Tutorial 4

SCS 1112

-W.D.Vihanga-

Question 1

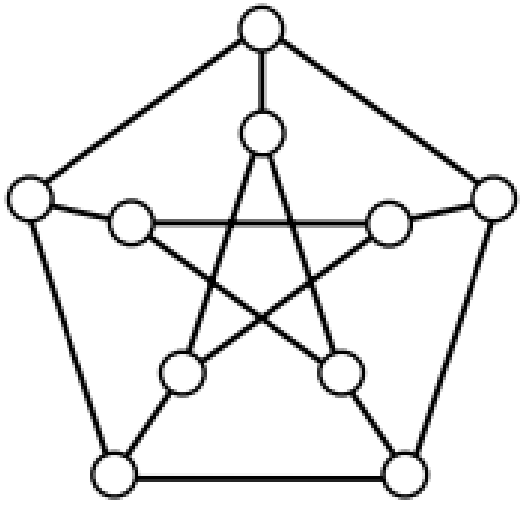
Discuss the 3-colourability problem of a graph G. Show that 3-colourabilty is NP complete

3-Colourability of Graph

Try Out:

Problem

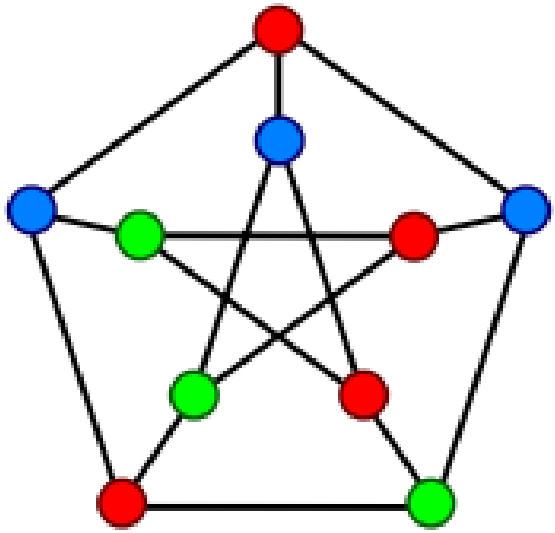
Given a graph, determine whether each node can be coloured red, blue, or green, so that the endpoints of each edge have different colours.



3-Colourability of Graph

Problem

Given a graph, determine whether each node can be coloured red, blue, or green, so that the endpoints of each edge have different colours.



What is NP complete

- A problem is said to be NP-Complete if:
  i. It is NP
  ii. The problem can be reduced to a known NP Complete problem

http://www.cs.toronto.edu/~bor/373s13/L15.pdf

3-Colorability Time Complexity

- Prove 3- Colorability is NP Complete

I. If a problem is in NP class it must be polynomially verifiable

   The 3-colorability problem is NP because we can guess a coloring of vertices with 3 colors and check in quadratic time that the coloring is correct [For each of the |V| vertices check the color of upto |V|-1 of it's neighbors]

3-Colorability Time Complexity

II. The problem can be reduced to a known NP Complete problem

    3-colorability problem can be reduced to 3-satisfiability problem(3-SAT). 3-SAT is an NP complete problem

- Considering I & II, one can claim that 3-Colorability of Graph is NP-Complete

Boolean Satisfiability

- **Boolean Satisfiability Problem** (**SAT**) is the problem of determining if there exists an interpretation that satisfies a given Boolean formula. In other words, it asks whether the variables of a given Boolean formula can be consistently replaced by the values TRUE or FALSE in such a way that the formula evaluates to TRUE. If this is the case, the formula is called *satisfiable*.
- On the other hand, if no such assignment exists, the function expressed by the formula is FALSE for all possible variable assignments and the formula is *unsatisfiable*.
- For example, the formula "*a* AND NOT *b*" is satisfiable because one can find the values *a* = TRUE and *b* = FALSE, which make (*a* AND NOT *b*) = TRUE. In contrast, "*a* AND NOT *a*" is unsatisfiable.

- https://en.wikipedia.org/wiki/Boolean_satisfiability_problem

3 - SAT

- A boolean formula is in conjunctive normal form (CNF) if it is a conjunction (and) of several clauses, each of which is the disjunction (or) of several literals, each of which is either a variable or its negation.
- For example:

$$\overbrace{(a \vee b \vee c \vee d)}^{\text{clause}} \wedge (b \vee \bar{c} \vee \bar{d}) \wedge (\bar{a} \vee c \vee d) \wedge (a \vee \bar{b})$$

- http://www.dcs.fmph.uniba.sk/~fduris/VKTI/3color_HamCycle.pdf

3 - SAT

- A 3CNF formula is a CNF formula with exactly three literals per clause;
- Eg: (a v b v c) ^ (b v ~c v d) ^ (~a v ~b v c) ^ (e v f v g)

- 3SAT is just SAT restricted to 3CNF formulas — given a 3CNF formula, is there an assignment to the variables that makes the formula evaluate to true?

Question 2

Consider the job shop scheduling problem again. Map this to the knapsack problem. Use (earliest) starting time and (largest) job duration as parameters, and assume that jobs can be selected non-FCFS

- Consider maximizing machine utilization by assigning largest jobs first
- Can we consider Start Time/ Job Duration of each job and order from Min to Max?
- Can we find out the job with earliest starting time and largest duration
- Greedily add jobs considering the above fraction and start and finishing times of 2 jobs
- Is this a 1-1 mapping to Knapsack problem?

Question 3

Given a set of numbers, how can a non empty subset that sums to K can be selected? What is the brute force complexity? Is there a P time algorithm?

Subset Sum Problem

- Brute Force Solution:
  - Explore all subsets of the main set
  - 2^N subsets in a set of N elements
  - Hence exponential time complexity

Brute force Recursive Solution

- Let isSubSetSum(int set[], int n, int sum) be the function to find whether there is a subset of set[] with sum equal to *sum*. n is the number of elements in set[].

- The isSubsetSum problem can be divided into two subproblems
  …a) Include the last element, recur for n = n-1, sum = sum – set[n-1]
  …b) Exclude the last element, recur for n = n-1.
  If any of the above the above subproblems return true, then return true.

Brute force Recursive Solution

- Following is the recursive formula for isSubsetSum() problem.

isSubsetSum(set, n, sum) = isSubsetSum(set, n-1, sum) || isSubsetSum(set, n-1, sum-set[n-1])

Base Cases: isSubsetSum(set, n, sum) = false, if sum > 0 and n == 0 isSubsetSum(set, n, sum) = true, if sum == 0

Try all the subsets in the worstcase. Hence Exponential

http://www.geeksforgeeks.org/dynamic-programming-subset-sum-problem/

Pseudo-polynomial time Solution

- Uses Dynamic Programming

- Take the input in *nums[1]*, *nums[2]*, … , *nums[n]*
  Use a 2-D array named *ispossible[ ][ ]* of size(*N*+1) x (*M*+1).
- *ispossible[i][j]* can be either 0 or 1. Its value will be 1 if it is possible to select some numbers from the set {*nums[1]*,*nums[2]*,*nums[3]*,…,*nums[i]*} so that their sum equals to *j*; otherwise it will be 0.

- Our final goal is to find the value of ***ispossible[N][M]***, which will be 1 if it is possible to obtain a subset of {*nums[1]*,*nums[2]*,*nums[3]*,…,*nums[N]*} with sum equal to *M*.
  To find the value of *ispossible[N][M]* we need to find the value of each element in the two dimensional array *ispossible[ ][ ]*.

Pseudo-polynomial time Solution

- In general, *ispossible[i][j]* = 1 iff one of the following two conditions is true:

- 1. ***ispossible[i-1][j]*** **is 1**. If *ispossible[i-1][j]* is 1 it means that it is possible to obtain a sum of j by selecting some numbers from {*nums[1]*,*nums[2]*,*nums[3]*,…,*nums[i-1]*}, so obviously the same is also possible with the set {*nums[1]*,*nums[2]*,*nums[3]*,….,*nums[i-1]*,*nums[i]*}.

- 2. ***ispossible[i-1][j-nums[i]]*** **is 1**. If *ispossible[i-1][j-nums[i]]* is 1 it means that it is possible to obtain a sum of (*j–nums[i]*) by selecting numbers from {*nums[1]*, *nums[2]*,…,*nums[i-1]*}. Now if we select *nums[i]* too, then we can obtain a sum of (*j–nums[i]*)+*nums[i]* = *j*. Therefore *ispossible[i][j]* is 1.

- Note that first it is important to set *ispossible[i][0]* = 1 (for 0<=*i*<=*N*). Because obviously its possible to obtain a sum = 0 from any value of *i*.

Pseudo-polynomial time Solution

- The previous two points serve as the **DP equation** for calculating all values in *ispossible[ ][ ]*.
- This algorithm has time complexity **O(N*M)** and space complexity **O(N*M)**.

- https://codeaccepted.wordpress.com/2014/02/27/dynamic-programming-subset-sum-problem/

Example

- Number Set : {2,3,7,8,10} , Sum : 11

	0	1	2	3	4	5	6	7	8	9	10	11
2	T	F	T	F	F	F	F	F	F	F	F	F
3	T	F	T	T	F	T	F	F	F	F	F	F
7	T	F	T	T	F	T	F	T	F	T	T	F
8	T	F	T	T	F	T	F	T	T	T	T	T
10	T	F	T	T	F	T	F	T	T	T	T	T

- Answer : True {3,8}
- https://www.youtube.com/watch?v=s6FhG--P7z0

Question 4

Has the following problem got a solution(answer)? 'If the only barber in the town shaves only those who do not shave themselves, who shaves the barber?'