CSC 395 Homework 5 Spring 2024

Due: Friday, April 26

In this homework assignment, you will be focusing on Linear Programming and Network Flow algorithms. In the first part of the assignment, you will utilize `scipy`'s build in Simplex algorithm to solve the simple LP from class, to get some practice with both using matrix-vector notation for LPs and interacting with built-in numerical packages for python. In the second part of the assignment, you will be implementing your own code to solve Network Flow problems. Your solutions should go in the files `hw5p1.py` and `hw5p2.py`, which will be submitted through the course gradescope.

Part 1. **Using Simplex**

For the first part of this assignment, you will use `scipy`'s built in LP solver to solve the simple problem from class.

You should begin by familiarizing yourself with the function you will be using: `scipy.optimize.linprog`.

The documentation for this function can be found at the link below:

https://docs.scipy.org/doc/scipy/reference/optimize.linprog-simplex.html

As you read through this documentation, pay close attention to the format of the inputs and how they encode the objective function and constraints for the LP: this function uses matrix-vector notation.

Next, recall the LP example from class (with the sprockets):

We have a company that produces sprockets to sell, but the demand for sprockets is highly seasonal, making it difficult to plan our production. The company analyst has obtained demand estimates for the months of the upcoming year (d_1, d_2, ..., d_{12}), and they are very uneven, ranging from as little as 440 per month to as much as 920 per month. We would like to exactly meet these production demands at the lowest possible cost.

Currently our company has a workforce of 30 employees who each make $2,000 a month, and who each produce 20 sprockets per month.

Clearly this is more than sufficient for some months, but will not be enough for other months. We have 3 alternative strategies for handling the changing demand:

- We can pay workers overtime to produce extra sprockets. However, workers get paid an extra 80% wages for overtime work, and each worker is limited to 30% extra work time.
- We can hire and fire workers as needed, but there are associated HR and governmental costs to doing this: it costs $320 to hire a new worker and $400 to fire a current worker.

- We can store surplus sprockets from one month to the next, but this will cost \$8 per stored sprocket per month. We currently have no sprockets stored, and we want to avoid having any excess sprockets at the end of the year.

To formulate this as a linear program, we define the following variables:

- w_i = the number of workers during the ith month; $w_0 = 30$.
- x_i = the number of sprockets made during the ith month.
- o_i = the number of overtime sprockets in the ith month.
- h_i = the number of workers hired in the ith month.
- f_i = the number of workers fired in the ith month.
- s_i = the number of sprockets stored at end of month i; $s_0 = 0$.

With these variables, we obtain the following objective function. The terms of this expression in order are: employee salaries, hiring costs, firing costs, storage costs, overtime costs (where each worker is paid \$180 per sprocket instead of the standard \$100 per sprocket).

$$cost = 2000 \cdot \Sigma\, w_i + 320 \cdot \Sigma\, h_i + 400 \cdot \Sigma\, f_i + 8 \cdot \Sigma\, s_i + 180 \cdot \Sigma\, o_i$$

Our goal will be to minimize this cost function subject to the constraints of our problem.

So our next step is to formally define those constraints:

- First, all variables must be nonnegative:

  $$w_i,\, x_i,\, o_i,\, h_i,\, f_i,\, s_i \geq 0, \quad i = 1, 2, \ldots, 12.$$

- The total sprockets made per month (x_i) is regular amount (20 per worker) plus overtime (o_i):

  $$x_i = 20w_i + o_i, \quad i = 1, 2, \ldots, 12.$$

- The number of workers can change at the start of each month:

  $$w_i = w_{i-1} + h_i - f_i, \quad i = 1, 2, \ldots, 12.$$

- The number of stored sprockets is the amount stored from the previous month (s_{i-1}), plus the number we made (x_i), minus the number we sold (d_i):

  $$s_i = s_{i-1} + x_i - d_i, \quad i = 1, 2, \ldots, 12.$$

  Overtime is limited (30% of 20 sprockets per worker per month):

  $$o_i \leq 6w_i, \quad i = 1, 2, \ldots, 12.$$

Now that we have the full problem defined, it is time for Simplex! In the file `hw5p1.py` you should import the necessary `scipy` library, create the vectors and matrices to represent this LP, and call the `linprog` function to obtain a result. Your result will be in the form of a vector with something like 72 entries. Please take a couple minutes to write a comment at the bottom of your file that includes the vector, as well as providing a short summary (i.e., try to make it easy to read how many workers for each month, how many sprockets made in each month, how many overtime sprockets made in each month, etc.). Additionally, please briefly comment on the quality of your solution: is this a good solution for this problem?

Part 2. **Implementing Edmonds-Karp**

In class we introduced the classic Network Flow problem: Given a directed, weighted graph G along with a specific starting node s and destination node t, find the maximum possible *flow* from s to t. Each edge in the graph is assigned an amount of flow, which cannot exceed its designated capacity (the weight of the edge in the graph G). Flow starts at s and arrives at t. For all other vertices, we have a conservation of flow: the flow into a vertex v is equal to the flow out of the vertex.

To solve this problem of finding the maximum flow in the graph, we turn first to the general Ford-Fulkerson algorithm:

- Create what we call the *residual graph*, which starts as a copy of graph G. All modifications described below will be performed on this residual graph.
- Define an *augmenting path* to be a path from s to t that still has remaining capacity, i.e., each edge has nonzero capacity remaining.
- Find any augmenting path in the current residual graph.
- Compute the maximum flow that can possibly be pushed along that path (which is equal to the capacity of the minimum edge along the path, since that edge will act as a bottleneck for the flow).
- Push that flow along the path. For every edge e along the path:
  - Decrease the weight of edge e by the amount of pushed flow.
  - If e gets 'filled' and has its weight set to 0, it should be removed from the graph.
  - If there currently exists a reverse edge for e, increase that reverse edge's weight by the amount of pushed flow (allowing us to 'regret' the choice and undo it later).
  - If there does not currently exist a reverse edge for e, add it into the graph and set its weight equal to the amount of pushed flow.
- Repeat this process until there are no longer any augmenting paths in the residual graph.

It is common to also create a copy of graph G where all edges are initially removed. As the algorithm progresses and flow is pushed along augmenting paths, the flow is recorded in this *flow graph*. Importantly, there are no reverse edges allowed here. If we choose to push flow in the opposite direction as a previous choice (i.e., if we are about to add an edge that already has a reverse edge in the flow graph), we instead cancel the two edges: the one with the larger weight remains with its weight decreased by the amount of the reverse edge.

Under some reasonable conditions about the graph G, this process is guarenteed to find the maximum flow in the graph, and will terminate in at most $C \cdot |E|$ iterations, where $|E|$ is the number of edges and C is the maximum capacity (edge weight) in the graph.

This runtime is undesirable, and it turns out that we can do better. The algorithm known as Edmonds-Karp is able to promise termination in at most $O(|V| \cdot |E|)$ iterations (where $|V|$ is the number of vertices). It obtains this bound with a slight modification to the original algorithm: **when searching for the next augmenting path in the residual graph, use breadth-first search.**

For this second part of the assignment, your task is to implement Edmonds-Karp to solve a network flow problem given an input directed, weighted graph G.

For this task, you will be using the `networkx` package in `python`. You can find the documentation at the following link:

https://networkx.org/documentation/stable/reference/introduction.html

There is also a tutorial at the following link, which I strongly recommend you read over before starting this assignment. You will need to create your own graphs which include info for edges and nodes and update the edges as your algorithm progresses, so it will be important to read into how to add edge/node data, how to add/remove edges from the graph, and how to find out if a specific edge currently exists in the graph.

https://networkx.org/documentation/stable/tutorial.html

Once you have familiarized yourself with the `networkx` package, it is time to start working on your Edmonds-Karp implementation. You should put your submission in the file `hw5p2.py` to be submitted on gradescope.

As you write your code, you will likely want to introduce a number of helper functions. In particular, I would suggest separate functions for implementing BFS to find an augmenting path, finding the flow to push on that augmenting path, actually pushing the flow in the residual graph, and correctly recording the flow in the flow graph. Your BFS code will need to be able to recognize when it has failed to find an augmenting path, and signal to your Edmonds-Karp implementation that it has reached termination.

If you are not familiar with BFS, the wikipedia page has some useful info along with pseudocode demonstrating its implementation. Note that once you have found the goal vertex t, you will have to retrace the *parent* info to recreate the augmenting path (or find that t's *parent* info was not updating, meaning that there was no path).

https://en.wikipedia.org/wiki/Breadth-first_search

Note that `python` lists can act like the queue that you will need for BFS:

- The queue can be initialized as an empty list.
- The `append` method can act like an enqueue operation: pushing an element into the rear of the queue.
- Calling `lst.pop(0)` will remove and return the element at the front of the list `lst`, so this can function as a dequeue operation that removes an element at the front of the queue.

As a final note on your BFS implementation: you will need to repetitively find paths in the same graph object, but the BFS algorithm requires that you update *parent* and *visited/explored* data for each node for each run of BFS. You should be careful to reset these

values each time you start BFS, to avoid bugs where information from previous paths is inadvertently re-used.

Once you have completed your implementation of Edmonds-Karp, you should create the graph from the example provided in class (and the pdf reading), and use your code to find the maximum flow in that graph. Check over your result (it will help to draw a picture of the graph by hand), and convince yourself that your code has successfully found the maximum flow in the graph.

Submit both your `hw5p1.py` and `hw5p2.py` to the course gradescope.