Brandon Smith Dec. 5, 2019

Comp 360: Assignment 5

1Blue-3CoL:

1) For each v_i in V $\quad V$ = set of nodes v_i in G
2) $G_i \leftarrow$ copy G.
3) Remove v_i and connected edges from G_i (effectively colouring it B).
4) if 2-CoL(G_i)
5) return true
6) else
7) return false

2-CoL(G):

1) Pick random v_i from V colouring it red.
2) Perform DFS on G starting on v_i alternating node colours on each step.
3) IF ever a neighbour of any node has same colour
4) Then return false
5) IF DFS finishes without any neighbours sharing colours
6) Then return true.

2-CoL is already polynomial time on the number of nodes travelled and neighbours checked. The algorithm 1Blue-3-CoL simply loops over each v_i in V and removes v_i and it's edges then calls 2-CoL. so $\approx O(|V| \times O(2\text{-CoL})) \Rightarrow$ 1Blue-3-CoL is polynomial time because poly-time multiplied by poly-time is still poly-time.

---

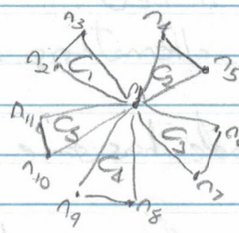
2. Algorithm I: For any given triangle C_i there are 4 cases for an optimal solution:
   1) C_i has 0 nodes connected to any other triangle C_j in which case the optimal would remove 1 node.
   2) C_i has 1 node connected to one or more C_j's in which case the optimal would remove at most 1 node from C_i.
   3) C_i has 2 nodes connected to one or more C_j's in which case the optimal would remove at most 2 nodes from C_i

4) C_i has 3 nodes connected to 2 or more C_j's in which case the optimal would remove at most 3 nodes from C_i.

The optimal solution would find the node v_i with the largest number of connecting triangles (C_j's) and remove v_i. However, as a result of the above 4 cases, if Any of the connected C_j's are found by the algorithm then all 3 nodes in C_j will be removed (including node v_i) so we will end up with a result that removes EXACTLY removes 3 times the number of nodes than the optimal solution.

$\Rightarrow$ The algorithm is a strict 3-factor approximation algorithm.

Algorithm 2:



Optimal removes the center node.

$\Rightarrow opt = 1$

Alg: - Picking C_1, remove n_2
- Picking C_1, remove n_3
- Picking C_2, remove n_4
etc...

$Alg = 10$

$\Rightarrow$ Algorithm 2 is <u>not</u> a 3-factor approximation.

Algorithm 3:

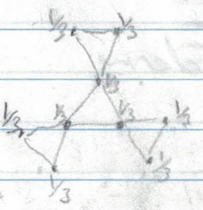
Finding the node with the most connected triangles removes the largest possible number of triangles in the first removal, and then the largest possible number of triangles in the second removal, etc. By the 4 cases listed above, this is itself an optimal solution to the problem. Therefore this problem is not a 3-factor approximation algorithm but instead

$\Rightarrow$ Algorithm 3 is a (1-factor) optimal algorithm.

Algorithm 4:

Given X^* = solution to L.P.

Worst case scenario would assign $(\frac{1}{3}, \frac{1}{3}, \frac{1}{3})$ to every single triangle in graph G. This is because if any node is not $\frac{1}{3}$ then the algorithm will return 1 or 2 of the nodes of any given C_i and any connected C_j's will consequently also not be $(\frac{1}{3}, \frac{1}{3}, \frac{1}{3})$ so that further reduces the set of returned nodes.



So the worst case scenario will return at most 3 nodes per triangle C_i (not exactly 3 because of nodes with multiple triangles attached). The optimal solution would return a list of the highest triangle degree nodes which reduces to at most 1 node per triangle.

⇒ Worst case for Algorithm 4 is 3 times as bad as worst case for optimal ⇒ Alg 4 ≤ 3× optimal

By the construction of the L.P. the higher the triangle degree of a node the higher the weight given to said node so the best case scenario will return the highest triangle degree nodes.

⇒ Best case for Algorithm 4 is 1 times optimal. ⇒ Alg 4 ≥ optimal

⇒ optimal ≤ Alg 4 ≤ 3× optimal.

---

3. Ordering $V_1, V_2, V_3, \ldots, V_{n-1}, V_n$

special case: V_1, V_2, V_3, V_4, V_5, and V_6 can each have ≤ n-neighbours.

Pattern 2)

$V_7 \leq n-1$	$V_{n-2} \leq n-(n-5)+2$	$V_i \leq n-(n-5)+n-i$
$V_8 \leq n-2$	$V_{n-1} \leq n-(n-5)+1$	$V_i \leq 5+n-i$
$V_9 \leq n-3$	$V_n \leq n-(n-5)$	

Showing 6-Col

At most 5 "backward" edges per node so we can assure that for every given node, every backward edge could be differently coloured using 6 colours. For node V_i we have ≤ n-i "forward" edges

however so long as we just use these edges for verification we won't need to impose any rules or restrictions other than just both neighbours need to be differently coloured.

Take colours 6, 5, 4, 3, 2, 1. Using the reverse of ordering $v_1, v_2, \ldots, v_n$,

→For $i = n, n-1, \ldots, 2, 1$
  For $j = 6, 5, \ldots, 2, 1, 0$
    if $j == 0$ then return False //Not 6-CoL
    else if v_i does not have neighbour with colour j
      then colour v_i as j
return (True, List of (v_i, j) pairs)

The algorithm has running time $O(|V| \times |E|)$ by looping over each node and for each node checking all neighbours.

The algorithm works because we've imposed both an ordering and restrictions on the number of "backwards" edges. If we start from the end, we're guaranteed that we can colour it and it has 5 "backwards" neighbours different colours. So if we colour each node by looping through the ordering backwards, then we're always guaranteed that what we previously coloured will not stop us from finding a colour for this new node.

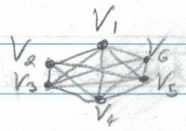
5-CoL Approximation

Attempt 1: Slightly modify the above algorithm as:

Take colours 5, 4, 3, 2, 1. Using the reverse of ordering $v_1, v_2, \ldots, v_n$

→For $i = n, n-1, \ldots, 2, 1$
  For $j = 5, 4, 3, 2, 1$
    if $j == 1$
      then colour v_i as j
    else if v_i does not have neighbour with colour j
      then colour v_i as j
return List of (v_i, j) pairs

The worst case situation that still satisfies the problem constraints is a complete graph with 6 vertices because it has the minimum number of vertices and edges such that it is not trivially 5-colourable.



$\Rightarrow \frac{6(6-1)}{2} = 15$ edges.

$\Rightarrow$ Algorithm: $[(V_6, 5), (V_5, 4), (V_4, 3), (V_3, 2), (V_2, 1), (V_1, 1)]$

All but the edge connecting V_1 with V_2 are properly coloured so in the worst case:

$$Alg \geq \frac{14}{15} \text{optimal}$$

The best cases are trivially any graph that is easily 5 or less colourable (for example the subset of our graphs with restrictions that the "backwards" edges are at most 4 per vertex instead of 5).

$$\Rightarrow \quad \text{optimal} \geq Alg \geq \frac{14}{15} \text{optimal}$$

$\Rightarrow$ Algorithm is a $\frac{14}{15}$ approximation algorithm

---

4. a) Given $D_{ab} = \{d_1, d_2, \ldots, d_i\}$ = Set of del calls from a->b
$I_{ab} = \{i_1, i_2, \ldots, i_k\}$ = Set of insert calls from a->b
$S_{ab} = \{s_1, s_2, \ldots, s_j\}$ = Set of set calls from a->b.

Assume without loss of generality that D_{ab} occurs first, I_{ab} occurs second and S_{ab} occurs last.

We know $d(a,b) = |D_{ab}| + |I_{ab}| + |S_{ab}|$. So if we can prove that $d(b,a) = |S_{ba}| + |I_{ba}| + |D_{ba}|$ where $|S_{ba}| = |S_{ab}|, |I_{ba}| = |D_{ab}|$, and $|D_{ba}| = |I_{ab}|$, AND the sets S_{ba}, I_{ba}, D_{ba} are exactly the inverse of S_{ab}, D_{ab}, I_{ab} respectively, then we have shown $d(a,b) = d(b,a)$.

In this problem inverse means for every set operation we perform the opposite set operations (set(x, i, 0)) => set(x, i, 1)),
For every delete operation we perform an insert operation in the reverse order, and
For every insert operation we perform a delete operation in the reverse order.

For example: $D_{ab} = \{del(x,1), del(x,3), del(x,2)\}$
$\Rightarrow I_{ba} = \{insert(x,2,b), insert(x,3,b), insert(x,1,b)\}$

And vice versa for $I_{ab} \Rightarrow D_{ba}$

Also the order of operation changes to $I_{ba} \rightarrow D_{ba} \rightarrow S_{ba}$.

So every delete operation has an inverse insert operation that reverses the original delete. Every insert has an inverse delete that reverses the original insert. And every set has an inverse set that reverses the original set.

So if $d(a, b)$ is finite, then there exists an inverse set of operations as described above taking us from b to a in the same number of steps.

$\Rightarrow \quad d(a, b) = d(b, a)$

---

b) d(a, b): // Let a[0...n], b[0...m]
matrix[n][m]
//Fill in matrix for if b is empty and we just need deletions
For i = 0, 1, ..., n
matrix[i][0] = i + 1
//Fill in matrix for if a is empty and we just need insertions
For i = 0, 1, ..., m
matrix[0][i] = i + 1

```
for i = 0, 1, ..., M-1
    for j = 0, 1, ..., N-1
        if a[i] == b[j]
            cost = 0
        else
            cost = 1
        matrix[i+1][j+1] = min( matrix[i][j+1] + 1,   // delete
                                matrix[i+1][j] + 1,   // insert
                                matrix[i][j] + cost   // substitution
return matrix[n][m]
```

The matrix always keeps track of the minimum between the costs of deletion, insertion, and substitution in that order respectively in the code. By iterating over both strings this will always compute the correct minimum for any given possible step but never re-computing the previously computed values like a recursive approach would do.

c) Intuition tells me that we're looking for string d as the largest common substring amongst the three strings a, b, and c. This would restrict the d(a, d), d(b, d), d(c, d) calls to only using delete and insert operations.

The optimal solution, I think, would find string e such that for each a, b, c the characters of e appear as a scattered subsequence of a, b, and c.

Largest disparity would be such that the optimal finds a solution of $|d| = \min(|a|, |b|, |c|)$ and the largest common substring is $\emptyset$. For this to occur, one of the two of a, b, c that are not minimal would need to have at least 2 times the length of the minimal in order to ensure that no substrings occur.

Assume $|a|$ minimal, $|b| \geq 2|a|$, $|c| \geq |a|$

In this situation the algorithm would output $d(a,d) = |a|$, $d(b,d) = |b|$, $d(c,d) = |c|$.

$\Rightarrow$ output $= |a| + |b| + |c|$
$\geq 4|a|$

The optimal would output $d(a,d) = 0$, $d(b,d) = |a|$, $d(c,d) = |c| - |a|$

$\Rightarrow$ optimal $= 0 + |a| + |c| - |a|$
$\geq |a|$

4-factor $\Rightarrow$ So clearly I have backed up the wrong tree.

I'm still confident of the optimal solution, just need to find a better approx. algorithm.