Take Home Project
Kinetic

Russell Spiewak

December 2, 2024

Introduction

This document contains the assumptions and analysis for the two problems on the take home project for the Research Engineer position at Kinetic. The text and ideas in this document are governed by the same license as found in the accompanying repository (https://github.com/r-spiewak/kinetic_project). This document is found in the repository at this location.

1 Graphs

1.1 Problem Statement

We have a simple connected graph G with vertices V. Given an arbitrary $v \in V$ we would like to find all directed subgraphs g of G where: (1) v is in g, (2) no vertex is a source or a sink, (3) an edge in G appears at most once in g, i.e., g is an oriented graph, and (4) the number of edges in g is less than k and greater than 0.

Demonstrate your algorithm with $k = 10$ on a random graph with 10 vertices and 20 edges. If you can't enumerate all valid subgraphs, find as many as you can. Explain if your solution is practical for much larger graphs and how it could be improved.

It may be fruitful to think in terms of directed cycles and their unions.

1.2 Analysis

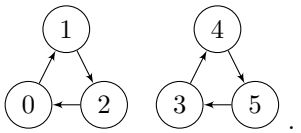
A graph $G = (V, E)$. A graph may have self-loops (in which a single vertex has an outgoing edge leading directly back into itself). A graph may also be directed or undirected. An undirected graph can be considered as a directed graph in which every edge from vertex i to vertex j has a corresponding reverse edge from vertex j to vertex i.

Here, the Problem Statement states $v \in V$, which, according to the strict mathematical definition of $\in$, indicates that v is a single vertex. As such, we will proceed under the assumption that $g = (v_g, e_g)$, and that $v \in v_g$.

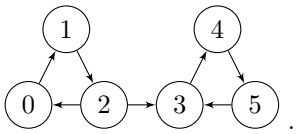
Here it will be noted that in a graph $g = (v_g, e_g)$ a set of vertices v_g may contain a group of vertices v_s where $v \notin v_s$ that is connected to the rest of the graph by a single edge. If , after further filtering, this set of vertices v_s does not contain any vertex of interest (e.g., the purpose of this implementation is to find paths from v to some other $v_i \in v_g$ where $v, v_i \notin v_s$), this set of vertices v_s can be safely pruned from g (partially or in its entirety) and be functionally equivalent to g. These two (or more) functionally equivalent graphs may be regarded as the same g in "all directed subgraphs" in the Problem Statement, or as separate g. However, pruning the graphs in such a way and finding such equivalencies would take extra computation, and, while possibly useful for further steps in filtering subgraphs (see example above), would not be within the scope of this problem. As such, here they will be regarded as separate g if that is easiest.

Similarly, the Problem Statement does not specify that the subgraphs g must be fully connected, or even that the vertex v that is contained in the subgraph g must be connected within the subgraph (this is even considering the extrapolation on condition (2) in the Problem Statement; see discussion below). Including the assumption of full connectivity will enable tree based search approaches, but not including that assumption may be easier for matrix based approaches. Therefore, even though such non-fully connected subgraphs may not be useful in application, the assumption for this problem will depend on what is easier for the specific approach.

Condition (2) in the Problem Statement states "no vertex is a source or a sink". This can be interpreted to imply that in a subgraph g, all vertices must have at least one incoming edge and one outgoing edge. This can be expressed as saying that all vertices in g must have in-degree and out-degree greater than 0. (This does allow still for non-fully connected graphs, as in the following example:



While this is a valid subgraph based on the analysis thus far, certain methods, such as neighbor based methods, would not discover the part of the graph not connected to the root node.) This does not, however, preclude a group of vertices from being a source or a sink, only single vertices. For example, the following would still be a valid subgraph:



This example has a source cluster and sink cluster, but is a valid subgraph since no individual vertex is a source or sink. Furthermore, similar to the subgraph with disconnected areas, some methods, such as some neighbor based methods,

may not discover the "source" cluster of vertices, if, for example, the root node is in the "sink" cluster. It will also be noted that a self-loop on a vertex will preclude that vertex both from being a source and from being a sink.

Next, condition (4) in the Problem Statement states $0 < |e_g| < k$ (note that the condition clearly states "less than k" and not "less than or equal to"). Thus, since $|e_g|$ is an integer and not $\emptyset$, $k \geq 2$.

Combining the analysis of condition (4) with condition (2), only one edge in the graph (if there are no self-loops) would indicate one vertex is a source and the other vertex is a sink. Since there can be no source vertices or sink vertices, if self-loops are not allowed, $|e_g| \geq 2$. Thus, $2 \leq |e_g| < k$, and the previous result would become $k \geq 3$.

For the example given in the problem statement, neither the random graph nor the arbitrary v is specified. Therefore, a random graph will be constructed, and an arbitrary v will be chosen, while demonstrating the algorithm.

1.2.1 Approach 1: Brute Force Algorithm

The simplest algorithm here is sort of a brute force algorithm. It involves first starting with the vertex v, and enumerating all subsets of V which contain v. Then, for each of these subsets, the next step is enumerating all possible subgraphs where $0 < |e_g| < k$ with $e_g \subseteq E$. Finally, the next step is eliminating from this enumeration any subgraph which does not meet the stated conditions (for example, any subgraph that has vertices which are sources or sinks would be eliminated).

The number of sets of vertices in this approach is

$$\sum_{i=1}^{|V-v|} \binom{|V-v|}{i} = 2^{|V-v|}, \tag{1}$$

where $i \geq 1$ since in order to have at least one edge in a graph there must be at least two vertices (in the case of self-loops, one vertex can serve both purposes) which are connected by that one edge. Then for each set of vertices the number of subgraphs is, at maximum (i.e., worst case scenario),

$$\sum_{j=1}^{k-1} \binom{|E|}{j} = \sum_{j=1}^{k-1} \frac{|E|!}{(|E|-(k-1))!} \binom{k-1}{j} = \frac{|E|!}{(|E|-(k-1))!} 2^{k-1}, \tag{2}$$

where $|E| \geq k-1$. This second count is, of course, reduced from $|E|$ to the length of the corresponding set of edges for which the corresponding vertices are in the subset. Thus, (an overestimate of) the worst case scenario number of valid subgraphs is

$$\sum_{i=1}^{|V-v|} \binom{|V-v|}{i} \sum_{j=1}^{k-1} \binom{|E|}{j} = 2^{|V-v|} \frac{|E|!}{(|E|-(k-1))!} 2^{k-1} \tag{3}$$

$$= \frac{|E|!}{(|E|-(k-1))!} 2^{|V-v|+k-1}. \tag{4}$$

For the example given in the Problem Statement for demonstration, $G = (V, E)$ with $|V| = 10$, $|E| = 20$, and $k = 10$, the count becomes

$$\begin{aligned}
\text{count} &= \frac{|E|!}{(|E| - (k-1))!} 2^{|V-v|+k-1} && (5)\\
&= \frac{20!}{(20 - (10-1))!} 2^{9+10-1} && (6)\\
&= \frac{20!}{(20-9)!} 2^{9+9} && (7)\\
&= \frac{20!}{11!} 2^{18} && (8)\\
&= 20 \cdot 19 \cdot 18 \cdot 17 \cdot 16 \cdot 15 \cdot 14 \cdot 13 \cdot 12 \cdot 2^{18} && (9)\\
&= 1.59775 \times 10^{16}. && (10)
\end{aligned}$$

This is a really big number of subgraphs and would not be feasible to enumerate.

1.2.2 Approach 2: Iterative Edge Removal and Graph Pruning

Instead of a bottom-up approach attempting to list all possible subgraphs and then remove the subgraphs that are invalid, as in the previous approach which was shown to be infeasible, a top-down more refined and systematic approach will be more fruitful, especially when taking into consideration the conditions for subgraphs to be valid. These conditions, if considered appropriately and intelligently, can eliminate large swaths of the search space without having to consider individual subgraphs.

First, as noted above, all vertices in any subgraph g must have both in-degree and out-degree greater than 0. Thus, the first step is to remove from the graph G all vertices whose in-degree or out-degree is not greater than 0. The simple way to determine this is as follows: Consider the $|V| \times 1$ vector $\vec{v}$ representing all vertices in the graph with 1, i.e.,

$$\vec{v}_{|V|\times 1} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix}. \tag{11}$$

If the adjacency matrix of a graph G is the $|V| \times |V|$ matrix A, the in-degree of vertex i is the i^{th} entry in $\sum_j a_{ij}$ (where a_{ij} is the element of A in the i^{th} row and the j^{th} column), or $\left(\sum_j a_{ij}\right)_i$. Similarly, the out-degree is the i^{th} entry in $\sum_j a_{ji}$, or $\left(\sum_j a_{ji}\right)_i$. Thus, any vertex i for which $\left(\sum_j a_{ij}\right)_i < 1$ or $\left(\sum_j a_{ji}\right)_i < 1$ should be removed, or pruned, from G, as it is a source or a sink and is thus precluded from being in any valid subgraph g. A simple method for

Algorithm 1 Iterative Edge Removal and Graph Pruning

procedure GRAPH PRUNING(g, v)
  $\vec{v_i} \leftarrow 1 \, \forall v_g$
  if $\left\{ i : \left(\sum_j a_{ij} \right)_i < 1, \left(\sum_j a_{ji} \right)_i < 1 \right\} = \emptyset$ **then**
    return g
  else
    $g \leftarrow g \setminus l \, \forall l \in \left\{ i : \left(\sum_j a_{ij} \right)_i < 1, \left(\sum_j a_{ji} \right)_i < 1 \right\}$ ▷ *Remove sinks and sources*
    if $v \notin g$ **then**
      return ▷ *Eliminate subgraphs without v*
    else
      return GRAPH PRUNING(g, v)
procedure ITERATIVE EDGE REMOVAL(g, k, v)
  if $0 < |e_g| < k$ **then**
    print g ▷ *A valid subgraph*
  for all $e \in e_g$ **do**
    $g_i \leftarrow g \setminus e$
    $g_i \leftarrow$GRAPH PRUNING(g_i, v)
    return ITERATIVE EDGE REMOVAL(g_i, k, v)

this is, for each i^{th} index for which $\left(\sum_j a_{ij} \right)_i < 1$ or $\left(\sum_j a_{ji} \right)_i < 1$, to replace all entries in the i^{th} row and column of A with 0.

Now that some vertices and corresponding edges have been removed from the graph G, some vertices which were previously not sources or sinks may have become sources or sinks. Thus, the above in- and out-degree calculations and subsequent removal of corresponding edges must be repeated iteratively, until no vertices are sources or sinks.

If at any point the vertex v is removed in this way from G, there are no subgraphs that meet the criteria in the Problem Statement.

Additionally, if the total number of edges in this subgraph is less than k, this subgraph g (which also contains vertex v, has no vertices which are sinks or sources) meets the criteria in the Problem Statement, and should be added to a list of enumerated valid subgraphs to be returned as the answer to the Problem.

Next, we can iteratively remove a single edge from this subgraph, and repeat the process until we have enumerated all valid subgraphs.

This approach, in the worst case scenario (in which removing each edge results in the creation of no additional source or sink vertices, which is impossible to occur on every iteration, so this is again an overestimate), performs two summations of matrix indices for each edge removed, and iterates over each of the edges in the graph. Thus, the complexity order of calculations (with a factor of the optimized complexity order of the matrix operation implementations in the built-in libraries such as `numpy`) to be performed is $O(|E|^2)$, since it looks through each edge recursively.

This approach has been implemented in the repository, and a Jupyter notebook running the example from the Problem Statement ca be found here. For a small random graph of 5 vertices and 8 edges, the algorithm took an imperceptible amount of time in the Jupyter notebook. For the example with 10 vertices and 20 edges, the simple implementation of the algorithm took 67 minutes and 12.5 seconds in the Jupyter notebook. The resulting enumerated list of subgraphs was also too large for the Jupyter notebook to store in the cell output.

It is possible for some subgraphs to be produced multiple times by different iteration paths, and this algorithm as implemented will then include that subgraph (and all valid children subgraphs) multiple times in the output (once for each iteration path leading up to that subgraph). Somehow keeping track of each subgraph encountered (both vertices and edges, presumably in some kind of hashmap-like structure on the tuple of adjacency matrix and vertex labels) and avoiding iterating through encountered subgraphs again will both save on computation time and prevent these subgraphs from being duplicated in the output. This hashmap-like structure is implemented in the version used in this python script (with all enumerated subgraphs output to this text file). This script was significantly faster. The whole script, including both iteration and writing the output file, took 0.054 minutes. Without further experimentation, it is not known whether that is because of the hashmap implementation or running it as a script instead of a Jupyter notebook.

One thing that could be improved in this implementation is to turn the function into a generator and print each generated subgraph as it is produced, instead of collecting each valid subgraph in a list and printing the list at the end. Furthermore, the recursion can be parallelized to use computational resources more efficiently and produce results faster. However, given the speed of the current implementation on the example in the Problem Statement, additional effort will not be put into further optimizing this problem at this time.

2 Optimization

2.1 Problem Statement

Given functions $f_i : \Re_{\geq 0} \to \Re_{\geq 0}$ with $f_i(0) = 0$ and $\forall a < b, f_i(a) \leq f_i(b)$, we want to numerically solve the optimization problem:

$$u^* = \arg\max_{\substack{0 \leq u_i \leq 1 \\ \sum_i u_i = 1}} \sum_i f_i(u_i)$$

Note that f_i are not necessarily differentiable or even continuous. Your solution can be approximate and must be more efficient than a simple grid search. How would you get the best performance if there were thousands of functions? Can you make use of the fact that the functions are non-decreasing?

Demonstrate your algorithm on the following problem and plot the objective

function over the two dimensional simplex with the optimal value overlaid:

$$f_1(x) = 2x^{\frac{1}{3}} \tag{12}$$

$$f_2(x) = 5x^2 \tag{13}$$

$$f_3(x) = \begin{cases} 0 & x \leq 0.5 \\ x + 1 & x > 0.5 \end{cases} \tag{14}$$

2.2 Analysis

Here it will be noted that the condition $\sum_i u_i = 1$ can be rewritten

$$u_i = 1 - \sum_{i-1} u_{i-1}, \tag{15}$$

further demonstrating the relationship between the arguments to the functions. Specifically, this formulation demonstrates that any one argument is constrained by the values of all other arguments.

This can also be considered in a different way as well. With $u_i \in x_i \forall i$, and considering an n dimensional coordinate space (where n is the total number of functions), the conditioning equation

$$\sum_{i=1}^{n} u_i = 1 \tag{16}$$

defines a surface in the coordinate space on which the solution must reside, or a simplex in the $n - 1$ dimensional space.

2.2.1 Approach 1: Maximum Value Weighting Approximation

Since $f_i(0) = 0 \forall i$, $\forall a < b, f_i(a) \leq f_i(b)$, and $0 \geq u_i \geq 1 \forall i$,

$$\arg\max f_i(x_i) = 1 \forall i \tag{17}$$

(considering each function individually, without including the condition on u^* that $\sum_i f_i(u_i) = 1$).

Therefore, a simple weighting approximation for each u_i can be

$$u_i = \frac{f_i(1)}{\sum_i f_i(1)}. \tag{18}$$

For the given example,

$$f_1(1) = 2 \tag{19}$$
$$f_2(1) = 5 \tag{20}$$
$$f_3(1) = 2 \tag{21}$$
$$\sum_i f_i(1) = 9 \tag{22}$$
$$\therefore u_1 = \frac{2}{9}, u_2 = \frac{5}{9}, u_3 = \frac{2}{9} \tag{23}$$
$$\sum_i f_i(u_i) = 2\left(\frac{2}{9}\right)^{\frac{1}{3}} + 5\left(\frac{5}{9}\right)^2 + 0 \tag{24}$$
$$\approx 2.755 \tag{25}$$

While the time complexity for this algorithm is $O(i)$, it can be easily shown that this does not produce the optimal results. For example, if $u_1 = u_2 = \frac{2}{9}, u_3 = \frac{5}{9}$, $\sum_i f_i(u_i) \approx 3.014 > 2.755$.

2.2.2 Approach 2: Argument of Maximum Valued Function

In the given example, from visual inspection, $u_1 = u_3 = 0, u_2 = 1$ produces $\sum_i f_i(u_i) = 5$, which, in this case, is likely the optimal solution (or, at least, fairly close to it).

Turning this into an algorithm,

$$u_j = 1 \text{ for } \max_j f_i(1) \tag{26}$$
$$u_i = 0 \ \forall i \neq j. \tag{27}$$

For cases in which multiple $f_i(1) = f_j(1)$ for $i \neq j$ (as in the example if only f_1 and f_3 are considered), either argument would work equally effectively. In such cases, since a decision must be made, the smaller index will be chosen.

This approach relies on the fact that the functions all start $f_i(0) = 0$ and are non-decreasing.

This approach also has time complexity $O(i)$.

However, it can be shown by counterexample that this approach does not work. Consider the following example:

$$f_1(x) = 3 \tag{28}$$
$$f_2(x) = 4 \tag{29}$$
$$f_3(x) = 5. \tag{30}$$

This approach would suggest $u_1 = u_2 = 0\, u_3 = 1$ is the optional solution. However, it can be easily seen that $u_1 = u_2 = u_3 = \frac{1}{3}$ (and any number of other sets of arguments) produces a more optimal result.

2.2.3 Approach 3: Subgradient Descent on Augmented Lagrangian

As noted above, this problem can be regarded in the following way. With $u_i \in x_i \forall i$, and considering an n dimensional coordinate space (where n is the total number of functions), the conditioning equation

$$\sum_{i=1}^{n} u_i = 1 \tag{31}$$

defines a surface in the coordinate space on which the solution must reside. The objective can be stated as maximizing the arguments of the function

$$f(x_1, x_2, ..., x_n) = \sum_{i=1}^{n} f_i(x_i) \tag{32}$$

on the space $\Re^n_{[0,1]}$ given the constraint condition

$$\sum_{i=1}^{n} x_i = 1. \tag{33}$$

This constraint can be rewritten as

$$c(x_1, x_2, ..., x_n) = \sum_{i=1}^{n} x_i - 1. \tag{34}$$

The objective to find the arguments that maximize the function above is equivalent to an objective to find the arguments that minimize the negative of that function, namely

$$g(x_1, x_2, ..., x_n) = -\sum_{i=1}^{n} f_i(x_i) \tag{35}$$

with the same constraint.

The constraint can be embedded into the objective in a similar fashion as a penalty term or Lagrange multiplier method, for example using an Augmented Lagrangian:

$$L(x_1, x_2, ..., x_n, \lambda, \rho) = g(x_1, x_2, ..., x_n) + \lambda c(x_1, x_2, ..., x_n) + \frac{\rho}{2} c(x_1, x_2, ..., x_n)^2 \tag{36}$$

$$= -\sum_{i=1}^{n} f_i(x_i) + \lambda \left(\sum_{i=1}^{n} x_i - 1 \right) + \frac{\rho}{2} \left(\sum_{i=1}^{n} x_i - 1 \right)^2, \tag{37}$$

where the penalty term ρ and the Lagrangian multiplier λ are sufficiently large so as to severely penalize any constraint violations.

Then, a subgradient descent method can be employed to explore the parameter space and locate the maximum (or minimum, in the negated formulation of the problem), and arguments that lead to the maximum. The subgradient descent method relies on the subgradient, which is defined as any vector v satisfying

$$f(y) \geq f(x) + v \cdot (y - x), \forall y. \tag{38}$$

Like the gradient for a differentiable function, a subgradient of a function at a point is the slope of a line that is below the function everywhere, i.e., is a global under-estimator of the function. The set of all such subgradients at a point is the subdifferential. If a function f is continuously differentiable, its subdifferential reduces to the singleton

$$\partial f = \{\nabla f\}. \tag{39}$$

Furthermore, x is a global minimizer of f if $0 \in \partial f(x)$.

The subgradient can be defined individually for each function, but it can also be numerically estimated for all functions as

$$\nabla f(x) \approx \frac{f(x + \epsilon) - f(x)}{\epsilon}, \tag{40}$$

a finite difference method.

The subgradient descent method begins with initial values for the coordinates x_i satisfying the constraint condition. In each iteration, it computes the subgradient of the Augmented Lagrangian in each iteration as

$$\frac{\partial L}{\partial x_i} = -\nabla_{x_i} f_i(x_i) + \lambda \frac{\partial c(x)}{\partial x_i} + \rho c(x) \frac{\partial c(x)}{\partial x_i} \tag{41}$$

$$= -\nabla_{x_i} f_i(x_i) + \lambda + \rho c(x) \tag{42}$$

and updates the coordinates appropriately. It ensures the coordinates are in the domain, and calculates the new amount of constraint violation. It then updates the Lagrangian multiplier λ, the penalty parameter ρ if necessary (if constraint violations are not being sufficiently penalized and are consistently too large in magnitude), and the step size η. It continues iterating until either it reaches the maximum number of iterations, or it has reached a minimum with constraint satisfaction.

Note that the subgradient descent method does not have the same guarantees regarding finding the global maximum as regular gradient descent.

To increase the chances of finding the global maximum, a number of initialization points can be used, and this method can be run in parallel for each of those initial points. Then, the arguments of the method result which produced the largest value of the objective function can be taken as approximately the argmax of the objective function.

Algorithm 2 Subgradient Descent on Augmented Lagrangian

Require: $\rho > 0$, $\beta > 1$, $0 < \gamma < 1$

procedure SUBGRADIENT DESCENT ON AUGMENTED LAGRANGIAN($f_i(x)$)
  $x \leftarrow [x_1, x_2, ..., x_n]$ with $x_i \in [0, 1] \forall i = 1, 2..., n$ ▷ *Initialize coordinates*
  $\lambda \leftarrow 0$, $\eta \leftarrow 1$
  while iteration $<$ max iterations **do**
    $\nabla_x g \leftarrow 0$
    for $i = 1, 2, ..., n$ **do**
      $\nabla_{x_i} g \leftarrow \nabla_{x_i} g + \frac{f_i(x_i+\epsilon) - f_i(x_i)}{\epsilon}$ ▷ *Calculate Subgradients*
    $\nabla_x L \leftarrow \nabla_x g + \lambda + \rho c$ ▷ *Calculate new Augmented Lagrangian*
    $x_i \leftarrow x_i - \eta \cdot \nabla_{x_i} L$ ▷ *Update coordinates*
    $x_i \leftarrow \text{clip}(x_i, 0, 1)$ ▷ *Ensure domain compliance*
    $c \leftarrow c(x)$ ▷ *Calculate new constraint violations*
    $\lambda \leftarrow \lambda + \rho \cdot c$ ▷ *Update Lagrangian multiplier*
    if $|c| < tol$ **then**
      break ▷ *Objective minimized with constraint satisfaction*
    if $|c| < \frac{tol}{10}$ **and** iteration > 0 **then**
      ▷ *Update ρ if not sufficiently penalizing violations of c, after the first iteration* ◁
      $\rho \leftarrow \beta \cdot \rho$
    $\eta \leftarrow \gamma \cdot \eta$ ▷ *Update step size*
  return x

This approach has been implemented, and run for the example in the Problem Statement in this python script with the results

$$u^* = \begin{bmatrix} 0.01876 \\ 0.99775 \\ 0 \end{bmatrix} \tag{43}$$

$$\sum_i f_i(u_i) = 5.50895 \tag{44}$$

(note that the implementation does not seed the random number generator, so each run will yield slightly different results even given the same input arguments). This run used 20 initialization points, and took a total of 0.054826 seconds to complete.

In order to plot the objective function on the two dimensional simplex with the best result overlaid, this implementation was rerun in this Jupyter notebook, where it took 0.1088 s to complete, and resulted in

$$u^* = \begin{bmatrix} 0.02404 \\ 0.97596 \\ 0 \end{bmatrix} \tag{45}$$

$$\sum_i f_i(u_i) = 5.33971. \tag{46}$$

The plot can be found in the aforementioned Jupyter notebook, and uses the result obtained in the notebook (as opposed to the script; see note earlier about the lack of seeding on the random number generator for an explanation as to why the two runs produced different results).

The results from this approach can be improved by decreasing the tolerance of constraint violations to provide a smaller violation in the result. Additionally, a larger number of initialization points would also result in a wider search, and would also increase the likelihood of honing in on the most accurate possible result.