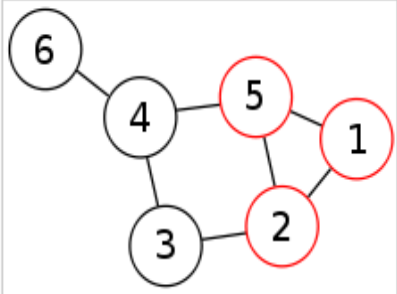
Clique

An undirected graph is formed by a finite set of vertices and a set of unordered pairs of vertices, which are called edges. By convention, in algorithm analysis, the number of vertices in the graph is denoted by n and the number of edges is denoted by m. A clique in a graph G is a complete subgraph of G; that is, it is a subset S of the vertices such that every two vertices in S are connected by an edge in G. A maximal clique is a clique to which no more vertices can be added; a maximum clique is a clique that includes the largest possible number of vertices, and the clique number $\omega(G)$ is the number of vertices in a maximum clique of G.[1]



The graph shown has one maximum clique, the triangle {1,2,5}, and four more maximal cliques, the pairs {2,3}, {3,4}, {4,5}, and {4,6}.

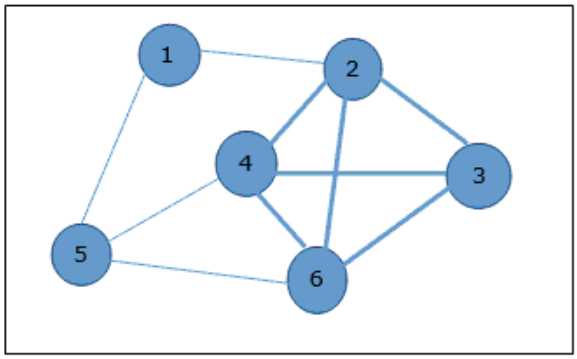
Several closely related clique-finding problems have been studied.

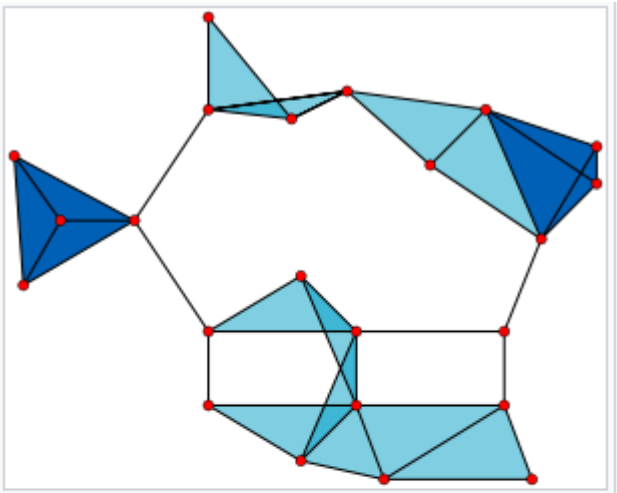
- In the maximum clique problem, the input is an undirected graph, and the output is a maximum clique in the graph. If there are multiple maximum cliques, only one need be output.
- In the weighted maximum clique problem, the input is an undirected graph with weights on its vertices (or, less frequently, edges) and the output is a clique with maximum total weight. The maximum clique problem is the special case in which all weights are equal.
- In the maximal clique listing problem, the input is an undirected graph, and the output is a list of all its maximal cliques. The maximum clique problem may be solved using as a subroutine an algorithm for the maximal clique listing problem, because the maximum clique must be included among all the maximal cliques.
- In the k-clique problem, the input is an undirected graph and a number k, and the output is a clique of size k if one exists (or, sometimes, all cliques of size k).
- In the clique decision problem, the input is an undirected graph and a number k, and the output is a Boolean value: true if the graph contains a k-clique, and false otherwise.

The first four of these problems are all important in practical applications; the clique decision problem is not, but is necessary in order to apply the theory of NP-completeness to clique-finding problems.

Example

Take a look at the following graph. Here, the sub-graph containing vertices 2, 3, 4 and 6 forms a complete graph. Hence, this sub-graph is a **clique**. As this is the maximum complete sub-graph of the provided graph, it's a **4-Clique**.





A graph with

- 23 × 1-vertex cliques (the vertices),
- 42 × 2-vertex cliques (the edges),
- 19 × 3-vertex cliques (light and dark blue triangles), and
- 2 × 4-vertex cliques (dark blue areas).

The 11 light blue triangles form maximal cliques. The two dark blue 4-cliques are both maximum and maximal, and the clique number of the graph is 4.

Exact Algorithms for MCP (maximum clique problem)

We can address the decision and optimization problems with an exact algorithm, such as a backtracking search .

Backtracking search incrementally constructs the set C (initially empty) by choosing a candidate vertex from the candidate set P (initially all of the vertices in V) and then adding it to C. Having chosen a vertex the candidate set is then updated, removing vertices that cannot participate in the evolving clique. If the candidate set is empty then C is maximal (if it is a maximum we save it) and we then backtrack. Otherwise P is not empty and we continue our search, selecting from P and adding to C.

We start with a simple algorithm, similar to Algorithm 1 in [8]. Fahle's Algorithm 1 uses two sets: C the growing clique (initially empty) and P the candidate set (initially all vertices in the graph). C is maximal when P is empty and if |C| is a maxima it is saved, i.e. C becomes the champion. If |C| + |P| is too small to unseat the champion search can be terminated. Otherwise search iterates over the vertices in P in turn selecting a vertex v, creating a new growing clique C0 where C0 = C [{v} and a new candidate set P0 as the set of vertices in P that are adjacent to v (i.e. P0 = P \ neighbours(v)), and recursing. We will call this MC.

In an undirected graph, a **clique** is a complete sub-graph of the given graph. Complete sub-graph means, all the vertices of this sub-graph is connected to all other vertices of this sub-graph.

The Max-Clique problem is the computational problem of finding maximum clique of the graph. Max clique is used in many real-world problems.

Let us consider a social networking application, where vertices represent people's profile and the edges represent mutual acquaintance in a graph. In this graph, a clique represents a subset of people who all know each other.

To find a maximum clique, one can systematically inspect all subsets, but this sort of brute-force search is too time-consuming for networks comprising more than a few dozen vertices.

```
Algorithm: Max-Clique (G, n, k)
S := ϕ
for i = 1 to k do
   t := choice (1…n)
   if t ∈ S then
      return failure
   S := S ∪ t
for all pairs (i, j) such that i ∈ S and j ∈ S and i ≠ j do
   if (i, j) is not a edge of the graph then
      return failure
return success
```

Analysis

Max-Clique problem is a non-deterministic algorithm. In this algorithm, first we try to determine a set of **k** distinct vertices and then we try to test whether these vertices form a complete graph.

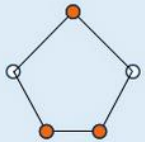
There is no polynomial time deterministic algorithm to solve this problem. This problem is NP-Complete.

Vertex Cover Problem

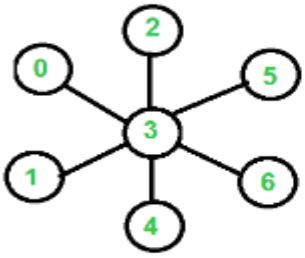
- In the mathematical discipline of graph theory, "A *vertex cover* (sometimes node cover) of a graph is a subset of vertices which "*covers*" every edge.
- An edge is *covered* if one of its endpoint is chosen.
- In other words "A *vertex cover* for a graph G is a set of vertices incident to every edge in G."
- The *vertex cover problem*: What is the minimum size vertex cover in G?

Vertex Cover Problem

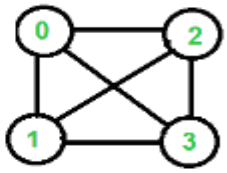
Problem: Given graph G = (V, E), find *smallest* $V' \subseteq V$ *s. t.* if (u, v) $\in$ E, then u $\in$ V′ or v $\in$ V′ or both.



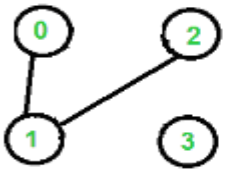
Following are some examples.



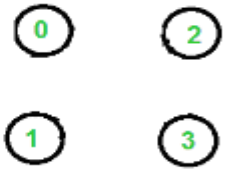
Minimum Vertex Cover is {3}



Minimum Vertex Cover is {0, 1, 2} or {0, 1, 3} or {1, 2, 3}



Minimum Vertex Cover is {1}



Minimum Vertex Cover is empty {}

Vertex Cover Problem is a known NP Complete problem

Vertex Cover : Greedy Algorithm(1)

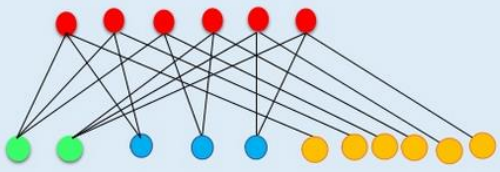
➢ **Idea:** Keep finding a vertex which covers the maximum number of edges.

Step 1: Find a vertex *v* with maximum degree.

Step 2: Add *v* to the solution and remove *v* and all its incident edges from the graph.

Step 3: Repeat until all the edges are covered.

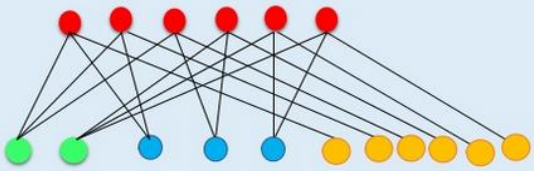
Greedy Algorithm(1): Analysis



Optimal Solution = 6, select all red vertices.

➢ Greedy approach does not always lead to the best approximation algorithm.

Greedy Algorithm(1): Analysis



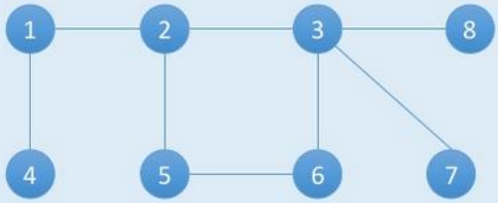
Unfortunately if we select the vertices in following order, then we will get worst solution for this vertex cover problem-

- First we might choose all the green vertices.
- Then we might choose all the blue vertices.
- And then we might choose all the orange vertices. Solution=11;

Vertex Cover : Algorithm(2)

APPROX-VERTEX-COVER
1: C ← Ø ;
2: E′ ← E
3: **while** E′ ≠ Ø; **do**
4: let (u, v) be an arbitrary edge of E′
5: C ← C ∪ {(u, v)}
6: remove from E′ all edges incident on either u or v
7: **end while**

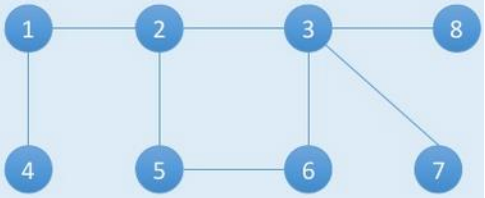
Algorithm(2): Example



Initially C = Ø

E′ = {(1,2) (2,3) (1,4) (2,5) (3,6) (5,6) (3,7) (3,8)}

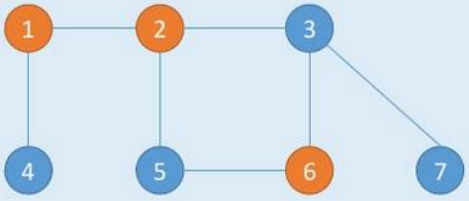
Algorithm(2): Example



C = {1, 2, 3, 6}

E′ = {~~(3,6) (5,6) (3,7) (3,8)~~}

Algorithm(2): Example (Cont...)

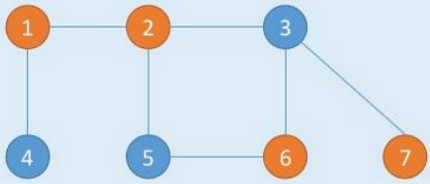


Are the red vertices a vertex-cover?

No……. why?

Edge (3, 7) is not covered by it.

Algorithm(2): Example (Cont...)



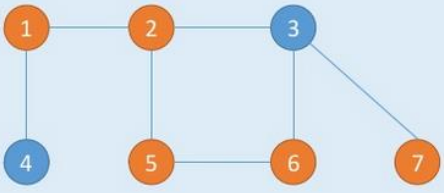
Are the red vertices a vertex-cover?

Yes

What is the size?

Size = 4

Algorithm(2): Example (Cont...)



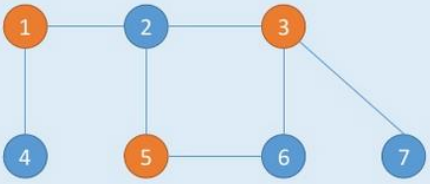
Are the red vertices a vertex-cover?

Yes

What is the size?

Size = 5

Algorithm(2): Example (Cont...)



Are the red vertices a vertex-cover?

Yes

What is the size?

Size = 3

Graph Coloring | Set 2 (Greedy Algorithm)

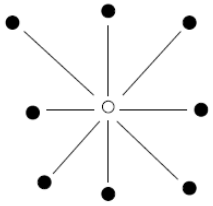
We introduced graph coloring and applications in previous post. As discussed in the previous post, graph coloring is widely used. Unfortunately, there is no efficient algorithm available for coloring a graph with minimum number of colors as the problem is a known NP Complete problem. There are approximate algorithms to solve the problem though. Following is the basic Greedy Algorithm to assign colors. It doesn't guarantee to use minimum colors, but it guarantees an upper bound on the number of colors. The basic algorithm never uses more than d+1 colors where d is the maximum degree of a vertex in the given graph.

Basic Greedy Coloring Algorithm:

```
1. Color first vertex with first color.
2. Do following for remaining V-1 vertices.
          a) Consider the currently picked vertex and color it with the
             lowest numbered color that has not been used on any previousl
             colored vertices adjacent to it. If all previously used colo
             appear on vertices adjacent to v, assign a new color to it.
```

Vertex Coloring

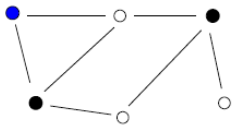
Consider a graph $G = (V, E)$



Edge coloring: no two edges that share an endpoint get the same color

Vertex coloring: no two vertices that are adjacent get the same color

Use the minimum amount of colors
This is the chromatic number

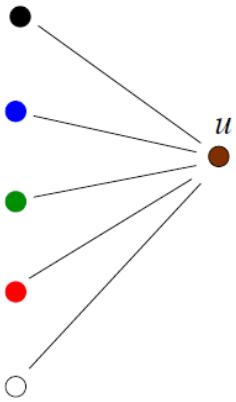


Number between 1 and $|V|$ (why?)

Greedy Algorithm 1

Colors are indicated by numbers $1, 2, \ldots$

- ☐ Consider the nodes in some order
- ☐ At the start, each node is uncolored (has color 0)
- ☐ Give each node the smallest color that is not used to color any neighbor

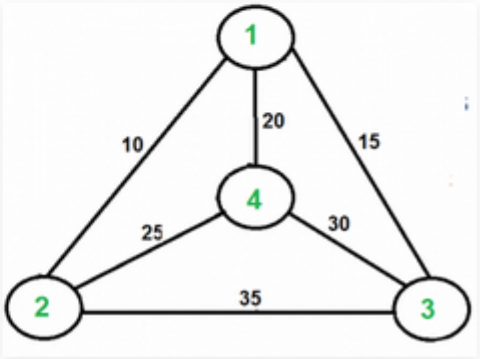


Travelling Salesman Problem | Set 1 (Naive and Dynamic Programming)

Travelling Salesman Problem (TSP): Given a set of cities and distance between every pair of cities, the problem is to find the shortest possible route that visits every city exactly once and returns to the starting point.

Note the difference between Hamiltonian Cycle and TSP. The Hamiltoninan cycle problem is to find if there exist a tour that visits every city exactly once. Here we know that Hamiltonian Tour exists (because the graph is complete) and in fact many such tours exist, the problem is to find a minimum weight Hamiltonian Cycle.

For example, consider the graph shown in figure on right side. A TSP tour in the graph is 1-2-4-3-1. The cost of the tour is 10+25+30+15 which is 80.



The problem is a famous NP hard problem. There is no polynomial time know solution for this problem.

Naive Solution:

1) Consider city 1 as the starting and ending point.

2) Generate all (n-1)! Permutations of cities.

3) Calculate cost of every permutation and keep track of minimum cost permutation.

4) Return the permutation with minimum cost.

Time Complexity: ?(n!)

Travelling Salesman Problem | Set 2 (Approximate using MST)

We introduced Travelling Salesman Problem and discussed Naive and Dynamic Programming Solutions for the problem in the previous post,. Both of the solutions are infeasible. In fact, there is no polynomial time solution available for this problem as the problem is a known NP-Hard problem. There are approximate algorithms to solve the problem though. The approximate algorithms work only if the problem instance satisfies Triangle-Inequality.

Triangle-Inequality: The least distant path to reach a vertex j from i is always to reach j directly from i, rather than through some other vertex k (or vertices), i.e., dis(i, j) is always less than or equal to dis(i, k) + dist(k, j). The Triangle-Inequality holds in many practical situations.
When the cost function satisfies the triangle inequality, we can design an approximate algorithm for TSP that returns a tour whose cost is never more than twice the cost of an optimal tour. The idea is to use **M**inimum **S**panning **T**ree (MST). Following is the MST based algorithm.

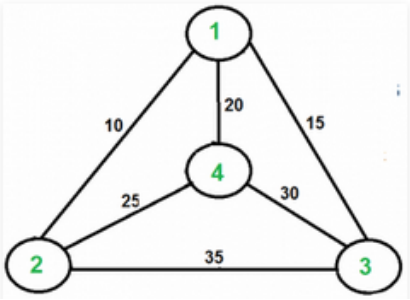
Algorithm:

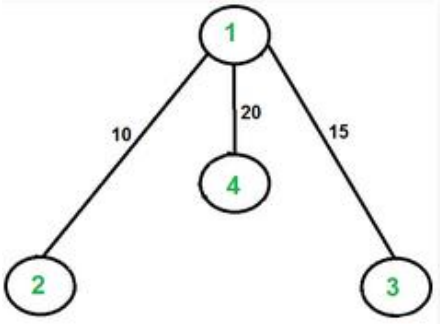
1) Let 1 be the starting and ending point for salesman.

2) Construct MST from with 1 as root using Prim's Algorithm.

3) List vertices visited in preorder walk of the constructed MST and add 1 at the end.

Let us consider the following example. The first diagram is the given graph. The second diagram shows MST constructed with 1 as root. The preorder traversal of MST is 1-2-4-3. Adding 1 at the end gives 1-2-4-3-1 which is the output of this algorithm.





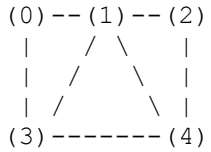
Minimum Spanning Tree of the graph on left side with 1 as root. The Preorder Traversal of MST is 1-2-4-3. So the output is 1-2-4-3-1

In this case, the approximate algorithm produces the optimal tour, but it may not produce optimal tour in all cases.

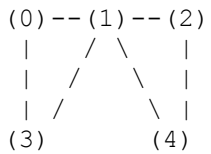
Backtracking (Hamiltonian Cycle)

Hamiltonian Path in an undirected graph is a path that visits each vertex exactly once. A Hamiltonian cycle (or Hamiltonian circuit) is a Hamiltonian Path such that there is an edge (in graph) from the last vertex to the first vertex of the Hamiltonian Path. Determine whether a given graph contains Hamiltonian Cycle or not. If it contains, then print the path. Following are the input and output of the required function.

For example, a Hamiltonian Cycle in the following graph is {0, 1, 2, 4, 3, 0}. There are more Hamiltonian Cycles in the graph like {0, 3, 4, 2, 1, 0}



And the following graph doesn't contain any Hamiltonian Cycle.



Naive Algorithm
Generate all possible configurations of vertices and print a configuration that satisfies the given constraints.

There will be n! (n factorial) configurations.

```
while there are untried configurations
{
   generate the next configuration
   if ( there are edges between two consecutive vertices of this
      configuration and there is an edge from the last vertex to
      the first ).
   {
      print this configuration;
      break;
   }
}
```

Backtracking Algorithm
Create an empty path array and add vertex 0 to it. Add other vertices, starting from the vertex 1. Before adding a vertex, check for whether it is adjacent to the previously added vertex and not already added. If we find such a vertex, we add the vertex as part of the solution. If we do not find a vertex then we return false.