SAT Solver

PATIA
Ait gueni ssaid Abderrahmane Tangara Abdoulhakim
Abderrahmane.Ait-gueni-ssaid@etu.univ-grenoble-alpes.fr
Abdoulhakim.Tangara@etu.univ-grenoble-alpes.fr

28 april 2023

Table des matières

Introduction

Au sein du cours PATIA, nous avons exploré l'univers de la planification automatique. Dans un premier temps, nous avons étudié la modélisation de problèmes grâce à l'utilisation de PDDL4J. Nous avons ensuite mis en pratique nos connaissances en écrivant un domaine de Sokoban ainsi qu'un encodeur permettant de transformer les parties du jeu en problèmes PDDL. Enfin, nous avons poursuivi notre apprentissage en écrivant un encodeur de problèmes PDDL4J en format SAT-CNF, avant de résoudre ces problèmes avec le solveur Sat4J.

1 Lien du Git

https ://github.com/aaitguenissaid/patia_22-23

2 Instructions pour l'exécution

Exécutions à partir de la racine du dépôt GIT :
BlocksWorld :

```
# afficher le èproblme
cat exercice1_Blocksworld/blocks_my_problem.pddl
```

BlocksWorld :

```
# afficher le domaine
cat exercice5_Pursuit-Evasion/domain.pddl

# afficher le probleme
cat exercice5_Pursuit-Evasion/p01.pddl

# executer la resolution du probleme
bash exercice5_Pursuit-Evasion/run_pursuit_pddl4j.sh
```

Sokoban :

```
# afficher le domaine
cat ProjetSokoban/SokobanPDDLParser/src/main/resources/domain.pddl

# compiler
bash ProjetSokoban/compile.sh

# executer
# remplacer <niveau> par : test0, test1, ..., test30
bash ProjetSokonan/run.sh <niveau>

# afficher un probleme parse
# Les noms des problemes sont sous la forme [Scoria/Scoria2/Scoria3]-Level[1..10]
# exemple 1 :
cat ProjetSokoban/SokobanPDDLParser/results/PDDLproblems/Scoria-Level1.pddl
# exemple 2 :
cat ProjetSokoban/SokobanPDDLParser/results/PDDLproblems/Scoria2-Level5.pddl
```

SAT encoder (se déplacer dans le dossier SAT/scripts) :

```
 cd SAT/scripts/

# compiler
 bash compile.sh

# executer le solveur SAT sur un domain et un provleme
bash run.sh <domaine.pddl> <probleme.pddl>

# encoder et resoudre le premier probleme avec le SAT solver:
bash run_gripper.sh
bash run_depot.sh
bash run_logistics.sh
bash run_robot.sh
bash run_robot.sh
bash run_robot_advanced.sh

# encoder et resoudre le premier probleme avec HSP:
bash run_blocksworld_pddl4j.sh
bash run_depot_pddl4j.sh
bash run_gripper_pddl4j.sh
bash run_logistics_pddl4j.sh

# script de tests pour lancer le solveur SAT ou HSP sur tout :
# Pas sur qu'il marche sur d'autres machines il faut les adapter
# et aussi compiler le validateur avec le bon chemin.
bash run_all_HSP.sh
bash run_all_SAT.sh
```

3 Travail Effectué

3.1 Modélisation de problème - Blocksworld

— Modélisation d'une instance de problème pddl pour le domaine blocksworld
— Code dans le dossier exercice_1_Blocksworld

3.2 Modélisation de domaine et problème - Poursuit Évasion

— Modélisation d'un domaine et création d'un problème pour l'exercice poursuit-évasion
— Code dans le dossier exercice_5_Poursuit-évasion

3.3 Sokoban

Le sokoban est un jeu de puzzle dont le but est de déplacer des boîtes sur des emplacements cibles dans un entrepôt. Le joueur incarne un gardien qui peut soit se déplacer dans 4 directions (haut, bas, droite, gauche) ou pousser une boite dans une direction, si aucun obstacle ne l'empêche). Le but ici a été de coder le domaine du jeu Sokoban, transformer un niveau fourni au format .json en un problème pddl, trouver le plan, puis convertir celui-ci en actions qui, lorsque exécutées par le serveur de jeu, résoudra le puzzle.

3.3.1 Le Domaine

Nos objets (types) :
— direction : les 4 directions (up > haut , down > bas , right > droite , left > gauche).
— position : un emplacement sur le niveau, qu'il soit destination, libre, ou occupé par une boite.
— guard : le gardien.
— box : une boite à déplacer vers une case cible.

Nos prédicats :
— guardOn ?x - position ?y - guard : Un gardien y est sur une position x.
— boxOn ?x - position : Une boite se trouve sur une position x.
— isNeighbor ?x - position ?y - position ?how - direction : Expression de l'adjacence de deux emplacements dans une direction donnée.
— isFree ?x - position : La position x est inoccupée.

Nos actions :
— move (?g - guard ?from - position ?to - position ?how - direction) : Déplacer le gardien "g" d'une position "from" à une autre "to", dans la direction "how".
— push (?g - guard ?box - box ?from - position ?middle - position ?to - position ?how direction) : Le gardien "g" sur une position "from" pousse la boite "box" sur une position "middle" vers une position "to" dans la direction "how".

3.3.2 Encodage d'un Niveau

Pour encoder un niveau nous avons créé une classe SokobanDomainParser dans laquelle on analyse le fichier .json correspondant dont le chemin a été passé en paramètre. On en extrait le champ "title" qui correspond au titre du niveau, et le champ "testIn" qui est une chaine de caractères représentant la grille de jeu. Les lignes sont séparées par le caractère "n\n" Nous avons converti le champ testIn en un tableau 2D (une dimension par ligne).

Pour chaque élément que nous pouvions rencontrer, nous avons créé un attribut :

```
1      char wall = '#';
2      char box = '$';
3      char destination = '.';
4      char boxOnStoragePlace = '*';
5      char guard = '@';
6      char guardOnStoragePlace = '+';
7      char floor = ' ';
```

Pour chaque case que l'on rencontre, on crée un objet position, l'objet qui s'y trouve, et le prédicat d'état initial correspondant à l'état de l'objet, ou de la relation entre les objets aux mêmes coordonnées (une case libre, une boite sur un emplacement, ...). Une position est identifiée par P suivi de XY sa coordonnée (ex. P12, P24, ...). On confère aux boites la même nomenclature avec B en préfixe, et le gardien quant à lui est identifié par un G (il n'y en a qu'un seul).

Pour chaque position, on effectue une recherche de voisin, et on ajoute les prédicats isNeighbor correspondants à l'état initial en cours de construction. Si une position est une destination (identifiée par un "."), on ajoute un prédicat boxOn aux états finaux.

Une fois que les informations du problème ont été collectées, la méthode parse crée un fichier .pddl. Dans ce fichier, on y écrit le titre du problème, le domaine, les objets, l'état initial et l'objectif (avec les éléments nécessaires au formattage comme les parenthèses, ...). Le fichier est créé dans le répertoire results/PDDLproblems et porte le nom du titre du niveau.

3.3.3 Recherche du Plan

Une fois que nous avons converti le niveau en un problème au format .pddl, on utilise dans une nouvelle classe SokobanSolutionSolver, le solveur HSP de la librairie pddl4j pour nous fournir un plan. Dans le but d'exploiter ses informations, on redirige la sortie de l'exécution vers un fichier. Ensuite, nous filtons les lignes contenant les actions move et push, pour obtenir un fichier contenant uniquement la série d'instruction que nous souhaitons transformer en un chemin.

3.3.4 Extraction du chemin

Ici, une classe SokobanResultInterpreter s'occupe de lire le ficher filtré contenant la série d'actions à réaliser.

Pour déterminer dans quelle direction déplacer le gardien, on examine les différences des coordonnées x et y des positions "from" et "to" pour les actions "move", et les différences entre les positions "from" et "middle" si l'action est "push" (comme indiqué précédemment, nous faisons apparaitre la coordonnée X et Y de la position dans son nom).

— Si la différence en X est 1, on ajoute R (right) au chemin, L (left) si -1.
— Si la différence en Y est 1, on ajoute D (down) au chemin, U (up) si -1.

La chaine de caractères correspondant au chemin est ensuite stockée dans un fichier texte.

3.3.5 Résolution du puzzle

Exécution du code du jeu Sokoban, qui va lire le chemin obtenu, puis visualisation du résultat sur le navigateur à l'adresse fournie par le serveur codingame lors de l'exécution.

3.4 SAT encoder

Travail réalisé dans le dossier SAT du dépôt git.

Cette partie du travail a consisté en la création d'un code, permettant de convertir une instance d'un Problème de planification (pddl4j), en un objet encodé sous forme de clause SAT-CNF, qui es par la suite passé en entrée de solveur SAT (sat4j), et en extraire un plan valide s'il existe.

Dans notre implémentation, on fait appel à la fonction *estimate* de la classe StateHEuristic de la libraire pddl4j pour obtenir une estimation du nombre d'étapes à réaliser.

Pour encoder le problème, nous avons appliqué les formules décrites dans les diapositives pour les items suivants :

— L'état initial
— Les actions
— Les transitions
— L'état final

L'ensemble de clauses obtenu (CNF) a ensuite été passé en entrée du solveur SAT4J, qui en sortie a retourné un modèle (une série d'identifiants d'actions). Dans notre implémentation, il a fallu parcourir dans le sens inverse le modèle et prendre les valeurs qui correspondent aux actions, puis les décoder avant de les ajouter au plan séquentiel.

On a pu optimiser l'encodage en rajoutant une contrainte pour enlever les actions qui ne sont pas valide, (celle qui ont l'intersection des effets positifs et négatifs non-vide et celle qui ont aucun effet ni positif ni négatifs. Cela a permis d'avoir des encodage un peu plus court mais il n'est pas tres significative.

3.4.1 Interprétation des résultats

Le seul Domaine qui a un graphe facile a interpréter est Logistics, On voit bien que le solveur SAT on est plus rapide, propose pour certains problèmes des plans plus courts. On voit que c'est pareil pour Blocksworld sauf que le solveur SAT n'arrive pas a résoudre les problèmes qui sont plus grands. c'est a cause de l'explosion du nombre de clauses générées. pour le reste des domaines les résultats ne sont pas concluants.

Mais on pense que c'est intéressant de voir comment optimiser encore plus l'encodage, pour raccourcir le temps de calcul pour le solveur. Sinon ces résultats indique que le solveur HSP est meilleur globalement.

Les solutions proposées par le solveur SAT sont validées par le Validateur. On a remarqué que le domaine grippers n'a aucun plan trouvé ni par SAT ni par HSP. les plans proposés ont été refusés par le validateur. les graphes de ce derniers sont juste a titre indicatifs.

3.5 SAT et HSP Benchmarks

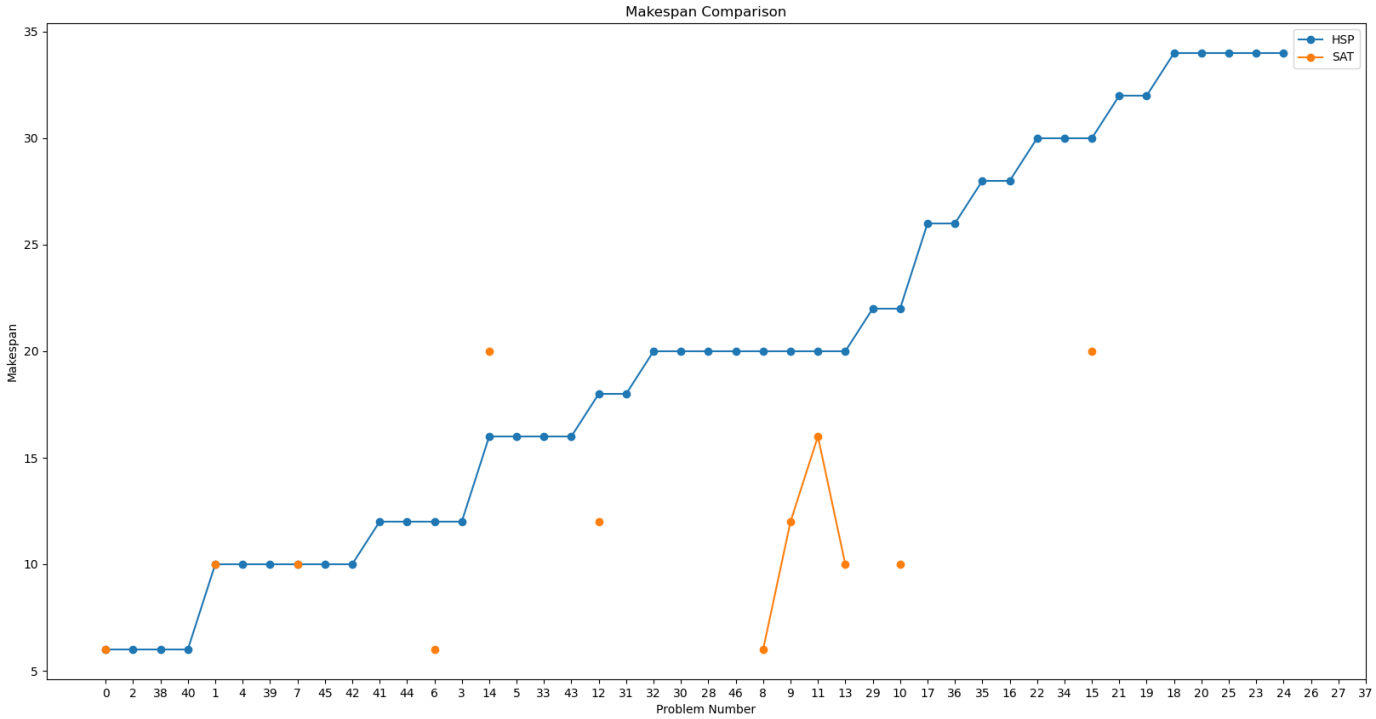

FIGURE 1 – Blocksworld : makespan

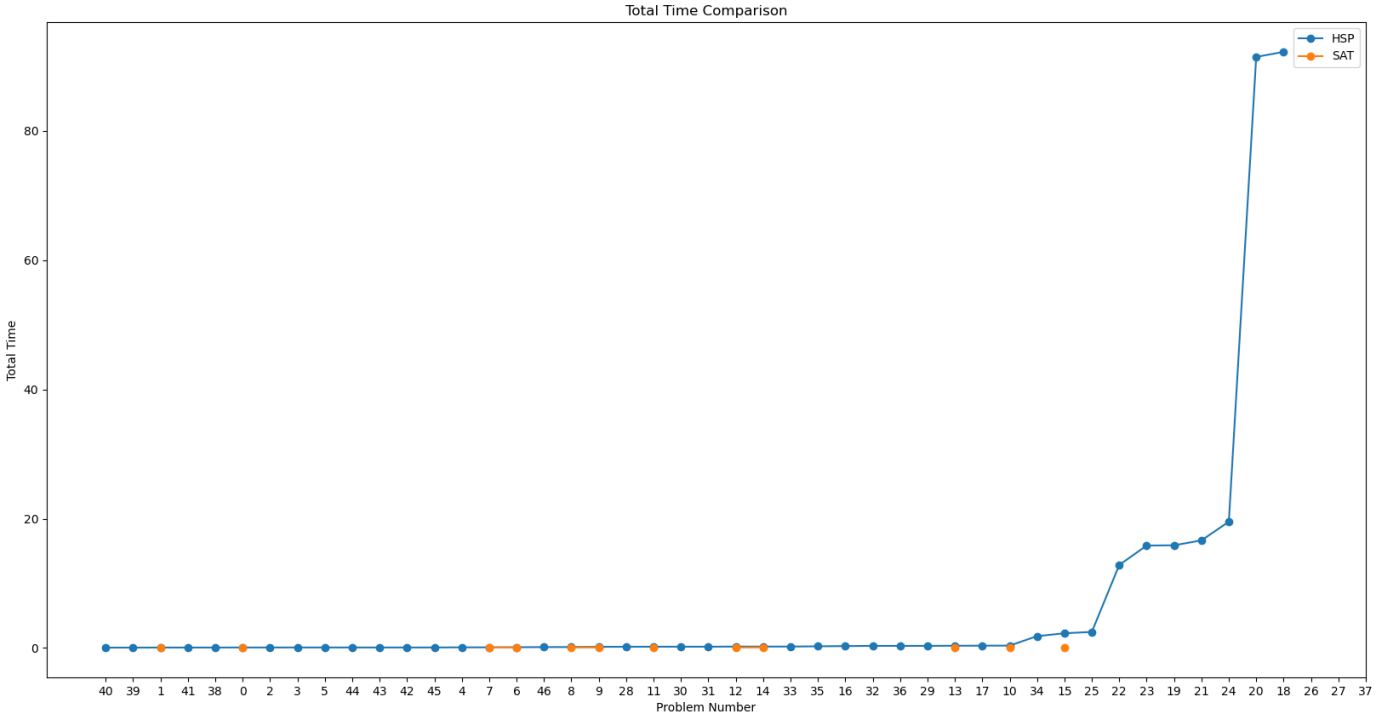

FIGURE 2 – Blocksworld : temps total

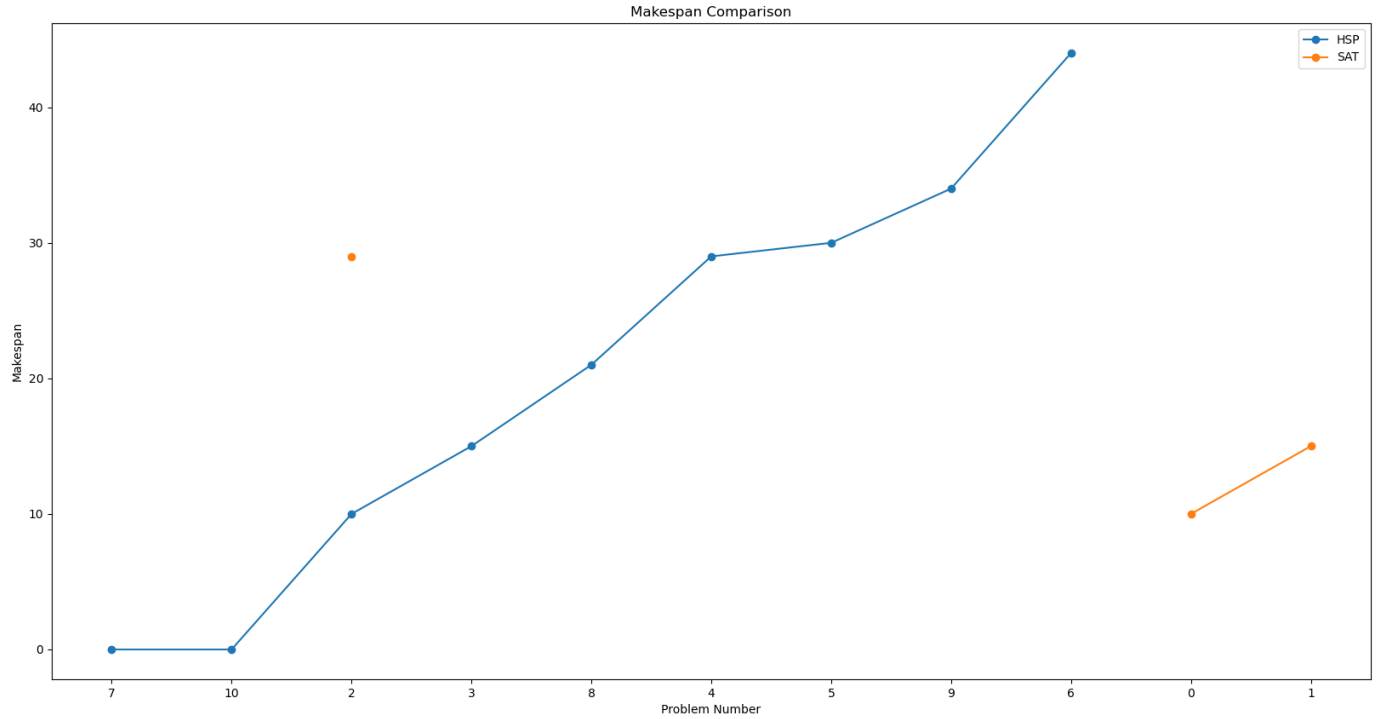

FIGURE 3 – Depots : makespan

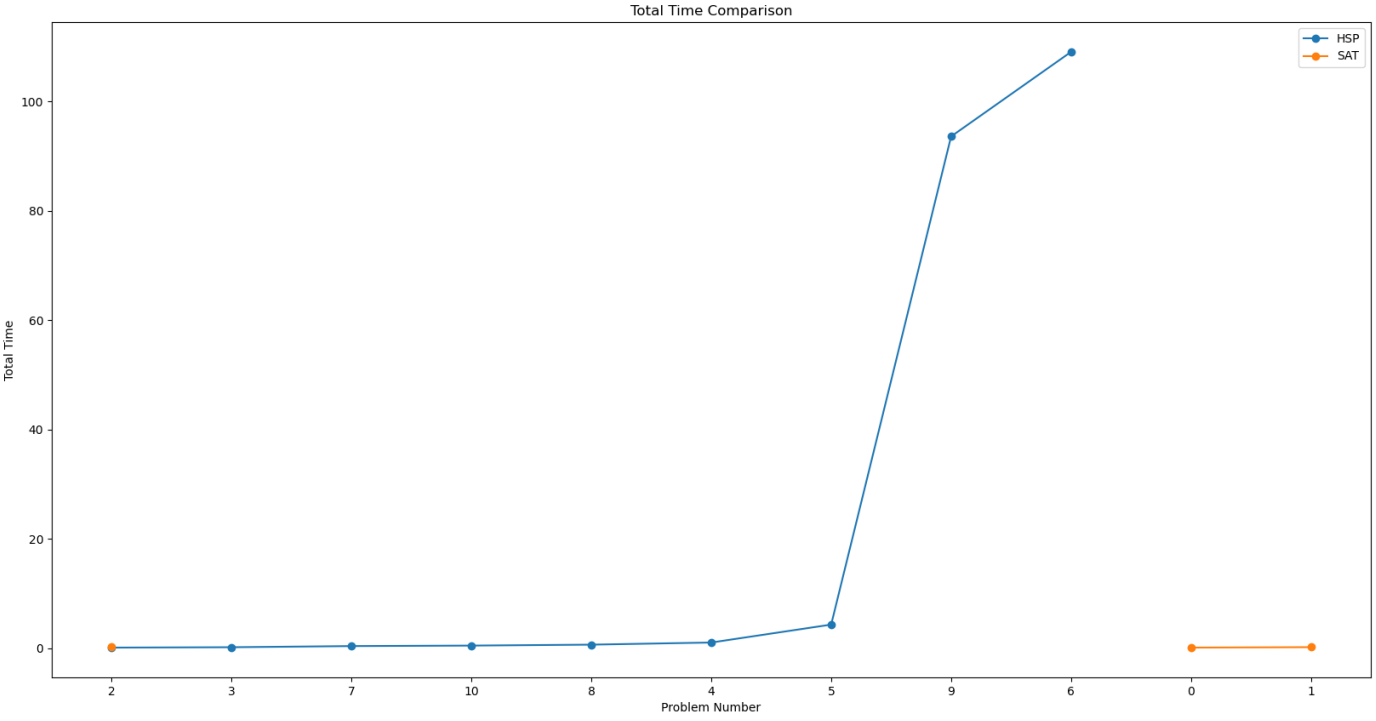

FIGURE 4 – Depots : temps total

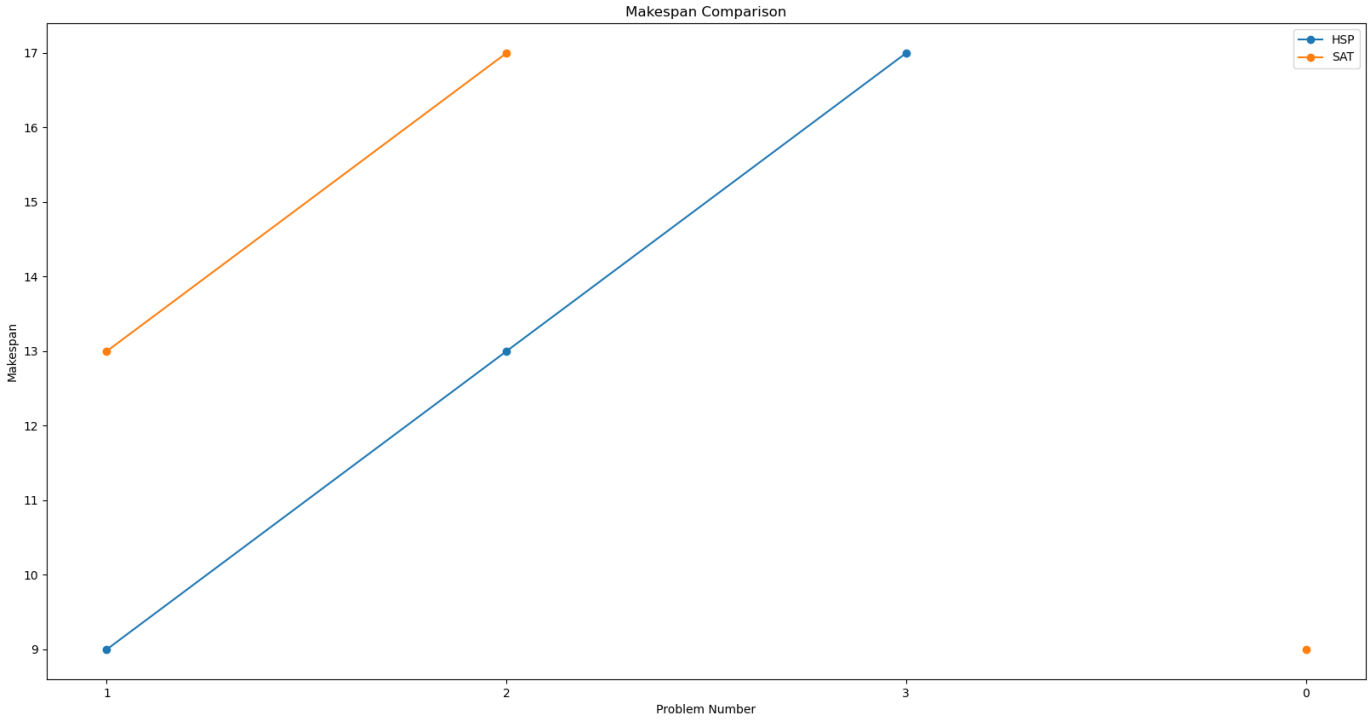

FIGURE 5 – Gripper : makespan

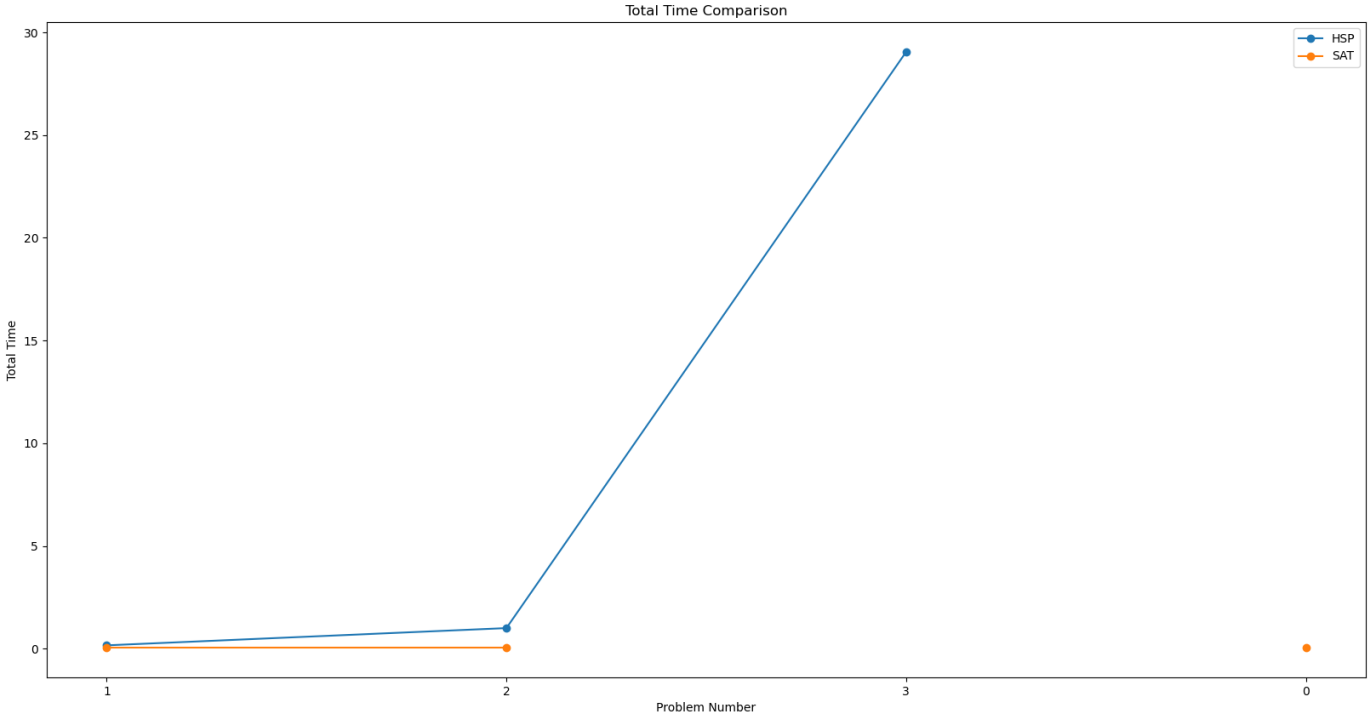

FIGURE 6 – Gripper : temps total

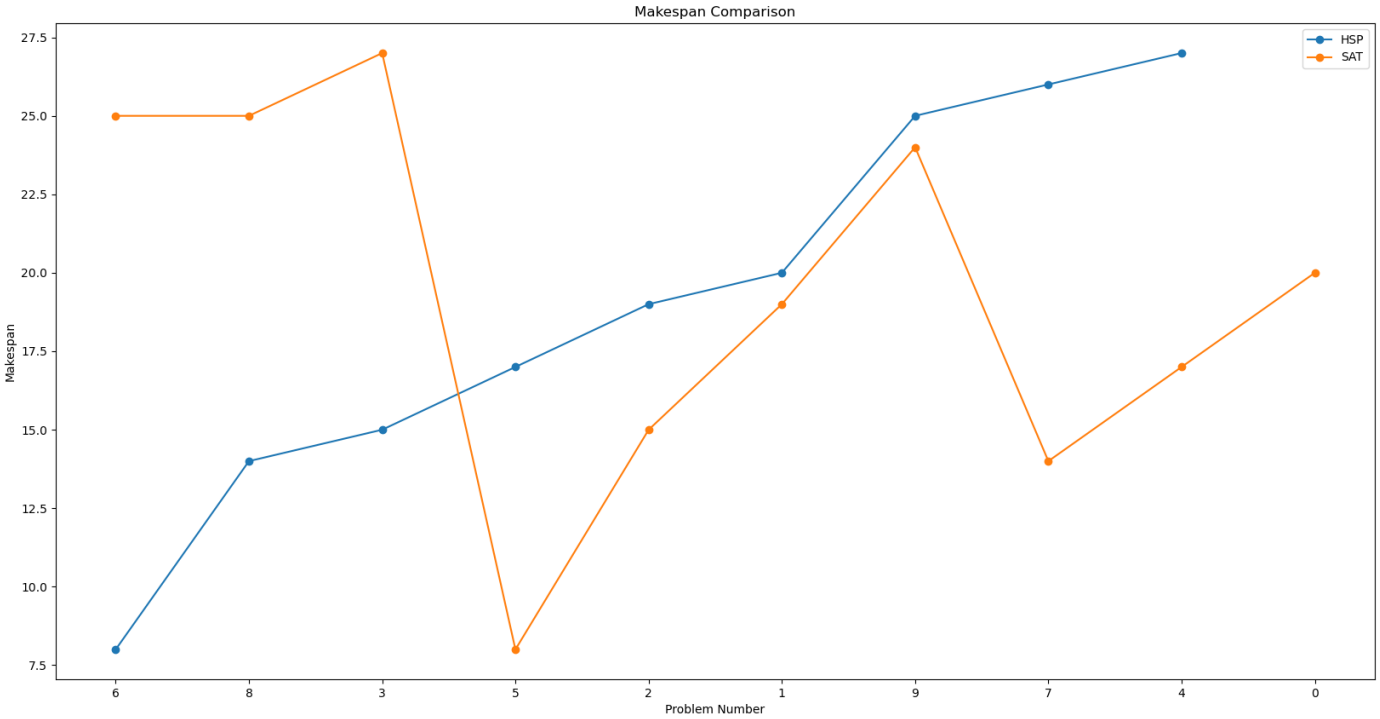

FIGURE 7 – Logistics : makespan

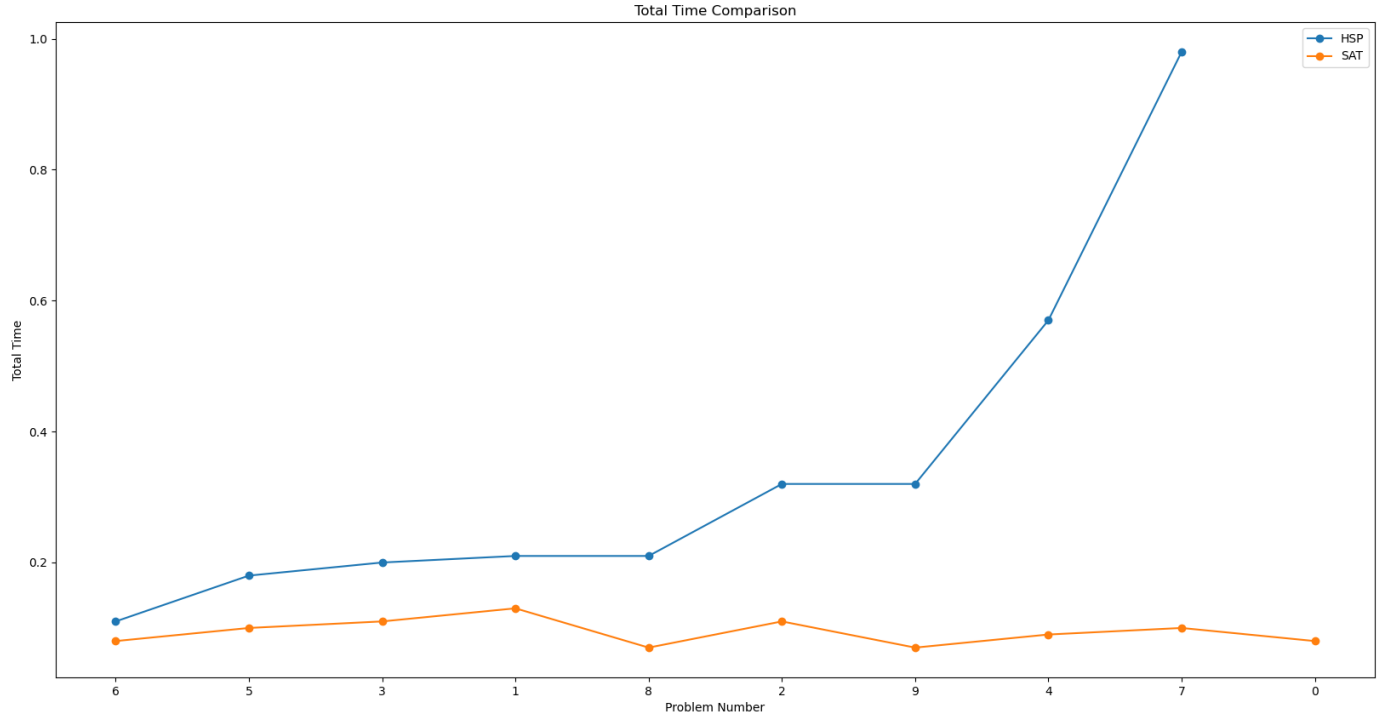

FIGURE 8 – Logistics : temps total