EE 3980 Algorithms

Homework HW12 Travelling Salesperson Problem

105061110 周柏宇

2020/6/7

1. Introduction

In this homework, we want to solve the Travelling Salesperson Problem as efficiently as possible. We analyze its time and space complexity. In the end, we record the CPU time and output the result.

2. Analysis & Implementation

2.1 Problem Description

The Travelling Salesperson Problem is a problem that aims to minimize the cost of a Hamiltonian circuit. A Hamiltonian circuit is a closed path that visits every vertex once with no repeats. Given a graph with n vertices, there is a total number of $(n-1)!$ distinct visiting order if fixing the starting city, which makes the Travelling Salesperson Problem intractable to find its minimum cost and visiting order.

2.2 Least-Cost Search with Branch and Bound

We already know that to find the minimum cost, there is no better way than trying all the possibilities whether using depth-first or breadth-first

search. However, there are some ways to “estimate” or “bound” the cost at a given node. With that, we can compare it with the minimum cost so far, and decide whether to span the rest of the tree.

One method of bounding the cost of a circuit is that, given a cost matrix, assuming the diagonal is set to infinite, we can find the minimum of every row, subtract the elements in the row by the minimum and sum them up every minimum of the row. Similarly, we can do it for every column. Find the minimum of every column, subtract every element in the column by the minimum and sum up the minimum. Why these two procedures (we refer to as row reduce and column reduce) provide a lower bound of the circuit is because, for a Hamiltonian circuit, a node has exactly one outward link and one inward link. Hence, we have to pick a number in every row as a cost for an outward link. If we simply choose the minimum of every row, which may not from a valid circuit, it surely provides a lower bound. By subtracting out the minimum, we can prevent double-counting when we do the column reduce. The pseudo codes of row reduce and column reduce are provided.

```
1. // Return row-reduced cost matrix.
2. // Input: n x n cost matrix, binary variable doit
3. //        indicating whether to modify the matrix
4. // Output: cost reduced
5. Algorithm RowR(n, cost, doit)
6. {
7.     s_min := 0; // cumulated minimum
8.
9.     for i := 1 to n do {
10.          min := ∞;
11.          for j := 1 to n do {
12.              if (cost[i][j] < min) then {
13.                  // update the minimum of the row
14.                  min := cost[i][j];
15.              }
16.          }
17.          if (min != ∞) then {
18.              if (doit) then { // modify the table of cost
19.                  for j := 1 to n do {
20.                      if (cost[i][j] != ∞) then {
21.                          cost[i][j] := cost[i][j] - min;
22.                      }
23.                  }
24.              }
25.              // accumulate the minimum
26.              s_min := s_min + min;
27.          }
28.      }
29.      return s_min;
30.  }
```

```
1. // Return column-reduced cost matrix.
2. // Input: n x n cost matrix, binary variable doit
3. //        indicating whether to modify the matrix
4. // Output: cost reduced
5. Algorithm ColR(n, cost, doit)
6. {
7.     s_min := 0; // cumulated minimum
8.     for i := 1 to n do {
9.         min := ∞;
10.          for j := 1 to n do {
11.              if (cost[j][i] < min) then {
12.                  // update the minimum of the column
13.                  min := cost[j][i];
14.              }
15.          }
16.          if (min != ∞) then {
17.              if (doit) then { // modify the table of cost
18.                  for j := 1 to n {
19.                      if (cost[j][i] != ∞) then {
20.                          cost[j][i] := cost[j][i] - min;
21.                      }
22.                  }
23.              }
24.              // accumulate the minimum
25.              s_min := s_min + min;
26.          }
27.      }
28.      return s_min;
29.  }
```

When we select a link, say city u to city v, we can further increase the lower bound by the cost of the link, and the lower bound provided by row reduce and column reduce after setting the row of u, column of v and the link from city v to city u to infinite to ensure the choices of links from

a Hamiltonian circuit.

Given a node, we can calculate each lower bound for the links, and prioritize the search for the links with a smaller lower bound. When we reach the bottom of the tree, the lower bound converges to the real cost. With the real cost, it can further rule out unpromising attempts, thus reducing the search space. This concludes the least-cost search with branch and bound algorithm.

```
1. // Initialization call of least-cost search.
2. // Input: Total n cities with cost table cost.
3. // Output: none
4. Algorithm LC_Call(n, cost)
5. {
6.     for i := 1 to n do visited[i] := 0;
7.     visited[1] := 1; // mark city 1 as visited
8.     LB := ∞; // minimum lower bound of the circuit
9.     // current lower bound
10.      R := RowR(cost, 1) + ColR(cost, 1);
11.      LC_d(n, 1, R, cost, LB); // start from city 1
12.  }
```

```
1. // Recursive least-cost search with branch and bound.
2. // Input: Total n cities, current city u has lower bound R.
3. //        Cost table cost, minimum lower bound LB.
4. // Output: none
5. Algorithm LC_d(n, u, R, cost, LB)
6. {
7.     Set row u of cost to ∞;
8.     i_adj := 1;
9.     for i := 1 to n do {
10.          if (visited[i] = 0) then { // try unvisited cities
11.              Set column i of cost and cost[i][u] to ∞;
12.              adj[i_adj].idx := i;
13.              adj[i_adj].R :=
14.              R + cost[u][i] + RowR(cost, 0) + ColR(cost, 0);
15.              Recover column i and cost[i][u];
16.              i_adj := i_adj + 1;
17.          }
18.     }
19.     if (i_adj = 2) then { // only one city to travel
20.         Recover row u of cost to its original values;
21.         if (adj[1] < LB) then LB := adj[1];
22.         return;
23.     }
24.     // Try next city in least-cost order,
25.     // stop if the lower bound is dominated.
26.     while ((s := nextA(adj, a_adj - 1, min) != -1) and
27.            (min.R <= LB)) do {
28.         a_adj := a_adj - 1;
29.         v := min.idx;
30.         visited[v] := 1; // set city v as visited
31.         Set column v of cost and cost[v][u] to ∞.
32.         // modify the cost table
33.         RowR(cost, 1); ColR(cost, 1);
34.         LC_d(n, v, min.R, cost, LB);
35.         visited[v] := 0;
36.     }
37.     Recover row u of cost to its original values;
38. }
```

```
// Return minimum of array.
// Input: array A with n elements, return ret
// Output: exit status
Algorithm nextA(A, n, ret)
{
    if (n <= 0) return -1; // no city to return
    else if (n = 1) { // only one city
        ret = A[1];
        return 0;
      }
      min := A[1];
      for i := 2 to n do {
          if (A[i].R < min) {
              min := A[i].R;
              i_min := i;
          }
      }
      // move the minimum to the end
      ret := A[i_min]; A[i_min] := A[n]; A[n] := ret;
      return 0;
 }
```

Despite all the effort we make to reduce the search space, in complexity theory, this is still an algorithm with exponential time complexity $\mathcal{O}(n!)$ with n being the number of cities

As for the space complexity, the tree search has a maximum depth of n. In each `LC_d`, we make a copy of `cost` because we may need to go back to the parent, which has the unreduced cost matrix. Thus, the maximum memory we allocate at a time is $\mathcal{O}(n^3)$. Because for our testing inputs, the biggest n is 30, otherwise we need to make sure stack overflow does not occur. There are also some

variables like the name of cities, the array to mark visited cities, etc., but the space they occupy is negligible compare with $\mathcal{O}(n^3)$.

3. Result and Observation

To measure the performance, we execute the program to solve the Travelling Salesperson Problem with a different number of cities and record the CPU time.

Input file	*# cities n*	*# nodes visited*	$\frac{\#\ visited}{n!}$	*Distance*	*Execution time* [s]
t1.dat	5	11	0.09166	28	0.00002
t2.dat	10	265	0.00007	84	0.00090
t3.dat	15	593	4.53e-10	105	0.00314
t4.dat	20	44317	1.82e-14	132	0.32444
t5.dat	25	521363	3.36e-20	153	6.19763
t6.dat	30	168494	6.35e-28	166	2.99428

As the table shows, with branch and bound, we can solve a problem with $\mathcal{O}(n!)$ in a reasonable amount of time. Also, notice that the least-cost search does not guarantee the minimum number of nodes to try. Some cases like t5.dat and t6.dat, the number of nodes tried is more in t5.dat than in t6.dat. This tells us if we somehow find a high lower bound early, we can solve the problem fast.