**TSP Project Report, Group 29**

**Group members:**

James Whiteley IV

Gerald Wilson

Eugene Pak

**Outline:**

1. **Pseudocode + description greedy algorithm - DONE**
2. **Pseudocode + description linear programming algorithm - DONE**
3. **Pseudocode + description dynamic programming algorithm**
4. **Description of chosen algorithm**
5. **Pseudocode of chosen algorithm**
6. **Best tours for 3 example files and the time it took - DONE**
7. **Best competition solutions for 3min time limit and unlimited time limit - DONE**

**Greedy Algorithm for TSP**

The greedy algorithm for TSP will choose the smallest distance from the current city to all cities adjacent to the current city. For example, if you are currently in San Jose and the cities adjacent to you are San Francisco, Paso Robles, and Los Angeles, the algorithm will choose San Francisco since it’s the closest city with the least distance. The problem is that the greedy algorithm chooses the locally optimal solution without taking into account the globally optimal solution (which is what we want) and will generally lead to a sub-optimal solution.

Pseudocode:

//start is starting position on graph g

//distance(c1, c2) is a function that returns the distance between two cities c1 and c2

//adj(c1) is a function that returns a list of cities adjacent to c1

//This function takes a graph and starting position (city)

// It returns the order each city is visited as well as the total distance traveled

TSP(start, g):

Visited = [] //track all visited cities so no duplicates

Total\_distance = 0 //add all selected cities to total\_distance for final solution

Not\_visited = g.remove(start) //list of all cities in graph without starting city

Cur = start

While not\_visited != []: // while there are still cities you haven’t visited

adjacent = adj(cur) // list of all cities adjacent to current city

min\_dist = None

min\_city = None

for city in adjacent: //check which city is least distance

if min\_dist == None: // first city checked, make current min

min\_dist = distance(cur, city)

min\_city = city

elif dist(cur, city) < min\_dist: //distance from cur city to this is less than prev smallest

min\_dist = distance(cur, city)

min\_city = city

visited.append(min\_city) //add city to visited list

total\_distance += min\_dist //add distance traveled to total

not\_visited.remove(min\_city) //remove city traveled to from graph

cur = min\_city //set current city to traveled to city for next iteration

visited.append(start) //travel back to starting city once all cities have been visited

total\_distance += dist(cur, start) //add traveling distance from final city to beginning city

return visited, total\_distance

**Integer Linear Programming Algorithm for TSP**

This algorithm attempts to solve the Traveling Salesman problem using the PROC OPTMODEL[1].

This method uses the PROC OPTMODEL to first find integral matching. However, this is not necessarily a tour, and would then not fit the problem criteria. If the solution is a disconnected graph, it is not a tour and violates a subtour constraint. These constraints are added to formulation and the integer program is solved again. This repeats until a solution is a tour of the graph.[1]

Pseudocode:

*\*The code below was created with assistance from SAS documentation on TSP and Integer Linear Programming[1] along with assistance from Yong Wang’s video on Integer Programming with the Traveling Salesman Problem [2]*

/\* iterative solution using the subtour formulation \*/

proc optmodel;

set VERTICES;

set EDGES = {i in VERTICES, j in VERTICES: i > j};

num xc {VERTICES};

num yc {VERTICES};

num numsubtour init 0;

set SUBTOUR {1..numsubtour};

/\* read in the instance and customer coordinates (xc, yc) \*/

read data tspData into VERTICES=[var1] xc=var2 yc=var3;

/\* the cost is the euclidean distance rounded to the nearest integer \*/

num c {<i,j> in EDGES}

init floor( sqrt( ((xc[i]-xc[j])\*\*2 + (yc[i]-yc[j])\*\*2)) + 0.5);

var x {EDGES} binary;

/\* minimize the total cost \*/

min obj =

sum {<i,j> in EDGES} c[i,j] \* x[i,j];

/\* each vertex has exactly one in-edge and one out-edge \*/

con two\_match {i in VERTICES}:

sum {j in VERTICES: i > j} x[i,j]

+ sum {j in VERTICES: i < j} x[j,i] = 2;

/\* no subtours (these constraints are generated dynamically) \*/

con subtour\_elim {s in 1..numsubtour}:

sum {<i,j> in EDGES: (i in SUBTOUR[s] and j not in SUBTOUR[s])

or (i not in SUBTOUR[s] and j in SUBTOUR[s])} x[i,j] >= 2;

/\* this starts the algorithm to find violated subtours \*/

set <num,num> EDGES1;

set INITVERTICES = setof{<i,j> in EDGES1} i;

set VERTICES1;

set NEIGHBORS;

set <num,num> CLOSURE;

num component {INITVERTICES};

num numcomp init 2;

num iter init 1;

num numiters init 1;

set ITERS = 1..numiters;

num sol {ITERS, EDGES};

/\* initial solve with just matching constraints \*/

solve;

call symput(compress('obj'||put(iter,best.)),

trim(left(put(round(obj),best.))));

for {<i,j> in EDGES} sol[iter,i,j] = round(x[i,j]);

/\* while the solution is disconnected, continue \*/

do while (numcomp > 1);

iter = iter + 1;

/\* find connected components of support graph \*/

EDGES1 = {<i,j> in EDGES: round(x[i,j].sol) = 1};

EDGES1 = EDGES1 union {setof {<i,j> in EDGES1} <j,i>};

VERTICES1 = INITVERTICES;

CLOSURE = EDGES1;

for {i in INITVERTICES} component[i] = 0;

for {i in VERTICES1} do;

NEIGHBORS = slice(<i,\*>,CLOSURE);

CLOSURE = CLOSURE union (NEIGHBORS cross NEIGHBORS);

end;

numcomp = 0;

do while (card(VERTICES1) > 0);

numcomp = numcomp + 1;

for {i in VERTICES1} do;

NEIGHBORS = slice(<i,\*>,CLOSURE);

for {j in NEIGHBORS} component[j] = numcomp;

VERTICES1 = VERTICES1 diff NEIGHBORS;

leave;

end;

end;

if numcomp = 1 then leave;

numiters = iter;

numsubtour = numsubtour + numcomp;

for {comp in 1..numcomp} do;

SUBTOUR[numsubtour-numcomp+comp]

= {i in VERTICES: component[i] = comp};

end;

solve;

call symput(compress('obj'||put(iter,best.)),

trim(left(put(round(obj),best.))));

for {<i,j> in EDGES} sol[iter,i,j] = round(x[i,j]);

end;

/\* create a data set for use by gplot \*/

create data solData from

[iter i j]={it in ITERS, <i,j> in EDGES: sol[it,i,j] = 1}

xi=xc[i] yi=yc[i] xj=xc[j] yj=yc[j];

call symput('numiters',put(numiters,best.));

quit;

**Best tours for 3 example files + time:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **File name** | **Best tour** | **Optimal Tour** | **Ratio** | **Total time (sec)** |
| tsp\_example\_1.txt | 131471 | 108159 | 1.215 | 0.006530 |
| tsp\_example\_2.txt | 2952 | 2579 | 1.144 | 0.022675 |
| tsp\_example\_3.txt | 1806314 | 1573084 | 1.148 | 58.542190 |

**Best competition solutions for 3min time limit AND unlimited time limit:**

|  |  |  |
| --- | --- | --- |
| **File name** | **Best tour** | **Total time (sec)** |
| test-input-1.txt | 5582 | 0.005387 |
| test-input-2.txt | 8046 | 0.006425 |
| test-input-3.txt | 14419 | 0.015760 |
| test-input-4.txt | 19254 | 0.047382 |
| test-input-5.txt | 26807 | 0.185257 |
| test-input-6.txt | 38780 | 0.870871 |
| test-input-7.txt | 58854 | 6.112696 |

**References**

[1] (Nov. 2017). SAS [Online]. Available: <http://support.sas.com/documentation/cdl/en/ormpug/63975/HTML/default/viewer.htm#ormpug_milpsolver_sect020.htm>

[2] Wang, Y. (2017, April 10) Available: <https://www.youtube.com/watch?v=nRJSFtscnbA>