MATH182 FINAL
DUE July 31, 2020

Question 1. *(5pts) Write an algorithm which implements depth-first search without recursion. You might find a* stack *data structure to be useful. Your algorithm should take as input a graph* $G = (V, E)$ *and upon termination should successfully assign the* predecessor*,* discovery time*, and* finishing time *attributes to all vertices. You can also use the* color *attribute.*

Solution. Following is pseudocode[1] for a non-recursive algorithm that implements depth-first search:

DFS(G)

```
1  // properly initialising vertices' attributes
2  for each vertex u ∈ G.V
3       u.colour = WHITE
4       u.π = NIL
5  time = 0        // setting global time
6  // for each non-visited node, visit node and all its descendants
7  for s in G.V
8       if s.colour == WHITE
9            DFS-VISIT(G, s, time)
```

DFS-VISIT($G, s, time$)

```
 1  whiteStack = empty stack          // stack of discovered but (possibly) unexplored nodes
 2  blackStack = empty stack          // stack of discovered and fully explored nodes
 3  PUSH(whiteStack, s)               // s is the first node to explore
 4  // iterate while there are still descendants to explore
 5  while whiteStack not empty
 6       u = POP(whiteStack)          // extract node added last
 7       // explore node if unexplored
 8       if u.colour == WHITE
 9            time = time + 1
10            u.d = time              // set discovery time (only set once)
11            u.colour = GREY         // set to GREY (for cases u ∈ G.Adj[u])
12            // iterate over unexplored adjacent nodes
13            for each v in G.Adj[u]
14                 if v.colour == WHITE
15                      v.π = u                 // may happen multiple times; last π is true π
16                      PUSH(whiteStack, v)     // insert node into stack of unexplored nodes
17            u.colour = BLACK        // node's immediate descendants have been discovered
18            PUSH(blackStack, u)     // push to stack of finished nodes
19 // blackStack is a stack of finished nodes in the order they were finished (last on top)
20 while blackStack not empty
21       u = POP(blackStack)          // extract last finished node
22       time = time + 1              // update time
23       u.f = time                   // set finish time
```

[1] C++ code in Appendix I

Note that in line 14, we actually traverse through the vertices adjacent to u in the opposite order to get exactly the same forest as we would in recursive DFS, since in our algorithm, the adjacent vertex we visit last will be explored first. Following is a description of the algorithm.

The function DFS is identical to the DFS defined in notes; it's the implementation of DFS-VISIT that we change to a non-recursive implementation. (Of course, both functions could easily be combined into one function, but we define them separately to keep the pseudocode a little neater). Also note that in our code, a vertex is WHITE if it's exploration has not yet begun (regardless of whether it has been discovered), GREY if it is currently being explored, and BLACK if it has been explored. We say a vertex has been "explored" if all of its adjacent vertices have been discovered and pushed to *whiteStack*. Correspondingly, the attribute d refers to the time the exploration of a vertex begins, not when it's discovered.

Iterating over vertices in DFS, we call DFS-VISIT on every vertex that has not yet been visited. In DFS-VISIT lines 1-3, we initialise *whiteStack* and *blackStack* to be empty stacks; *whiteStack* will store all vertices that have been discovered but not necessarily fully explored, and *blackStack* stores all vertices that have been fully explored. Naturally, we push s to *whiteStack* and start the **while** loop in lines 5-18 from there.

In lines 6-7, we extract the vertex last added to *whiteStack* and check if it's unexplored. If indeed it is unexplored, we start its exploration by setting its d attribute to the incremented time and its *colour* attribute to GREY. In lines 13-16, we insert into *whiteStack* all vertices adjacent to u whose exploration has not yet started after setting their predecessor attribute π to u. Note that a vertex's predecessor attribute may be set multiple times — appropriately, the last value it is assigned is the true predecessor, since the vertex will be explored according to its most recent insertion in *whiteStack*. After we have stacked all the unexplored vertices of u for exploration in *whiteStack*, we mark u as explored (setting $u.colour$ to BLACK) and push it onto *blackStack*.

While we may add the same vertex to *whiteStack* multiple times, we explore each vertex exactly once, thanks to the status check in line 8. We iterate this process until *whiteStack* is empty, indicating all the nodes reachable from s have been explored. At this stage, *blackStack* is a stack of finished nodes in the order they were finished, with the first finished vertex on bottom and the last on time. From this, by Parenthesis Theorem, we can define the finish times for all the explored nodes as in lines 21-23.

DFS-VISIT therefore visits every node reachable from s in a depth-first order.

Question 2. *(5pts) Give the optimal parenthesization for a matrix chain product* $\langle A_1, A_2, A_3, A_4, A_5, A_6 \rangle$*, where*

- A_1 *has size* 5×10
- A_2 *has size* 10×3
- A_3 *has size* 3×12
- A_4 *has size* 12×5
- A_5 *has size* 5×50
- A_6 *has size* 50×6

Solution. We use the MATRIX-CHAIN-ORDER algorithm[2] from the notes:

[2] C++ code in Appendix III

MATRIX-CHAIN-ORDER(p)

```
1  n = p.length − 1
2  let m[1..n, 1..n] and s[1..n − 1, 2..n] be new tables
3  for i = 1 to n
4        m[i, i] = 0
5  for l = 2 to n
6        for i = 1 to n − l + 1
7              j = i + l − 1
8              m[i, j] = ∞
9              for k = i to j − 1
10                   q = m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
11                   if q < m[i, j]
12                         m[i, j] = q
13                         s[i, j] = k
14 return m and s
```

We define $p := [5, 10, 3, 12, 5, 50, 6]$, the array of dimensions (in order) of given matrices, and call MATRIX-CHAIN-ORDER(p). In lines 1-2, we set $n = p.length - 1 = 6$ and create tables $m[1..n, 1..n]$, which will store the optimal solutions to $A_{i..j}$ subproblems, and $s[1..n, 1..n]$, which will store the index k which achieves the optimal cost for the $m[i, j]$ subproblem. Since computing the single matrix A_i (viewed as a 1-element matrix product) requires zero scalar multiplications, we initialise all $m[i, i] = 0$.

In the outer **for** loop in lines 5-13, we iterate over the lengths of nontrivial matrix chains in increasing order. For each length l, we iterate over all possible first indices i from 1 to $n - l + 1$. In the innermost **for** loop in lines 9-13, we find the optimal solution to the $A_{i..j}$ subproblem and store the total number of multiplications in the optimal solution in $m[i, j]$ and the relevant parenthesisation in $s[i, j]$.

Iterating over this process, at the end of our algorithm, m and s are in the following state:

$$m = \begin{array}{c} \begin{matrix} 1 & 2 & 3 & 4 & 5 & 6 \end{matrix} \\ \left[\begin{matrix} 0 & 150 & 250 & 405 & 1655 & 2010 \\ & 0 & 360 & 330 & 2430 & 1950 \\ & & 0 & 180 & 930 & 1770 \\ & & & 0 & 3000 & 1860 \\ & & & & 0 & 1500 \\ & & & & & 0 \end{matrix}\right] \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{matrix} \end{array} \qquad s = \begin{array}{c} \begin{matrix} 2 & 3 & 4 & 5 & 6 \end{matrix} \\ \left[\begin{matrix} 1 & 2 & 2 & 4 & 2 \\ & 2 & 2 & 2 & 2 \\ & & 3 & 4 & 4 \\ & & & 4 & 4 \\ & & & & 5 \end{matrix}\right] \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{matrix} \end{array}$$

We now use the PRINT-OPTIMAL-PARENS function[3] defined in lecture notes:

PRINT-OPTIMAL-PARENS(s, i, j)

```
1  if i == j
2        print "A_i"
3  else print "("
4        PRINT-OPTIMAL-PARENS(s, i, s[i, j])
5        PRINT-OPTIMAL-PARENS(s, s[i, j] + 1, j)
6        print ")"
```

From the matrix s generated from MATRIX-CHAIN-ORDER(p), we call PRINT-OPTIMAL-PARENS($s, 1, 6$). We then get the parenthesisation $\underline{(A_1A_2)((A_3A_4)(A_5A_6))}$, which has total cost $s[1, 6] = 2010$.

[3] C++ code in Appendix II

Question 3. *(5pts) Suppose $G = (V, E)$ is a connected, undirected graph. Show that if an edge (u, v) is contained in some minimum spanning tree of G, then it is a light edge crossing some cut of the graph.*

Solution. Let T be a minimum spanning tree of G that contains the edge (u, v) and has total weight W. We show (u, v) must be a light edge crossing some cut of the graph.

Since T is a minimum spanning tree, removing the edge (u, v) yields two disjoint minimum spanning trees. Let the minimum spanning tree containing u be T_u with weight W_u and that containing v be T_v with weight W_v when the edge (u, v) is removed from T. Let V_u be the set of vertices in T_u and V_v be the set of vertices in T_v.

Since T_u and T_v are disjoint minimum spanning trees, V_u and V_v must also be disjoint; furthermore, since $T_u \cup \{(u, v)\} \cup T_v = T$, we know $V_u \cup V_v = V$. Consider the cut (V_u, V_v) of G and assume towards contradiction that (u, v) is not a light edge across this cut. By assumption, there exists some edge (s, t) crossing the cut such that the weight of (s, t) (say, $w(s, t)$) is less than the weight of (u, v) (say, $w(u, v)$). Now consider the tree $T' := T_u \cup \{(s, t)\} \cup T_v$, i.e., the tree formed by adding the edge (s, t) to the graph $T_u \cup T_v$. Since (s, t) is now connected the two formerly-disjoint spanning trees, T' must be a spanning tree of G.

Let W' be the weight of this new spanning tree. By definition of T' and (s, t), we have

$$W' = W_u + w(s, t) + W_v < W_u + w(u, v) + W_v = W \implies W' < W$$

This contradicts our assumption that T is a minimum spanning tree. Therefore the edge (u, v) must be a light edge of the cut (V_u, V_v). ■

Question 4. *(5pts) Describe an efficient algorithm that, given a set $\{x_1, x_2, \ldots, x_n\}$ of points on the real line, determines the smallest set of unit-length closed intervals that contains all of the given points. Argue that your algorithm is correct.*

Solution. We assume the set $X = \{x_1, x_2, \ldots, x_n\}$ is sorted such that $x_1 \leq x_2 \leq \ldots \leq x_n$. The following pseudocode illustrates an efficient greedy algorithm to find the desired set of intervals:

```
FindMinimalIntervalSet(X)
 1  S = φ     // set of intervals
 2  i = 1     // counter for first/smallest element x_i not yet in some interval
 3  // iterating until all x_i are in some interval
 4  while i ≤ n
 5        S = S ∪ {[x_i, x_i + 1]} // inserting interval [x_i, x_i + 1] into S
 6        // finding first/smallest k such that x_k is not yet in some interval
 7        k = i + 1
 8        while k ≤ n and x_k ≤ x_i + 1
 9              i = i + 1
10        i = k // updating i
11  return S
```

This algorithm simply iterates over the ordered elements x_i in X that aren't yet in some interval in S and makes the greedy choice to insert the interval $[x_i, x_i + 1]$ into S. We show making this greedy choice yields an optimal solution. In particular we show for a non-empty, ordered set of points X_k with smallest point x_1, the interval $[x_1, x_1 + 1]$ is included in some minimal set of unit-length intervals that contains all $x_j \in X_k$.

Let X_k be a non-empty, ordered set of points. Furthermore, let S_o be an ordered, minimal set of unit-length intervals that contains all $x_j \in X_k$, and let $x_1 := X_k[1]$ be the smallest element in

X_k. x_1 must lie in the "earliest" interval $[a, a+1]$ in S_o (i.e., the interval located leftmost on the real line) since otherwise, the set is not optimal.

If $[a, a+1] = [x_1, x_1+1]$, we are done. Otherwise, by assumption $x_1 \in [a, a+1]$, we have $a < x_1 \leq a+1 < x_1+1$. Since for all $x_j \in X_k \cup [a, a+1]$, we know $x_j \geq x_1$ (by assumption on x_1) and $x_j \leq a+1 < x_1+1$, we also have $x_j \in [x_1, x_1+1]$. We can therefore replace the interval $[a, a+1]$ with $[x_1, x_1+1]$ without needing to add any additional intervals. In other words, the set $S := S_o \backslash \{[a, a+1]\} \cup \{[x_1, x_1+1]\}$ is also an optimal set.

Therefore, since the set X is ordered, when some x_i has not yet been included in an interval, we have the equivalent of the subproblem $X_k := \{x_k : i \leq k \leq n\}$. Making the greedy choice every time we encounter an element from x_1 onwards that has not yet been included in S therefore gives an optimal solution. ■

Question 5. *(True/False) For each of the following statements indicate whether they are **true** or **false**. Each question is worth 2pts, a blank answer will receive 1pt. Recall that "true" means "always true" and "false" means "there exists a counterexample".*

(1) Since Max-Heapify *runs in* $O(\lg n)$ *time, the process* Build-Max-Heap *runs in* $\Theta(n \lg n)$ *time.*

(2) In the rod-cutting problem, the naive recursive solution Cut-Rod *runs in polynomial time.*

(3) If a problem has a greedy algorithm solution, then it necessarily does not have a dynamic programming solution; and vice versa.

(4) The algorithm BFS *visits every vertex of the graph.*

(5) During DFS, if a vertex v has its color changed from white to gray while u is gray, then $u.f < v.f$.

Solution.

(1) False. The Build-Max-Heap algorithm actually runs in $\Theta(n)$ time.

(2) False. The Cut-Rod algorithm actually runs in exponential time.

(3) False. Since the properties a problem must have to be solvable with a greedy algorithm and those it must have to be solvable with a dynamic programming algorithm are not mutually exclusive, a problem may have both a greedy algorithm solution and a dynamic programming solution. For instance, the activity selection can be solved with both a greedy algorithm and with a dynamic programming solution.

(4) False. Breadth-first search only visits every vertex of the graph reachable from the source vertex.

(5) False. If a vertex v has its color changed from white to gray while u is gray, it is necessarily discovered after u is discovered, i.e., $u.d < v.d$. By the Parenthesis Theorem, this means $u.f > v.f$.

APPENDIX

I. C++ code for functions DFS and DFS-Visit. Structures `Graph` and `Node` from `question1.h` defined in Appendix II.

```cpp
#include <iostream>
#include <vector>
#include <stack>
#include <map>
#include "question1.h"
using namespace std;

void DFS(const Graph G);
void DFSVisit(const Graph& G, Node* s, int& time);

int main() {

    // testing DFS
  // creating nodes (same as example from class)
  Node y("y"); Node z("z"); Node s("s"); Node f("f");
  Node x("x"); Node w("w"); Node v("v"); Node u("u");
  map<Node*, vector<Node*>> Adj{
    {&s, vector<Node*> {&z,& w}},
    {&z, vector<Node*> {&y,& w}},
    {&y, vector<Node*> {&x}},
    {&x, vector<Node*> {&z}},
    {&w, vector<Node*> {&x}},
    {&v, vector<Node*> {&s,& w}},
    {&f, vector<Node*> {&v,& u}},
    {&u, vector<Node*> {&f}},
  };

  vector<string> ordering = { "s", "z", "y", "x", "w", "f", "v", "u" };

  Graph G(Adj, ordering);
  DFS(G);
  // the same forest tree as that in class is generated
}

void DFS(const Graph G) {
  for (Node* v : G.V) {
    v->colour = WHITE;
    v->pred = nullptr;
  }
  int time = 0;
  for (Node* s : G.V)
    if (s->colour == WHITE)
      DFSVisit(G, s, time);

  // printing results
  for (Node* s : G.V) {
    cerr << s->name << endl;
    cerr << "Start time:  " << s->d << endl;
```

```
    cerr << "End time:     " << s->f << endl;
    if (s->pred != nullptr)
      cerr << "Predecessor: " << s->pred->name << endl;
    cerr << endl;
  }
}

void DFSVisit(const Graph& G, Node* s, int& time) {
  stack<Node*> whiteStack;
  stack<Node*> blackStack;
  whiteStack.push(s);
  while (!whiteStack.empty()) {
    Node* u = whiteStack.top(); whiteStack.pop();
    if (u->colour == WHITE) {
      time++;
      u->d = time;
      u->colour = GREY;
      vector<Node*> adjacents = G.Adj.find(u)->second;
      for (int vIndex = adjacents.size() - 1; vIndex >= 0; vIndex--) {
        Node* v = adjacents[vIndex];
        if (v->colour == WHITE) {
          v->pred = u;
          whiteStack.push(v);
        }
      }
      u->colour = BLACK;
      blackStack.push(u);
    }
  }

  while (!blackStack.empty()) {
    Node* u = blackStack.top(); blackStack.pop();
    time++;
    u->f = time;
  }
}
```

II. Definitions for structures `Graph` and `Node`.

```
#pragma once

#include <vector>
#include <string>
#include <map>
using namespace std;

enum Colour { WHITE, GREY, BLACK };

struct Node {
  // Node name
  string name;
  // Node properties
  Colour colour;
  Node* pred;
```

```
  int d;
  int f;

  Node(string name)
    : name(name) {}
};

struct Graph {
  // vertices in graph
  vector<Node*> V;
  // map of vertices to adjacent vertices
  map<Node*, vector<Node*>> Adj;

  Graph(map<Node*, vector<Node*>> Adj, vector<string> ordering)
    : Adj(Adj) {
    // inserting vertices
    for (string v : ordering)
      for (map<Node*, vector<Node*>>::iterator itr = Adj.begin(); itr != Adj
    .end(); itr++)
        if (itr->first->name == v) {
          V.push_back(itr->first);
          break;
        }
    }

};
```

III. C++ code for MatrixChainOrder and PrintOptimalParens

```
#include <vector>
#include <iostream>
#include <string>
using namespace std;

void PrintOptimalParens(const vector<vector<int>>& s, int i, int j);

void MatrixChainOrder(vector<int> p) {
  // finds optimal matrix chain parenthesisation

  const int n = p.size() - 1;
  vector<vector<int>> m(n + 1, vector<int>(n + 1, 0));
  vector<vector<int>> s(n, vector<int>(n + 1, 0));
    // we create larger matrices than is really needed for 1-based indexing

  for (int i = 0; i <= n; i++)
    m[i][i] = 0;

  for (int l = 2; l <= n; l++) {
    for (int i = 1; i <= n - l + 1; i++) {
      int j = i + l - 1;
      m[i][j] = INT_MAX;  // sentinel
      for (int k = i; k <= j - 1; k++) {
        int q = m[i][k] + m[k + 1][j] + p[i - 1] * p[k] * p[j];
        if (q < m[i][j]) {
```

```
          m[i][j] = q;
          s[i][j] = k;
        }
      }
    }
  }

  // returns s (with 1-based indexing)
  PrintOptimalParens(s, 1, n);
}

void PrintOptimalParens(const vector<vector<int>>& s, int i, int j) {
  if (i == j)
    cout << "A" << i;
  else {
    cout << "(";
    PrintOptimalParens(s, i, s[i][j]);
    PrintOptimalParens(s, s[i][j] + 1, j);
    cout << ")";
  }
}
```