Comparison of Naive Implementation of Big M Simplex Method Linear Programming against Naive Implementation of Dijkstra's for Solving Shortest Path Problems

Will Darragh

May 28, 2020

Executive Summary

Whereas some mathematical problems are difficult to solve explain in simple terms, many graph theory problems are incredibly simple to explain. One of the most essential graph theory problems, the shortest path problem, is so simple in formulation, it hardly sounds like a problem of any mathematical interest. The problem is thus: given a map with several roads, intersections between roads, and given distances between these intersections, what is the fastest way to get from point A to point B. Though the problem is simple to formulate, creating an efficient algorithm to solve this problem is not trivial.

From the field of Computer Science, we have a well known process known as Dijkstra's algorithm. Dijkstra's algorithm starts at a given intersection and finds the shortest paths to local intersections before continuing to travel through the map until it reaches the goal. Dijkstra's is fairly efficient because it keeps track of intersections it has already traveled to and it always travels to intersections that cause it to travel the least distance.

Although Dijkstra's is not the only shortest path algorithm, it is certainly the most famous. Since the time of Dijkstra's invention, however, a new mathematical tool has emerged that can be applied to a variety of problems. This tool, linear programming, has endless applications, from optimizing profits against material cost, to creating systems of maximum flow. Linear programs are only limited in the sense that they require very specific formulation. This limitation is also their strength, as any problem that can be formulated correctly can be solved by linear programming.

This paper analyzes these two algorithms on the basis of their utility to a college programming competition team. In these competitions, students solve a variety of mathematical and computer science related problems in a short period of time. At higher levels of competition, not only must students write a program that correctly solves these problems, they must write programs that solve given problems quite quickly, often in a matter of seconds. Thus, students need extremely efficient algorithms. They can use reference material in writing their programs, so overall efficiency and utility is more important than difficulty of writing the algorithm.

Definitions, Key Terms, and Clarifications

This section is included to clarify the language that will be used in this paper. Especially considering that this paper draws inspiration from Computer Science, Mathematics, and various sub-fields thereof, this section may prove useful.

For the rest of this paper, we will use the formal terms for parts of a graph.

- Graph - The whole set of nodes and edges defined by the problem. For the purposes of the shortest path problem, the graphs considered are more formally directed multi-graphs, but we will simply refer to them as graphs.
- Node - A point in the graph connected by edges. Nodes will be labeled numerically, starting at 0 unless otherwise noted. To make the problem easier to define, if there are a total of n nodes, then node 0 will always be the start and node $n-1$ will always be the goal.
- Edge - A connection between two nodes in a graph. Note that for the purposes of the a shortest path problem, all edges are weighted edges, meaning they have a certain cost associated with traveling, but they will simply be referred to as edges.

There are many different ways of implementing linear programming. The simplex method is the earliest such method, and perhaps the easiest to understand in terms of its algorithm. The Big M method is a way of handling linear programs which have equations with $=$ in them [7]. For the rest of this paper, any reference to linear programming, the simplex method, or the Big M method all refer to the same thing.

Although Dijkstra's algorithm is quite interesting, it is well documented and explored. The rest of this paper will focus on linear programming, so general references to a process, method, or algorithm will refer to linear programming unless explicitly stated otherwise. Dijkstra's is the benchmark, but linear programming is the real source of interest here.

All code referred to in this paper was written in Python3. This is one of the most commonly used languages in college programming competitions, and it is the language of choice for the Mercer Binary Bears Programming Team. Competitions do not allow any packages or libraries to be imported that do not come with base Python3. With certain libraries, it could be that the two algorithms in question perform differently.

Finally, inspiration for the naming a variables and terms was drawn primarily from two difference sources. One is a Princeton professor's slides on how to implementing the a simplified simplex method in code [5], and the other is a fantastic online calculator which shows the step by step results of applying the Big M algorithm [6]. The notations of these two sources were somewhat different. The author takes responsibility for and apologizes for any inconsistencies with notation.

1 Assumptions

The shortest path problem is very general. This paper is concerned with the development of a general algorithm that handles any valid graph input. The only assumption made is that the graph given is solvable, such that there is a possible path between the start node and the goal node.

2 Variables

Z = Variable for evaluation of objective function

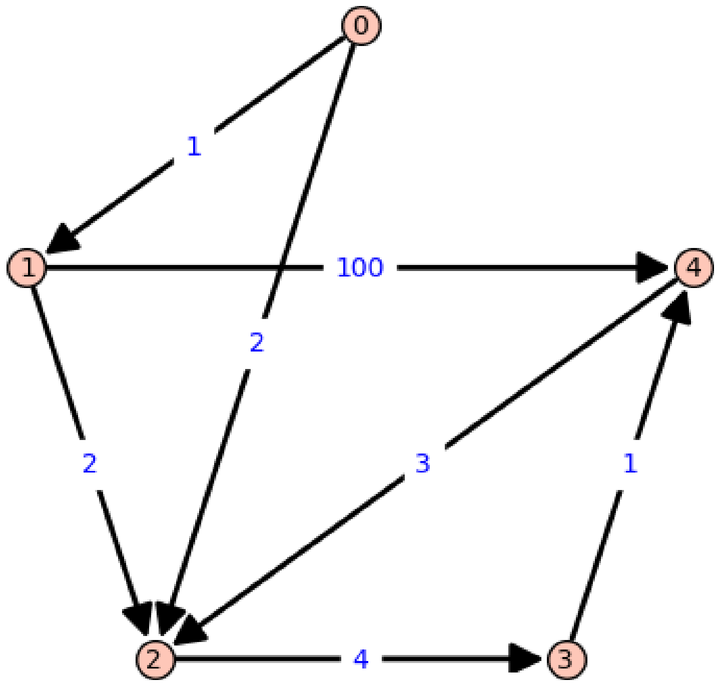
$x_0 ... x_{n^2-1}$ = Binary variables for each edge where n is the number of nodes edge from i to j is $x_{i \cdot n+j}$

$a_0 ... a_{n-1}$ = artificial variables which will be forced to be zero in order to satisfy equality for equations

3 Parameters

$M = 1000000$ = Some arbitrarily large number for the big M method

The parameters for this problem vary depending on the input. The graph [2] for the first test case is below.



Thus the parameters are as follows

$$
\begin{aligned}
c_1 &= 1 \\
c_2 &= 2 \\
c_7 &= 2 \\
c_9 &= 100 \\
c_{13} &= 3 \\
c_{19} &= 1 \\
c_{22} &= 3 \\
c_i &= 0 \text{ for all other } i < n
\end{aligned}
$$

4 Objective

Given a graph, we want to find the shortest path from the start node to the goal node. We also want to compare this result and the time it takes against the traditional Dijkstra's algorithm.

5 Model Equations

Objective Function

$$\sum_{i=0}^{n^2-1} c_i \cdot x_i$$

Constraints

Constraints vary depending on input, constraints for the first test case follow.

$$\begin{aligned} x_1 + x_2 &= 1 \\ -x_1 + x_7 + x_9 &= 0 \\ -x_2 - x_7 - x_{22} + x_{13} &= 0 \\ -x_{13} + x_{19} &= 0 \\ -x_{22} + x_{19} + x_9 &= 1 \end{aligned}$$

Note that the first constraint established that the start node has exactly one edge going out. The last constraint is normally written as a series of terms summing to -1, meaning the goal node has an edge connected, but for the purposes of the Big M method, both sides of the equation are multiplied by -1.

6 Solution

The linear program is generated by the parameters, objective function and the constraints. Code for the linear program solver can be found in the appendix.

For the first example, the linear program found that the best solution is

edge from 0 to 2,
edge from 2 to 3,
edge from 3 to 4,
with a final cost of 7

The simplex algorithm took around a million nanoseconds, whereas running the same test case through Dijkstra's algorithm took only ten thousand nanoseconds. That's one hundred times faster!

Generating further test cases in not a trivial task, as verifying the validity of a graph is not simple, and the simplex code will get stuck on graphs without a valid solution.

Nonetheless, we tested using a valid graph generated by a Dijkstra's visualization web app. [3] The specific input for this test case can be found in the comparison code in the appendix. There were a total of 18 nodes in this example. The simplex algorithm took almost a tenth of a second, while Dijkstra's took only 25 thousand nanoseconds. That's about 3500 times faster. So, when increasing the size of the test case by a factor of about 4, we drastically increased the difference in run times between the two algorithms.

Of course, it would only be fitting to test the algorithms against a real world example. A previous problem from the ACM IPCP world finals included a graph [1]. Input for this problem included graphs with thousands of nodes. The problem itself is a variation of the shortest path problem, involving finding the shortest path from a start node to all other nodes. We only tested finding the shortest path along one route. Dijkstra's ran for less than a thousandth of a second before finding one route, whereas the connection to a Linux server terminated before the simplex code finished running. Needless to say, that length of run time would not be viable in a programming competition.

Conclusions

In theory, Dijkstra's should always beat out linear programming. Even a very optimized algorithm for linear programming [4] has a run time on the order of $O(n^3)$, meaning that the run times scales with the cube of the size of the input. On the other hand, a poor version of Dijkstra's runs at $O(n^2)$ at worst, and basic optimization can make Dijkstra's scale near linearly.

Linear programming always correctly solves problems, and it can occasionally solve some problems quite efficiently. It appears, however, that for large graphs, solving the shortest path problem is simply too computationally complex. Perhaps there is an extremely optimized version of linear programming that is efficient for this problem, but the naive implementation is certainly not that.

Perhaps too, linear programming would prove an effective tool in other programming team type problems, but further research is required. Certainly writing a linear program solver is quite difficult, and it is quite finicky to get running correctly. The variety of problems it can solve is appealing, but overall, it unfortunately does not seem like the right tool for the tasks of a programming team.

References

[1] Icpc past problems 2016. `https://icpc.baylor.edu/worldfinals/problems`.

[2] Jeffery Denny. Example graph from class.

[3] David Galles. Dijkstra shortest path. `https://www.cs.usfca.edu/~galles/visualization/Dijkstra.html`.

[4] Johnathan A. Kelner and Daniel A. Spielman. A randomized polynomial-time simplex algorithm for linear programming. pages 51–60, 2006.

[5] Robert Sedgewick. Linear programming. `https://www.cs.princeton.edu/~rs/AlgsDS07/22LinearProgramming.pdf`.

[6] Piyush N Shah. Atozmath simplex method caculator. `https://cbom.atozmath.com/CBOM/Simplex.aspx?q=sm`.

[7] Clifford Stein. The big m method. `http://www.columbia.edu/~cs2035/courses/ieor3608.F05/david-bigM.pdf`.

Appendix A - Code

Comparison Program

```
#!/usr/bin/python3

import dijkstras_shortest_path as dijkstras
import simplex_shortest_path as simplex

from time import perf_counter as timer

def compare(edges, n):
  # Setup both
  dijkstras.setup(edges, n)
  simplex.setup(edges, n)

  # Time dijksras
  start = timer()

  dijkstras.solve()

  stop = timer()

  dijkstras_time = stop-start

  # Results
  print(f'\nDijkstras:\n')
  dijkstras.show()
  print(f'Time␣was␣{int(dijkstras_time*(10**9))}␣nanoseconds')
  print(f'Time␣was␣{dijkstras_time:.4f}␣seconds')

  # Time simple
  start = timer()

  simplex.solve()

  stop = timer()

  simplex_time = stop-start

  print(f'\nSimplex:\n')
  simplex.show()
  print(f'Time␣was␣{int(simplex_time*(10**9))}␣nanoseconds')
  print(f'Time␣was␣{simplex_time:.4f}␣seconds')

# Case - example from class

edges = [    [ 0, 1, 1.0 ],
    [ 0, 2, 2.0 ],
    [ 1, 4, 100.0 ],
    [ 1, 2, 2.0 ],
    [ 2, 3, 4.0 ],
    [ 3, 4, 1.0 ],
    [ 3, 2, 3.0 ] ]

n = 5
```

```
compare(edges, n)

# Case - more difficult example

n = 18

edges = [
[0,8,8.0],
[0,14,4.0],
[1,0,1.0],
[1,4,2.0],
[1,5,7.0],
[2,1,3.0],
[3,6,8.0],
[4,5,1.0],
[5,2,9.0],
[5,9,6.0],
[6,2,5.0],
[6,3,9.0],
[6,10,7.0],
[7,11,1.0],
[8,4,8.0],
[8,5,8.0],
[8,9,4.0],
[9,5,8.0],
[9,6,8.0],
[9,10,2.0],
[9,12,7.0],
[10,3,1.0],
[11,7,5.0],
[12,11,5.0],
[12,13,9.0],
[14,11,7.0],
[14,15,6.0],
[15,11,1.0],
[15,16,3.0],
[16,12,9.0],
[16,13,2.0],
[16,15,7.0],
[16,17,8.0]
]

compare(edges, n)
```

Big M Simplex Method

```
#!/usr/bin/python3

# Edges are tuples like [node1, node2, weight]
# n is number of nodes
def setup(edges, n):

  global M,N,a,C,B,CJ

  # Adjacency Matrix
  matrix = [ [0.0 for i in range(n)] for j in range(n) ]

  for i,j,weight in edges:
    matrix[i][j] = weight

  M = n

  N = n**2

  # Tabluex
  a = [ [0.0 for i in range(M+N+1)] for j in range(M+1) ]

  for i in range(n):
    for j in range(n):
      if matrix[i][j] != 0:
        a[i][M*i + j] = 1.0
        a[j][M*i + j] = -1.0
        a[M][M*i + j] = -1.0*matrix[i][j]

  # First node going out
  a[0][M+N] = 1.0

  # Adjust last row so last node in value can be 1 not -1
  for j in range(M+N):
    a[M-1][j] *= -1

  # Last node going in
  a[M-1][M+N] = 1.0

  # Artificial variables
  for j in range(N,M+N):
    a[j-N][j] = 1.0

  # Arbitrarily large number
  c = 1000000.0
  # Mutiplicative value
  C = [-1.0*c for i in range(M)]

  # Variable going in
  B = [f'A{i}' for i in range(M)]

  # Original Objective
  CJ = [i for i in a[M]]

  # Set object constant for artifical variables
  for i in range(M):
    CJ[M+N-i] = -1.0*c

# Standard pivot
def pivot(p, q):

  global M,N,a,C,B,CJ

  for i in range(M+1):
    for j in range(M+N+1):
      if ( i != p and j != q ):
        if ( a[p][q] != 0 ):
          a[i][j] -= (a[p][j] * a[i][q]) / a[p][q]
        else:
          a[i][j] -= (a[p][j] * a[i][q]) * 1000000000.0

  for i in range(M+1):
    if ( i != p ):
      a[i][q] = 0.0

  for j in range(M+N+1):
    if ( j != q ):
      if ( a[p][q] != 0 ):
        a[p][j] /= a[p][q]
      else:
        a[p][j] *= 1000000000.0

  a[p][q] = 1.0

def solve():

  global M,N,a,C,B,CJ

  # Z value, relative objective function
  Z = lambda col: sum( [ C[i]*a[i][col] for i in range(M) ] )
  while(True):

    p,q = 0,0

    # While Z value minus objective negative for all in objective row
    for q in range(M+N):
      if ( (Z(q) - CJ[q]) < 0 ):
        break
    else:
      break

    q = 0

    # Find the column with a minimum Z - C value
    for j in range(M+N):
      if ( Z(j) - a[M][j] < Z(q) - a[M][q] ):
        q = j

    # Find a valid row
    for p in range(M):
      if ( a[p][q] > 0 ):
        break

    # Find row with smallest ratio
    for i in range(M):
      if ( a[i][q] > 0 ):
        if ( a[i][M+N] / a[i][q] < a[p][M+N] / a[p][q] ):
          p = i

    # Set the incoming variable
    B[p] = f'x{q}'

    # Adjust the multiplicative
    C[p] = CJ[q]

    pivot(p, q)

def show():

  global M,N,a,C,B,CJ

  value = 0.0

  # Print value of non-zero variables
  for i in range(M):
    if B[i][0] == 'x':
      if a[i][M+N] != 0:
        #print(f'{B[i]} = {a[i][M+N]}')

        index = int(B[i][1:])

        print(f'Take␣edge␣from␣{index//M}␣to␣{index%M}')

        value += a[i][M+N]*CJ[index]

  # Final result
  print(f'Final␣cost␣of␣{-1*value}')

if __name__ == '__main__':

  # Test Case

  edges = [ [ 0, 1, 1.0 ],
      [ 0, 2, 2.0 ],
      [ 1, 2, 2.0 ],
      [ 1, 4, 100.0],
      [ 2, 3, 4.0 ],
      [ 3, 4, 1.0 ],
      [ 4, 2, 3.0 ] ]


  setup(edges, 5)

  solve()

  show()
```

Dijkstra's Algorithm

```
#!/usr/bin/python3

COST = 0
FROM = 1

INF = 1000000

# Edges are tuples like [node1, node2, weight]
# n is number of nodes
def setup(edges, n):
  global matrix, table, visited, goal, N

  N = n

  goal = n-1

  visited = []

  # Adjacency Matrix
  matrix = [ [0.0 for i in range(n)] for j in range(n) ]

  for i,j,weight in edges:
    matrix[i][j] = weight

  # Table
  table = [ [INF, -1] for j in range(n) ]

  # Mark first as visited
  table[0][COST] = 0

  visited.append(0)

def solve():

  global matrix, table, visited, goal, N

  curr = 0

  while not ( goal in visited ):

    next_node = None
    next_cost = INF

    for i in range(N):
      if ( matrix[curr][i] != 0 ):
        dist = table[curr][COST] + matrix[curr][i]

        if ( dist < table[i][COST] ):
          table[i][COST] = dist
          table[i][FROM] = curr

          if ( dist < next_cost ):
            next_node = i

    visited.append(curr)
    curr = next_node

def show():

  global table,goal

  curr = goal

  while ( curr != 0 ):
    a = table[curr][FROM]
    print(f'Take␣edge␣from␣{a}␣to␣{curr}')

    curr = a

  print(f'Final␣cost␣of␣{table[goal][COST]}')


if __name__ == '__main__':

  # Test Case

  edges = [ [ 0, 1, 1.0 ],
      [ 0, 2, 2.0 ],
      [ 1, 2, 2.0 ],
      [ 1, 4, 100.0],
      [ 2, 3, 4.0 ],
      [ 3, 4, 1.0 ],
      [ 4, 2, 3.0 ] ]


  setup(edges, 5)

  solve()

  show()
```