Outline du rapport :

- Principe général de la méthode, espace de recherche, fonction objective
- Algorithmes utilisés
- Explication du code
- Tests et résultats
- Commentaires

1°/ Heuristique du MAX STABLE :

Principe général :

il est difficile, voire impossible, de trouver un k-coloring d'un grand graphe G (par exemple $|V| \geq 1000$) avec k proche de $\chi(G)$ en appliquant directement un algorithme donné α sur G. Pour colorier les grands et très grands graphes en appliquant directement un algorithme de coloration α donné sur G. Dans ce cas, une approche possible est d'appliquer un prétraitement pour extraire i grands ensembles indépendants (i stables maximales) du graphe afin d'obtenir un graphe résiduel beaucoup plus petit qui devrait être plus facile à colorier que le graphe initial. Un algorithme de coloration α est ensuite invoqué pour colorier le graphe résiduel.
Puisque chacun des i ensembles forme une classe de couleur, 1 plus le nombre de couleurs nécessaires pour le graphe résiduel donne une borne supérieure pour $\chi(G)$.

Les méthodes conventionnelles pour la phase de prétraitement opèrent de manière greedy en extrayant à chaque fois un ensemble indépendant du graphe G.
Puis supprimer de G les sommets de l'ensemble indépendant ainsi que leurs arêtes incidentes. Nous pouvons affiner le choix de l'ensemble indépendant à supprimer et en prenant l'ensemble indépendant qui est connecté au plus grand nombre possible de sommets du graphe restant.

Un k-coloring légal d'un graphe donné G = (V, E) correspond à une partition de V en k ensembles indépendants. Supposons que l'on veuille colorier G avec k couleurs. Supposons maintenant que nous extrayons t < k ensembles indépendants I1, ..., It de G. Il est évident que si nous pouvions maximiser le nombre de sommets couverts par les t ensembles indépendants (c'est-à-dire que $|I1 \cup ... \cup It|$ est aussi grand que possible), nous obtiendrions un graphe résiduel avec moins de sommets lorsque $I1 \cup ... \cup It$ sont retirés de G. Cela pourrait à son tour entraîner une diminution du nombre de sommets et en se ramenant à une borne supérieure de coloration inférieure ou égale à k - t couleurs, car il reste moins de sommets dans le graphe résiduel.

Définition d'un max stable et complexité algorithmique :

Nous avons utilisé une heuristique qui se base principalement sur la relation entre le nombre chromatique et le cardinal de l'ensemble indépendant maximal.
$\chi(G) + \alpha(G) <= n+1$ où n est le nombre de sommets du graphe.

Preuve par Induction : On remarque qu'un seul sommet a $\chi(G) + \alpha(G) = 2$.En ajoute ensuite chaque sommet un par un. Remarquez qu'en ajoutant un seul sommet, on ne peut les augmenter que d'une unité. En faisant cela, nous convertissons G en G*. Disons que le nouveau sommet v est connecté aux sommets 1 à r de notre G précédent et n'est pas connecté aux sommets r+1 à n. Pour que l'inégalité soit résolue conformément à notre hypothèse d'induction, les deux doivent être augmentés en un point.

Pour que le $\chi(G)$ augmente, les sommets 1 à r, v est connecté aux r couleurs dans toute coloration correcte de G. Ainsi $|r| >= \chi(G)$. De plus, l'ensemble indépendant maximum dont v doit faire partie doit être contenu dans G - r. Donc $|G - r| >= \alpha(G)$ En ajoutant les deux inégalités, on obtient $|G| = n >= \chi(G) + \alpha(G)$

Maintenant les deux augmentent, montrant que $n+2>=\chi(G^*) + \alpha(G^*)$ Ainsi $\chi(G^*) + \alpha(G^*) <= |G^*| + 1$ CQFD La borne est valide pour les graphes complets et les graphes nuls.

Nous comprenons aussi que ce problème est NP-complet, mais il existe un algorithme séquentiel très efficace (en terme de temps d'exécution) permettant de calculer ce stable. La complexité est clairement O(m) où m est l'ordre du graphe, or d'un point de vue technique cet algorithme peut très bien être parallélisé et donc optimisé pour les graphes très larges.

Algorithme d'extraction du stable maximum :

Étant donné un graphe G(V,E), il est facile de trouver un seul MIS en utilisant l'algorithme suivant :

1. Initialiser I à un ensemble vide
2. Tant que V n'est pas vide :
    - Choisir un nœud $v \in V$;
    - Ajouter v à l'ensemble I ;
    - Retirer de V le nœud v et tous ses voisins.
3. Retourner I

2°/ Explication du code :

```python
23 """
24     Retourne un max stable sous-graphe de G, en choisissant à chaque itération de la boucle
25     while le sommet de degré minimal
26 """
27 def _maximal_independent_set(G):
28     result = set()
29     #Initiailiser le graph restant à G
30     remaining = set(G)
31     #Pour tout noeud dans le graph restant
32     while remaining:
33         #Choisir le noeud avec le degré min (pour éviter la suppression de beaucoup de noeuds)
34         G = G.subgraph(remaining)
35         v = min(remaining, key=G.degree)
36 
37         #Ajouter le noeud v au stable
38         result.add(v)
39         #Enlever le noeud v du Graph ainsi que ses voisins
40         remaining -= set(G[v]) | {v}
41     return result
```

- Ligne 28 : initialisation d’un ensemble vide qui contiendra
- Ligne 30 : Initialisation de l’ensemble des sommets du sous-graphe restant au graph initial
- Lignes 32 - 40 : tant que le graph restant n’est pas encore vide, nous choisissons à chaque itération le sommet de degré minimal min(remaining, key=G.degree), nous optons pour ce degré pour permettre à l’algorithme d’explorer plus de sommets en maximisant le stable, une fois v calculé, on l'enlève du graph ainsi que ses voisins.
- Ligne 41 : nous retournons ainsi l’ensemble des sommets qui n’ont aucune relation d’adjacence entre eux.

```python
43 """
44     Cet fonction permet d'appeler la procédure d'extraction du max-indep-set (stable maximal)
45     et soustrait l'ensemble du graphe de base,
46 """
47 def strategy_independent_set(G, colors):
48     remaining_nodes = set(G)
49     while len(remaining_nodes) > 0:
50         nodes = _maximal_independent_set(G.subgraph(remaining_nodes))
51         remaining_nodes -= nodes
52         yield from nodes
53     return nodes
```

- Ligne 48 : initialisation de l’ensemble des sommets restants
- Lignes 49 - 52 : Tant qu’il y en a encore des sommets dans le graphes, extraire le stable maximum, enlever les sommets de ce dernier du graph orignal
- Ligne 53 : nous retournons uniquement les sommets du stable, car chacun des sommets restant se verra attribuer la première couleur non utilisée (greedy coloring) sur ce dernier.

```python
55 def GraphColoring(G):
56     colored_graph = G
57     if len(G) == 0:
58         return {}
59     colors = {}
60     nodes = strategy_independent_set(G, colors)
61     for u in nodes:
62         # Ensemble des voisins du noeud courant
63         neighbour_colors = {colors[v] for v in G[u] if v in colors}
64         # Trouver la première couleur non utilisé
65         for color in itertools.count():
66             if color not in neighbour_colors:
67                 break
68         # Donner la couleur au noeud courant
69         colors[u] = color
70 
71     color_map = []
72     for node in colored_graph:
73         color_map.append(colors_option[colors[node]])
74 
75     colors_used = set(color_map)
76     return color_map, len(colors_used)
```

- Lignes 61 - 69 : pour chacun des sommets retourné par la fonction strategy_independent_set, qui représente le stable maximal, nous lui attribuons la première couleur non utilisée par ses voisins.
- Lignes 71 - 73 : pour tous les sommets, s'il appartient au stable alors on lui a déjà attribué une couleur dans la ligne 69, sinon lui attribuer une nouvelle couleur.