Travel model notes

Gitanjali Bhattacharjee

January 2020

Contents

Figures

Listings

1 Traffic model overview

1.1 Bugs

I have identified three errors in the original implementation of the traffic assignment algorithm and traffic model – they are described in detail in Section 3. In brief, they are:

- Inconsistent order of iteration over origins and destinations when assigning traffic, which leads to different network performance results for the same damage maps. For example, the no-damage network performance metrics can vary by at least an order of magnitude depending on the order in which we assign traffic to the road network.
- Incorrect travel time computation – when trips were added to edges in the original implementation, the new travel time over that edge was computed as the travel time due to only the additional trips rather than to all trips on the edge. This would lead to smaller travel times over edges, which could distort the results of the traffic assignment algorithm since shortest paths are found based on edge travel times.
- Incorrect trip assignment – if a trip between a particular origin and destination cannot be made, it's the result of a zero-capacity edge somewhere in the shortest path between the origin and destination. In the original implementation, the flow between the origin and destination would be assigned to all edges in the shortest path even if the trip between the origin and destination could not be made because of a zero-capacity edge in the shortest path. This increased the total travel time over the network by an order of magnitude.
- Incorrect graph edge weights – in the original graph of the road network, the traversal times t_a of edges with zero capacity were also 0. This meant that these edges were selected as part of shortest paths between origins and destinations but that trips couldn't actually be assigned to them, resulting in false connectivity losses.

2 Needs

My research focuses on developing methods to identify which bridges in the road network to retrofit such that the performance of the network is optimized, in terms of total travel time, trips made, vehicle miles travelled, or some function of those metrics. We retrofit bridges because doing so reduces the probability that they will sustain damage during an earthquake, and because we assume that the road network functions better when bridges are not damaged. Better functionality or performance is defined as smaller travel times, larger numbers of trips made, and fewer vehicle miles travelled. The methods I am developing are built on those assumptions – therefore, the models I use to implement and demonstrate the value of the methods I'm developing should also reflect those assumptions. The traffic model has caused some concern in the past because there have been specific cases in which it appears to violate those assumptions. Even having fixed the bugs identified in Section 1.1, I find cases in which these assumptions are violated.

- The performance of the network when not damaged should be consistent – that is, the values of the no-damage total travel time, trips made, and vehicle miles travelled should be constant. **Status: Met.**
- When the network is not damaged, the metrics reflecting its performance – travel time, vehicle miles travelled, and the number of trips made – should be no worse than metrics that reflect the performance of a damaged network. Ideally, the metrics reflecting the performance of the undamaged network should indicate better performance than those of a damaged network. **Status: Complicated.** There are cases in which bridges are damaged and one or two of the three network performance metrics improves (e.g. travel time decreases, trips made increases). See the next item for more detailed explanation.
- The network should function no worse with a particular bridge than without it – that is, the network's performance should not improve when a bridge is damaged relative to when that bridge is undamaged. **Status: Complicated.** There are three metrics that we can track using the travel model: the total

Damaged bridges	Δ Travel time (s)	Δ Vehicle miles	Δ Trips made
none	2.978×10^{13}	3.505×10^{7}	7.628×10^{6}
966	-2.294×10^{12}	3.553×10^{7}	3.022×10^{3}
917, 972	4.725×10^{12}	3.474×10^{7}	1.623×10^{3}

Table 1: Changes in network performance metrics when bridges are damaged. Δ = current condition - no-damage condition. Positive values indicate an increase, whie negative values indicate a decrease.

travel time, trips made, and vehicle miles travelled. None of these are guaranteed to get worse – that is, reflect network performance worse than the no-damage performance – when bridges are damaged. Consider the cases shown in Table 1: in both cases when bridges are damaged, the number of trips made on the network increases, which taken by itself would suggest improved network performance. When bridge 966 is damaged, more trips are made (an improvement) and the total travel time decreases (an improvement). A cost model that only took into account these two metrics would indicate that damaging bridge 966 improves the performance of the network. However, the vehicle miles travelled more than double when bridge 966 is damaged, indicating that the trips that do get made are more circuitous as a result of the bridge damage, though they are made more quickly. When bridges 917 and 972 are both damaged, the number of trips made increases (an improvement), the travel time increases (a deterioration), and the vehicle miles travelled increase (a deterioration). These sample cases suggest that implementing a cost model to go along with this traffic model must be done with *extreme* care. It is not enough to consider only the number of trips made and the total travel time to get a full picture of the how the network performance changes. Possible solutions include (1) considering all three network performance metrics when computing the cost of the network performance or (2) encoding the assumptions that bridge damage will lead to worse network performance in the cost model itself, e.g. if bridges are damaged, impose a cost per bridge.

3 Bugs

3.1 Order of origin, destination iteration

A dictionary is a particular type in Python. It contains keys that, when looked up, return particular values. Say we have a dictionary `menu`. One key in `menu` is `spaghetti`. The value of `spaghetti` is `7.50`. Thus, if we called `menu[spaghetti]`, we would get the price of the spaghetti: `7.50`. Say we add the key-value pair `(pizza, 10.00)` to `menu`.

We can look up all the keys in a particular dictionary by calling `menu.keys()`, from which we would expect to get the resulting list: `[spaghetti, pizza]`. We might expect to get that exact list because the keys are listed according to the order in which they were inserted in the dictionary. However, the documentation for Python 2 (in which the traffic model is written) states, "Keys and values are listed in an arbitrary order which is non-random, varies across Python implementations, and depends on the dictionary's history of insertions and deletions." Thus, we might get the list of menu items in another order.

In the traffic assignment code `ita.py`, iteration over origins and destinations was originally implemented as follows.

```
for origin in self.demand.keys():
   paths_dict = nx.single_source_dijkstra_path(self.G, origin, cutoff = None, weight = '
       ↪ t_a')
   for destination in self.demand[origin].keys():
```

Since the code is written in `Python 2.7`, the results of `self.demand.keys()` and `self.demand[origin].keys()` are not guaranteed to remain the same when called at different times. In fact, we have proof that the lists do not remain the same when called at different times.

3.1.1 Significance

The order of the menu items may not matter for a restaurant application. However, consider our fixed-demand travel model. The demand between the origins and destinations that we consider is stored as a nested dictionary called `demand`. The first key of `demand` is an origin, and the second key of `demand` is a destination. Calling `demand[origin][destination]` returns a numerical value. To assign traffic in `ita.py`, we iterate over two lists: first, a list of the origins (which we get by calling `demand.keys()`) and then a list of destinations (which we get by calling `demand[origin].keys()`). The order in which the origins and destinations appear in those lists is not guaranteed to remain the same. In fact, we have demonstrated that when the methods in the traffic model are called in parallel (using the `parallel-python` package), the order of the origins *always* changes from when the methods are not called in parallel. The order in which the traffic assignment algorithm iterates over origins and destinations has a non-negligible impact on the values of the traffic metrics that describe the performance of the road network – the no-damage travel time over the network according to the traffic assignment algorithm is $6.12E12$ seconds for one ordering of the origins and $2.12E14$ seconds for another ordering of the origins. The no-damage travel time can thus vary by at least $5.7E10$ hours depending on the order in which we iterate over the origins.(There are 34 super-nodes, each of which can be both an origin and a destination – I have not tested the no-damage traffic metrics for every ordering of super-nodes in origin and destination lists).

Why is the inconsistency that stems from different orders of origins and destinations a problem? First, because it means that there have been instances in which the performance of the network when damaged has been better than the performance of the undamaged network. This undermines the case for retrofitting bridges – why bother, if the performance of the road network apparently improves when bridges are damaged? Second, if the order in which we iterate over origins and destinations matters so much and also varies, then there may be cases in which a more-damaged network (i.e., one with fewer damaged bridges) seems to perform better than a less-damaged network. Third, the performance of the undamaged network may be inconsistent – in fact, it may vary a great deal. This variation might, unintentionally, do a good job of reflecting the real world – however, in terms of modeling how retrofits affect the road network's performance, this variation means the goal-posts are constantly shifting. If the goal of retrofitting bridges is to minimize the disruption to the road network, huge variations in the baseline condition of the road network make improvements difficult to discern, to both to human eyes and to agents or analytical tools that assume smaller costs (associated with smaller total travel times) represent improvements in the road network performance rather than essentially meaningless variations in performance due to the order in which traffic was assigned.

For example, I am using global variance-based sensitivity analysis to measure how changes in the fragilities of highway bridges affect the expected performance of the road network. This analytical setup assumes that changes in the performance of the road network are explained by which bridges are damaged (a function of the hazard and the bridges' fragility). However, if the largest variations in the travel time stem from changes in the order over which origins and destinations are iterated, the results of any variance-based sensitivity analysis using code as written *must* be assumed to be flawed unless the number of samples used to conduct the sensitivity analysis is sufficiently large to cover all the orderings of origins and destinations that might occur. In practice, it does not seem that the origins and destinations are returned in random and different orders each time they are called – however, there is no guarantee against such a situation.

3.1.2 Solutions

3.1.2.1 Sort lists by integer values The simplest fix I can think of is to ensure that origins and destinations are iterated over in a fixed order. To implement this, I have made the following change to `ita.py`. Ordering the destinations in this fashion is necessary because each origin has a different set of associated destinations.

```python
origins = [int(i) for i in self.demand.keys()] # get SD node IDs as integers
origins.sort() # sort them
origins = [str(i) for i in origins] # make them strings again
for origin in origins:
    paths_dict = nx.single_source_dijkstra_path(self.G, origin, cutoff = None, weight = '
        ↪ t_a')
    destinations = [int(i) for i in self.demand[origins].keys()]
    destinations.sort()
    destinations = [str(i) for i in destinations]
    for destination in destinations:
```

3.1.2.2 Create an origin-destination `defaultdict`

Another potential fix is to create an additional dictionary – let's call it `origins_destinations` – in which the keys are origins and the values are lists of destinations. Because the order of elements in a list is guaranteed to be persistent in Python, this would eliminate the need to sort the list of destinations for every origin, as is done in the snippet code above. That is, we could rewrite the code above as:

```python
origins = [int(i) for i in self.demand.keys()] # get SD node IDs as integers
origins.sort() # sort them
origins = [str(i) for i in origins] # make them strings again

origins_destinations = build_od() # returns a defaultdict with keys of origin IDs and
    ↪ values of lists of destination IDs

for origin in origins:
    paths_dict = nx.single_source_dijkstra_path(self.G, origin, cutoff = None, weight = '
        ↪ t_a')
    destinations = origins_destinations[origin] # this is a list of destination node IDs
    for destination in destinations:
```

To make the above piece of code feasible, we would have to create the dictionary `origins_destinations` as an instance of the class `defaultdict` (the regular dictionary class does not allow key-value pairs in which the value is a list, while the `defaultdict` class does). However, we need to ensure that the order of destinations remains the same not only between calls to the iterative traffic assignment code but also between runs of a larger piece of code – therefore, we would have to sort the lists of destinations when creating the `origins_destinations` in the first place.

Why not simply change `demand` from a regular dictionary to an instance of the class `defaultdict` and then call `origins` as `demand.keys()`? As before, the order of the elements in the resulting list would not be guaranteed.

I implemented the following method (`build_od`) in `bd.py`.

```python
def build_od(demand_dict):

  od_dict = defaultdict() # keys are origins, values are lists of destinations

  for origin in demand_dict.keys():
    temp_destinations = []
    for d in demand_dict[origin].keys():
      temp_destinations.append(d)

    destinations = [int(i) for i in temp_destinations]
    destinations.sort()
    destinations_sorted = [str(i) for i in destinations]

    od_dict[origin] = destinations_sorted

  return od_dict
```

3.1.3 Additional notes

In doing some bare-bones testing, it seems that the type of the keys in a dictionary has an effect on the order in which they are returned. I created two dictionaries, `test_dict_a` and `test_dict_b`.

```
test_dict_a = {}
test_dict_a['04'] = 4
test_dict_a['01'] = 1
test_dict_a['02'] = 2
test_dict_a['03'] = 3

test_dict_b = {}
test_dict_b['spaghetti'] = 7.50
test_dict_b['pizza'] = 10.00
```

By Python convention, the `test_dict_a.keys()` ought to return the keys in order of insertion (though again, this is not guaranteed), i.e. ['04', '01', '02', '03']. Instead, it returns ['02', '03', '01', '04']. This remains true when the keys are accessed in parallel using `parallel python`. However, `test_dict_b.keys()` produces the expected (though not guaranteed) result – `[spaghetti, pizza]` – both in a standard call and in multiple calls made in parallel using `parallel python`. The origins and destinations in the `demand` dictionary are numbered in much the same way that the keys of `test_dict_a` are – that is, with numbers stored as strings.

3.2 Travel time computation

This bug was identified and shared by Jorge M. Lozano.

3.2.1 Significance

In the original traffic assignment code (see Listing 1), the travel time along an edge is computed as the travel time induced by the additional flow rather than the total flow along the edge. This is obviously problematic and would result in underestimates of the total travel time across the network.

3.2.2 Solution

The solution, as shown in line 33 of Listing 2, is to compute the travel time along an edge using the total flow assigned to that edge.

3.3 Trip assignment

3.3.1 Significance

To track connectivity losses due to bridge damage, I implemented a counter for the number of trips made on the road network to `ita.py`. In doing so, I found that not all trips demanded of the road network are made when there is no damage.

There are 11,179,420 trips demanded of the road network. Under no-damage conditions, I would expect almost (if not) all trips to be assigned to the road network. However, only 7.7 million trips are made on the undamaged road network when the order in which we iterate over origins and destinations is held constant as described in Section 3.1. When the order of origins and destinations is not fixed, about 7.9 million trips are assigned, suggesting this problem is not an unintended result of fixing the order in which we iterate over origins/destinations.

3.3.2 Solution

The original mechanism by which trips were or were not made between origin-destination pairs is depicted schematically in Figure 2 and was carried out as follows. Say we have an origin $\mathcal{O}$ and a destination $\mathcal{D}$. The number of trips demanded between $\mathcal{O}$ and $\mathcal{D}$ is T. The shortest path between $\mathcal{O}$ and $\mathcal{D}$ is returned as a list of nodes, L, as shown in Figure 1. We keep track of whether the trips between $\mathcal{O}$ and $\mathcal{D}$ are made using a boolean, `od_made`. We initially assume that all the trips can be made and so set `od_made = True`. We then have to check this assertion by iterating, in order, over the nodes in L and adding the trips T to the flow on the series of edges defined therein. Each edge – defined by a pair of nodes – has a capacity. If an edge's capacity is greater than 0, we add the flow (T) to that edge. If the edge's capacity is not greater than 0, we do not assign T, and we change `od_made` to `False`, since the path between $\mathcal{O}$ and $\mathcal{D}$ has been interrupted.

Note that in the above description, if the path between $\mathcal{O}$ and $\mathcal{D}$ was interrupted and the trip was not made, we would *still* assign the trips demanded to subsequent edges that have non-zero capacities, as shown in Figure 2 . This didn't make sense to me. Therefore, I modified the the code in `ita.py` such that if we encounter a zero-capacity edge while traversing the shortest path from $\mathcal{O}$ to $\mathcal{D}$, we will now not iterate over the subsequent edges, regardless of their capacity, since they must not be reachable from $\mathcal{O}$. (This is accomplished by adding the `break` in Line 36 of Listing 2). Thus, the flow assigned to edges downstream of the zero-capacity edge will be 0, as shown in Figure 3. If we adhered to the original implementation and did iterate over the subsequent edges, the total travel time and the vehicle miles travelled would be artificially inflated, though the number of trips made (the counter I implemented) would not be.

Despite all the changes described above, not all trips demanded are assigned to the undamaged road network. This is most likely because even in the undamaged network, more than 25% of all edges (representing roads) have capacities of zero. Trips cannot be assigned to zero-capacity edges.

Code version	Total travel time	Total flow	Total vehicle miles travelled
original	2.23×10^{14} s	7.69×10^{6} trips	1.12×10^{8} miles
updated	2.98×10^{13} s	7.63×10^{6} trips	3.51×10^{7} miles

Table 2: Traffic metrics on the undamaged road network.

For complete transparency, I have included the original traffic assignment code from `ita.py` in Listing 1 and the code with my modifications in Listing 2. Note that Listing 2 includes all other modifications listed in this document (i.e. ordered iteration over origins and destinations, correct time computation).

3.4 Graph edge weights

3.4.1 Significance

Trips between point A and point B are assigned by first identifying the shortest path between A and B in the graph of the road network. The shortest path is the path with the smallest sum of the traversal times t_a of each edge in the path. For each edge in the path, we check that its capacity is greater than 0 before adding flow to it.

However, the graph of the road network includes edges with zero capacity (i.e. that cannot be assigned any traffic) but finite traversal times. Therefore, an edge with 0 capacity and a finite traversal time could be identified as being on the shortest path between point A and point B – but in the next step, we would not be able to assign any flow to that edge, and the trip would not be made. This results in the false loss of more than 3 million trips when the network is undamaged.

3.4.2 Solution

The solution is to correct the graph of the road network before assigning any traffic to its edges. The correction is simple – any edge with zero capacity should have a traversal time $t_a = \infty$. The implementation of this correction is shown in Listing 3. It need only be carried out once and the resulting graph saved as a `gpickle`. The correction affects more than 25% of edges in the graph.

The result of this correction is that on the undamaged road network, 10.8 million trips (97%) of the 11.1 million trips demanded can be made. The remaining 0.3 million trips cannot be completed because there is no shortest path between their origin that does not include edges with infinite traversal times t_a.

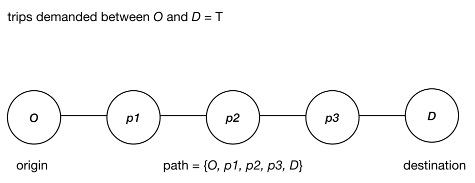

Figure 1: Schematic of shortest path between an origin and destination node.

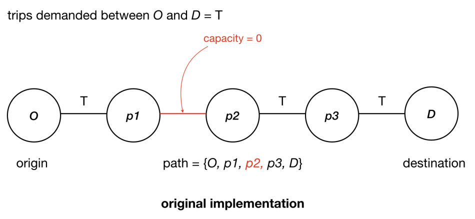

Figure 2: Schematic of original method assigning flow to the edges along the shortest path between an origin and destination node. Even if flow can't be assigned to the edge between p1 and p2, it will be assigned to subsequent edges along the path. Thus, while the trip from the origin to the destination couldn't be made along the shortest path, the travel time and vehicle miles travelled would be calculated as though the trip were made except for a skipped edge. This method of flow assignment would presumably lead to artificially high total travel times and vehicle miles travelled.

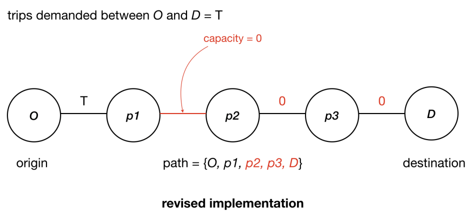

Figure 3: Schematic of revised method of assigning flow to the edges along the shortest path between an origin and destination node. Now, if flow can't be assigned to a particular edge, then the trip cannot be made and the trips demanded will not be assigned to subsequent edges along the path between an origin and a destination.

Listing 1: The original implementation of traffic assignment in ita.py, written by Mahalia Miller. Some comments have been omitted for brevity.

```
def assign(self):
  for i in range(4): #do 4 iterations
    for origin in self.demand.keys():
      paths_dict = nx.single_source_dijkstra_path(self.G, origin, cutoff = None, weight
        ↪ = 't_a')
      for destination in self.demand[origin].keys():
        od_flow = iteration_vals[i] * self.demand[origin][destination]*0.053
        path_list = paths_dict[destination] #list of nodes

        for index in range(0, len(path_list) - 1):
          u = path_list[index]
          v = path_list[index + 1]

          if self.G.is_multigraph():
            num_multi_edges = len(self.G[u][v])
            if num_multi_edges >1: #multi-edge
              best = 0
              best_t_a = float('inf')
              for multi_edge in self.G[u][v].keys():
                new_t_a = self.G[u][v][multi_edge]['t_a'] #causes problems
                if (new_t_a < best_t_a) and (self.G[u][v][multi_edge]['capacity']>0):
                  best = multi_edge
                  best_t_a = new_t_a
            else:
              best = 0
            if (self.G[u][v][best]['capacity']>0):
              self.G[u][v][best]['flow'] += od_flow
              t = util.TravelTime(self.G[u][v][best]['t_0'], self.G[u][v][best]['capacity
                ↪ '])
              travel_time = t.get_new_travel_time(od_flow)
              self.G[u][v][best]['t_a'] = travel_time
          else:
            try:
              if (self.G[u][v]['capacity']>0):
                self.G[u][v]['flow'] += od_flow
                t = util.TravelTime(self.G[u][v]['t_0'], self.G[u][v]['capacity'])
                travel_time= t.get_new_travel_time(od_flow)
                self.G[u][v]['t_a'] = travel_time #in seconds
            except KeyError as e:
              print 'found␣key␣error:␣', e
              pdb.set_trace()

  return self.G
```

Listing 2: The modified implementation of traffic assignment in ita.py, with fixes for the order of iteration over origins and destinations, the computation of travel time, and the assignment of trips.

```
def assign(self):

total_flow = 0 # GB ADDITION -- tracker for number of trips made on the whole network

# GB ADDITION -- adding sorting
origins = [int(i) for i in self.demand.keys()] # get SD node IDs as integers
origins.sort() # sort them
origins = [str(i) for i in origins] # make them strings again

od_dict = bd_test.build_od(self.demand)

for i in range(4): #do 4 iterations
  for origin in origins:

    paths_dict = nx.single_source_dijkstra_path(self.G, origin, cutoff = None, weight
        ↪ = 't_a')

    for destination in od_dict[origin]:

      od_flow = iteration_vals[i] * self.demand[origin][destination]

      path_list = paths_dict[destination] #list of nodes

      od_made = True

      for index in range(0, len(path_list) - 1):
        u = path_list[index]
        v = path_list[index + 1]

        try:
          if (self.G[u][v]['capacity']>0):
            self.G[u][v]['flow'] += od_flow
            t = util.TravelTime(self.G[u][v]['t_0'], self.G[u][v]['capacity'])
            travel_time= t.get_new_travel_time(self.G[u][v]['flow']) # GB MODIFICATION
            self.G[u][v]['t_a'] = travel_time #in seconds

          else:
            od_made = False
            break # GB ADDITION -- we should not continue assigning traffic to edges
                ↪ between origin and destination if destination is not reachable from
                ↪ origin!

        except KeyError as e:
          print('found␣key␣error:␣', e)
          pdb.set_trace()

      total_flow += od_made*od_flow

return self.G, total_flow
```

Listing 3: A method to correct the original road network graph prior to any traffic assignment by ensuring that zero-capacity edges have infinite traversal times.

```

    def correct_graph(G):
        count = 0
        count0 = 0
        count1 = 0
        for edge in G.edges():
                if G[edge[0]][edge[1]]['capacity'] == 0:
                        if G[edge[0]][edge[1]]['t_a'] != float('inf'):
                                G[edge[0]][edge[1]]['t_a'] = float('inf')
                                count += 1
                        if G[edge[0]][edge[1]]['t_0'] != float('inf'):
                                G[edge[0]][edge[1]]['t_0'] = float('inf')
                                count0 += 1
                if G[edge[0]][edge[1]]['distance_0'] == 0:
                        G[edge[0]][edge[1]]['capacity'] = 0
                        G[edge[0]][edge[1]]['t_0'] = float('inf')
                        G[edge[0]][edge[1]]['t_a'] = float('inf')
                        G[edge[0]][edge[1]]['distance'] = 0

                        count1 += 1

        print 'corrected␣', count, 'zero-capacity␣and␣', count1, '␣zero-length␣edges␣in␣
            ↪ graph'

        return G
```