Continuation of NP-Complete/NP-Hard problems in graph theory

1. Independent Set

Reminder: An **independent set** is a set of vertices in a graph, where no 2 vertices are adjacent.

As we have already proven, finding an independent set of size k is a NP-Complete problem.

How to solve it?

Brute force: Checking all possible subsets of size k to find if one is an independent set. Complexity? Number of subsets of size k – Combinations of k taken by n: C_n^k => O($\frac{n!}{k!(n-k)!}$).

For the optimization problem – find the *maximum* independent set – the brute force approach is to check all possible subsets, which is 2^n

The best algorithms use backtracking, with different rules. (One obvious rule is: once you find that a pair of vertices are neighbors, remove all other subsets that contain that pair from the search space – i.e backtrack).

2. Clique

Reminder:

- *G'* is called an **induced** subgraph if ∀ v_1, v_2 ∈ *V'*, if (v_1, v_2) ∈ *E* then (v_1, v_2) ∈ *E'*.
  In other words, an induced subgraph contains a subset of vertices and all edges that exist between those vertices in the original graph.
- A simple graph that has edges between every vertex pair is called **complete**.
- **Independence number** is the independent set with greatest size.

Def 1: A **clique** in an undirected graph, is a subset of vertices of a graph, such that the induced subgraph is complete.

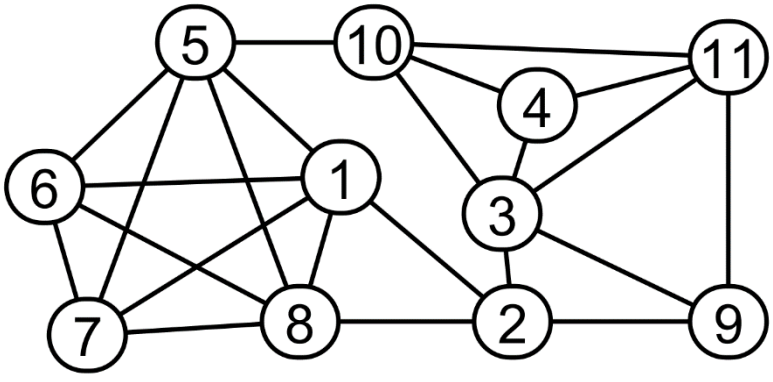


Figure 1

Take for example the graph in Figure 1. A clique in this graph is {1,8,7,6,5}, as the induced subgraph is complete. Another clique is {11, 10, 4, 3}, while other examples might be {2,3,9} or {1,8,2}. Note that any subset of a clique is also a clique (e.g. {5,1,7,6}, {8,1,6}, etc.).

Theorem: A set of vertices is a clique of a simple graph G if and only if it is an independent set of the *complement* of G (the complement has the same vertices but has edges where the original doesn't and vice versa).

It is trivial to prove: since in a clique all vertices are adjacent, in the complement, where all existing edges are removed, none of the vertices are adjacent anymore, therefore they form an independent set.

⇨ Independence number = **clique number** (max size of clique).

That means that cliques and independent sets are equivalent, any independent set problem can be stated/solved as a clique problem. Then all clique problems are in the same complexity class as the independent set problems (i.e. the decision problem is NP-Complete and the optimisation problem is NP-Hard).

The k-clique problem: given a graph and an integer k, is there a clique of size k.

The optimization problem: find the maximum clique (the clique of a graph that has the maximum possible size of all the cliques).

How to solve it?

Brute force: Check all subsets of V => $O(2^n)$

Better algorithms include backtracking, same as for independent sets (when you find 2 vertices that are not neighbors, remove all subsets with those 2 vertices from the search space).

3. Dominating Sets

Reminder:

- A **dominating set** is a set of vertices in a graph, such that every vertex is either in this set or adjacent to a vertex in it.
- The **dominance number** is the size of the smallest possible dominating set of a graph.

Finding the minimum dominating set is NP-Hard.

As a decision problem, finding if a graph has a dominating set of size k is an NP-Complete problem.

How to solve it?

Brute force algorithm: same as independent sets: check all possible subsets.

Backtracking is **not** as useful for this problem, as there is no clear rule when 2 vertices should be part of the set or not.

Faster algorithms are based on *approximations*: finding a dominating set that has size close or equal to the dominance number. Most approximation algorithms use a *greedy* technique: we construct our solution by always adding elements but never removing, based on some rule (heuristic) that should get us close to the optimal solution. Greedy algorithms are designed to

much more efficient compared to exact algorithms – if you are getting just an approximation at least you could get it fast.

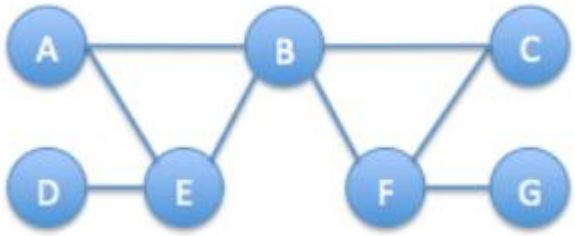
For dominating sets a reasonably good approximation is always adding the vertex with highest degree, then remove all of its neighbors from the graph – continue until no vertices are left.

This will always give a dominating set, and the approximation of the minimum is fine, especially for smaller graphs. If *dn* is the dominance number and *size* is the size of the found dominating set, then:

$$dn \leq size < dn * (\ln(n) + 1)$$

Which is fine, but as n grows ln(n) can easily reach 2-digit values, meaning that *size* can be 10 or even 20 times higher than the actual dominance number.

Counterexample for when greedy does not find the minimum dominating set:



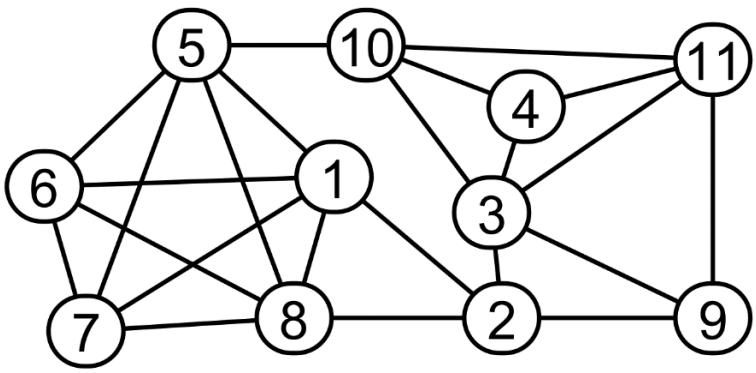
Greedy chooses B then D then G, while the minimum dominating set is E and F.

4. Vertex Cover

Def 2: A **vertex cover** of a graph is a set of its vertices such that any edge is incident to a vertex in this set.

Formally: let $G = (V, E)$ be an undirected graph. $M \subseteq V$ is a vertex cover of G if $\forall (v_1, v_2) \in E$ either $v_1 \in M$ or $v_2 \in M$ or both. A vertex cover is called minimum if it has the minimum size of all the possible vertex covers.

Let's consider again the graph from figure 1 (copied below). A vertex cover for this graph might be: {5,1,8,7,3,4,11,9}. This vertex cover has size 8 which is the smallest possible size for a vertex cover for this graph.

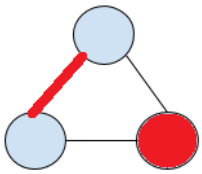


The vertex cover optimization problem is to find the minimum vertex cover – the cover with the smallest possible size for a given graph. The associated decision problem is "is there a vertex cover of size k?".

Finding a minimum vertex cover of a graph has various implementations such as finding the minimum number of stores that need to be built in an area such that any road reaches at least one store.

The vertex cover problem might look similar to the dominating set problem, but they are not quite the same. Still: for connected graphs, every vertex cover *is* a dominating set, but not every dominating set is a vertex cover (a dominating set can have fewer vertices in it).

Example: the set of red vertices is a dominating set but not a vertex cover.



NOT a **Vertex Cover.**

Because the red line does not have: "at least 1 node connected in the (red) set of nodes."

However, not all vertex covers are dominating sets for *unconnected* graphs. Why? Because isolated vertices *must* be in a dominating set, while they don't have to be in the vertex cover (since no edge is incident to them, they add nothing to the vertex cover).

There is an interesting link between vertex covers and independent sets: all edges from the graph must have at least one endpoint in the vertex cover, meaning that the vertices that are left cannot have any connections between them (if they had any, that edge would not be covered by the vertex cover). Therefore, the vertices that are left form an independent set.

In other words, for a graph $G=(V, E)$, if M is a vertex cover for G then $V \setminus M$ is an independent set.

Indeed, if we take the previous example we see that it is true. For the graph in figure 1 we found the vertex cover {5,1,8,7,3,4,11,9}. The rest of the vertices form the set {2, 6, 10} which is an independent set.

That means that the problem of finding a vertex cover of size k is the same as the one of finding an independent set of size n-k. And finding a minimum vertex cover is the same as finding a maximum independent set.

So, the decision version "is there a vertex cover of size k" is an NP-Complete problem. And the optimization version "find the minimum vertex cover" is an NP-Hard problem.

How to solve?

We still want to find a subset of vertices, so the complexity of brute force search is the same:

- Decision version: $O(C_n^k) \Rightarrow O(\frac{n!}{k!(n-k)!})$
- Optimization version: $O(2^n)$

O the other hand backtracking is not as useful for this problem, since, similarly to dominating sets, there is no clear or as useful rule when 2 or more vertices should be added to the cover or not (it is usable if we just try to find an independent set of size n-k).

There are greedy algorithms, however.

One algorithm works by always choosing the highest degree vertex and removing its *incident edges*, and so on until there are no more edges. This is similar to the one for dominating sets but we remove the incident edges not the neighbors.

As for dominating sets, this will give an acceptable approximation. If *vn* is the size of the smallest vertex cover and *size* is the size of the found vertex cover, then:

$$vn \leq size < vn * \log(n)$$

As before, this is good especially for smaller graphs, but as the size increases, it can result in a cover that is 10 or 20 times larger or more than the actual minimum vertex cover.

Fortunately, for vertex covers there is a much better greedy algorithm: take a random edge that is not covered and add both endpoints to the cover. Continue until all edges are covered.

This simple heuristic will give us a cover at most twice the size of the minimum cover. That is:

$$vn \leq size < vn * 2$$

5. Graph coloring

For this course we will be considering only vertex coloring. Edge coloring also exists, but it is a different problem.

Def 3: a **vertex coloring** of a graph is the assignment of colors to its vertices such that no two adjacent vertices have the same color.

Formally:

Given an undirected graph $G = (V, E)$, a **vertex coloring** of G is a function c: $V \rightarrow C$, which assigns to vertices in V a color from some set of colors C, such that $\forall (v_1, v_2) \in E$, $c(v_1) \neq c(v_2)$.

The set of colors are conventionally positive integers, i.e. C = {1, 2, 3, ... *k*}. Actual colors are sometimes used if the number of colors (k) is small (e.g. C = {red, green, blue, white}).

A coloring example is given in Figure 2. If the colors are {red, blue, green} then the graph can be colored with these colors in the following way: red:{10, 1, 8, 2}, blue:{4, 5, 3}, green:{6, 7, 9}. As a side note, this type of graph is called a Peterson graph.

A coloring of a graph using *k* colors is called a *k*-coloring. A graph that has a *k*-coloring is called *k*-colorable.

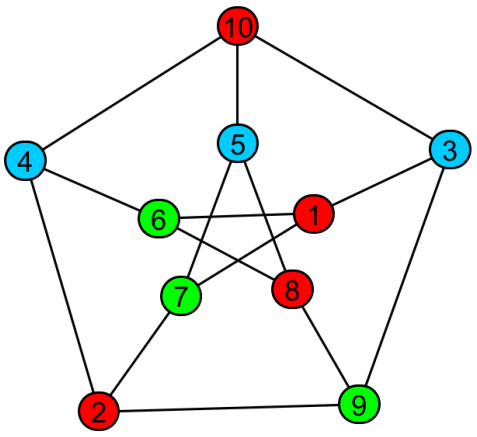


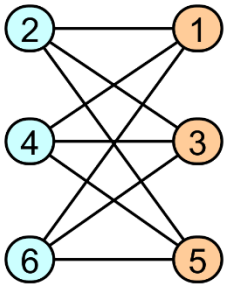
Figure 2

The graph in Figure 2 is 3-colorable, with a 3-coloring highlighted in the figure.

Other than map-making graph coloring also have other uses, for example register allocation in compilers[1].

Q: Any link between coloring and bipartite graphs?

A: A graph is bipartite if and only if it is 2-colorable. In Figure 3 you can see an example 2-colored bipartite graph. It is clear that any bipartite graph admits a 2-coloring: since no vertices from one partition are adjacent then they can all have the same color.



In general, a k-coloring is actually a partition of the vertices in k parts such that in any partition no vertices are adjacent. So a partition is a subset of vertices where no 2 vertices are adjacent. So, what is a partition? It is an *independent set*.

A k-coloring of a graph is a partitioning of the vertices of the graph into k independent sets. Then, finding a k-coloring is the same as finding k independent sets of size n/k. Meaning that the independent set problem can be reduced to finding a k-coloring => Finding a k-coloring of a graph is NP-Complete.

Def 4: the **chromatic number** of a graph is the smallest number of colors needed to color the graph (or the smallest k for which the graph admits a k-coloring).

Q: What is the chromatic number of a complete graph?

A: n, because in a complete graph each vertex is adjacent to all others, so each vertex needs a different color.

Q: Any link between chromatic number and cliques?

The chromatic number is at least as small as the greatest clique in the graph. Since in a complete graph we need n colors, and a clique represents a complete subgraph, that means that for a clique of size *s* we need *s* colors to color it.

=> As for finding the clique of maximum size, finding the chromatic number is also NP-Hard (though there are some efficient algorithms for testing some constants like 2 or 3).

So, given a k and a graph, how to find a k-coloring?

Brute force: checking all possible partitions in k parts of the vertex set. The number of partitions of n elements in k parts is given by the Stirling numbers of the second kind, S(n, k), where S(n, k) = k*S(n-1, k) + S(n-1, k-1) (no direct formula). A Stirling number of the second kind is $O(n^k)$. So brute force is verry inefficient – $O(n^k)$.

Backtracking is a good technique to solve this problem more efficiently with an obvious first rule – always check if no adjacent vertices have the same color.

Greedy algorithm to estimate the chromatic number and give a coloring with the number of colors of the found estimation also exist. For example: always select the highest degree uncolored vertex and color it with the minimum color that does not conflict with its neighbors.

1. https://www.hcltech.com/sites/default/files/documents/resources/whitepaper/files/register_allocation_via_graph_coloring_meena_jain_-_v2.0.pdf