



**UNIVERSITÉ
DE GENÈVE**

University of Geneva
Department of Science
Bachelor's Degree in Computer Science

Année Académique 2023-2024

Projet Bachelor

Superviseur :
Prof. Pierre LEONE

Etudiant :
David Gabriel PAPA

27 juin 2024

Abstract

Ce travail a pour but de simuler des conditions de circulations en synthétisant ceux-ci sous forme de problèmes logiques. Nous utiliserons Z3 afin de résoudre nos situations représentant un rétrécissement de voie. Nous verrons à quelle vitesse ce problème est résolvable et que faut-il éviter pour perdre le moins de temps possible.

Mots-clés : logique, satisfaisabilité, problème, Z3, trafic, solver, optimiser, formule.

Table des matières

1	Introduction	4
1.1	Problèmes de satisfaisabilité booléenne	4
1.2	Opérations SAT/SMT	4
1.2.1	Variables	4
1.2.2	Opérateurs logiques	5
1.2.3	Conjonction	5
1.2.4	Disjonction	5
1.2.5	Théorie de l'égalité	5
1.2.6	Arithmétique linéaire	6
1.3	Solver Z3	6
1.3.1	Description	6
1.3.2	Utilisation Python	7
1.3.3	Applications avancées de Z3	7
2	Problème	9
2.1	Description	9
2.2	Variables	9
2.2.1	Nombres de voitures	9
2.2.2	Temps	9
2.2.3	Vitesse	10
2.2.4	Position	10
2.2.5	Distance de freinage	10
2.2.6	Distance	11
2.3	Conditions initiales	11
2.4	Évolution des variables	12
2.4.1	Vitesse	12
2.4.2	Position	13
2.4.3	Distance de freinage	13
2.5	Conditions de simulation	14
2.6	Conditions finales	15
2.7	Récupération des résultats	16
2.8	Temps maximum	16
3	Analyse	18
3.1	Explication de code	18
3.1.1	Solutions multiples	18
3.1.2	Variations du nombre de véhicules	18
3.2	Évolution des véhicules	19
3.3	Vitesse quelconque vs vitesse croissante	20
3.4	Temps maximal	21
3.4.1	Disposition initiale	21
3.4.2	Évolution des positions	24

1 Introduction

1.1 Problèmes de satisfaisabilité booléenne

Les problèmes de satisfaisabilité, aussi appelé plus communément problèmes SAT, constituent un domaine fondamental de la théorie de la complexité et de l'informatique théorique. Ces problèmes abordent une question centrale : étant donné une formule logique, existe-t-il une assignation des variables qui rende cette formule vraie ?

Ainsi, nous cherchons à comprendre les mécanismes de satisfaisabilité en examinant des formulations logiques propositionnelles où chaque problème peut être vu comme un casse-tête logique. Nous tentons de découvrir si ce casse-tête peut être résolu ou non, c'est-à-dire si une configuration spécifique permet de satisfaire toutes les contraintes fixées.

Admettons que nous souhaitons acheter une maison. Nous aurons plusieurs critères à remplir pour que nous considérions cette maison comme intéressante :

- Elle coûte moins que un million.
- Elle est au bord du lac ou n'a pas de voisins.
- Elle est rouge.

Nous voulons que chaque restriction soit ajoutée aux autres conditions. Il est facile d'ajouter des conditions mais plus celle-ci sont complexes et nombreuses, moins il y a de chance de trouver une solution.

Un point important de notre problème SAT est qu'il s'agit généralement d'une formule ou d'une équation à satisfaire. Pour l'appliquer à des situations de la vie courante il faut donc commencer par mathématiser notre cas. Décrire une disposition et des règles sous forme d'équation requiert beaucoup de temps et de précision car une mauvaise reformulation mène automatiquement à une erreur dans les résultats. La retranscription de notre problème sous forme mathématique sera décrite plus tard [2].

1.2 Opérations SAT/SMT

Les opérations SAT reposent sur la logique propositionnelle, où des formules sont composées de variables, d'opérateurs logiques et de formes différentes. Nous détaillons ces parties afin de mieux comprendre leur utilisation future [5] :

1.2.1 Variables

Les variables en logique propositionnelle représentent des propositions qui peuvent être soit vraies soit fausses. Leur état est à déterminer pour trouver une solution satisfaisante.

1.2.2 Opérateurs logiques

Il existe cinq opérateurs logiques :

- *NOT* ou \neg : est un *NON* logique, pour une opération $\neg x$ vraie, on doit avoir x faux.
- *AND* ou \wedge : est un *ET* logique, pour une opération $x \wedge y$ vraie, on doit avoir x ET y qui sont vrai.
- *OR* ou \vee : est un *OU* logique, pour une opération $x \vee y$ vraie, on doit avoir x OU y qui est vrai. (les deux peuvent être vrai)
- *IMPLIES* ou \rightarrow : est une *IMPLICATION* logique, pour une opération $x \rightarrow y$ vraie, on doit avoir x faux ou y vrai.
- *EQUIVALENCE* ou \leftrightarrow : est une *EQUIVALENCE* logique, pour une opération $x \leftrightarrow y$ vraie, on doit avoir x et y qui sont identique, donc soit les deux sont vrais, soit les deux sont faux.

1.2.3 Conjonction

Une conjonction est une proposition contenant comme dernière opération à effectuer des *AND* [1] : $x \wedge y \wedge z$ est donc une conjonction. Il est important de noter que lors de l'ajout d'éléments pour résoudre notre système, nous effectuons une conjonction mais son fonctionnement sera décrit plus bas [1.3.2].

1.2.4 Disjonction

Une disjonction est une proposition contenant comme dernière opération à effectuer des *OR* [1] : $x \vee y \vee z$ est donc une disjonction. Nous nous en servons pour permettre la variations de différentes manière notamment pour la vitesse [2.4.1].

1.2.5 Théorie de l'égalité

La Théorie de l'égalité concerne les propriétés et les manipulations des égalités entre les termes, elle traite des équations entre variables et constantes et de la capacité à déterminer si ces équations peuvent être satisfaites simultanément.

Une formule peut inclure des équations de la forme $x = y$, où x et y sont des atomes. Les atomes peuvent être complexes et inclure des combinaisons d'opérateurs logiques [1.2.2], de variables et de constantes.

Par exemple, pour la formule suivante [1] :

$$(y = z \vee (\neg(x = z) \wedge x = 2))$$

Une solution satisfaisant cette égalité serait : $x \mapsto 2, y \mapsto 2, z \mapsto 0$. Lorsque les solveurs SAT sont étendus pour inclure la Théorie de l'égalité, on parle souvent de SMT (Satisfiability Modulo Theories). Les solveurs SMT combinent des techniques de SAT avec des

théories comme l'égalité pour résoudre des problèmes plus complexes qui incluent non seulement des variables booléennes mais aussi des contraintes arithmétiques et d'égalité.

1.2.6 Arithmétique linéaire

L'arithmétique linéaire joue un rôle essentiel dans de nombreux domaines comme l'optimisation, la programmation linéaire et la théorie des nombres. Dans le contexte de la logique et des solveurs de satisfiabilité (SAT), elle est souvent utilisée dans la Satisfiabilité Modulo Theories (SMT), où elle est combinée avec d'autres théories pour résoudre des problèmes complexes.

Dans l'arithmétique linéaire, nous pouvons trouver des opérateurs de calculs : $+$, $-$, $*$, $/$ mais aussi des opérateurs d'inégalités : $<$, \leq , $>$, \geq . Combiner cela à des constantes et des variables ainsi que les concepts décrits précédemment permet de formuler un grand nombre d'opérations plus complexes.

Le solveur Z3 que nous utiliserons est justement capable d'utiliser toutes ces méthodes afin d'optimiser ou simplement trouver des solutions.

1.3 Solver Z3

1.3.1 Description

Le solveur Z3 est un solveur SMT (Satisfiability Modulo Theories) qui est un problème de décision pour les formules logiques par rapport aux combinaisons de théories de fond telles que l'arithmétique, les vecteurs binaires, les tableaux et les fonctions non interprétées, développé par Microsoft Research. Z3 est capable de traiter des problèmes de satisfiabilité logique combinés avec des théories arithmétiques et algébriques. Il est largement utilisé pour la vérification formelle, l'optimisation, la vérification de logiciels et matériels, et d'autres applications de l'informatique théorique et pratique [3]. En Python, Z3 est accessible via une bibliothèque qui permet de construire, manipuler et résoudre des formules logiques.

Z3 est utilisé dans beaucoup de domaines grâce à sa simplicité, son efficacité et sa versatilité. Étant un solveur SMT, il est couramment sollicité pour des tâches d'analyse de contraintes, en résolvant des problèmes de satisfaction de contraintes dans divers contextes, de vérification de logiciels ou, comme nous l'utiliserons dans ce travail, en optimisation pour résoudre des problèmes impliquant un perfectionnement dans leur fonctionnement.

La différence notable de Z3 avec d'autres solveurs se situe dans ses spécificités, il est capable de supporter une multitude de théories tel que [3] :

- L'arithmétique avec des nombres réels.
- Les tableaux et les tuples qui peuvent servir pour des données et que nous utiliserons comme expliqué dans la partie sur nos variables [2.2].
- Les bit-vecteurs qui sont des opérations de bas niveau et servent généralement à de la

vérification de matériel.

- Les types de données algébriques comme les arbres binaires.
- Les séquences et les chaînes de caractères.

1.3.2 Utilisation Python

La librairie *z3-solver* disponible sur Python est celle que nous utiliserons afin de résoudre notre problème. Son utilisation est très intuitive et le solveur a été conçu pour une interactivité simplifiée et donc facile à manipuler.

Tout d'abord, nous pouvons définir plusieurs types de variables dans notre programme comme des entiers, des réels ou même des booléens :

```
from z3 import *  
  
x = Int('x')  
y = Real('y')  
z = Bool('z')
```

Il est aussi possible d'inclure les opérateurs logiques qui appliquent leur opération entre toutes les parties de leur contenu :

```
formule1 = And(x > 0, Or(y < 5.5, Not(z)))  
  
formule2 = Or(z, x < 5)
```

Enfin, nous créons un solveur afin de résoudre notre proposition en y ajoutant nos formules. Il est important de noter que lors de l'ajout de plusieurs formules, une conjonction de celles-ci est effectuée dans notre solveur. Ainsi, nous visualisons la solution proposée grâce au *model* de celui-ci :

```
solver = Solver()  
solver.add(formule1)  
solver.add(formule2)  
if solver.check() == sat :  
    model = solver.model()  
    print(model)  
else :  
    print("La formule n'est pas satisfaisable.")
```

Il nous suffit de ces quelques opérations pour réaliser notre problème sur le trafic routier qui sera détaillé dans la partie suivante [2]. Il est aussi possible d'utiliser un optimiseur dans Z3 où nous pouvons lui demander certains objectifs spécifiques. Nous le mentionnons ici car il y a une partie de notre code qui en fait l'utilisation, cependant, il s'agit simplement d'un essai qui s'est avéré non pertinent pour notre tâche à réaliser.

1.3.3 Applications avancées de Z3

Z3, en tant que solveur SMT polyvalent, trouve des applications au-delà de l'utilisation que nous en faisons, notamment dans des domaines complexes où il joue un rôle crucial dans l'analyse et la vérification formelle. Voici quelques-unes de ses utilisations avancées :

Z3 est intégré dans les outils de vérification formelle pour analyser et garantir la conformité du code source avec des spécifications précises. Il permet de générer des invariants de programme pour vérifier la sécurité et la fiabilité du code, analyser des chemins d'exécution pour identifier des erreurs potentielles, comme des violations d'accès mémoire. Mais aussi, d'automatiser la détection de bugs en modélisant des conditions d'erreur et en résolvant les contraintes associées. On peut retrouver l'utilisation de Z3 dans les outils qui traduisent le code en représentations SMT. [4]

Il permet l'optimisation de plusieurs caractéristiques à l'aide des capacités d'optimisation permettant de trouver des solutions optimales pour plusieurs objectifs simultanément [3]. Cette fonctionnalité permet notamment dans un cas de consommation énergétique, de minimiser l'énergie tout en maximisant la couverture.

Z3 peut être utilisé pour la synthèse de programmes, où il aide à générer automatiquement du code ou des configurations qui satisfont un ensemble de contraintes définies par l'utilisateur. Son application s'étend de la correction automatique de code défectueux à de la génération de tests qui couvrent des cas particuliers ou des scénarios extrêmes.

Il est aussi possible de vérifier des propriétés dans les systèmes matériels pour valider des circuits et des systèmes numériques complexes en vérifiant les propriétés de timing pour garantir que les signaux respectent des contraintes de temps précises. Z3 peut vérifier les propriétés logiques aussi pour s'assurer que les systèmes matériels se comportent conformément aux spécifications logiques.

Z3 est souvent utilisé dans l'analyse symbolique et le test concolic [6] pour explorer l'espace des états d'un programme et générer des tests qui couvrent différents chemins d'exécution. Il aide à explorer des scénarios rares qui peuvent ne pas être facilement testés par des méthodes conventionnelles et à automatiser les tests unitaires pour une couverture de code plus exhaustive.

2 Problème

2.1 Description

Pour notre problème, nous nous pencherons sur le rétrécissement d'une route, en passant notamment de deux à une voie, en recherchant le temps minimum nécessaire afin d'obtenir un état sûr et nécessaire pour les automobilistes. Nous pouvons exprimer plusieurs conditions à satisfaire pour assurer la sécurité ainsi que des conditions initiales qui font référence à un rétrécissement de la route.

Pour exécuter notre problème, il nous faut d'abord définir plusieurs variables afin de poser notre situation sous forme de problème SAT [1.2].

2.2 Variables

Tout ce qui sera expliqué dans cette partie et les suivantes se trouve dans le notebook *David_Papa_projet_bachelor.ipynb* [2] qui contient l'entièreté du code pour ce travail.

2.2.1 Nombres de voitures

Afin de voir l'évolution de nos solutions, on peut varier le nombre de véhicules dans notre schéma initial. On définit le nombre de voiture n :

$$2 \leq n \leq 10$$

```
for k in range(2, 11) :  
    # ... (autre code)  
    n = k
```

Nous nous arrêtons à 10 voitures, car au delà le programme prend énormément de temps à s'exécuter.

2.2.2 Temps

Nous définissons une valeur pour le temps t qui sera la valeur que nous essaierons de réduire au maximum. Nous aurons donc un programme séquentiel avec des résultats pour chaque temps. La notion du temps sera développer plus loin [2.7] mais est instanciée ainsi :

```
for l in range(2, 300) :  
    # ...  
    t = l  
    # ...  
    break
```

Le temps maximum de notre problème sera défini par la variable t_{max} . Ce temps maximum varie en fonction du nombre de véhicules.

2.2.3 Vitesse

Chaque véhicule possède une vitesse en tout temps quel que soit l'état de la route. Pour représenter cela, nous l'enregistrerons dans un double tableau de la forme :

$$s[numero\ du\ vehicule][temps\ instantane]$$

`vmax = 6`

```
s = [[Int('s_{}_{}'.format(i, j)) for j in range(t)] for i in range(n)]
```

Le premier argument représente donc le véhicule que nous souhaitons regarder et est compris entre 0 et n ($0 \leq numero\ de\ vehicule < n$) et le second correspond au temps auquel nous aimerions connaître la vitesse, donc compris entre 0 et t_{max} ($0 \leq temps\ instantane < t_{max}$).

Afin de rester dans des valeurs raisonnables, nous définirons une vitesse maximale $v_{max} = 6$ qui limitera la vitesse des véhicules.

2.2.4 Position

Chaque voiture a une position fixe pour chaque temps. Nous allons donc enregistrer ces données de la même manière que pour la vitesse [2.2.3], dans un double tableau de la forme :

$$pos[numero\ du\ vehicule][temps\ instantane]$$

```
pos = [[Int('pos_{}_{}'.format(i, j)) for j in range(t)] for i in range(n)]
```

Le premier argument représente donc le véhicule que nous souhaitons regarder et est compris entre 0 et n ($0 \leq numero\ de\ vehicule < n$) et le second correspond au temps auquel nous aimerions connaître la position, donc compris entre 0 et t_{max} ($0 \leq temps\ instantane < t_{max}$).

2.2.5 Distance de freinage

Nous définirons la distance de freinage de chaque véhicule afin de pouvoir l'utiliser dans nos calcul par la suite :

$$df[numero\ du\ vehicule][temps\ instantane]$$

```
df = [[Int('df_{}_{}'.format(i, j)) for j in range(t)] for i in range(n)]
```

Ce tableau correspond donc à la distance de freinage d'une voiture à un certain temps et est défini comme :

$$df[i][j] = \frac{s[i][j] * (s[i][j] + 1)}{2}$$

```
for j in range(1, t) :
    for i in range(n) :
        solver.add(df[i][0] == (s[i][0] * (s[i][0] + 1)) / 2)
        solver.add(df[i][j] == (s[i][j] * (s[i][j] + 1)) / 2)
```

Ainsi en connaissant la vitesse d'une voiture i à un temps j , nous pouvons définir sa distance de freinage df .

2.2.6 Distance

Pour facilité l'utilisation du programme, nous définissons aussi un tableau triple qui indiquera la distance entre une voiture et une autre à un temps précis :

$$d[\text{voiture } i][\text{voiture } j][\text{temps instantane}]$$

```
d = [[[Int('d_{}_{_}') .format(i, j, k)) for k in range(t)] for j in range(n)] for i in
    ↪ range(n)]
```

La valeur correspondra à l'écart strict entre deux voitures qui part du premier véhicule i au deuxième j au temps *temps instantane*. Nous verrons plus loin [2.5] que nous ne calculons pas tous les écarts entre toutes les voitures mais simplement avec le véhicule qui le précède (j est la voiture devant i).

2.3 Conditions initiales

Dans cette partie, nous verrons comment sont fixés les conditions de bases pour nos variables afin de simuler un rétrécissement de la route. Au temps $t = 0$ pour $n = 3$ voitures nous pourrions avoir :




Pos[0]	Pos[1]	Pos[2]	Pos[3]	Pos[4]	Pos[5]	Pos[6]	Pos[7]	Pos[8]	Pos[9]
									
$s_0=4$	\longleftrightarrow		$s_1=5$	\longleftrightarrow				$s_2=5$	
$df[0][0]=10$	$d[0][1][0]=2$		$df[1][0]=15$	$d[1][2][0]=4$				$df[2][0]=15$	

FIGURE 1 – *Disposition initiale possible pour trois voitures*

Pour conserver des valeurs logiques, nous commencerons par définir la position initiale de tous les véhicules comme étant plus grand ou égal à 0 :

$$pos[i][0] \geq 0$$

$$\forall i : 0 \leq i < n$$

Et avec la spécificité que la voiture la plus en retrait commence à la position 0 tel que :

$$pos[0][0] = 0$$

```
solver.add(pos[0][0] == 0)
for i in range(n) :
    solver.add(pos[i][0] >= 0)
```

La vitesse des voitures sera quant à elle bornée entre 0 et la vitesse maximale fixée :

$$0 \leq s[i][0] < v_{max}$$

$$\forall i : 0 \leq i < n$$

```
for i in range(n) :
    solver.add(s[i][0] >= 0, s[i][0] <= vmax - 1)
```

Nous interdissons les véhicules d'avoir la vitesse maximum afin d'obtenir des résultats plus pertinents et une évolution des vitesses pour toutes les voitures engagées.

Cette dernière valeur nous permet de calculer la distance de freinage de chaque véhicule comme défini au point [2.2.5] :

$$df[i][0] = \frac{s[i][0] * (s[i][0] + 1)}{2}$$

$$\forall i : 0 \leq i < n$$

```
for i in range(n) :
    solver.add(df[i][0] == (s[i][0] * (s[i][0] + 1)) / 2)
```

2.4 Évolution des variables

2.4.1 Vitesse

Pour la vitesse, nous utiliserons deux manières de la modifier ce qui va nous donner des résultats surprenant à analyser.

Premièrement, le cas classique où le véhicule aura le droit d'accélérer, de freiner ou de maintenir une vitesse constante toujours en restant dans la tranche de vitesse autorisée :

$$0 \leq s[i][t] \leq v_{max}$$

$$\forall i : 0 \leq i < n, \forall t : 0 < t < t_{max}$$

$$s[i][t] = (s[i][t-1]) \vee (s[i][t-1] - 1) \vee (s[i][t-1] + 1)$$

$$\forall i : 0 \leq i < n, \forall t : 0 < t < t_{max}$$

```
for time in range(1, t) :
    for i in range(n) :
        #vitesse toujours bornée
        solver.add(s[i][time] >= 0, s[i][time] <= vmax)
        #modification de la vitesse (+/- 1 ou constante)
        solver.add( Or( s[i][time] == s[i][time - 1],
                        s[i][time] == s[i][time - 1] + 1,
                        s[i][time] == s[i][time - 1] - 1))
```

Le deuxième cas est très similaire au premier à une exception près, la vitesse n'a pas le droit de diminuer :

$$0 \leq s[i][t] \leq v_{max}$$

$$\forall i : 0 \leq i < n, \forall t : 0 < t < t_{max}$$

$$s[i][t] = (s[i][t-1]) \vee (s[i][t-1] + 1)$$

$$\forall i : 0 \leq i < n, \forall t : 0 < t < t_{max}$$

```
for time in range(1, t) :
    for i in range(n) :
        #vitesse toujours bornée
        solver.add(s[i][time] >= 0, s[i][time] <= vmax)
        #modification de la vitesse (+/- 1 ou constante)
        solver.add( Or( s[i][time] == s[i][time - 1],
                        s[i][time] == s[i][time - 1] + 1))
```

2.4.2 Position

La nouvelle position de chaque voiture sera simplement calculer en fonction de sa position précédente et de sa vitesse passée :

$$pos[i][t] = pos[i][t-1] + s[i][t-1]$$

$$\forall i : 0 \leq i < n, \forall t : 0 < t < t_{max}$$

```
for time in range(1, t) :
    for i in range(n) :
        solver.add(pos[i][time] == pos[i][time-1] + s[i][time-1])
```

2.4.3 Distance de freinage

Comme pour l'initialisation [2.2.5], la formule pour la distance de freinage restera identique, dépendant uniquement du moment auquel celui-ci est calculé :

$$df[i][t] = \frac{s[i][t] * (s[i][t] + 1)}{2}$$

$$\forall i : 0 \leq i < n, \forall t : 0 < t < t_{max}$$

```
for time in range(1, t) :
    for i in range(n) :
        solver.add(df[i][time] == (s[i][time] * (s[i][time] + 1)) / 2)
```

2.5 Conditions de simulation

Pour obtenir des résultats pertinents et compréhensibles, nous ajouterons des restrictions qui permettront de simuler correctement une route avec un surnombre de véhicules sur une distance trop petite comme nous pourrions voir après le rétrécissement d'une voie.

Pour simplifier nos calculs et avoir une lecture plus agréable du problème, nous ajouterons la condition :

$$pos[i][t] < pos[i + 1][t]$$

$$\forall i : 0 \leq i < n - 1, \forall t : 0 \leq t < t_{max}$$

```
for time in range(t) :
    for i in range(n - 1) :
        solver.add( pos[i][time] < pos[i + 1][time] )
```

Qui indique que la voiture $i + 1$ sera toujours devant la voiture i afin d'avoir les automobiles dans un ordre croissant (le véhicule 0 est derrière le 1, etc.).

Avec cette contrainte, il est maintenant plus facile de calculer la distance [2.2.6] entre deux véhicules. Le calcul se fera uniquement entre deux entités successives (donc entre i et $i + 1$) et indiquera le nombre de cases les séparant. Une distance de 0 ne veut donc pas dire que les voitures sont au même endroit, mais simplement collées l'une derrière l'autre.

$$d[i][i + 1][t] = pos[i + 1][t] - pos[i][t] - 1$$

$$\forall i : 0 \leq i < n - 1, \forall t : 0 \leq t < t_{max}$$

```
for time in range(t) :
    for i in range(n - 1) :
        solver.add( d[i][i + 1][time] == pos[i + 1][time] - pos[i][time] - 1 )
```

Cette définition de la distance entre des véhicules nous sert à définir la formule pour une distance de sécurité. Il s'agit d'une condition à satisfaire en tout temps tel que : la somme entre la distance entre deux voitures et la distance de freinage de la voiture de devant doit être supérieure ou égale à la distance de freinage de la voiture de derrière pour éviter tout accident :

$$d[i][i + 1][t] + df[i + 1][t] \geq df[i][t]$$

$$\forall i : 0 \leq i < n - 1, \forall t : 0 \leq t < t_{max}$$

```
for time in range(t) :
    for i in range(n - 1) :
        solver.add( d[i][i + 1][time] + df[i + 1][time] >= df[i][time] )
```

2.6 Conditions finales

La simulation sera terminée lorsqu'elle atteindra un état stable satisfaisant les conditions fixées plus bas, marquant une reprise fluide du trafic routier. La valeur t_{max} indiquera le dernier temps de notre problème et donc l'état final.

Pour avoir un état sûr, nous ajouterons une condition plus restrictive sur les distances de sécurité [2.5] qui consisterait à obtenir une vitesse finale inférieure ou égale à la distance avec la voiture de devant :

$$s[i][t_{max}] \leq d[i][i+1][t_{max}]$$

$$\forall i : 0 \leq i < n-1$$

```
for i in range(n) :
    if (i != n - 1) :
        solver.add( s[i][t-1] <= d[i][i + 1][t-1] )
```

Pour ne pas avoir un état initial qui satisferait aussi cette condition, nous rajoutons sa condition inverse uniquement pour l'état initial :

$$s[i][0] > d[i][i+1][0]$$

$$\forall i : 0 \leq i < n-1$$

```
for i in range(n) :
    if (i != n - 1) :
        solver.add( s[i][0] > d[i][i + 1][0] )
```

Cette condition n'empêche pas de satisfaire l'équation pour les distances de sécurité [2.5] définies au départ.

Enfin, pour avoir un résultat optimal, nous nous assurons que toutes les voitures aient reprises leur vitesse maximum [2.2.3] :

$$s[i][t_{max}] = v_{max}$$

$$\forall i : 0 \leq i < n$$

```
for i in range(n) :
    solver.add( s[i][t-1] == vmax )
```

Nous pouvons visualiser ceci avec nos exemples [Figure 2] :

Où avec $t_{max} = 4$:

$$s[0][4] = s[1][4] = s[2][4] = v_{max} = 6$$

$$s[0][4] = 6 \leq d[0][1][4] = 6$$

$$s[1][4] = 6 \leq d[1][2][4] = 6$$




Pos[16]	Pos[17]	Pos[18]	...	Pos[23]	Pos[24]	Pos[25]	...	Pos[30]	Pos[31]	Pos[32]
										
	$s_0=6$	\longleftrightarrow			$s_1=6$	\longleftrightarrow			$s_2=6$	
	$df[0][4]=21$	$d[0][1][4] = 6$			$df[1][4]=21$	$d[1][2][4] = 6$			$df[2][4]=21$	

FIGURE 2 – *Disposition finale possible pour trois voitures*

2.7 Récupération des résultats

Comme nous avons plusieurs programmes avec des conditions différentes à satisfaire, nous utiliserons des dictionnaires pour enregistrer les valeurs de nos variables. Ces dernières nous serviront pour afficher nos résultats sur des graphiques côte à côte, afin de faciliter notre analyse.

Pour faire varier le nombre de véhicules présents dans notre problème, nous utilisons simplement une boucle `for` allant de 2 à 10 voitures comprises. Augmenter d'avantage le nombre de véhicules augmente de manière exponentielle le temps. Par conséquent, effectuer des opérations avec plus de véhicules serait trop long à exécuter pour obtenir des résultats, donc nous resterons sur cette fourchette.

Afin de trouver le temps le plus petit satisfaisant notre solution pour un nombre de véhicule précis, nous aurons également une boucle `for` qui, pour le premier résultat trouvé, ajoutera sa solution à notre dictionnaire et arrêtera de chercher d'autres solutions pour un nombre de véhicules identiques. Ceci signifie qu'en relançant le programme, nous pouvons obtenir des résultats légèrement différents mais remplissant toujours toutes nos conditions dans un laps de temps identique.

`dico_vitesse_pas_restreint[k]` contient un dictionnaire avec comme clé : le nombre de voitures et comme valeur : le temps minimum pour résoudre le problème.

`dico_vitesses_voitures[k][l][m]` contient les vitesses pour k voitures, pour la voiture numéro l au temps m .

Nous aurons un programme identique avec la condition de vitesse croissante afin de récupérer ces différents résultats et de les comparer sur des graphiques.

2.8 Temps maximum

Afin de pousser les recherches un peu plus loin, nous chercherons à savoir combien de temps mettrait notre problème à être résolu si la situation initiale était la moins optimale.

Pour cela, nous utiliserons les premiers tests effectués dans notre notebook [2] afin de réinjecter une solution dans notre Solver et d'exclure cette réponse pour obtenir des résultats différents (détails au point [3.1.1]). La subtilité ici est que seule la disposition

initiale nous intéresse et, donc il faut simplement ajouter nos variables au temps 0 pour chercher des valeurs différentes :

```
for d in model :  
    if (str(d).endswith("_0")) :  
        cp = model[d] == d()  
        b.append(cp)
```

Avec toutes nos conditions définies précédemment, nous possédons un nombre limité de conditions initiales dépendant surtout du nombre de véhicules au départ. Par conséquent, nous testons toutes les dispositions possibles avec le temps minimum pour les résoudre et, lorsqu'il n'y a plus de solutions, le plus grand résultat obtenu sera notre temps maximal pour k voitures. (voir [Figure 6](#))

Pour des raisons de temps et de manipulation, il est impossible de faire ces opérations sur des cas où plus de cinq véhicules seraient considérés au commencement. Le temps d'exécution du programme est beaucoup trop long.

3 Analyse

3.1 Explication de code

Dans notre Solver, nous utilisons plusieurs méthodes différentes afin d'avoir une multitude de données à analyser. Afin d'être sûr de leur comportement correct, nous avons tester des parties de code séparément qui sont présent dans notre fichier qui contient tout notre code : *David_Papa_projet_bachelor.ipynb* [2].

3.1.1 Solutions multiples

Lorsque notre Solver fonctionne, il va s'arrêter à la première solution qu'il trouve, cependant nous aurons besoin de voir plusieurs solutions dans un problème plus tard [3.4]. C'est pourquoi nous avons cette partie de code :

```
while solver.check() == sat :
    model = solver.model()
    print(model)
    blocks = []
    for d in model :
        cp = model[d] == d()
        blocks.append(cp)

    solver.add( Not ( And ( blocks )))
```

Qui est en fait une boucle tant que nous possédons des solutions pour notre solver. Nous itérons sur les variables et leur valeur dans notre `model` afin de toutes les connaître et d'ajouter une nouvelle condition à notre solver qui exclue la solution qu'il vient de trouver. Nous utilisons un bloc relié par des *AND* logique afin d'avoir une nouvelle solution même si celle-ci ne diffèrerait que pour une seule variable. Nous pouvons visualiser les différents résultats afin de s'assurer de son fonctionnement dans notre code.

3.1.2 Variations du nombre de véhicules

Pour que nous puissions trouver nos solutions les plus efficaces pour un nombre différent de voitures, nous avons appliqué une boucle `for` sur le nombre de véhicules et sur le temps nécessaire à avoir une solution :




```
for k in range(2, 11) :
    for l in range(2, 300) :
        #n voitures et t temps max
        n = k
        t = l
```




Nous avons donc notre nombre de voiture qui va augmenter et notre temps aussi. Comme nous souhaitons simplement connaître le meilleur temps pour chaque nombre de voitures, nous cassons la boucle sur le temps à l'aide d'un `break` dès qu'une solution est trouvée et ainsi passer directement à l'itération suivante de notre première boucle. Nous affichons nos résultats (en affichant la valeur de nos variables) pour s'assurer que tout est correct.




3.2 Évolution des véhicules




Pour mieux comprendre la méthode de résolution de notre problème, nous nous penchons sur le cas pratique pour une résolution en un temps $t_{max} = 4$ pour $n = 3$ véhicules. Il s'agit d'une des vingt-neuf solutions existantes que nous pouvons apercevoir sur la Figure 6.

Voici l'évolution de nos véhicules pour chaque itération de temps :

Pos[0]	Pos[1]	Pos[2]	Pos[3]	Pos[4]	Pos[5]	Pos[6]	Pos[7]	Pos[8]	Pos[9]
									
$s_0=4$	\longleftrightarrow		$s_1=5$	\longleftrightarrow				$s_2=5$	
$df[0][0]=10$	$d[0][1][0]=2$		$df[1][0]=15$	$d[1][2][0]=4$				$df[2][0]=15$	

Pos[3]	Pos[4]	Pos[5]	...	Pos[7]	Pos[8]	Pos[9]	...	Pos[12]	Pos[13]	Pos[14]
										
	$s_0=4$	\longleftrightarrow			$s_1=5$	\longleftrightarrow			$s_2=6$	
$df[0][1]=10$	$d[0][1][1]=3$			$df[1][1]=15$	$d[1][2][1]=4$			$df[2][1]=21$		

Pos[7]	Pos[8]	Pos[9]	...	Pos[12]	Pos[13]	Pos[14]	...	Pos[18]	Pos[19]	Pos[20]
										
	$s_0=4$	\longleftrightarrow			$s_1=5$	\longleftrightarrow			$s_2=6$	
$df[0][2]=10$	$d[0][1][2]=4$			$df[1][2]=15$	$d[1][2][2]=5$			$df[2][2]=21$		

Pos[11]	Pos[12]	Pos[13]	...	Pos[17]	Pos[18]	Pos[19]	...	Pos[24]	Pos[25]	Pos[26]
										
	$s_0=5$	\longleftrightarrow			$s_1=6$	\longleftrightarrow			$s_2=6$	
$df[0][3]=15$	$d[0][1][3]=5$			$df[1][3]=21$	$d[1][2][3]=6$			$df[2][3]=21$		




Pos[16]	Pos[17]	Pos[18]	...	Pos[23]	Pos[24]	Pos[25]	...	Pos[30]	Pos[31]	Pos[32]
										
	$s_0=6$	\longleftrightarrow			$s_1=6$	\longleftrightarrow			$s_2=6$	
$df[0][4]=21$	$d[0][1][4]=6$			$df[1][4]=21$	$d[1][2][4]=6$			$df[2][4]=21$		

FIGURE 3 – Résolution d'un problème à trois voitures en cinq temps

Nous remarquons assez rapidement un certain pattern qui est mis en évidence en rouge (pour indiquer les variations). Le véhicule le plus avancé et qui donc ne possède aucune contrainte vers l'avant accélère directement pour atteindre la vitesse maximale $v_{max} = 6$ [2.2.3]. La différence de vitesses entre deux voitures au temps t augmentera la distance qui les sépare de cette même valeur au temps $t + 1$.

$$s[1][2] = 5, s[2][2] = 6 \Rightarrow diff = s[2][2] - s[1][2] = 1$$

$$d[1][2][2] = 5, d[1][2][3] = d[1][2][2] + diff = 6$$

Ainsi, lorsque la distance est juste suffisante pour satisfaire la condition finale du point [2.6] qui vaut au minimum $v_{max} = 6$, c'est la voiture suivante qui va accélérer et atteindre la vitesse maximale. De ce fait, elle maintiendra son écart avec la voiture de devant et continuera d'augmenter la distance avec le véhicule de derrière même si celui-ci accélère aussi.

Du point de vue d'un véhicule, on va essayer de ne pas avoir un grand écart de vitesse avec le véhicule le précédent afin de pouvoir arranger l'espace entre deux voitures et stopper toutes variations entre elles rapidement.

3.3 Vitesse quelconque vs vitesse croissante

Durant le calcul du temps optimal pour obtenir une solution [2.1], nous avons pu constater que pour atteindre notre résultat, il fallait que certains véhicules ralentissent. C'est ainsi qu'est venu l'idée de comparer quelle serait les différences si nous obligeons la vitesse à croître [2.4.1]. En visualisant les différents graphiques, nous remarquons que lorsque la vitesse ne peut pas diminuer, un véhicule devant un autre aura toujours une vitesse supérieure ou égale à celui de derrière lui.



FIGURE 4 – Evolution des vitesses entre voitures pour une vitesse sans restrictions et une vitesse croissante

Dans ces graphes, nous pouvons facilement distinguer les différentes évolutions. Nous constatons un graphique beaucoup plus propre pour les vitesses croissantes contre des courbes assez chaotiques provenant de vitesses non contraintes.

Nous pourrions imaginer que la solution optimale pour résoudre notre problème serait celui apporté par les véhicules évoluant à vitesse croissante, cependant le résultat est tout autre en comparant leurs différences :

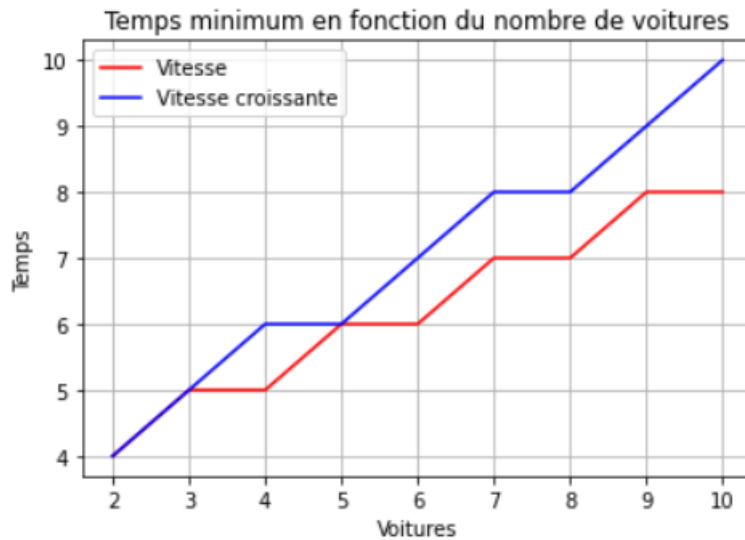


FIGURE 5 – Comparaison du temps minimum selon le nombre de véhicules pour une vitesse libre et une vitesse croissante

Nous distinguons une croissance régulière du temps minimum en fonction du nombre de véhicules pour nos deux variations de vitesses. Néanmoins, le temps minimum pour une vitesse croissante est bien moins efficace que l'autre manière.

Ces constatations pourraient expliquer ce que l'on remarque dans la vie courante avec le trafic en accordéon qui est un phénomène fréquent lors d'embouteillages et qui semblerait donc être la solution la plus rapide de retrouver un état stable et constant.

3.4 Temps maximal

Avec notre manière de construction du problème, il existe un nombre fini de positions initiales pour un nombre de voitures données. C'est pourquoi nous pouvons vérifier quelle disposition est sous-optimale pour notre problème.

3.4.1 Disposition initiale

À l'aide de la partie précédente [3.1.1], lorsque notre Solver trouvera une solution, on lui demandera de chercher une autre solution en interdisant la même disposition initiale :

```
if (len(blocks) > 0) :
    for elem in blocks :
        solver.add( Not ( And ( elem )))

while solver.check() == sat :
    model = solver.model()
    b = []
    for d in model :
```

```

if (str(d).endswith("_0")) : #changer que les positions initiales
    cp = model[d] == d()
    b.append(cp)

blocks.append(b)
solver.add( Not ( And ( b )))

```

Avec cette partie de code, nous avons `blocks` qui contient toutes les conditions initiales déjà essayées et que nous ne souhaitons pas réutiliser. Cette variable sera réinitialiser lorsque nous recommencerons le calcul pour un nombre de véhicules différents [3.1.2] contrairement à la variable `b` qui, elle, est la solution courante testée. Grâce à cela, nous aurons bien un essai de chaque disposition initiale en prenant un intervalle de temps assez grand. Avec notre variable `blocks`, nous disposons de toutes les conditions initiales possibles et donc en prenant le dernier élément de celui-ci, nous accéderons au dispositif initiale le moins optimal.

En comptant le nombre de positions initiales qui sont résolubles en un temps minimum, nous pouvons logiquement observer un nombre croissant de conditions initiales ce qui implique un plus long temps d'exécution du programme pour un plus grand nombre de voitures. C'est pourquoi dans cette partie nous limiterons nos calculs à cinq véhicules simultanés contrairement aux dix précédentes [2.2.1] :

Pour 2 voitures :	Pour 4 voitures :
6 solutions en 4 temps	3 solutions en 5 temps
15 solutions en 5 temps	221 solutions en 6 temps
10 solutions en 6 temps	845 solutions en 7 temps
3 solutions en 7 temps	401 solutions en 8 temps
1 solutions en 8 temps	35 solutions en 9 temps
	1 solutions en 10 temps
Pour 3 voitures :	
29 solutions en 5 temps	Pour 5 voitures :
128 solutions en 6 temps	66 solutions en 6 temps
68 solutions en 7 temps	1751 solutions en 7 temps
12 solutions en 8 temps	5212 solutions en 8 temps
1 solutions en 9 temps	1823 solutions en 9 temps
	132 solutions en 10 temps
	1 solutions en 11 temps

FIGURE 6 – Nombre de cas possibles en fonction du temps

Nous pouvons noter une croissance exponentielle du nombre de cas initiaux comme attendu mais aussi une solution qui semble toujours pire que les autres et qui est unique. En nous penchant sur cette solutions sous-optimale et en visualisant son résultat, nous pouvons constater des similitudes :

```
Pour 2 voitures :
Sens du mouvement ---->

case_0 | case_1 | case_2 | case_3 | case_4 | case_5 | case_6 | case_7 | case_8 |
v0, s1 | v1, s1 | -      | -      | -      | -      | -      | -      | -      |

Pour 3 voitures :
Sens du mouvement ---->

case_0 | case_1 | case_2 | case_3 | case_4 | case_5 | case_6 | case_7 | case_8 |
v0, s1 | v1, s1 | v2, s1 | -      | -      | -      | -      | -      | -      |

Pour 4 voitures :
Sens du mouvement ---->

case_0 | case_1 | case_2 | case_3 | case_4 | case_5 | case_6 | case_7 | case_8 |
v0, s1 | v1, s1 | v2, s1 | v3, s1 | -      | -      | -      | -      | -      |

Pour 5 voitures :
Sens du mouvement ---->

case_0 | case_1 | case_2 | case_3 | case_4 | case_5 | case_6 | case_7 | case_8 |
v0, s1 | v1, s1 | v2, s1 | v3, s1 | v4, s1 | -      | -      | -      | -      |
```

FIGURE 7 – *Pire disposition initiale venant du code [2]*

	Pos[0]	Pos[1]	Pos[2]	Pos[3]	Pos[4]	Pos[5]	Pos[6]	Pos[7]	Pos[8]
n=2									
	$s_0=1$	$s_1=1$							
	$df[0][0]=1$	$df[1][0]=1$							

	Pos[0]	Pos[1]	Pos[2]	Pos[3]	Pos[4]	Pos[5]	Pos[6]	Pos[7]	Pos[8]
n=3									
	$s_0=1$	$s_1=1$	$s_2=1$						
	$df[0][0]=1$	$df[1][0]=1$	$df[2][0]=1$						

	Pos[0]	Pos[1]	Pos[2]	Pos[3]	Pos[4]	Pos[5]	Pos[6]	Pos[7]	Pos[8]
n=4									
	$s_0=1$	$s_1=1$	$s_2=1$	$s_3=1$					
	$df[0][0]=1$	$df[1][0]=1$	$df[2][0]=1$	$df[3][0]=1$					

	Pos[0]	Pos[1]	Pos[2]	Pos[3]	Pos[4]	Pos[5]	Pos[6]	Pos[7]	Pos[8]
n=5									
	$s_0=1$	$s_1=1$	$s_2=1$	$s_3=1$	$s_4=1$				
	$df[0][0]=1$	$df[1][0]=1$	$df[2][0]=1$	$df[3][0]=1$	$df[4][0]=1$				

FIGURE 8 – *Illustration de la pire disposition initiale*

Sur l'image ci-dessus, nous remarquons que la pire disposition initiale est en fait très intuitive. En considérant notre route à une voie, la disposition initiale sous-optimale est identique quelle que soit notre nombre de véhicules. Toutes nos voitures ont la vitesse minimale possible de 1 et sont toute une derrière l'autre avec un espacement nul.

3.4.2 Évolution des positions

Étant donné que nos conditions initiales sous-optimales suivent un même pattern, il est pertinent de se demander s'il en est de même pour leur évolution jusqu'à l'obtention d'un état stable. C'est pourquoi nous commençons par regarder le modèle à deux et à trois véhicules :

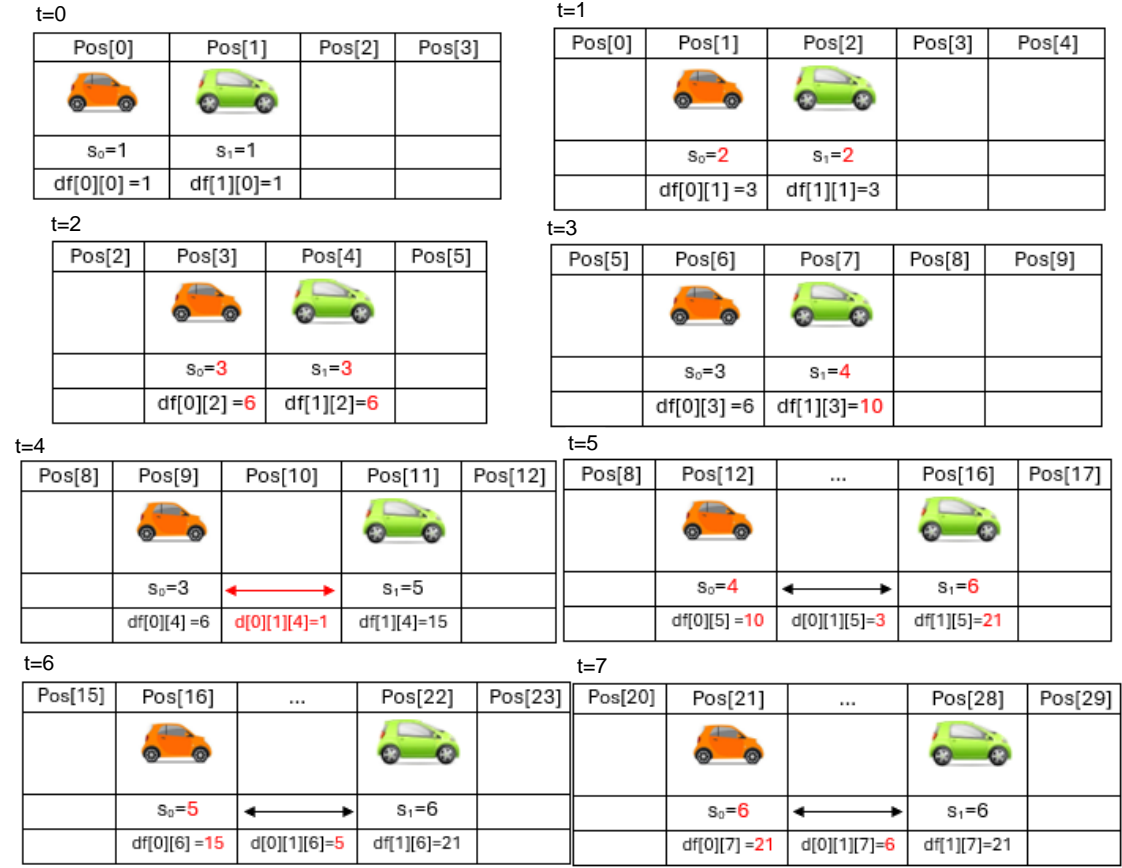


FIGURE 9 – Évolution de la pire disposition initiale pour 2 voitures



FIGURE 10 – Évolution de la pire disposition initiale pour 3 voitures

Avec ces illustrations plus visuelles, il est plus évident de repérer une gestion particulière de notre optimisation. On remarque que notre véhicule 0 peut même ralentir pour accélérer notre processus comme nous avons pu le voir au point [3.3]. Il faudra comparer l'évolution pour quatre véhicules pour faire des rapprochement :

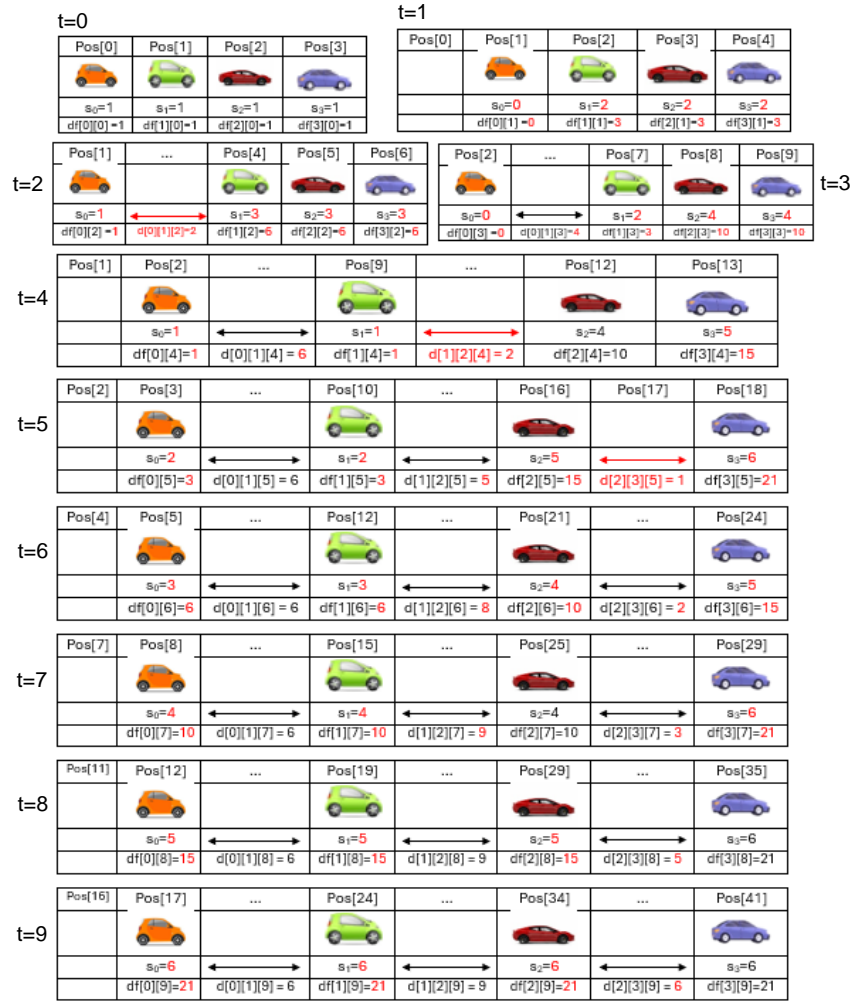


FIGURE 11 – Évolution de la pire disposition initiale pour 4 voitures

Avec ce dernier visuel, on remarque que l'on cherche d'abord à régler la distance qui séparer les deux voitures de dernière (voitures 0 et 1) avant de remonter notre colonne de véhicules. Une fois que la distance entre nos véhicules est correct, les voitures auront une vitesse semblable afin de maintenir cette distance, que leur vitesse soit élevée ou non. Nous avons donc une sorte de mise à niveau à partir de l'arrière de notre file de véhicules et non par l'avant comme on aurait pu le penser. Cette observation est donc contraire à ce que nous pourrions voir dans la vie courante et signifierait qu'un comportement "agressif" de conducteurs à l'arrière de la file aurait en fait tendance à prolonger le temps nécessaire pour retrouver un état sûr et fluide.

Références

- [1] Daniel KROENING et Ofer STRICHMAN. *Decision Procedures An Algorithmic Point of View*. Texts in Theoretical Computer Science. An EATCS Series. Springer Berlin, Heidelberg, 2016. DOI : <https://doi.org/10.1007/978-3-662-50497-0>.
- [2] *Lien du dépôt GIT contenant le code*. URL : <https://github.com/DavidGabrielPapa/Travail-Bachelor/tree/master>.
- [3] Lev Nachmanson NIKOLAJ BJØRNER Leonardo de Moura et Christoph WINTERSTEIGER. *Programming Z3*. URL : <https://theory.stanford.edu/~nikolaj/programmingz3.html>.
- [4] Gaurav PARTHASARATHY, Peter Müller THIBAUT DARDINIER Benjamin Bonneau et Alexander J. SUMMERS. *Towards Trustworthy Automated Program Verifiers : Formally Validating Translations into an Intermediate Verification Language*. URL : <https://pm.inf.ethz.ch/publications/ParthasarathyDardinierBonneauMuellerSummers24.pdf>.
- [5] WIKIPÉDIA. *Calcul des propositions*. URL : https://fr.wikipedia.org/wiki/Calcul_des_propositions.
- [6] WIKIPÉDIA. *Concolic testing*. URL : https://en.wikipedia.org/wiki/Concolic_testing.