Supplementary Material

Complexity

Note that we could also transform the original problem into a "multiple choice branching" problem. Which is to find a subset A'∈A in a directed graph G=(V,A) and a partition of A into disjoint sets A1, A2, …, Am with the sum of weights in subset A' larger than a given positive integer K such that A' contains no cycles and at most one arc from each partition. By reducing into a 3-SAT problem, the "multiple choice branching" problem turns out to be NP-Complete (Garey, Johnson and Stockmeyer 1976) (Garey and Johnson 1979). To be more specific, it remains NP-Complete even if G is strongly connected and all weights are equal as finding maximum weight branching was viewed as a 2-matroid intersection problem that could be solved in polynomial time (Garey, Johnson and Tarjan 1976). In other words, a maximum weight branching can be viewed as a maximum weight directed spanning tree for a strongly connected graph. Similarly, if the graph is symmetric, the problem become equivalent to the multiple choice span-ning tree problem, another 2-matroid intersection problem that can be solved in polynomial time (Hassin and Levin 2012)(Henn 2007).

In spite of its $O(n^4)$-seemingly structure depicted, the time complexity for our proposed BDPST algorithm ends up in $O(SEV/S)$ or $\boldsymbol{O(EV)}$ since the worst case is to traverse each grid in all directions. Where E refers to the number of maximum directions one spanning tree could try (this variable is set to 2 in our Algorithm, equals to branching factor in the worst case), S refers to the total number of selected and recommended stations, and V indicates the number of grids labelled with road in the given area. Note that V is far greater than S and E. Meanwhile, the preprocessing time is $O(V^2)$ for grid-preprocessing with station-recommendation of constructing a look-up table for further indexing. Which indicates that runtime for our BDPST is not strongly related to the total number of selected and recommended stations (S) but mainly affected by the size of given area (V).

As we mentioned, existing route-planning algorithms are not applicable in our case. But for comparison, the time complexity to deal with our problem utilizing Brute-force searching turns out to be $O(E^V)$, A* algorithm to be $O(E^V)$, Dijkstra's algorithm and Prim's algorithm to be $O(EVlogV)$ or $O(EV+VlogV)$ depending on implementation, and bidirectional search to be $O(SE^{V/S})$. Since V is far greater than S and E in general (note that E refers to the branching factor instead of total number of edges in graph here), could turn out to find approximate solutions in affordable/reasonable time compared to all aforementioned route-planning algorithms..

Pseudocode

Algorithm BDPST

```
1   MV ← {/*must_visit_grids given by user*/}
2   RM ← ∅
3   foreach g ∈ MV do
4     foreach g' ∈ area do
5       gd(g',g) ← G(g',g)×pf(g)
6       agd(g') ← agd(g')+gd(g',g) /*precomputed*/
7   foreach r from 0 to number_of_recommendation
8     foreach g ∈ area and g ∉ MV ∪ RM do
9       Let max be the maximum g of pf(g)-agd(g)
10    RM ← RM ∪ {max}
11    foreach g' ∈ area do
12      gd(g') ← G(g',g)×pf(g)
13      agd(g') ← agd(g')+gd(g',g) /*look-up table instead*/
14  foreach g ∈ area do
15    sid(g) ← -1
16    if g ∈ MV∩RM then
17      root[starter_id] ← g
18      set[starter_id,0].add(root[starter_id])
19      sid(g) ← starter_id
20      count(starter_id) ← 0
21      foreach g' ∈ area do
22        agd[starter_id,0](g') ← agd(g')-2×gd(g',g)
23  route ← ∅
24  foreach i from 0 to infinite do
25    foreach s ∈ MV∩RM do
26      foreach st ∈ set[s,i] do
27        if count(s) ≥ 2 then break
28        nst ← st /*deep copy*/
29        pt(nst,s) ← 0
30        foreach g ∈ nearby and reachable in road do
31          if sid(g) = s then
32            Renew parameters if current path is better
33          else
34            Let choice be the maximum g of
                  (agd[s,i](g)+distScore(unvisitedMV))
35        if choice ≠ null
36          nst.visit(choice)
37          if sid(g) ≠ 1 and count(sid(g)) < 2 then
38            route.addseg(nst, obj(g))
39            count(s) ← count(s)+1
40            count(sid(g)) ← count(sid(g))+1
41            if success to form a route then break
42            foreach g' ∈ area do
43              agd[s,i](g')
                    ← agd[s,i](g')-2×gd(g',root[sid(g')])
44            if count(sid(g)) = 2 /*count(s) = 2*/ then
45              foreach st ∈ MV∩RM and g' ∈ area do
46                agd[st,i](g')←agd[st,i](g')-
                                  2×gd(g',root[sid(g')])
47          else
48            set[s,i+1].add(nst)
49            sid(g) ← s
50            obj(g) ← nst
51        pt(st,s) ← pt(st,s)+1
52        if pt(st,s) > 1 then
53          set[s,i].delete(st)
54      set[s,i+1].add(set[s,i])
55      agd[s,i+1] ← agd[s,i]
56    if success to form a route then break
```