residual graph of the flow in *Figure 3*.

7. **Augmenting Path**: A path from s to t within the *residual* graph.
   a. The Ford-Fulkerson and Edmonds-Karp algorithms work by selecting augmenting paths one at a time and filling them as much as possible (that is, limited by the lowest-capacity edge along the path). If traversing a backwards edge, we *subtract* flow from it instead.
8. **Cut** – a partition of V into two sets A, B where $s \in A, t \in B$.

9. Capacity of a cut:

$$C(A,B) = \sum_{\substack{uv \in E \\ u \in A \\ v \in B}} c(uv)$$

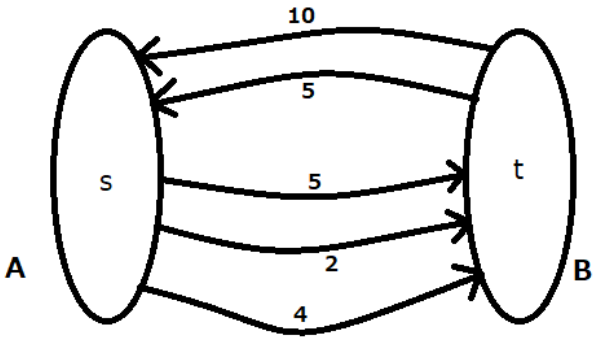

Figure 5. An example cut. What is the capacity of this cut?

10. $MaxFlow \leq C(A,B)$, where (A,B) is any cut.
    Equivalently: $MaxFlow \leq \min_{(A,B)} C(A,B)$ $0 \leq f(e) \leq c(e)$
11. **Min Cut**: The cut with the smallest capacity.

$$MinCut = \min_{(A,B)} C(A,B)$$

12. **Theorem**: $MaxFlow = MinCut$
    Makes sense – there's no way any flow could exceed the capacity of the most restrictive cut (i.e., bottleneck) of the flow network.
    Also, note that the *residual graph* of a max-flow/min-cut will have *no paths* from s to t. Were this not the case, it would still be possible to add more flow, so we wouldn't have a max-flow.
13. $Value(f) \leq C(A,B)$

$$Value(f) = \sum_{\substack{uv \in E \\ u \in A \\ v \in B}} f(uv) - \sum_{\substack{vu \in E \\ v \in B \\ u \in A}} f(vu)$$

14. Let $R_G(f)$ be the residual graph at the termination of the algorithm.
    Let $A = \{u \text{ s.t. } u \text{ can be reached from } s \text{ in } R_G(f)\}$.
    Then: $B = V - A$

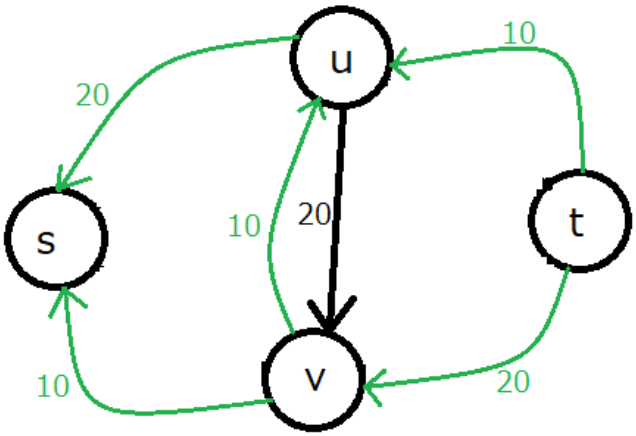

Figure 6. $A = \{s\}, B = \{u, v, t\}$ in this example.

15. The final flow graph, G_f

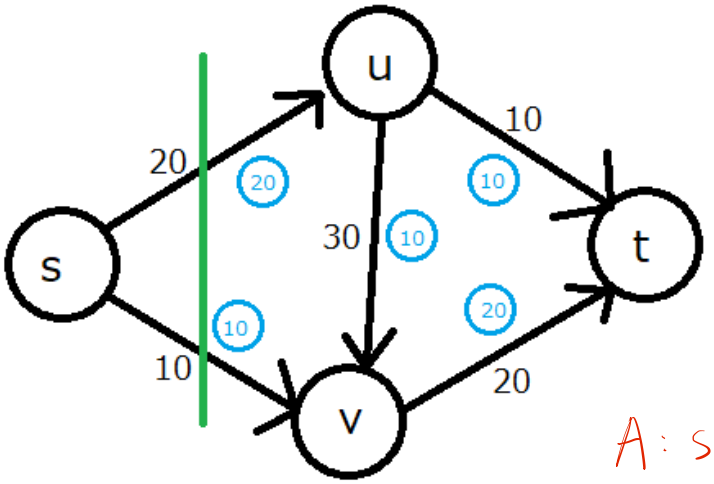

Figure 7. G_f for our ongoing example.

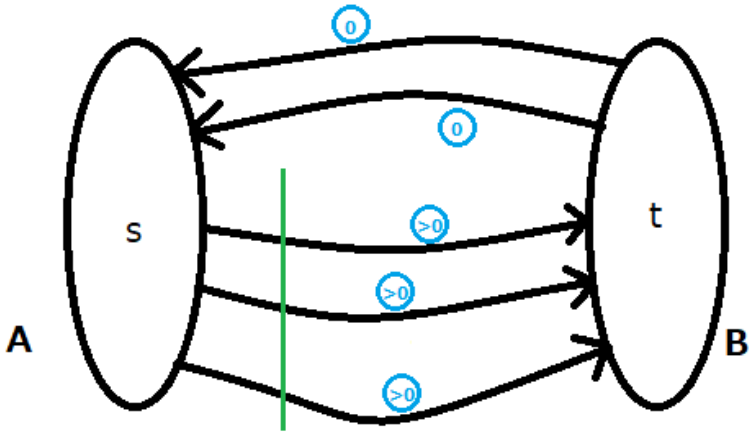

Figure 8. G_f in general. All edges from B to A have zero flow; were this not the case, the residual graph would have A-to-B edges, so that's not possible. In short:

$$C(A,B) = \sum_{\substack{uv \in E \\ u \in A \\ v \in B}} c(uv)$$

II. The Goal

Faced with a problem that (most likely) doesn't initially present itself as a network flow problem, we would like to do the following.

1. Represent the problem as a network flow problem – either as a "max-flow" or "min-cut" problem.
   a. If you can formulate the problem as a matching problem, then it is most definitely a max-flow problem.
      i. Spotting matching problems may not always be easy. One of the example problems is a slightly disguised matching problem.
   b. Many problem involves disconnecting some set of nodes from another while minimizing some cost. This is most definitely a min-cut problem.
   c. If the problem feels less straightforward, then you should try to think of a representative problem that it seems similar to. For min-cut, project selection and baseball elimination are both very useful.

2. Once you've decided on max-flow or min-cut, set up the flow network. Describe all node/edge types in detail.
3. Describe how to extract a solution from the max-flow or min-cut of your network. For example, in bipartite matching, the correct matchings are derived by taking the saturated intermediate edges.
4. Choose an algorithm (options will be discussed in the next section) and figure out the complexity accordingly.
5. Prove correctness.
   a. For max-flow, you want to prove the following.
      i. Your nodes and edge weights do not violate any of the constraints of the problem (e.g., in bipartite matching, no one gets matched with more than one person).
      ii. A max-flow of your graph and necessarily corresponds to a solution, and vice-versa.
      iii. You correctly extract a solution from the network.
   b. For min-cut, you want to prove the following.
      i. Consider what kinds of edges can potentially cross a min-cut and, consequently, what kinds of nodes can end up in A/B. It would likely be helpful to draw a diagram like those shown in class.
      ii. The following scenarios are common, and may require slightly different proofs:
         - If the problem asks you to disconnect some group of nodes, show that the right nodes end up in A or B, and that your network accomplishes the disconnection with minimum cost.
         - If the problem asks you to maximize some quantity Q, then the proof will likely be quite mathematical. In general, you want to add up the capacities of the edges crossing the cut and relate them to the quantity that you want to maximize. Let C be the value of the cut and let X be some *constant* that you derive. You will likely end up with an equation of the following form:

           $$Q = X - C$$

           This shows that minimizing C will directly maximize your quantity.
         - If your problem falls under neither of these cases, then your proof will likely be a mathematical one that shows how the value of your cut solves the problem.

III. Algorithms for Max-Flow

1. Ford-Fulkerson

a. Variables:
   i. $G = V, E$
   ii. $C = \sum_{su \in E} c(su)$. We don't know in advance the exact value of the max-flow, so this is our upper bound for it.
      1. A tighter bound: $C = \min\{\sum_{su \in E} c(su), \sum_{vt \in E} c(vt)\}$

b. Algorithm: Repeat until termination.
   i. Find an augmenting path.
   ii. Fill the path as much as possible.

c. Complexity:
   i. Each iteration takes $\theta(|V| + |E|)$ to find an augmenting path.
   ii. Each non-final iteration adds an *integer value of at least 1*. Therefore, there will be at most C iterations.
      1. If an iteration adds no value, then that means that there are no augmenting paths left.
   iii. Total complexity: $\boldsymbol{O\big((|V| + |E|)C\big)}$
   iv. **Pseudopolynomial** – looks polynomial, but C could be exponentially large.

d. Notes:

i. *Only* guaranteed to terminate for integer capacities, as we mentioned above that each iteration adds *integer* values.
   1. If all of the capacities are *rational* numbers, then one can multiply them by a constant to make them integers. Keep in mind that this may greatly increase C and, consequently, the runtime.

2. (Scaling) Ford-Fulkerson

a. Variables:
   i. $G = V, E$
   ii. $C = \sum_{su \in E} c(su)$.
   iii. T: A threshold. When finding paths, ignore edges with capacity $< T$.

b. Algorithm: Repeat until termination.
   i. Find an augmenting path for which no edge has capacity $< T$.
      1. If no such path exists: halve T. $T \leftarrow \frac{T}{2}$
   ii. Fill the path as much as possible.

c. Complexity:
   i. Each iteration still takes $\theta(|V| + |E|)$ if a valid path exists.
   ii. The total complexity is $O((|V| + |E|)m \log C \approx \boldsymbol{O(|E|^2 \log C)}$.
      1. $|V| + |E|$ is generally on the order of $|E|$.
      2. See the textbook for details on the remaining $|E| \log C$ factor.

d. Notes:
   i. It is very much just Ford-Fulkerson, but more selective with paths.
   ii. As such, it has the same limitation of guaranteed termination only for integer capacities.

3. Edmonds-Karp

a. Variables:
   i. $G = V, E$

b. Algorithm: Repeat until termination.
   i. Find the augmenting path with the *fewest edges* (BFS), breaking ties arbitrarily.
   ii. Fill the path as much as possible.

c. Complexity:
   i. Finding a path takes, again, $\theta(|V| + |E|)$.
   ii. There are at most mn iterations (find proofs online if curious).
   iii. Therefore, the total complexity is $\boldsymbol{O(|E|^2|V|)}$.

d. Notes:
   i. This algorithm's complexity does *not* depend on C, and thus the algorithm is *strongly* polynomial.
      1. A network *could* still have exponentially many nodes or edges, thus potentially making another algorithm better.

4. Preflow-Push

a. Variables:
   i. $G = V, E$

b. The implementation details of this algorithm are out of scope, but can be found in the textbook. With that said, the algorithm has some important properties that you should know about.
   i. The algorithm does not use augmenting paths, and instead adds flow one edge at a time.
   ii. The algorithm **does** work even if there are edges whose capacities are irrational numbers.

c. Complexity:

i. The total complexity is $\boldsymbol{O(|V|^2|E|)}$, which is generally better than that of Edmonds-Karp.

IV. Example Problem 1 – Max-Flow (Straightforward)

We will do the following problem from the course website.

Problem 16: Job Assignment

You are given a set of n jobs and a set of m machines. For each job you are given a list of machines capable of performing the job. An assignment specifies for each job one of the machines capable of performing it. The overhead for an assignment is the maximum number of jobs performed by the same machine. The job assignment problem is: given such lists and an integer $1 \leq k \leq n$, is there an assignment with overhead at most k?

Input: n jobs and m machines, compatibility lists for each machine
Output: Yes/no: is it possible to complete all n jobs?

We should start by listing key properties of the problem.

- n jobs
- m machines
- Each machine can handle at most k jobs, where $1 \leq k \leq n$.
- Each machine can do some subset of the jobs.

This suggests a straight-forward matching problem, and thus a max-flow problem.

Nodes:

- s, t
- m nodes representing the machines: $m_i, \forall i \in \{1,2,\dots,m\}$
- n nodes representing the jobs: $j_j, \forall j \in \{1,2,\dots,n\}$

Edges:

- $s \to m_i, \forall i \in \{1,2,\dots,m\}$ with capacity k, which ensures that each machine can have at most k incoming flow.
- $j_j \to t, \forall j \in \{1,2,\dots,n\}$ with capacity 1, which ensures that each job can be done at most once.
- $m_i \to j_j, \forall i, j$ where machine i can do job j, with capacity 1. This ensures that any given machine can only contribute to jobs that it can do, and that it can only do any given job once.

Solution: If the max-flow has value equal to n, then the assignment exists. Otherwise, the assignment does not exist. As matchings, output the $m_i \to j_j$ edges that have flow.

Complexity/Algorithm:

- There are $O(m+n)$ nodes.
- There are $O(mn)$ edges.
- $C = n$, as it is limited by the number incoming flow capacity of t.
- Because $C \leq n \leq mn$, **Ford-Fulkerson** appears to be a good option. Ford-Fulkerson has complexity $O\big((|V|+|E|)C\big)$, so the complexity is $O\big((m+mn)n\big) = O(mn+mn^2) = \boldsymbol{O(mn^2)}$.

Correctness:

1. Machines can have incoming flow at most k, thus ensuring that they can't be overworked.
2. Jobs have outgoing flow at most 1, thus ensuring that each job can be done at most once.
3. Machine→job edges exist only if the machine can do the job, so incompatible matches won't arise.

4. Each matched machine contributes a value of 1, which ensures the following:
   - If the max-flow has value n, then that means all jobs were matched to some machine.
   - If all jobs were matched to some machine, then the max-flow has value n.
   - Each job was matched with at most one machine and, if the max-flow has value n, then exactly one machine. Thus, the saturated $m_i \to j_j$ edges correctly correspond to the matchings.
5. By the three points of #4, our algorithm is correct.

V. Example Problem 2 – Max-Flow (Trickier)

We will do the following problem from the course website:

Problem 6: Domino Tiling

You are given a subset of the $n \times n$ grid of squares, described as an $n \times n$ boolean matrix. The question is whether it is possible to cover precisely this subset with 1×2 rectangular dominoes, subject to the following constraints: Each domino must be placed either horizontally or vertically, covering two adjacent squares of the grid. Both squares must be in the subset. No two dominoes can cover the same square. Each square in the subset must be covered by some domino.

Give an efficient algorithm for deciding whether it is possible, given as input the array of booleans describing the subset True meaning the square is in the subset, False meaning it is not.

Input: The grid size n, a list of squares that must be covered by dominoes
Output: Yes/no: Is it possible to cover all requisite squares with dominoes?

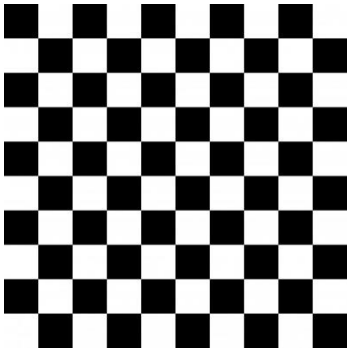

Figure 9. A checkerboard. For reasons that will soon become apparent, this will be our visualization of the grid.

We will again start by listing our understanding of the problem.

1. There are n^2 squares.
2. Some subset of the squares must be covered. Let m be the number of such squares.
   a. Each domino covers exactly 2 squares. Thus, in order for the problem to be solvable, m must be even; otherwise, we can immediately conclude that there are no solutions.
3. Each tile, if covered, is covered alongside exactly one adjacent tile.
   a. The key observation: each domino will cover *exactly one* black tile and *exactly one* white tile.
   b. Now, we see this as a bipartite matching problem: we match each black tile to one of its adjacent white tiles. As a matching problem, it can be solved with max-flow.

Nodes:

- s, t
- $\frac{m}{2}$ nodes representing the black tiles that must be covered: $b_i, \forall i \in \left\{1,2,\dots,\frac{m}{2}\right\}$
- $\frac{m}{2}$ nodes representing the white tiles that must be covered: $w_j, \forall j \in \left\{1,2,\dots,\frac{m}{2}\right\}$

Edges:

- $s \to b_i, \forall i \in \left\{1,2,\dots,\frac{m}{2}\right\}$ with capacity 1. Each black tile must be matched.

- $w_j \to t, \forall j \in \left\{1,2,\dots,\frac{m}{2}\right\}$ with capacity 1. Each white tile must be matched.
- $b_i \to w_j, \forall i,j$ where black i is adjacent to white j, with capacity 1.

Solution: If m is odd, then there is no solution. Otherwise, if the max-flow has value $\frac{m}{2}$, then we have a solution; if not, we don't. If we do have a solution: cover together the tiles for which $b_i \to w_j$ has flow.

Complexity/Algorithm:
- There are $O(m)$ nodes.
- There are $O(m)$ edges, as each black node has up to 4 outgoing edges.
- $C = \frac{m}{2}$ since the max-flow is limited by the number of pairs.
- Again, C is small relative to $|V|$ and $|E|$, so **Ford-Fulkerson** will work well. Our total complexity is $O\left((m+m)\frac{m}{2}\right) = \boldsymbol{O(m^2)}$.

Correctness:
1. Black tiles have incoming flow at most 1, thus ensuring that they're matched at most once.
2. White tiles have outgoing flow at most 1, thus ensuring that they're matched at most once.
3. Black→White edges exist only if the tiles are adjacent, so incompatible matches won't arise.
4. Each match contributes a value of 1, which ensures the following:
   - If the max-flow has value $\frac{m}{2}$, then that means there are $\frac{m}{2}$ pairs and thus all m tiles are covered.
   - If all tiles are covered (matched), then the max-flow has value $\frac{m}{2}$.
   - Each black tile was matched with at most one tile and, if the max-flow has value $\frac{m}{2}$, then exactly one white tile. Thus, the saturated $m_i \to j_j$ edges correctly correspond to the matchings.
5. By the three points of #4, our algorithm is correct.

VI. Example Problem 3 – Min-Cut

We will do the following problem from the course website.

Problem 10: Electronic message transmission systems

Consider the problem of selecting sites for an electronic message transmission system. Any number of sites can be chosen from a finite set of potential locations. We know the cost c_i of establishing site i and the revenue r_{ij} generated between sites i and j, if they are both selected. Both the costs and revenues are non-negative. Find an efficient algorithm to determine the subset of vertices such that the sum of the edge revenues less the vertex costs is as large as possible.

Design an efficient polynomial-time algorithm.

Provide a high-level description of your algorithm, prove its correctness, and analyze its time complexity.

Input: n sites with associated costs $c_i \forall i \in \{1,2,\dots,n\}$; n^2 revenue values – one for each pair of sites.
Output: Subset of sites that maximizes total profit (revenues minus costs)

We will again start by listing our understanding of the problem.
1. Let n be the total number of sites.
2. Site i costs c_i to select.
3. If sites i and j are both selected, then we gain r_{ij} revenue.

This is quite similar to the project selection problem, and thus we see suspect it to be a min-cut problem. We will thus set up our flow network correspondingly and see if that works.

Nodes:

- s, t
- n nodes representing sites: $S_i, \forall i \in \{1,2,\dots,n\}$
- n^2 nodes representing revenue: $R_{ij}, \forall i \in \{1,2,\dots,n\}, j \in \{1,2,\dots,n\}$

I used capital letters for nodes in order to avoid confusion with other variables in the problem.

Edges:

- $s \to R_{ij}, \forall i \in \{1,2,\dots,n\}, j \in \{1,2,\dots,n\}$, with capacity r_{ij}
- $R_{ij} \to S_i$ and $R_{ij} \to S_j, \forall i \in \{1,2,\dots,n\}, j \in \{1,2,\dots,n\}$, with capacity ∞. We do not want these edges to be cross the cut.
- $S_i \to t, \forall i \in \{1,2,\dots,n\}$, with capacity c_i

Solution: Select all sites for which $S_i \to t$ is saturated (i.e., $S_i \in A$).

Complexity/Algorithm:

- There are $O(n^2)$ nodes.
- There are also $O(n^2)$ edges, but we expect the exact number of edges to be greater than the exact number of nodes.
- C can be arbitrarily large.
- Because we don't have a bound on C and the problem specifically asks for a polynomial-time algorithm, we don't want something that depends on C. We are thus limited to Edmonds-Karp and preflow-push. With our knowledge of the problem, we see that **preflow-push** scales better (albeit with the same complexity), so our total complexity is $O((n^2)^2 n^2) = \boldsymbol{O(n^6)}$.

Correctness:

1. Consider the types of edges crossing the min-cut.
   a. $s \to R_{ij}$: R_{ij} is in B. Let $\sum_{ij \notin A} R_{ij}$ be these nodes' contribution to the value of the cut.
   b. $S_i \to t$: S_i is in A. Let $\sum_{i \in A} S_i$ be these nodes' contribution to the value of the cut.
2. Let C be the capacity of the cut. Thus, $C = \sum_{ij \notin A} R_{ij} + \sum_{i \in A} S_i$.
   a. Rewrite: $\sum_{i \in A} S_i = C - \sum_{ij \notin A} R_{ij}$
3. R_{ij}, for any i, j, will only end up in A if both of i, j are in A (since the intermediate edges have infinite capacity. Thus, our gross profit is $\sum_{ij \in A} R_{ij}$.
4. Our cost is $\sum_{i \in A} S_i$, as we claimed that these are the sites that we choose.
5. The total profit is $T = \sum_{ij \in A} R_{ij} - \sum_{i \in A} S_i$; we will manipulate this equation.
   a. $T = \sum_{ij \in A} R_{ij} - \left(C - \sum_{ij \notin A} R_{ij}\right)$
   b. $T = \sum_{ij \in A} R_{ij} + \sum_{ij \notin A} R_{ij} - C = \left(\sum_{ij \in A} R_{ij} + \sum_{ij \notin A} R_{ij}\right) - C$
   c. $\boldsymbol{T = \sum_{ij} R_{ij} - C}$
6. $\sum_{ij} R_{ij}$ is a constant for this network, so we see now that by minimizing this cut, we maximize the profit. Our formulation of the flow network is therefore correct.

References

1. Kleinberg, Jon, and Éva Tardos. *Algorithm Design*. Boston: Pearson/Addison-Wesley, 2006. Print.