I- Problème SAT :

1- Complexité temporelle :

- **Vérification d'intégrité :** On considère n est le nombre de variables et m le nombre de clauses. Cela provient des vérifications effectuées sur les variables et les clauses. donc la complexité est de O(n + m)
- **Unité de propagation :** Parcourt les clauses à la recherche de clauses unitaires. À chaque itération, on applique des changements sur les clauses, avec une complexité de $O(m \cdot n)$, où m est le nombre de clauses et n le nombre de variables.
- **Fonction solve :** La fonction `solve` explore toutes les affectations possibles des variables en utilisant le backtracking (Division et Conquête), ce qui donne une complexité de $O(2^n)$ dans le pire des cas, où n est le nombre de variables. Cette partie domine la complexité de l'algorithme.

En conclusion la complexité temporelle est exponentielle ($O(2^n)$).

2- Complexité spatiale :

- **Stockage des clauses et variables :** $O(m \cdot n)$, où m est le nombre de clauses et n le nombre de variables.
- **Affectations :** Le dictionnaire `assignments` utilise $O(n)$ espace pour stocker les affectations des variables.
- **Appels récursifs :** L'espace des appels récursifs est $O(n)$ dans le pire des cas, en raison de la profondeur de la récursion.

  La complexité spatiale globale est donc O(n + m), mais peut atteindre O(n) supplémentaire en raison de la profondeur de récursion.

3- Graphes représentants le temps d'exécution et utilisation de la mémoire de l'algorithme SAT :

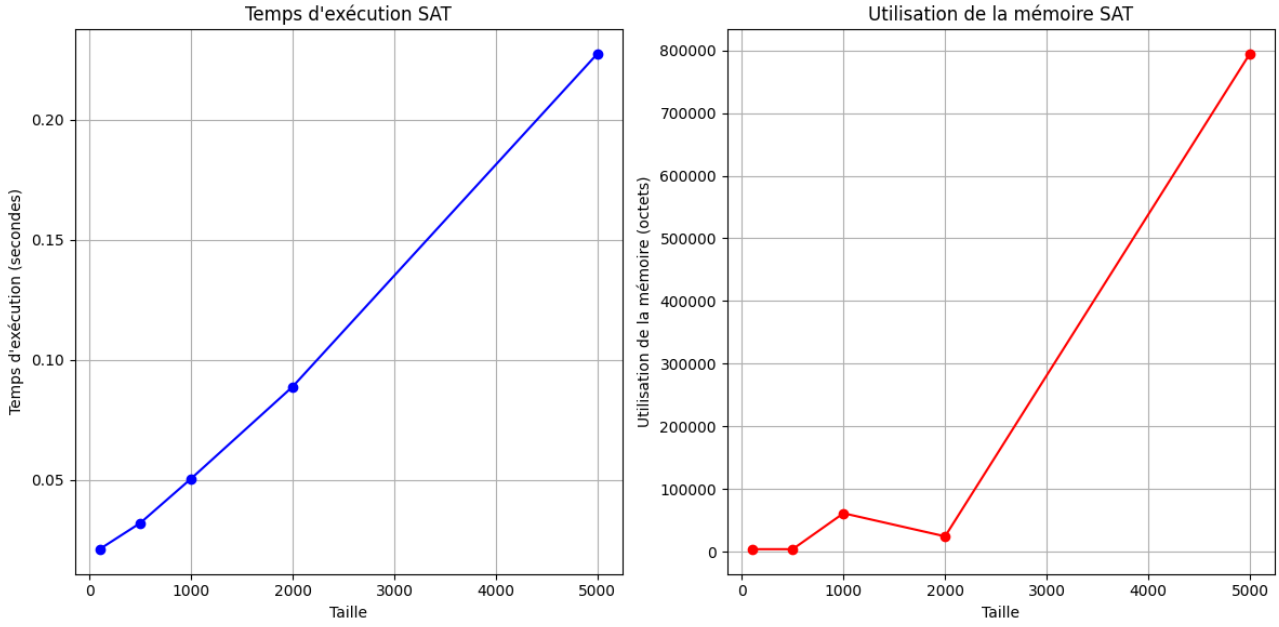


Figure 1:

5- optimisations et conditions qui réduisent la complexité de SAT :

- **Propagation unitaire :** La propagation unitaire simplifie les clauses. Lorsqu'une clause unitaire est rencontrée, une valeur est immédiatement affectée à la variable correspondante. Cette étape peut réduire le temps de calcul à $O(n)$ pour des instances satisfaites par propagation unitaire.
- **Backtracking :** Divise le problème en sous-problèmes en affectant successivement des valeurs `True` ou `False` aux variables, avec un retour en arrière en cas de conflit. La complexité dans le pire cas est exponentielle ($O(2^n)$). Des heuristiques judicieuses et un retour rapide améliorent l'efficacité.
- **Heuristiques de sélection des variables :** L'algorithme utilise des heuristiques comme le Maximum Occurrence in Clause (MOC) pour améliorer les performances.

II- Conversion SAT en 3SAT :

1- Complexité temporelle :

- **Boucle sur les Clauses :** Boucle sur les Clauses : Si le nombre total de clauses est alors cette boucle s'exécute m fois.
- **Transformation de chaque clause :**
  - 1 littéral : Ajout de 4 nouvelles clauses, $O(1)$ par clause.
  - 2 littéraux : Ajout de 2 nouvelles clauses, $O(1)$ par clause.
  - 3 littéraux : Clause inchangée, $O(1)$.
  - $\geq$ 4 littéraux : Découpage en plusieurs clauses avec variables intermédiaires, $O(k)$.

  Donc la complexité globale est : $O(m \cdot n)$, où m est le nombre de clauses et n le nombre moyen de littéraux par clause.

2- Complexité spatiale :

- **Espace pour les Variables**
  - Variables d'origine : $O(v)$, où v est le nombre de variables initiales.
  - Variables ajoutées : $O(m \cdot n')$, où n' est le nombre de nouvelles variables par clause.
  - Clauses d'origine : $O(m \cdot n)$.
  - Clauses transformées : $O(m \cdot n')$.
    Donc l'espace total est : $O(v + m \cdot n + m \cdot n')$.

L'espace total approximatif$O(v + m \cdot n)$.

3- Structures de données :

- **Variables :** Liste des variables initiales, étendue avec de nouvelles variables intermédiaires.
- **Clauses :** Liste de listes modifiée pour contenir exactement 3 littéraux.
- **Nouvelles variables :** Générées dynamiquement pour transformer les clauses.

4- Optimisations pour réduire la complexité temporelle et spatiale:

- **Minimisation des nouvelles variables :** Réutiliser des variables temporaires déjà générées lorsque possible pour économiser de l'espace.
- **Propagation unitaire :** Réduit v et m avant la recherche.
- **Détection des contradictions :** Permet de couper des branches entières de l'arbre de recherche.
- **Simplification des clauses :** Suppression des clauses redondantes ou tautologiques pour alléger les calculs.
- **Parallélisation :** Les transformations de clauses peuvent être réalisées indépendamment les unes des autres, ce qui est idéal pour une parallélisation.

III- 3-SAT :

1- Complexité temporelle :

- **Initialisation :** Chargement des variables et des clauses depuis le fichier JSON : $O(v + m)$, où :
  - v : Nombre de variables.
  - m : Nombre de clauses.
- **Extraction des clauses unitaires :** $O(m)$.
- **Mise à jour des clauses :** Parcourt chaque clause (longueur moyenne n) et modifie leur contenu. Pire cas : $O(m \cdot n)$.
- **Conversion en CNF (dans `is_satisfiable`) :**
  - Construction d'un dictionnaire pour mapper les variables :$O(v)$.
  - Conversion des clauses en littéraux entiers : $O(m \cdot n)$.
- **Recherche récursive :**
  - Pire cas : Exploration de toutes les affectations possibles des variables.
  - Nombre total d'assignations : 2^v.
  - Chaque étape inclut la propagation unitaire : $O(m \cdot n)$.

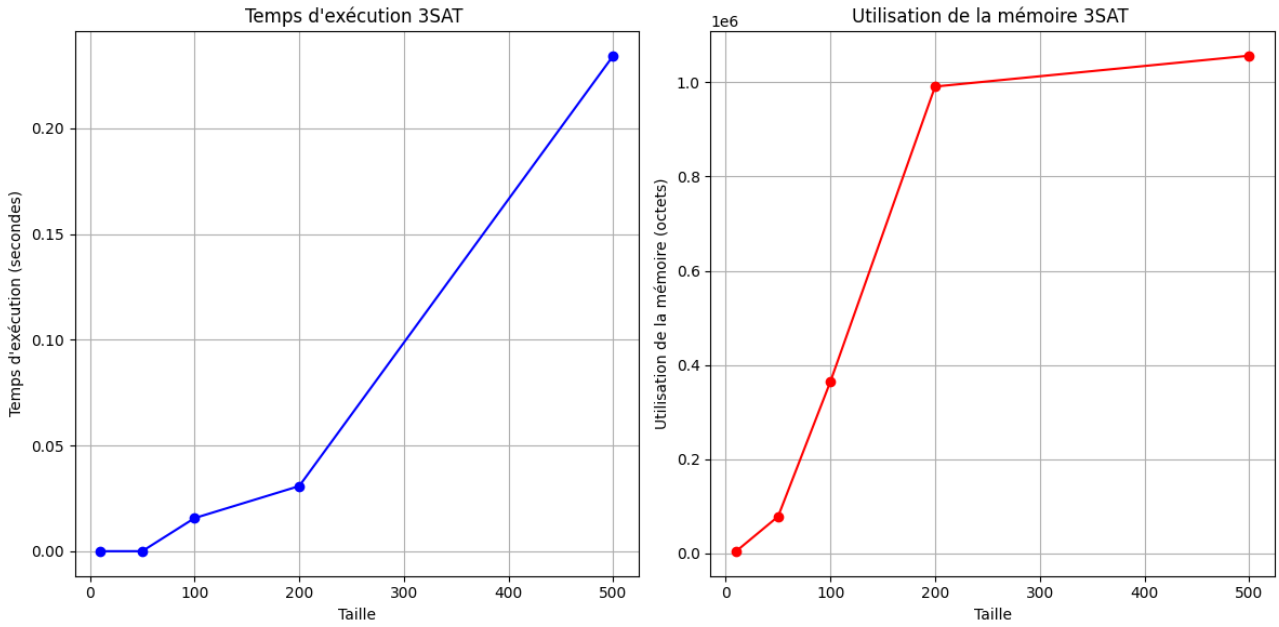
La complexité totale (pire cas) est : $O(2^v)$.

2- Complexité spatiale :

- **Représentation des données :**
  - Variables : Stockées dans une liste $O(v)$.
  - Clauses : Liste de listes ($O(m \cdot n)$, où n est le nombre moyen de littéraux par clause (ici $n = 3$)).
- **CNF (dans `is_satisfiable`) :**
  - CNF utilise des entiers au lieu de chaînes, ce qui peut économiser de l'espace.
  - Espace requis : $O(m \cdot n)$.
- **Espace auxiliaire :**
  - Assignations des variables : Dictionnaire de taille $O(v)$.
  - Clauses modifiées pendant la propagation unitaire : $O(m \cdot n)$.

D'où l'espace total est de $O(v + m \cdot n)$.

3- Graphes représentants le temps d'exécution et utilisation de la mémoire de l'algorithme 3-SAT :



4- Les structures de données utilisées :

- **Variables (`self.variables`) :** Liste de chaînes représentant les noms des variables du problème SAT, initialisée à partir d'un fichier JSON. Exemple : `["u1", "u2", "u3"]`.
- **Clauses (`self.clauses`) :** Liste de listes, où chaque sous-liste représente une clause contenant des littéraux. Exemple : `[["u1", "!u2", "u3"], ["!u1", "u2", "u3"]]`.

- **Assignments :** Dictionnaire dynamique pour les affectations des variables pendant la vérification de la satisfiabilité. Exemple : {`"u1": True, "u2": False, "u3": True`}.
- **Mapping des Variables :** Un dictionnaire variablemap est utilisé pour mapper les noms de variables (sous forme de chaînes) à des entiers pour faciliter la manipulation des littéraux lors de la résolution

5- Optimisations pour réduire la complexité de 3-SAT :

- **Propagation unitaire optimisée:** Simplifie les clauses en affectant directement les littéraux unitaires. Cela réduit v et m de manière significative, diminuant le nombre de variables actives avant la recherche.
  Utiliser une structure de données efficace pour suivre les clauses unitaires (ex. : une file d'attente)
- **Détection des contradictions :** Une contradiction immédiate (clauses contenant x et $\neg x$) permet de couper une branche entière de l'arbre de recherche, évitant des explorations inutiles.
- **Heuristiques de choix de variable :** Implémenter des stratégies pour choisir les variables à affecter en premier.
- **Simplification des clauses :** Avant la résolution, les clauses redondantes ou tautologiques peuvent être supprimées pour alléger les calculs. Cette technique réduit la taille initiale de m.

Environnement d'expérimentation :

- **Environnement matériel :**
  - Processeur (CPU) : AMD ryzen 5 3450U with Redeon Vega Mobile Gfx.
  - Carte graphique (GPU) : AMD Radeon(TM) Graphics.
  - RAM : 11,4Go.

- **Environnement logiciel :**
  - Système d'exploitation : windows 11.
  - Environnement de Développement : Visual studio code.
  - Langages de programmation : python.