CSE 202 Theory Problems: Introduction Modules

Hao-en Sung [A53204772] (wrangle1005@gmail.com)
Department of Computer Science
University of California, San Diego
San Diego, CA 92092

Abstract

CSE 202 Report for theory problem 1.

1 Theory problem 1: Sum of squares of Fibonacci numbers

It is known that $F_0 = 0$, $F_1 = 1$, $F_2 = 1$. With the mathematical induction, I can write down the following statements.

$$\text{When } n = 1,\ \sum_{i=1}^{1} F_i^2 = 1 = F_1 * F_2 = 1 * 1, \text{ which fulfills the given equation.} \tag{1}$$

$$\text{Assume that } n = k,\ \sum_{i=1}^{k} F_i^2 = F_k \cdot F_{k+1}, \tag{2}$$

$$\text{when } n = k+1,\ \sum_{i=1}^{k+1} F_i^2 = F_k \cdot F_{k+1} + F_{k+1}^2 = F_{k+1} \cdot (F_k + F_{k+1}) = F_{k+1} \cdot F_{k+2}. \tag{3}$$

Thus, I proved that $\sum_{i=1}^{n} F_i^2 = F_n \cdot F_{n+1},\ \forall n \geq 1$.

CSE 202 Theory Problems: Greedy Modules

Hao-en Sung [A53204772] (wrangle1005@gmail.com)
Department of Computer Science
University of California, San Diego
San Diego, CA 92092

Abstract

CSE 202 Report for theory problem 2 and 3.

Theory problem 2: Changing money

Proof for Greedy Property

I would like to claim a safe move is to take the coin with current largest possible denomination.

To prove this, assume that we did not take the coin with largest possible denomination in one of our previous step, and we currently have c_1 1-coins, c_5 5-coins, c_{10} 10-coins and r money unchanged, where $c_1 + c_5 \cdot 5 + c_{10} \cdot 10 + r = m$. One can always reduce the total number of used coins by changing some coins with smaller denominations to a larger one.

It is noticeable that this assumption can be held only if any larger denomination are integral multiple of smaller ones.

Time analysis

Since one can assume that the smaller denomination for a coin is 1-coin, one can always reduce value 1 from m. Thus the time complexity is $O(m)$.

Theory problem 3: Organizing a party

Proof for Greedy Property

I would like to claim a possible solution is to assume all n members would attend the party in the beginning. Later, a safe move is to remove any illegal member.

To prove this, one can tell that removing an illegal person immediately from the potential list will not affect the possible optimal solution, since he already has less than 2 friends or 2 non-friends. Thus, it as a safe move.

Time analysis

Since I at most need to remove n potential attendees from the list, there should be a outer for loop running in time $O(n)$. For each removal, one need to check all possible candidates, which also runs in time $O(n)$. At the end, one need to spend $O(n)$ time in checking whether one certain member has no less than 2 friends and no less than 2 non-friends. Thus, the overall time complexity should be $O(n \times n \times n) = O(n^3)$.

CSE 202 Theory Problems: Divide and Conquer Modules

Hao-en Sung [A53204772] (wrangle1005@gmail.com)
Department of Computer Science
University of California, San Diego
San Diego, CA 92092

Abstract

CSE 202 Report for theory problem 3-1, 3-2, and 3-3.

Theory problem 3-1: Fixed point in an array

Algorithm Statement

As I am given an array A with length n, current left boundary $l = 0$, and right boundary $r = n - 1$, my algorithm will first iteratively check whether $A[(l + r)/2]$ is equal to $(l + r)/2$ or not.

There are going to be three cases. If they are equal, then I have done my work. If $A[(l + r)/2] > (l + r)/2$, it is clear that all the elements in A after $(l + r)/2$ won't match their indexes because of the strictly increasing property for A. Thus, set $r = (l + r)/2 - 1$. Otherwise, if the third case occurs, I just choose the counter side, which is $l = (l + r)/2 + 1$. At the end, this algorithm would indicate its failure if there is no such match after checking $l = r$.

Proof for Correctness

The correctness of this algorithm is easy to tell. Every time I will discard half of the impossible candidates, and the match will be found in some iteration, if there exists any of them.

Time analysis

Since I will discard half of the candidate in each iteration, the overall time complexity is going to be $O(n)$.

Theory problem 3-2: Organizing a lottery

Algorithm Statement

My algorithm in fact is not so related to divide-and-conquer but only sorting. Firstly, I would separate each (a_i, b_i) pair into $(a_i, +1)$ and $(b_i + 1, -1)$, and sort them by the first element. Later, I would linearly go through all discrete points on the line, including x_i, a_i, or $b_i + 1$, with a integer s for summation. If current visiting point is x_i, I simply output the current summation value. Otherwise, I will perform $+1$ or -1 based on the second element of each sorted pair.

Proof for Correctness

The correctness of this algorithm is still straightforward. I basically separate each $[a_i, b_i] = 1$ segments as $[a_i, \infty] = +1$ and $[b_i + 1, \infty] = -1$. Thus, the answer should be correct even after sorting the boundary.

Time analysis

Assume there are n points and m segments. My algorithm is going to spend $O(2m\log(2m))$ in sorting and $O(n+2m)$ in traversing. Thus, the final time-complexity is going to be $O(2m\log(2m)+n+2m) = O(m\log(m)+n)$.

Theory problem 3-3: Finding a peak in sublinear time (advanced)

Algorithm Statement

My solution is derived during the discussion with one of my classmates, Ming-Lun.

Firstly, let me define the matrix M with left-column boundary l, right-column boundary r, top-row boundary u, and bottom-row boundary b. I am interesting to find out the maximum value in the center row and center column of M, which is denoted as $M((u+d)/2,:)$ and $M(:,(l+r)/2)$, respectively.

Without losing generality, let us assume that the maximum occurs at $M(p,(l+r)/2)$, which is in the upper part of central column, i.e. $M((u:(u+d)/2,(l+r)/2)$. The procedure obviously ends if

$$M(p,(l+r)/2) \geq M(p,(l+r)/2-1)$$
and
$$M(p,(l+r)/2) \geq M(p,(l+r)/2+1).$$

Otherwise, I would iterate the upper-left side of M if

$$M(p,(l+r)/2) \leq M(p,(l+r)/2-1)$$

by setting $r=(l+r)/2-1$ and $b=(u+d)/2-1$;
or iterate the top-right side of M if

$$M(p,(l+r)/2) \leq M(p,(l+r)/2+1)$$

by setting $l=(l+r)/2+1$ and $b=(u+d)/2-1$.

Proof for Correctness

The correctness of this algorithm can be proved as follows. It is noticeable that, for the proof simplicity, I would still take the upper-left corner as an example.

If for some p at the upper central column has maximum value among central row and central column but is smaller than the left element, i.e. $M(p,(l+r)/2) < M(p,(l+r)/2-1)$, I can have following derivations.

In view of $M(p,(l+r)/2-1)$ is the maximum value among central row and central column, one has

$$M(p,(l+r)/2) < M(p,(l+r)/2-1)$$
$$M(p,(l+r)/2) < \max(M((u+d)/2,:),M(:,(l+r)/2).$$

And thus, I proved that upper-left corner is a valid subproblem, where exists a locally peak value apart from the boundary. One then can confidently update $r=(l+r)/2-1$ and $b=(u+d)/2+1$.

Time analysis

Since for every iteration I can use $O(2*n)$ time to find out the next subproblem, the time complexity can be written in a recursive form

$$T(n) = T(n/4)+O(n).$$

Based on the Master Theorem, the overall complexity of solving this problem is $O(n) = o(n^2)$.

CSE 202 Theory Problems: Dynamic Programming Modules

Hao-en Sung [A53204772] (wrangle1005@gmail.com)
Department of Computer Science
University of California, San Diego
San Diego, CA 92092

Abstract

CSE 202 Report for theory problem 4-1 and 4-2.

Theory problem 4-1: Planning a trip

Algorithm Statement

The recursive formula of my algorithm can be written as follows.

$$f(i) = \min\left(f(j) + (a_j - a_i - 200)^2,\ \forall\ i < j \leq n\right),\ \forall\ 0 \leq i < n$$
$$f(n) = 0$$

Thus, for function call at each status i, I am going to iterate through all possible $j, \forall\ i < j \leq n$ and calculate the corresponding cost. At the end, I record the minimum cost for current status and return to previous function call.

Proof for Correctness

The recursive formula is pretty straightforward. On top of that, my memorization policy for dynamic programming function call assures the correct time complexity of solving this problem.

Time analysis

There are at most n status, i.e. $1, \cdots, n$. For each status, there is going to be a for loop iterates through at most n status afterwards. Thus, the overall time complexity is going to be $O(n^2)$.

Theory problem 4-2: Partitioning souvenirs

Algorithm Statement

The recursive formula of my algorithm can be written as follows. It is noticeable that $f(i, S_1, S_2)$ is true if any of the three cases is true.

$$f(i, S_1, S_2) = \begin{cases} f(i+1, S_1 \cup x_i, S_2) & \text{1st person get } x_i \\ f(i+1, S_1, S_2 \cup x_i) & \text{2nd person get } x_i\ ,\ \forall\ 0 \leq i < n \\ f(i+1, S_1, S_2) & \text{3rd person get } x_i \end{cases}$$

$$f(n, S_1, S_2) = \begin{cases} \text{True} & \text{If } \sum_{i \in S_1} x_i = \sum_{j \in S_2} x_j = \frac{1}{3}\sum_k s_k \\ \text{False} & \text{Otherwise} \end{cases}$$

Proof for Correctness

My recursive formula considers all possibilities, which assures the inclusion of the best solution. Besides, I use memorization policy for dynamic programming again here. So the time complexity is going to be polynomial but not exponential.

Time analysis

As we can tell: there are at most $n \cdot X \cdot X$ status, where X is the largest possible value of $\sum_i x_i$. For each status, there are going to be three possibilities that we need to take into consideration. Thus, the overall time complexity is $O(n \cdot X \cdot X \cdot 3) = O(nX^2)$.

CSE 202 Theory Problems:
Basic Data Structure Modules

Hao-en Sung [A53204772] (wrangle1005@gmail.com)
Department of Computer Science
University of California, San Diego
San Diego, CA 92092

Abstract

CSE 202 Report for theory problem 5-1.

Theory problem 5-1: Computing tree height

Algorithm Statement

For each input parent_i, I will build a directed edge from $\text{node}_{\text{parent}_i}$ to node_i. In practice, I would use **vector< vector<int> > edges** to record all edges, where **edges[i]** indicates a list of children of node_i. After building all the edges, I will just recursively calculate the height of node_i as

$$\text{height}_i = \left(\max_{\text{node}_j \in \textbf{edges[i]}} \text{height}_j \right) + 1.$$

It is clear that the base case would be the leaf nodes, for whom has height 1.

Proof for Correctness

The correctness of this algorithm is based on the recursive definition of tree's height.

Time analysis

Since each node would be concerned and visited exactly once, the overall time complexity would be $O(n)$.

CSE 202 Theory Problems: Priority Queues and Disjoint Sets

Hao-en Sung [A53204772] (wrangle1005@gmail.com)
Department of Computer Science
University of California, San Diego
San Diego, CA 92092

Theory problem 6-1: Computing k-th smallest element in a heap

Algorithm Statement

Firstly, I will create a min_priority_queue to maintain candidates in order, which are in the format of (value, position). The motivation is that all parent nodes are smaller or equal to their children. Thus, I need to add children nodes only if their parents have been taken into consideration. The pesudocode of my algorithm is attached as below.

Algorithm 1 K-th smallest element in a heap

```
1:  min_priority_queue < pair < int, int >> B;
2:  B.push(A[0], 0);
3:  count = k;
4:  while count > 1 do
5:      val, pos = B.top(); B.pop();
6:      count = count − 1;
7:      B.push(A[2 ∗ pos + 1], 2 ∗ pos + 1);
8:      B.push(A[2 ∗ pos + 2], 2 ∗ pos + 2);
9:  val, pos = B.top(); B.pop();
10: return val;
```

Proof for Correctness

If I am going to find the 1-th smallest element in a heap, the answer will be trivial — just the root.

If I have found $k-1$-th smallest element in a heap, then I would like to claim that k-th smallest element must have been added into my consideration or, in other words, is one of the children of previous $k-1$ smaller elements. It can be easily proved by contradiction: if it is not one of the children of previous $k-1$ smaller elements, there must be another node links between them, which cannot be true, or this node is not the k-th smallest one.

With two above statements, I prove the correctness of my algorithm by inductive reasoning.

Time analysis

It is obvious that my while loop, which is in line 4, will run at most $k-1$ times, thus, there are at most $1 + 2 \cdot (k-1) = 2k-1$ push operations. On top of that, since I will push two elements after one pop element for $k-1$ times, the maximum size will be $1 + (k-1) \cdot (2-1) = k$. Based on priority queue property, each push and pop operation will cost $O(\log k)$ time, and thus, my algorithm results in overall $O(k \log k)$ time complexity.

CSE 202 Theory Problems: Decomposition of Graphs

Hao-en Sung [A53204772] (wrangle1005@gmail.com)
Department of Computer Science
University of California, San Diego
San Diego, CA 92092

Theory problem 7-1: Computing sums of degrees of neighbors

Algorithm Statement

The solution is quite intuitive. Since every edge connects two nodes, I can just iterate through each edge and increase the summation for two endpoints by 1.

Algorithm 1 Computing sums of degrees of neighbors

```
1: vector< pair<int,int> > E;
2: vector<int> cnt_V;
3: for edge: edges do
4:     cnt_V[edge.first] = cnt_V[edge.first] + 1;
5:     cnt_V[edge.second] = cnt_V[edge.second] + 1;
6: return cnt_V;
```

Proof for Correctness

The degree for each node is the number of edges it has. Thus, iterating through each edge with corresponding updates will produce the correct results.

Time analysis

Since E edge is visited exactly once, and there are overall V points need to be concerned (or said returned), the overall time complexity is $O(V + E)$.

Theory problem 7-2: Checking Hamiltonicity of a DAG

Algorithm Statement

It is known that checking Hamiltonicity of a general graph is a NP-complete problem; however, it may not be the case on a specific graph, like DAG.

With the constraint of DAG, I can express the original graph in terms of topological sort, where vertex with lower visiting indexes will appear earlier than the larger ones. After that, I just need to check whether every two consecutive points sorted by visitng indexes form a edge in original graph or not.

Algorithm 2 Checking hamiltonicity of a DAG

```
function CHECK
    TS ← Find out the topological sort for G
    for i = 0 to n − 2 do
        if E[TS[i], TS[i + 1]] does not exist then
            return False;
    return True;
```

Proof for Correctness

According to the property of topological sort, the prerequisite points, or said lower visiting index points, will be in the former part of the list. Besides that, since Hamiltonicity does not allow me to "trace back" to previous points during the Depth First Search (DFS) process, the edges for all consecutive points in TS must exist in original graph; or there must be no valid Hamilton Path. For example, if TS can be expressed as $[n_1, n_2, n_3, \cdots, n_k]$ (sorted by visiting indexes), then $(n_1, n_2), (n_2, n_3), \cdots (n_{k-1}, n_k)$ must all exist in original graph to guarantee one Hamilton Path solution. It is noticeable that the case of not-connected graph can be also detected in this algorithm.

Time analysis

Finding the topological sort requires DFS through the whole graph, which costs $O(V + E)$ time. Moreover, checking whether the existence of edges for all consecutive points in TS costs $O(V)$ time with edges storing in adjacent matrix form. Thus, the overall time complexity is $O(V + E + V) = O(V + E)$.

CSE 202 Theory Problems: Paths in graphs

Hao-en Sung [A53204772] (wrangle1005@gmail.com)
Department of Computer Science
University of California, San Diego
San Diego, CA 92092

Theory problem 8-1: Finding a shortest cycle

Algorithm Statement

Firstly, it is noticeable that there is no E in time analysis. In other words, I must prevent my algorithm from extreme cases that there are multiple edges between two nodes, i.e. $O(V^2) < O(E)$. To solve this, if there are multiple edges between two nodes, I simply keep the smallest distance one and discard the rest of them. After that, there is at most one edge between every two nodes, i.e. $O(V^2) = O(E)$.

My solution is quite intuitive: if there is a cycle contains edge $e = (a, b)$, there must exist a path from b to a. That is to say, if I want to find a cycle with shortest distance summation, I just need to find out the shortest path from b to a after I discard edge (a, b).

Algorithm 1 Finding a shortest cycle

```
1: vector< vector< int > > E;          ▷ adjacent matrix with at most one edge between two nodes
2: if E(a, b) == ∞ then                ▷ additional check for existence of edge (a, b)
3:     return ∞;
4: else
5:     E(a, b) = ∞;
6:     return ShortestPath(b, a, E);
```

Proof for Correctness

A cycle can be decomposed into two parts: (a, b) and a path from b to a except self-cycle. Thus, to find out the smallest cycle is to simply discard edge (a, b) and find the shortest path from b to a in the remaining graph.

Time analysis

Condition analysis and the corresponding modification will cost constant time. The shortest path algorithm will cost $O(V^2)$ time when using Dijkstar algorithm with adjacency matrix implementation. Thus, the final time complexity of my algorithm is going to be $O(1 + V^2) = O(V^2)$.

Theory problem 8-2: Generalized shortest paths

Algorithm Statement

Similar to the previous problem, I need to prevent my algorithm from extreme cases that there are multiple edges between two nodes, i.e. $O(V^2) < O(E)$. I again only keep at most one edge with the smallest distance between each two nodes and discard the rest of them. After that, there is at most one edge between every two nodes, i.e. $O(V^2) = O(E)$.

Later, for a directed graph, I can simply "decompose" a node u with vertex cost c into an edge $(u_{in}, u_{out}) = c$. By doing so, I reduce this "generalized shortest paths" problem into normal shortest path problem and it can be solved accordingly.

Algorithm 2 Generalized shortest paths

```
vector< int > V;                                              ▷ vertex cost
vector< vector< int > > E;          ▷ adjacent matrix with at most one edge between two nodes
vector< vector< int > > nE;                                   ▷ new built graph with decomposed nodes
for i = 0 to i = V − 1 do
    for j = 0 to j = V − 1 do
        if E(i, j)! = ∞ then
            nE(V + i, j) = E(i, j);                           ▷ edge leaves from v becomes leaving v_out
    nE(i, V + i) = V(i)                                       ▷ decompose node i by adding edge (i, V + i) = V(i);
DistanceList = ShortestPathToAllPoints(s, E);                 ▷ DistanceList ∈ R^{V+V};
return DistanceList[V: V+V-1];
```

Proof for Correctness

From the previous algorithm statement, I not only decompose u into (u_{in}, u_{out}) with cost $V[u]$, but also assign $u = u_{in}$ and replace edge (u_{out}, v_{in}) for each edge (u, v). By doing so, to find out shortest path from s to u $, \forall\ u \in V$, i.e. $[s, v^1, \cdots, v^k, u]$, becomes to explore $[(s_{in}, s_{out}), (s_{out}, v^1_{in}), (v^1_{in}, v^1_{out}), \cdots, (v^k_{out}, u_{in}), (u_{in}, u_{out})]$. It then turns out to be a normal shortest path problem.

Time analysis

By decomposition of every node, there are going to be $2V$ nodes in new built graph. It is known that the Dijkstra algorithm with adjacent matrix implementation can be done in square of number of nodes. Thus, the overall time complexity becomes $O((2V)^2) = O(4V^2) = O(V^2)$.

CSE 202 Theory Problems:
Suffix Trees

Hao-en Sung [A53204772] (wrangle1005@gmail.com)
Department of Computer Science
University of California, San Diego
San Diego, CA 92092

Theory problem 9-1: Shortest Non-Shared Substring of Two Strings

Algorithm Statement

My algorithm works as follows.

1. Add all suffix of text_1 into suffix trees with color 1
2. Add all suffix of text_2 into suffix trees with color 2 (Notice that an edge shared by both text_1 and text_2 is $1|2 = 3$)
3. Traverse through the suffix tree once to find out the shortest path from root that is colored with color 1 only.

Algorithm 1 Shortest Non-Shared Substring of Two Strings

```
1: Tree = SuffixTree()                                  ▷ initialize an empty tree
2: Tree = addText(text_1 + "$", 1)                      ▷ insert all suffix substrings of text_1 with color 1
3: Tree = addText(text_2 + "$", 2)                      ▷ insert all suffix substrings of text_2 with color 2
4: DFS through Tree once, find out the shortest edge with color 1
```

Proof for Correctness

If two given texts are not exactly the same, there must exist an edge that are colored differently. Then our goal is to find it out while DFS through the tree once and recover the string we concerned during our function call return.

In fact, I have implemented it in *C++* and successively passed the Coursera judge. For more details about my implementation, please refer to Code 1

Time analysis

The time complexity of building up the tree twice is $O(|text|)$. Later, I need DFS the tree again within $O(|text|)$ time. Thus the overall time complexity is $O(|text|)$.

Appendix

```cpp
#include <cstdio>
#include <string>
#include <vector>
#include <utility>
#include <unordered_map>
#include <assert.h>

using namespace std;

struct Tree {
    int s, e, c;
    vector<Tree*> ch;
    Tree(): ch(5, NULL) {}
    Tree(int a, int b, int c): s(a), e(b), c(c), ch(5, NULL) {} // A, C, T, G, $
};

struct Sol {
    string str;
    bool operator < (const Sol &sol) const {
        return str.length() < sol.str.length();
    }
    Sol operator + (const Sol &sol) {
        Sol ret;
        ret.str = str + sol.str;
        return ret;
    }
    Sol(): str(2010, 'Z') {}
    Sol(string s): str(s) {}
};

unordered_map<char,int> um_ci;
unordered_map<int,char> um_ic;

void addText(Tree *root, string &n_text, string &o_text, int t, int c) {
    if (t == n_text.length()) {
        return;
    } else {
        int lb = um_ci.at(n_text[t]);
        if (root->ch[lb] == NULL) {
            root->ch[lb] = new Tree(t, n_text.length(), c);
        } else {
            int s = root->ch[lb]->s;
            int e = root->ch[lb]->e;
            int l = 0;
            while (s + l < e and o_text[s+l] == n_text[t+l]) {
                assert(t+l < n_text.length());
                l += 1;
            }

            if (s + l == e) {
                root->ch[lb]->c |= c;
                addText(root->ch[lb], n_text, o_text, t+l, c);
            } else {
                Tree *tmp = root->ch[lb];
                tmp->s = s+l;

                root->ch[lb] = new Tree(s, s+l, tmp->c | c);
                int nlb = um_ci.at(o_text[s+l]);
                root->ch[lb]->ch[nlb] = tmp;

                addText(root->ch[lb], n_text, o_text, t+l, c);
```

```
62              }
63          }
64      }
65  }
66
67  void dfs(Tree *root, string &text_1, string &text_2) {
68      for (int i = 0; i < 5; i++) {
69          if (root->ch[i] != NULL) {
70              int s = root->ch[i]->s;
71              int e = root->ch[i]->e;
72              int c = root->ch[i]->c;
73              printf("%d %d %d\n", s, e, c);
74              if (c == 1) {
75                  printf("text_1: ");
76                  for (int i = s; i < e; i++) {
77                      printf("%c", text_1[i]);
78                  }
79              } else if (c == 2) {
80                  printf("text_2: ");
81                  for (int i = s; i < e; i++) {
82                      printf("%c", text_2[i]);
83                  }
84              } else {
85                  printf("share: ");
86                  for (int i = s; i < e; i++) {
87                      printf("%c", text_1[i]);
88                  }
89              }
90              puts("");
91              dfs(root->ch[i], text_1, text_2);
92          }
93      }
94  }
95
96  Sol solve(Tree *root, string &text_1, string &text_2) {
97      Sol best_sol;
98      for (int i = 0; i < 4; i++) {
99          if (root->ch[i] == NULL) {
100             // do nothing
101         } else if (root->ch[i]->c == 1) { // edge owned by text_1
102             int s = root->ch[i]->s;
103             Sol sol(text_1.substr(s, 1));
104             best_sol = min(best_sol, sol);
105         } else if (root->ch[i]->c == 2) { // edge owned by text_2
106             // do nothing
107         } else { // shared edge
108             int s = root->ch[i]->s;
109             int e = root->ch[i]->e;
110             Sol sol(text_1.substr(s, e-s));
111             Sol ret = solve(root->ch[i], text_1, text_2);
112             best_sol = min(best_sol, sol+ret);
113         }
114     }
115     return best_sol;
116 }
117
118 int main() {
119     um_ci['A'] = 0; um_ic[0] = 'A';
120     um_ci['C'] = 1; um_ic[1] = 'C';
121     um_ci['G'] = 2; um_ic[2] = 'G';
122     um_ci['T'] = 3; um_ic[3] = 'T';
123     um_ci['$'] = 4; um_ic[4] = '$';
124
125     char c_text_1[2010], c_text_2[2010];
126     scanf("%s%s", c_text_1, c_text_2);
```

```
127
128     string text_1 = c_text_1;
129     text_1.push_back('$');
130
131     string text_2 = c_text_2;
132     text_2.push_back('$');
133
134     Tree *root = new Tree();
135     for (unsigned i = 0; i < text_1.length(); i++) {
136         addText(root, text_1, text_1, i, 1);
137     }
138     for (unsigned i = 0; i < text_2.length(); i++) {
139         addText(root, text_2, text_1, i, 2);
140     }
141     //dfs(root, text_1, text_2);
142
143     Sol sol = solve(root, text_1, text_2);
144     if (sol.str.length() == 2010) {
145         puts("Two strings are exactly the same.");
146     } else {
147         printf("%s\n", sol.str.c_str());
148     }
149 }
```

Listing 1: Non-shared Substring in Text 1 Only

CSE 202 Theory Problems: Flows in Networks

Hao-en Sung [A53204772] (wrangle1005@gmail.com)
Department of Computer Science
University of California, San Diego
San Diego, CA 92092

Theory problem 10-1: Number Puzzle

Algorithm Statement

To model this problem, I will introduce n nodes for rows, n nodes for columns, and two nodes for source and sink, respectively.

Algorithm 1 Number Puzzle

1: Introduce nodes R_1, ..., R_n, C_1, ..., C_n
2: Introduce source node S and sink node T
3: Build up edges (S, R_i) with capacity r_i, for all i
4: Build up edges (C_j, T) with capacity c_j, for all j
5: Build up edges (R_i, C_j) with capacity M, for all (i, j) pairs
6: Solve maximum flow problem from S to T

Proof for Correctness

It can be found out that the edge (S, R_i) properly restrict the summation of all values on the row i is below r_i, since each value in the matrix is non-negative. Similarly, (C_i, T) also set a proper constraint for the summation of column j. Edges with capacity M are then added as the upper-bound value for each cell in the origin matrix.

With these three capacity constraints, it is easy to tell that the solution found by solving this maximum flow problem is still a feasible solution to the original matrix filling problem. On the other hand, if there exists some solutions for matrix filling problem, maximum flow algorithm can also always find out at least one of them.

Time analysis

With modeling the original problem into maximum flow problem, there are going to be $2n + 2$ nodes and $n + n^2 + n = n^2 + 2n$ edges. If I apply Edmonds–Karp algorithm onto it, I am guaranteed to solve it in $O(V^2E)$ time, which is $O((2n + 2)^2 \cdot (n^2 + 2n)) = O(n^4)$ in this case.

CSE 202 Theory Problems: NP-complete Problems

Hao-en Sung [A53204772] (wrangle1005@gmail.com)
Department of Computer Science
University of California, San Diego
San Diego, CA 92092

Theory problem 11-1: Hitting Set

Algorithm Statement

To prove Hitting Set is a NP-complete problem, I need to reduce a well-known NP-complete problem to it, for example: Vertex Cover.

Lets define the NP-problem — Vertex Cover as following: given V nodes and E edges, and we need to check whether there is a subset of at most k nodes can cover the whole graph.

The reduction procedure is quite simple — regard each existing edges in E as a set, and the goal is to find a subset with at most k elements that can intersect each set.

Proof for Correctness

If there is a valid vertex cover using a subset of nodes with size s, where $s \leq k$, then the value on those nodes are also a valid subset of hitting set, because each edge u, v, which is covered in graph, also intersects with the subset of values. The proof for another direction is similar.

The time for reduction is also feasible. One needs to only transform each edge (u, v) in vertex cover into a set $\{u, v\}$ in hitting set, which takes $O(E)$ time.

CSE 202 Theory Problems: Coping with NP-completeness

Hao-en Sung [A53204772] (wrangle1005@gmail.com)
Department of Computer Science
University of California, San Diego
San Diego, CA 92092

Theory problem 12-1: Rescheduling Exams

Algorithm Statement

Following the instructions, it is intuitive to use three nodes to represent each node in the graph with three possible colors: R, G, and B. It is then easy for me to add some constraints in order to transform the original problem into a 2-SAT problem.

Those constraints include:

1. Exactly one of the other two colors different from the original one can be true.
2. Two adjacent nodes cannot share same color.
3. Each node cannot be the same as original color.

After turning it into 2-SAT problem, I can use *Strongly Connected Component (SCC)* to examine the satisfiability and find out one solution if it is satisfiable.

Algorithm 1 Rescheduling Exams

Given nodes V, colors C, and edges E
Introduce variables R_i, G_i, B_i, for $i \in [1, |V|]$
Add constraints: exactly one of $\{R_i, G_i, B_i\} \setminus C_i$ can be true, for $i \in [1, |V|]$
for $(u, v) \in E$ **do**
    Add constraint: R_u and R_v cannot be both true
    Add constraint: G_u and G_v cannot be both true
    Add constraint: B_u and B_v cannot be both true
for $i = 1 : |V|$ **do**
    Add constraint: R_i is false, if C_i is 'R'
    Add constraint: G_i is false, if C_i is 'G'
    Add constraint: B_i is false, if C_i is 'B'
solve_2SAT()

Proof for Correctness

Since all needed constraints are added, 2-SAT solver can return one feasible solution if any. Or, this problem is unsatisfiable. For more details of my implementation, please refer to Code 1.

Time analysis

With my settings, there are going to be $3|V|$ nodes, $2|V|$ clauses for "exactly one of other two colors" constraint, $3|E|$ clauses for "different adjacent color" constraint, and $|V|$ clauses for "different from original color" constraint. Therefore, there are at the end $6|V|$ nodes and $3|V|+3|E|$ edges in 2-SAT graph, which can be solved by SCC in $O(6|V|+(3|V|+3|E|)) = O(9|V|+3|E|) = O(|V|+|E|)$.

Appendix

```cpp
#include <cstdio>
#include <vector>
#include <utility>
#include <stack>
#include <string>
#include <unordered_map>
#include <assert.h>

using namespace std;

vector<int> dfn;
vector<int> low;
vector<bool> inStack;
stack<int> st;
vector<int> component;
vector<vector<int>> edges;

struct Var {
    int pa;
    int na;
    int pb;
    int nb;
    Var() {}
    Var(int va, int ca, int vb, int cb, const int n) {
        pa = 3*va + ca;
        na = pa + 3*n;
        pb = 3*vb + cb;
        nb = pb + 3*n;
    }
};

void dfs(int t, int &cnt, int &nc) {
    dfn[t] = low[t] = cnt++;
    st.emplace(t);
    inStack[t] = true;

    for (auto nt : edges[t]) {
        if (dfn[nt] == -1) {
            dfs(nt, cnt, nc);
            low[t] = min(low[t], low[nt]);
        } else if (inStack[nt] == true) {
            low[t] = min(low[t], low[nt]);
        }
    }

    if (dfn[t] == low[t]) {
        int top;
        do {
            top = st.top(); st.pop();
            component[top] = nc;
            inStack[top] = false;
        } while (top != t);
        nc += 1;
    }
```

```
}

int main() {
    unordered_map<char,int> um;
    um['R'] = 0;
    um['G'] = 1;
    um['B'] = 2;

    int n, m, a, b;
    char nodes[1010];
    scanf("%d%d%s", &n, &m, nodes);

    edges = vector<vector<int>>(6*n, vector<int>());
    for (int i = 0; i < n; i++) {
        int c = um.at(nodes[i]);
        Var var;
        if (c == 0) {
            var = Var(i, 1, i, 2, n);
        } else if (c == 1) {
            var = Var(i, 0, i, 2, n);
        } else {
            var = Var(i, 0, i, 1, n);
        }
        edges[var.na].emplace_back(var.pb);
        edges[var.nb].emplace_back(var.pa);
        edges[var.pa].emplace_back(var.nb);
        edges[var.pb].emplace_back(var.na);

        var = Var(i, c, i, c, n);
        edges[var.pa].emplace_back(var.na);
    }

    for (int i = 0; i < m; i++) {
        scanf("%d%d", &a, &b);
        a -= 1;
        b -= 1;
        for (int j = 0; j < 3; j++) {
            Var var(a, j, b, j, n);
            edges[var.pa].emplace_back(var.nb);
            edges[var.pb].emplace_back(var.na);
        }
    }

    dfn = vector<int>(6*n, -1);
    low = vector<int>(6*n, -1);
    inStack = vector<bool>(6*n, false);
    component = vector<int>(6*n, -1);

    st = stack<int>();
    int cnt = 0, nc = 0;
    for (int i = 0; i < 6*n; i++) {
        if (dfn[i] == -1) {
            dfs(i, cnt, nc);
        }
    }

    /* Check UNSATISFIABLE */
    for (int i = 0; i < 3*n; i++) {
        if (component[i] == component[3*n+i]) {
            puts("Impossible");
            return 0;
        }
    }

    string str;
```

```
    for (int i = 0; i < n; i++) {
        int cnt = 0;
        for (int j = 0; j < 3; j++) {
            if (component[3*i+j] < component[3*n+3*i+j]) {
                cnt += 1;
                if (j == 0) {
                    str.push_back('R');
                } else if (j == 1) {
                    str.push_back('G');
                } else {
                    str.push_back('B');
                }
            }
        }
        assert(cnt == 1);
    }
    printf("%s\n", str.c_str());
}
```

Listing 1: Non-shared Substring in Text 1 Only