**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 - DONE**
4. **Description of chosen algorithm - DONE**
5. **Pseudocode of chosen algorithm - DONE**
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;

**Dynamic Programming Algorithm for TSP**

Solving the traveling salesman problem using dynamic programming takes a bottom-up approach as opposed to a top down brute force approach. This is done by solving a smaller sub problem and using that answer to solve increasingly complex problems. The first step is to generate all possible subsets for all vertices excluding the start vertex. In this case, the smaller sub problem is calculating the smallest sub path and then using the result of this calculation to solver larger sub paths. This means paths between vertices will only need to be calculated once resulting in less computations needed to solve the overall problem. This algorithm solves the problem in exponential time O(2^n n^2) which is much better than factorial time using a brute force method.

Pseudocode:

minimum cost DP(start vertex, set, city root)

create city class with value, child vertices, selected

total cost = infinity

selected = i

root.child vertex = new city

if set is empty

returm cost from vertex and 0

for all vertices in set

create new city as set as child vertex

create new set

remove next vertex from set

vertex cost = calculate cost of visiting current vertex

next min cost = minimum cost DP(next vertex, new set, child vertex)

cur cost = next min cost + vertex cost

if cur cost < total cost

total cost = cur cost

selected = i

i++

child vertex.selected = true

return total cost

**Nearest Neighbor and 2-Opt**

Our approach was to use the greedy nearest neighbor algorithm then optimize the tour with the 2-Opt algorithm. The nearest neighbor algorithm starts at a vertex and selects the nearest neighbor meaning the neighboring vertex with the least cost. This process continues until all cities have been visited. This will visit all cities with the last move linking back to the original city. It is faster than a brute force approach but can lead to non-optimal results. 2-Opt is an heuristic algorithm based on the fact that 2 intersecting lines are not as efficient as 2 straight lines. 2 opt will evaluate all pairs of edges for two cities and do swaps/exchange if it results in the tour having a lower cost. It continues this evaluation and swapping process until there are no more improvements to be found. 2-Opt is an improvement heuristic so it improves upon the non-optimal tour created by the nearest neighbor algorithm.

Pseudocode:

nearest neighbor(adjacency matrix)

get number of cities

starting at city

add city to vector of toured cities

for i to number of cities

for j to number of cities

calculate distance

store distances in vector of distances

while the number of toured cities is less than the total number of cities

for all cities

compare every path of the city

calculate min distance from list

travel to the nearest neighbor

add the neighbor to the toured list

start at the neighbor

return toured cities

two opt(adjacency matrix, tour)

get number of cities

set improve to false

while improve is false

for i to number of cities-1

for j=i+1 to number of cities

calculate swap distance [tour[i]][tour[j]] + [tour[i+1]][tour[j+1]];

calculate old distance [tour[i]][tour[i+1]] + [tour[j]][tour[j+1]];

if old distance is greater than swap distance

swap edges for swap distance

set tour to new tour

keep improve as false

set improve to true

return toured cities

**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>