Graph Algorithms : Home Assignment

Valeran MAYTIE

Exact exponential algorithms for Graph Coloring Problem

1. The first non-trivial algorithm for 3-Colour.

   (a) Intuitively the root has 3 options then the children are constrained by their parents, so they only have 2 as we have a root and n-1 children then, there are $3 \times 2^{n-1}$ coloring options.

   We can proof this property by induction on n.

   - $n = 0$, the case where the tree contains no vertex is not interesting
   - $n = 1$, if we have one vertex, so we have $3 = 3 \times 2^{1-1}$ 3-coloring options
   - $n + 1$, by hypothesis induction, we know that a tree with n-vertices has $3 \times 2^{n-1}$ 3-coloring options.
     If we had a vertex on a tree, then it has two possible color options, so there is $3 \times 2^{n-1} \times 2 = 3 \times 2^n$ 3-coloring options.

   (b) The 3-coloring algorithm Which uses the *spanning_tree* which returns the spanning tree : S. We define *top* and *child*, the function who return the top of the tree and the child of a vertex. The function return a map with vertex key and color (C = $\{c_1, c_{2,3}\}$).

Algorithm 1 $\mathcal{O}^*(3^n)$ algorithm for 3-coloring with spanning tree

```
function REC_COLORING(G : graph, S: spanning_tree, v:vertex, ϕ: color)
    for n ∈ CHILD(S, v) do
        colors = possible_colors(G, n)
        for c ∈ colors do
            ϕ[n] ← c
            if REC_COLORING(G, S, n, ϕ) then
                break
            return false
    return true

function 3-COLORING(G : graph)
    S = SPANNING_TREE(G)
    top = TOP(S)
    ϕ = {top → c1}
    return  REC_COLORING(G, S, top(S), ϕ)
```

2. (a) To reduce to 2-Sat, we want to find clauses containing exactly 2 literals that allow us to find the colors of the nodes outside the dominating set (X : the dominating set and C : the set of colors).

We define the variable:

$$\mathcal{V} = \bigcup_{v \in V(G)\setminus X} \{v_c | \forall c \in C\}$$

All vertices have 3 variables representing if variable i is true then the node can have the color i.

We define a function Cl which return a set of clauses for a vertex v (represent the possible colors for v) :

$$Cl(v) = \begin{cases} \{(v_1 \vee v_1), (\neg v_1 \vee \neg v_1)\} & |\phi(X \cap N_G(v))| = 3 \\ \{(v_y \vee v_y) | \{y\} = \phi(X \cap N_G(v))\setminus C\} & |\phi(X \cap N_G(v))| = 2 \\ \{(v_i \vee v_j) | \{i, j\} = \phi(X \cap N_G(v))\setminus C\} & |\phi(X \cap N_G(v))| = 1 \end{cases}$$

$\phi(X \cap N_G(v))$ is the set of colors link to a vertex v.

The first case is here to make the formula false (we can not choose a color if the node is already linked to 3 different colors in X). In the second case y represent the last color.

Finally, we define the set $\mathcal{C}$ of clauses :

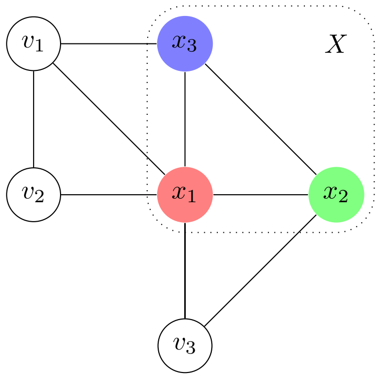
$$\mathcal{C} = \bigcup_{v \in V(G)\setminus X} Cl(v) \cup \{(\neg v_c \vee \neg n_c) | \forall n, c \in (N_G(v)\setminus X) \times (\phi(X \cap N_G(v))\setminus C)\}$$
$$\cup \{(\neg v_c \vee \neg v_c) | \forall c \in \phi(N_X(v)\}$$

The middle union represents the constraints between the external nodes of X. The last union prevents an external node to have the same colors that a neighbor in X.

Proof :

- Coloring $\Rightarrow$ 2-Sat :
  We assume that the vertices $V(G)\setminus X$ can be colored.
  We define for all v, $v_{c(v)} = true$ and for other color c we define $v_c = false$.
  All clauses generated by Cl are satisfied (the first case does not occur because the graph can be colored).
  We need to make a case disjunction for the other clauses of the form $(\neg v_c \vee \neg n_c)$. If we have $v_c = true$ then v has the color c has the color c and n cannot have the color c because they are neighbors, then $n_c = false$. Otherwise, v has not the color c, so $v_c = false$.
  Finally, the clauses of the form $(v_c \wedge v_c)$ (last union) is true because v is linked to a vertex in X which has the color c, so $v(x) \neq c$.
- 2-Sat $\Rightarrow$ Coloring :
  We assume that the formula for a dominating set X is satisfiable.
  So we have for all variable v in $V(G)\setminus X$ a c such that $v_c = true$, thanks to the function Cl.
  The same vertex v has no neighbor n such that $n_c = true$. Because we have the clause $(\neg v_c \vee \neg n_c)$, if $v_c = true$ then $n_c = false$.
  And thanks to the last union we can not have v_c is v is linked to a vertex who has the color c in X.

Example : $C = \{R, G, B\}$ (R : Red, G : Green, B : Blue)



For this graph we have the formula :

$$
\begin{aligned}
&(v_{1_G} \vee v_{1_G}) \wedge (\neg v_{1_B} \vee \neg v_{1_B}) \wedge (\neg v_{1_R} \vee \neg v_{1_R}) \wedge (\neg v_{1_G} \vee \neg v_{2_G}) \wedge \\
&(v_{2_G} \vee v_{2_B}) \wedge (\neg v_{2_R} \vee \neg v_{2_R}) \wedge (\neg v_{2_G} \vee \neg v_{1_G}) \wedge (\neg v_{2_B} \vee \neg v_{1_B}) \wedge \\
&(v_{3_B} \vee v_{3_B}) \wedge (\neg v_{3_R} \vee \neg v_{3_R}) \wedge (\neg v_{3_G} \vee \neg v_{3_G})
\end{aligned}
$$

Now, we can construct $c : V(G) \to [3]$. If the formula is not satisfiable then c can not be constructed. Otherwise, we can define c like that :

$$
c(x) = \begin{cases} \phi(x) & \text{if } x \in X \\ c \quad x_c = \text{true} \end{cases}
$$

If the formula is satisfiable we necessarily have c such that $x_c = true$ (by construction).

(b) Let G a graph such as $V(G) \geq 2$ and T the corresponding BFS tree. We have S_i the set of vertices at layer i of tree T.

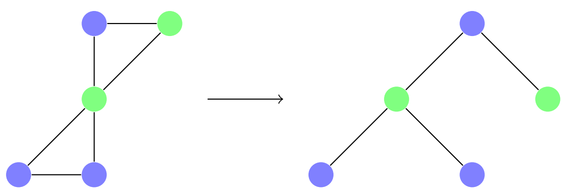
We define the sets $D_e = \{v | v \in S_i, \exists k, i = 2 \times k\}$ (vertices in an even layer) and $D_o = \{v | v \in S_i, \exists k, i = 2 \times k + 1\}$ (vertices in an odd layer).

D_e and D_o are distinct sets because in a BFT tree we can not have the same vertex several times, so the layer where the even and odd layers intersect is empty.

D_e De is a dominating set, because each layer is linked to the layer below and above it, so it is linked to all the other remaining vertices. We apply the same argument to D_o.

We can deduce that the upper bound of the smallest dominating set is $\frac{n}{2}$ $(n = |V(G)|)$. If $|D_e| < |D_o|$, then the smallest is D_e otherwise the smallest is D_o. So the worst case is when $|D_e| = |D_o| = \frac{n}{2}$.

Example :



(c) The *BFS_tree*, *bi_party* and *2-Sat* functions have a polynomial complexity. We use *3-coloring* (Algorithm-1) on a graph such that $|V(G)| = \frac{n}{2}$, then the complexity is $\mathcal{O}^*(3^{\frac{n}{2}}) = \mathcal{O}^*((\sqrt{3})^n)$. Therefor, the algorithm below has a complexity of $\mathcal{O}^*((\sqrt{3})^n)$.

Algorithm 2 $\mathcal{O}^*((\sqrt{3})^n)$ algorithm for 3-coloring with the smallest dominating set

```
function 3-COLORING-OPT(G : graph)
    T = BFS_TREE(G)
    D_e, D_o = BI_PARTY(T)
    X = D_e
    if |D_o| < B then
        X ← D_o
    φ = 3-COLORING(X)
    φ ← 2-SAT(G, X, φ)
```

3. (a) We have $X \subseteq V(G)$ and $1 \leq j < k$ such that $\chi(G[X]) \leq j$. We want to compute $Y \subseteq V(G)$ such that $\chi(G[Y]) \leq j+1$. The function $\mathcal{P}(X)$ give all possible subset of a set X. So for all $S \in \mathcal{P}(V(G))$ if S is on X then $\chi(G[S]) \leq j \leq j+1$. Otherwise, if we find $S_p \in \mathcal{P}(S)$ such as $S_p \in X$ (ϕ_p, j-coloring of S_p) and ϕ such that $\phi(x) = \phi_p(x)$ if $x \in S_p$ else $\phi(x) = j+1$ is a $j+1$-coloring of S, then S is in Y. For all subset we have to potentially calculate all the subset of a subset of cardinal $t(\mathcal{O}^*(2^n))$. We did this operation for all the subsets of vertices in the graph.

   So we have a complexity of

$$\sum_{t=0}^{n} \binom{n}{t} \mathcal{O}^*(2^t) = \mathcal{O}^*(3^n)$$

   (b) The final algorithm :

Algorithm 3 $\mathcal{O}^*(3^n)$ algorithm for k-coloring

```
function K-COLORING(G : graph)
    k = 0
    M = {}
    while V(G) not in M do
        k = k + 1
        for all C ⊆ V(G) do
            if C ∈ M then
                Continue
            for all S ⊆ C do
                if S ∉ M then
                    Continue
                φ_s = M[S] ∪{v → k|∀v ∈ C\S}
                if IS_COLORING(C, φ_s) then
                    M[C] ← φ_s
                    Break
    return k
```

The algorithms above finished, because a graph always has a colouring ($\leq n$). It repeats the algorithm detailed in the previous question at most n times, so it has a complexity of $\mathcal{O}^*(3^n)$.