Contents

1 Project Specification

In class, we studied the problem of finding maximum flows for source-sink network. Primarily, we discussed the Ford-Fulkerson algorithm (Chapter 24 of CLRS). Below is the algorithm, with a bit of detail added:

Ford-Fulkerson-Method(G, s, t)
// Inputs: G - Graph; s - source node; t - sink node
initialize flow f to 0
while there exists an augmenting path p in the residual graph G_f **do**
determine the maximum flow δ that can be pushed along p
augment the flow f along the path p by δ
update G_f
return f and its total cost

For this project, you will consider the following generalization of the maximum-flow problem (from Section 29.2 of CLRS). Suppose that, in addition to a capacity $c(u, v)$ for each edge (u, v), you are given a real-valued unit cost $a(u, v)$ (i.e., cost per unit flow). As in the maximum-flow problem, assume that $c(u, v) = 0$ if $(u, v) \notin E$ and that there are no anti-parallel edges. If you send f_{uv} units of flow over edge (u, v), you incur a cost of $a(u, v) \cdot f_{uv}$. You are also given a flow demand d (assume this to be less than or equal to the maximum flow of the network as determined by Fork-Fulkerson). You wish to send d units of flow from s to t while minimizing the total cost

$$\sum_{(u,v)\in E} a(u, v) \cdot f_{uv}$$

incurred by the flow. This problem is known as the ***minimum-cost-flow problem***.

Figure 1(a) shows an example of a minimum-cost-flow problem. For each edge, the capacities are denoted by c and unit costs by a. Vertex s is the source, and vertex t is the sink. The goal is

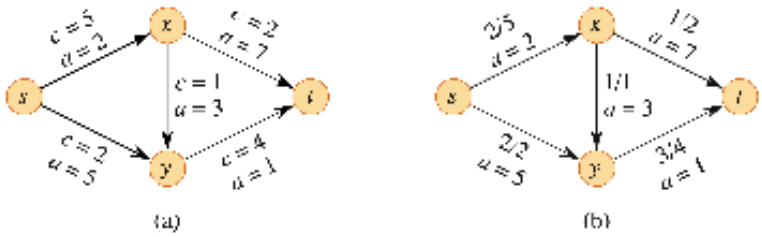

Figure 1: An example of a minimum-cost-flow problem.

to send 4 units of flow from s to t while incurring the minimum total cost. Figure 1(b) shows an optimal solution to the minimum-cost flow problem in which 4 units of flow are sent from s to t. For each edge, the flow and capacity are written as flow/capacity. The total cost is

$$\sum_{(u,v)\in E} a(u,v) \cdot f_{uv} = (2 \cdot 2) + (5 \cdot 2) + (3 \cdot 1) + (7 \cdot 1) + (1 \cdot 3) = 27.$$

A typical application of the minimum-cost flow problem is the commodity shipping problem. For example, a company wants to ship a certain amount of a commodity from its main warehouse to, say, a central distribution center. The commodity can be shipped along various routes such as road, rail, air and shipping, of different capacities. The goal is the find a route with enough capacity to handle the amount of commodity while minimizing the overall shipping cost.

The CLRS textbook discusses solving this problem with linear programming in Chapter 29. You will used linear programming for small test graphs as part of the verification of the correctness of your algorithms. The required linear program is discussed in Section 29.2 of CLRS. Instead of using linear programming, for this project, you will use adaptations of the Ford-Fulkerson algorithm to solve the minimum-cost flow problem.

The Ford-Fulkerson method, above, is primarily used for finding maximum flow in a source-sink flow network. However, to find a minimum-cost flow, variations of the Ford-Fulkerson method, such as the Successive Shortest Path algorithm, Cycle-Canceling algorithm, or Cost-Scaling algorithm are used. These algorithms adjust the Ford-Fulkerson method to not only look for feasible flows but also ensure the total cost is minimized. You will implement these adaptations in parts, and study their performances on randomly generated source-sink graphs.

In this project, as a team, you will do the following:

1. You will implement a random graph generator to create a directed weighted geometric (Euclidean) source-sink graphs having certain properties, which will be stored in an external file using the ASCII-based EDGES format.
2. This project will use eight (8) randomly generated graphs using your random graph generator with given expected properties.
3. You will determine the largest connected component (LCC) of each graph, and choose (algorithmically) a source and sink for each LCC.
4. You will then run three algorithms (described below) on the LCC of each graph, generating several metrics of your results.
5. You will also pick another algorithm other than the three described in this document, implement it, run it on the LCC of each graph, and record the resulting metrics.

6. The characteristics of the randomly generated graphs and the metrics for all these algorithms will be entered into tables, as shown below.

You may use a programming language of your choice, such as Python, Java, or C++, to create a functional implementation of the required algorithms.

1.1 Required implementations:

- Random Weighted Directed Euclidean Source-Sink Graph Generator

  For this project, you will need to randomly generate Euclidean neighbor graphs for the simulations to explore the behaviour of your different heuristic LSP algorithms.

  Consider the following pseudocode to generate a graph of n nodes with maximum distance r between node neighbors. After creating the n nodes, each node is assigned random (x, y) coordinates between 0 and 1. For each pair of vertices u and v within distance r, edge (u, v) or (v, u) (not both) is added to E with a probability of about 80% (some neighbors within r distance will not be connected by an edge). This creates a directed Euclidean neighbor graph.

```
GENERATE_SINK_SOURCE_GRAPH(n, r, upperCap, upperCost)
  // n is number of vertices, r is maximum distance between nodes sharing an edge,
  // upperCap is maximum capacity value, and upperCost is maximum unit cost value
  Define a set of vertices V such that |V| = n
  // assign each node in V random Cartesian coordinates (x, y) as follows
  for each vertex u ∈ V do
      u.x ← a uniform random number in [0..1]
      u.y ← a uniform random number in [0..1]
  // randomly assign edges of length ≤ r without creating parallel edges
  for each vertex u ∈ V do
      for each vertex v ∈ V do
          if (u ≠ v) && (u.x − v.x)^2 + (u.y − v.y)^2 ≤ r^2 then
              rand ← a uniform random number in [0..1]
              if rand < 0.3 then
                  if ((u, v) ∉ E) && ((v, u) ∉ E) then
                      E ← E ∪ {(u, v)}
              else if rand < 0.6 then
                  if ((u, v) ∉ E) && ((v, u) ∉ E) then
                      E ← E ∪ {(v, u)}
              else
                  // add no edge
  for each edge (u, v) ∈ E do
      (u, v).cap ← a uniform random integer in [1..upperCap]
      (u, v).unitcost ← a uniform random integer in [1..upperCost]
```

  Generate one graph for each set of n, r, $upperCap$ and $upperCost$ values (detailed instructions below on how to choose these parameters) and store the graph in an EDGES format file. This is an ASCII readable edge list format storing a graph as node pairs with no data, e.g., the directed edge $(1, 2)$ is stored on one line of the file as:

```
1 2
```

You can also extend this representation to also store the capacity and unit cost of an edge. For example, the directed edge $(1, 2)$ with capacity of 5 and unit cost of 3 can be stored on one line of the file as:

```
1 2 5 3
```

Note that this format cannot directly store isolated nodes with no neighbors. When running the simulations below, read in the stored Euclidean graph from the file to then use it.

- Minimum-Cost Flow Algorithms based on the Ford-Fulkerson Method

  Below are three algorithms that explore different aspects of a solution to the minimum-cost flow problem, that you will use for your simulation results. Some implementation details are not given (e.g. relating to calculating minimum-cost augumenting paths) that you will need to figure out yourself.

  Algorithm 1: Successive Shortest Path Algorithm

  This algorithm is simply Ford-Fulkerson using a short path algorithm such as Bellman-Ford to find the augmenting path with minimum cost at each iteration, until no augmenting paths exist in the residual graph.

SuccessiveShortestPaths(G, s, t, d)
  // Inputs: G - Graph; s - source node; t - sink node; d - required total flow
  initialize flow f to 0
  while $(d > 0)$ and (there exists an augmenting path p in the residual graph G_f) **do**
    find the minimum-cost path p' from s to t
    // using, e.g., Bellman-Ford algorithm using the unit cost of the edges instead of capacity
    // (all the edges of the minimum-cost path must have positive capacity)
    determine the maximum flow δ that can be pushed along p'
    if $\delta > d$ **then**
      $\delta \leftarrow d$ // we don't want f to be larger than the required total flow
    augment the flow f along the path p' by δ
    update G_f
    $d = d - \delta$
  if $d > 0$ **then**
    return null and -1 // failure: f is less than required total flow
  else
    return f and its total cost

Algorithm 2: Capacity Scaling Algorithm

This algorithm uses capacity scaling to improve the efficiency of the Successive Shortest Path algorithm.

Scale capacities and progressively refine the solution. The below algorithm uses a shortest path, but using DFS would work as well.

A nice YouTube video by William Fiset describing Capacity Scaling is here:
https://www.youtube.com/watch?v=1ewLrXUz4kk

CapacityScaling(G, s, t, d)
  // Inputs: G - Graph; s - source node; t - sink node; d - required total flow
  initialize flow f to 0
  set scaling factor Δ to maximum capacity ($upperCap$) of any edge in G
  while $\Delta \geq 1$ **do**
    while $(d > 0)$ and (there exists an augmenting path p in the residual graph G_f
        with residual capacity at least Δ) **do**
      find the shortest path p' from s to t with capacity at least Δ
      // (all the edges of the shortest path must have capacity $\geq \Delta$)
      determine the maximum flow δ that can be pushed along p'
      if $\delta > d$ **then**
        $\delta \leftarrow d$ // we don't want f to be larger than the required total flow
      augment the flow f along the path p' by δ
      update G_f
      $d = d - \delta$
    $\Delta \leftarrow \Delta/2$
  if $d > 0$ **then**
    return null and -1 // failure: f is less than required total flow
  else
    return f and its total cost

Algorithm 3: Integration of Algorithms 1 and 2

This algorithm uses capacity scaling to improve the efficiency of the Successive Shortest Path algorithm.

SuccessiveShortestPathsSC(G, s, t, d)
  // Inputs: G - Graph; s - source node; t - sink node; d - required total flow
  initialize flow f to 0
  set scaling factor Δ to maximum capacity ($upperCap$) of any edge in G
  while $\Delta \geq 1$ **do**
    while $(d > 0)$ and (there exists an augmenting path p in the residual graph G_f
        with residual capacity at least Δ) **do**
      find the minimum-cost path p' from s to t
      // using, e.g., Bellman-Ford algorithm using the unit cost of the edges
      // (all the edges of the shortest path must have capacity $\geq \Delta$)
      determine the maximum flow δ that can be pushed along p'
      if $\delta > d$ **then**
        $\delta \leftarrow d$ // we don't want f to be larger than the required total flow
      augment the flow f along the path p' by δ
      update G_f
      $d = d - \delta$
    $\Delta \leftarrow \Delta/2$
  if $d > 0$ **then**
    return null and -1 // failure: f is less than required total flow
  else
    return f and its total cost

Algorithm 4

Pick another Minimum-Cost Flow algorithm based on Ford-Fulkerson other than one of the three described above. Choices include, amoung other choices:

- Cycle-Canceling Algorithm
- Successive Augmentation Algorithm
- Out-of-Kilter Algorithm
- Primal-Dual Algorithm

Implement this algorithm as Algorithm 4.

1.2 Simulations and Results

Define the following metrics to be computed (one set of values for each heuristic and graph combination, to be presented in tables; see below):

- n: number of nodes in graph G, or $n = |V|$
- $|V_{LCC}|$: number of nodes in the largest connected component, LCC, of G
- $\Delta_{\text{out}}(LCC)$: the maximum out-degree of any node in $\|LCC\|$, or

$$\Delta_{\text{out}}(LCC) = \max_{v \in LCC} k_{\text{out}}(v)$$

where $k_{\text{out}}(v)$ is the out-degree of node v

- $\Delta_{\text{in}}(LCC)$: the maximum in-degree of any node in $\|LCC\|$, or

$$\Delta_{\text{in}}(LCC) = \max_{v \in LCC} k_{\text{in}}(v)$$

where $k_{\text{in}}(v)$ is the in-degree of node v

- $\overline{k(LCC)}$: the average degree of nodes in $\|LCC\|$, or

$$\overline{k(LCC)} = \frac{1}{|V_{LCC}|} \sum_{v \in LCC} (k_{\text{out}}(v) + k_{\text{in}}(v)) = \frac{|E_{LCC}|}{|V_{LCC}|}$$

where $|E_{LCC}|$ is the number of (directed) edges in the largest connected component, LCC, of G

- minimum cost (MC): total cost of the flow determined by the Minimum-Cost Flow algorithm based on Ford-Fulkerson
- flow (f): total flow determined by the Minimum-Cost Flow algorithm based on Ford-Fulkerson
- paths: the number of augmenting paths required until the Minimum-Cost Flow algorithm based on Ford-Fulkerson completes
- mean length (ML): average length (i.e., number of edges) of the augmenting paths
- mean proportional length (MPL): the average length of the augmenting path as a fraction of the longest acyclic path from s to t

Simulations I

For this project, first you have to determine eight source-sink networks for certain n, r, $upperCap$, and $upperCost$ values:

1. $n = 100$, $r = 0.2$, $upperCap = 8$, $upperCost = 5$
2. $n = 200$, $r = 0.2$, $upperCap = 8$, $upperCost = 5$
3. $n = 100$, $r = 0.3$, $upperCap = 8$, $upperCost = 5$
4. $n = 200$, $r = 0.3$, $upperCap = 8$, $upperCost = 5$
5. $n = 100$, $r = 0.2$, $upperCap = 64$, $upperCost = 20$
6. $n = 200$, $r = 0.2$, $upperCap = 64$, $upperCost = 20$
7. $n = 100$, $r = 0.3$, $upperCap = 64$, $upperCost = 20$
8. $n = 200$, $r = 0.3$, $upperCap = 64$, $upperCost = 20$

To determine the network to use for each set of parameters above, for each node of the randomly generated directed graph consider the size of the connected component that can be reached from that node, using, for example, depth-first search (DFS) (§20.3 of CLRS). For a largest connected component, use the "starting" node (of the DFS, say) associated with LCC as the start node s for that graph. Using Breadth-First Search, or similar shortest-path algorithm, determine the node that is furthest away from s (e.g., has the longest shortest-path in terms of number hops), and use this node as the sink node t.

With each selected graph, run an implementation of Ford-Fulkerson, such as Edmonds-Karp, to determine the maximum flow, $f_{\max}$, for that network.

For the final chosen graph for each set of parameters above, fill a table such as Table 1 with the characteristics of each graph.

Graph	n	r	upperCap	upperCost	$f_{\max}$	$\lvert V_{LCC}\rvert$	$\Delta_{\text{out}}(LCC)$	$\Delta_{\text{in}}(LCC)$	$\overline{k(LCC)}$
1									
2									
3									
4									
5									
6									
7									
8									

Table 1: Example results table for random graph characteristics

To ensure each algorithm completes on each network, we need to set d to be within the maximum flow possible for the network. So, set d to be something like $0.95 \cdot f_{\max}$ (your choice). When the minimum-cost flow algorithm completes, f should equal d. Record the results of running your four algorithms on the eight source-sink networks in a table such as Table 2.

Algorithm	Graph	f	MC	paths	ML	MPL
SSP	1					
CS	1					
SSPCS	1					
YOURS	1					
SSP	2					
CS	2					
SSPCS	2					
YOURS	2					
⋮						

Table 2: Example results table for each algorithm for the eight random graphs. SSP = SuccessiveShortestPaths; CS = CapacityScaling; SSPCS = SuccessiveShortestPathsSC; YOURS = your choice of Algorithm 4.

Simulations II

Consider the results you found in the *Simulations I* part. Consider the differences between the Minimum-Cost Flow algorithms. Propose your own set, or two, of n, r, *upperCap*, and *upperCost* values that highlight/emphasis these differences. Generate your source-sink graph(s) based on your chosen values and perform simulations of the four augmenting path algorithms on your source-sink network(s). Again, record the characteristics of your generated source-sink graph(s) in a table like Table 1 and the results of the four algorithms on these graph(s) in a table like Table 2.

2 Project Deliverables

1. A project report, submitted in PDF format. This report should be concise and clearly written and include the following sections:

   - Problem description
     In your words, what is the purpose of this project?
   - Implementation details
     Give details of any of the main function implementations **including pseudocode**. Repeating back the instruction sheet will only get part marks.
   - Implementation correctness
     How your tested the correctness of your implementations, both in terms of software correctness (e.g., boundary testing, etc.) but also that your graph algorithms are correct. For example, demonstrate the correctness of minimum-cost flow found, assuming $d = f_{\max}$, on relatively small (not too small!) example graphs where the output can be verified independently using linear programming – see Section 29.2 of CLRS.
   - Results
     Complete the tables as requested and including explanation of any choices you made during your simulations and how they affected the results.
   - Conclusions
     What overall conclusions can you draw from the results of your simulations?
   - Team work distribution
     In this part, clearly write down how the work was distributed to each member for evaluation purposes.
   - References
     Any references used to implement the algorithms in your project. You must not "cut'n'paste" code from any of sources (including LLMs).

2. A zip file containing:

   - ASCII encodings of all your source-sink graphs as input to your simulation code, one file per graph.
   - Your implementation code including a README to explain how to compile and run.