Homework #4 for Algorithms

Due: Thursday, March 28 by 5 PM

Chapter 4: Greedy

1) Use Prim’s algorithm to find a minimum spanning tree for the following graph. Show the steps as was done on the class handout.

2) Complete the trace of Dijkstra’s Algorithm (Algorithm 4.3) algorithm for the following graph. Show the steps by updating the length and touch arrays as was done in class.

3) Write a program using the greedy approach to solve the Traveling Salesperson problem. (Your program will not always find the minimum-length tour.) Run your program on the above graph for problem 2, and hand in the output showing the tour found.

Chapter 5. Backtracking

4) Consider the following 0-1 Knapsack problem with four items and a knapsack weight limit of W=13 oz.

<table>
<thead>
<tr>
<th>Item</th>
<th>Weight</th>
<th>Value</th>
<th>Value/Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4 oz.</td>
<td>$40</td>
<td>$10/oz.</td>
</tr>
<tr>
<td>2</td>
<td>7 oz.</td>
<td>$42</td>
<td>$6/oz.</td>
</tr>
<tr>
<td>3</td>
<td>10 oz.</td>
<td>$50</td>
<td>$5/oz.</td>
</tr>
<tr>
<td>4</td>
<td>3 oz.</td>
<td>$12</td>
<td>$4/oz.</td>
</tr>
<tr>
<td>5</td>
<td>3 oz.</td>
<td>$9</td>
<td>$3/oz.</td>
</tr>
</tbody>
</table>

a) Complete the backtracking state-space tree. Use a bound (e.g., best possible solution we could hope to achieve in the subtree) calculation of like we did in class of:

\[
\text{bound of a node} = \text{value in knapsack} + \text{fractional knapsack problem on remaining items}
\]

5) Write a program using Backtracking to solve the Traveling Salesperson problem. (Your program should always find the minimum-length tour.) Run your program on the above graph for problem 2, and hand in the output showing the tour found. Use a greedy algorithm to calculate if a node is NOT promising so pruning can be performed.
(1) HW #4

<table>
<thead>
<tr>
<th>2 3 4 5 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 5</td>
</tr>
<tr>
<td>-1 3 4 0 5</td>
</tr>
<tr>
<td>-1 1 4 2 3</td>
</tr>
<tr>
<td>2 3 4 5 6</td>
</tr>
<tr>
<td>-1 -1 4 -1 3</td>
</tr>
<tr>
<td>-1 -1 -1 -1</td>
</tr>
</tbody>
</table>

touch

<table>
<thead>
<tr>
<th>2 3 4 5 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1</td>
</tr>
<tr>
<td>1 2 2 1 1</td>
</tr>
<tr>
<td>1 2 2 3 3</td>
</tr>
<tr>
<td>1 2 2 3 3</td>
</tr>
<tr>
<td>1 2 6 3 3</td>
</tr>
</tbody>
</table>

MST

([Diagram of a network with nodes V1, V2, V3, V4, V5, V6 and edges labeled with distances])
Shortest paths from $v_1$

- $v_1 \rightarrow v_2$
- $v_1 \rightarrow v_3$
- $v_1 \rightarrow v_2 \rightarrow v_4$
- $v_1 \rightarrow v_2 \rightarrow v_3$


```cpp
#include <iostream>
#include <iostream>

using namespace std;

const int MAX_SIZE = 100;
const int NOT_ON_PARTIAL TOUR = -1;
const int TRUE = 1;
const int FALSE = 0;

// Prototypes:
void TSP(int n, int weights[][MAX_SIZE],
    int & totalWeight, int tour[MAX_SIZE]);

void ReadWeights(int n, int weights[][MAX_SIZE]);

int main() {
    int weights[MAX_SIZE][MAX_SIZE];
    int n;
    int totalWeight;
    int tour[MAX_SIZE+1];
    int toVertex;

    cout << "Enter n (# vertices): ";
    cin >> n;

    ReadWeights(n, weights);

    TSP(n, weights, totalWeight, tour);

    cout << endl << "The tour is: ";
    for (toVertex = 0; toVertex < n; toVertex++) {
        cout << " v" << tour[toVertex] << " ";
    }
    // end for
    cout << endl << "The total weight of this tour is " << totalWeight << endl;
} // end main

void TSP(int n, int weights[][MAX_SIZE], int & totalWeight, int tour[MAX_SIZE])
```

---

**tsp_greedy.cpp**

---

// Description: Homework #4 exercise. This program uses a
// greedy approach to find a good (but not necessarily globally optimal)
// solution to the traveling salesman problem (TSP). In the TSP
// problem you are given an n x n adjacency matrix W representing the
// weights of edges in a directed graph. W[i][j] represents the weight of
// the edge from vertex i to vertex j. The TSP is to find the minimum
// cost tour. A tour starts at some vertex, say vertex 0, visits every
// vertex exactly once and then returns to vertex 0.
//
// This greedy algorithm starts the tour at vertex 0 and then repeatedly
// extends the partial tour by follows the lowest weight edge from the last
// vertex on the tour that goes to a vertex not already in the tour.
```cpp
int vertex;
int lastVertexOnPartialTour;
int toVertex;
int foundFirstUnassignedToVertex;
int minWeightToVertex;
int vertexOnPartialTour[MAX_SIZE];

for (vertex = 0; vertex < n; vertex++) {
    vertexOnPartialTour[vertex] = NOT_ON_PARTIAL_TOUR;
} // end for (vertex...

tour[0] = 0;
totalWeight = 0;
lastVertexOnPartialTour = 0;
for (vertex = 0; vertex < n-1; vertex++) {
    foundFirstUnassignedToVertex = FALSE;
    // Greedy choice: extend partial tour by follow the lowest weight edge
    // from the last vertex on partial tour that goes to a vertex not already
    // in the partial tour.
    for (toVertex = 1; toVertex < n; toVertex++) {
        if (vertexOnPartialTour[toVertex] == NOT_ON_PARTIAL_TOUR) {
            if (!foundFirstUnassignedToVertex) {
                minWeightToVertex = toVertex;
                foundFirstUnassignedToVertex = TRUE;
            } else if (foundFirstUnassignedToVertex &&
                      weights[lastVertexOnPartialTour][toVertex] <
                      weights[lastVertexOnPartialTour][minWeightToVertex]) {
                minWeightToVertex = toVertex;
            } // end if
        } // end if (vertexOnPartialTour[toVertex]...
    } // end for (toVertex...
    totalWeight = totalWeight +
    weights[lastVertexOnPartialTour][minWeightToVertex];
    vertexOnPartialTour[minWeightToVertex] = TRUE;
    tour[vertex+1] = minWeightToVertex;
    lastVertexOnPartialTour = minWeightToVertex;
} // end for (vertex...

// Complete the tour back to vertex 0
for (vertex = n-1; vertex >= 0; vertex--) {
    tour[vertex+1] = vertex;
    totalWeight += weights[vertex][toVertex];
} // end for (vertex...

void ReadWeights(int n, int weights[][MAX_SIZE]) {

    int fromVertex;
    int toVertex;

    for (fromVertex = 0; fromVertex < n; fromVertex++) {
        cout << "Enter all " << n << " weights for fromVertex " << fromVertex << ": ";
        for (toVertex = 0; toVertex < n; toVertex++) {
            cin >> weights[fromVertex][toVertex];
        } // end for (toVertex ...
tsp_greedy.cpp

} // end for (fromVertex ...
} // end ReadWeights
tsp.cpp

Requirements: Mark Fienup
Description: This program performs a backtracking solution to the traveling salesman problem (TSP). It's promising function calculates a bound by summing the minimum edges leaving vertices not on the partial tour.

#include <iostream>
#include <iomanip>
#include <limits>

#define MAX 100
#define TRUE 1
#define FALSE 0

using namespace std;

// prototypes
void ReadArray(int E[][MAX], int & n);
void PrintArray(int E[][MAX], int n);
void tspInitialize(int included[], int path[]);
void tspInitialize(int included[], int path[]);
void tsp(int level, int pathLength);
int promising(int nextVertex, int pathLength, int level);

// global variables
int E[MAX][MAX];
int included[MAX];
int partialPath[MAX + 1];
in n;
in bestLength = INT_MAX;
in bestPath[MAX + 1];

int main (void) {
    int i;

    ReadArray (E, n);
    cout << "The adjacency matrix is:
";
    PrintArray (E, n);
    tspInitialize(included, partialPath);
    tsp(1, 0);

    cout << "Best tour length is " << bestLength << endl;
    cout << "The best tour is:
";
    for (i = 0; i <= n; i++) {
        cout "bestPath[i] " "
    } // end for (i...
    cout << endl;
}

void tsp(int level, int pathLength) {
    int completePathLength;
    int nextVertex;
    int i;
// cout << "level = " << level << " pathLength = " << pathLength << endl;

for (nextVertex = 1; nextVertex < n; nextVertex++) {
    // cout << "for nextVertex = " << nextVertex << " level = " << level << endl;
    if (!included[nextVertex]) { // only consider next vertices that are not
        // already on the partial tour
        if (level == n-1) { // for a complete tour see if it is the best
            completePathLength = pathLength + E[partialPath[level-1]][nextVertex]
                + E[nextVertex][0];
            if (bestLength > completePathLength) {
                bestLength = completePathLength;
                for (i = 0; i < level; i++) { // end for(i...)
                    bestPath[i] = partialPath[i];
                }
                bestPath[level] = nextVertex;
                bestPath[level+1] = 0;
                // cout << "new best length " << bestLength << endl;
            } // end if
        } else if (promising(nextVertex, pathLength, level)) { // only check promising nodes
            included[nextVertex] = TRUE; // update global arrays to reflect child
            partialPath[level] = nextVertex;
            tsp (level+1, pathLength + E[partialPath[level-1]][nextVertex]);
            included[nextVertex] = FALSE; // change back to the parent's state
        } // end if
    } // end if
} // end tsp

// this promising function calculates a bound by summing the minimum edges leaving
// vertices not on the partial tour
int promising(int nextVertex, int pathLength, int level) {
    int bound;
    int i, test;
    int minEdge;

    // update the partial path to reflect the child
    pathLength += E[partialPath[level-1]][nextVertex];
    included[nextVertex] = TRUE;

    // find the minimum edge leaving the vertex on the end of the partial tour.
    // This vertex cannot go back to vertex 0 (the start) because there are other
    // vertices that must be visited first.
    minEdge = INT_MAX;
    for (i = 1; i < n; i++) {
        if (!included[i] && E[nextVertex][i] < minEdge) {
            minEdge = E[nextVertex][i];
        }
    }
    bound = pathLength + minEdge;
tsp.cpp

// temporarily set to FALSE so vertex 0 is included in the minimum calculation
included[0] = FALSE;
for (test = 1; test < n; test++) {
    if (!included[test]) {
        minEdge = INT_MAX;
        for (i = 0; i < n; i++) {
            if (!included[i] && test != i && E[test][i] < minEdge) {
                minEdge = E[test][i];
            } // end if
        } // end for (i
        bound += minEdge;
    } // end if (!included...
} // end for (test...

included[0] = TRUE;
included[nextVertex] = FALSE;

if (bound < bestLength)
    return TRUE;
else
    return FALSE;
} // end promising

// Initial the partial tour to only include vertex 0
void tspInitialize(int included[], int path[]) {
    int i;

    included[0] = TRUE;
    path[0] = 0;

    for (i = 1; i < n; i++) {
        included[i] = FALSE;
    } // end for (i
} // end tspInitialize

// Read the number of vertices in the graph followed by the adjacency matrix.
void ReadArray(int E[][MAX], int & n) {
    int r, c;

    cin >> n;
    for (r = 0; r < n; r++) {
        for (c = 0; c < n; c++) {
            cin >> E[r][c];
        } // end for (c
    } // end for (r
} // end ReadArray

// Prints the adjacency matrix
void PrintArray(int E[][MAX], int n) {
    int r, c;

    for (r = 0; r < n; r++) {
        for (c = 0; c < n; c++) {
            cout << setw(7) << E[r][c] << "    ";
        } // end for (c
    } // end for (r
} // end PrintArray
tsp.cpp

cout << "\n";
} // end for (r
} // end ReadArray