**Algorithme A\* :**

Afin, d’appliquer cette méthode heuristique, nous devons savoir ce qu’on va utiliser comme attribut pour l’estimation, après réflexion, nous avons déterminé que le nombre de clauses satisfiables était notre choix pour guider la machine.

Il faut savoir que dans A\*, on commence toujours par un point de départ, et comme un état doit avoir ces variables définies, c.-à-d. pas de variables avec la valeur (-1), on génère donc d’abord une solution aléatoire qui va être notre point de départ, après, nous choisissons le nœud qui satisfait le plus grand nombre des clauses, et nous le développons jusqu’à aboutir à une solution positive.

Cette Méthode ressemble beaucoup plus à l’algorithme de recherche par glouton « Greedy algorithm » car chaque déplacement vers un nœud voisin a le même poids (1) et chaque nœud a exactement le nombre de déplacements vers celui-ci comme son poids, c.à.d. le prochain nœud à visiter a comme poids exactement N-nœuds-visités+1

On peut donc la considérer comme un cas particulier de la solution A\*, et on s’aperçoit donc que calculer F= G+H et comme une action redondante.

Nous devons certainement faire plus de recherche vis-à-vis notre solution qui suit le principe de l’algorithme glouton « Greedy algorithm » et donc à chaque étape nous obtenons un optimum local, et parfois on n’atteint pas un optimum global. Bien que rare, dans certain cas dans nos tests, l’algorithme passe beaucoup de temps dans un seul niveau (même distance). Mais se problème semble être résolu lorsque nous ne calculons pas la fonction G.

Fermé une fois exploréPicture 5

Pas encore exploréPicture 4

101 (2)

001 (3)

101 (2)

011 (2)

000 (2)

111 (1)

011 (2)

101 (2)

110 (0)

100 (1)

000 (2)

110 (0)

101 (2)

Est déjà fermé/ouvert et non créé Picture 6

Pas encore créé :Picture 8

La configuration de l’un des systèmes utilisé pour la prise de mesures de performances est la suivante:

* + intel core i5 4260u fréquence de base 1.4GHz. Lors de l’exécution (turbo boost) ~ 2.7GHz ( 2.3 en moyenne).
  + RAM 8 Go 1600 MHz DDR3
  + Système d’exploitation OSX 10.13.6 (macOS High Sierra) (a *UNIX based OS*)
  + Graphisme intel HD Graphics 5000.

La deuxième machine:

* + intel core i7 XXXXu @ 2.6GHz. turbo boost ~ 3.7GHz . a confirmer
  + RAM 8 Go 1600 MHz DDR3
  + Système d’exploitation Windows 10 version 1803
  + AMD Radeon R5 xxx . a confirmer.

La troisième machine:

* + AMD Ryzen 5 1600 @ 3.2GHz, boost 3.4GHz (tous les corps), 3.6GHz (un seul corps).
  + RAM 16 Go 2400 MHz DDR4
  + Système d’exploitation Windows 10 version 1803
  + Graphisme NViDIA GTX 1060 6go.

La carte graphique est bien plus utile si on utilise l’accélération graphique.

Dans un premier temps, nous avons implémenté des méthodes pour l’importation des fichier. Pour l’interface graphique, nous avons utilisé les librairies **javafx** et **jfoenix** pour l’affichage du chronomètre. pour la lecture des clauses de l’instance sélectionnée, nous avons conçu un -parser- avec une expression régulière afin de lire les clauses du fichiers CNF

Introduction

Dans le domaine de l’informatique et de la logique, le problème de satisfiabilité SAT, est un problème de décisionnel où l’on assigne à des variables des valeurs de vérité *{Vrai | Faux}* à un ensemble de clauses appelé instance. Une instance du problème SAT est un ensemble de clauses, qui est une conjonction de clauses ou chaque une d’elles est une disjonction de littéraux.

Le problème SAT fut le premier problème à être démontré NP-Complet, et représente une référence aux autre problèmes NP-Complet. (à developper / changer etc…)

**Structure de données :**

Dans un premier temps, nous allons représenter une instance sous forme matricielle.

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | 0 | 1 | -1 |
| -1 | 1 | 0 | 1 |
| 0 | 1 | -1 | 0 |
| 1 | -1 | 0 | 0 |

Chaque ligne représente une clause, et chaque colonne représente une variable.

Nous avons donc une matrice(n,m) de n clauses et m colonnes.

Tout d’abords, toutes les cases sont initialisées a « -1 ».

Chaque clause est représentée par un vecteur, chaque littéral présent est représentée par « 1 » respectivement « 0 » si sa valeur est « Vrai » respectivement « Faux » dans sa case correspondante (l’index de la colonne).

Soit X une instance, X = {x1, x2, x3, x4}, on a donc par exemple les clauses :

c1 = x1, !x2, x3

c2 = x2, !x3, x4

c3 = !x1, x2, !x4

c4 = x1, !x3, !x4

Avant de commencer l’exploration, il faut définir un ensemble de variables qui contient toutes les variables de notre problème.

V = {v1, v2, …, vn}

Partie 1

Dans cette première partie, nous nous intéressons à deux stratégies de recherche: La recherche aveugle et la recherche informée en utilisant des heuristiques.

**La stratégie de recherche aveugle:**

Dans cette partie, nous allons, dans un premier temps voir deux méthodes de recherche, la recherche en Profondeur d’abord, et la recherche en Largeur d’abord.

**Recherche en profondeur d’abord (DFS):**

Dans cette méthode, on commence par explorer un noeud choisi aléatoirement à partir de notre ensemble, on génère alors ses deux valeurs 1 et 0, et on ajoute le noeud à l’ensemble **Open** et on tire après un autre noeud tout en ajoutant l’état à l’ensemble **Open** jusqu’à arriver a une feuille ou on teste si la solution trouvé ne produit pas de contradiction, et on supprime ainsi cette branche de **Open**.

Sinon, on explore le noeud voisin tout en supprimant les branches non concluant à une solution ce qui permet la libération de la mémoire système.

de plus cette méthode s’avère assez efficace comparé à la méthode discutée ci dessous puisque on doit d’abord générer des valeurs à toutes les variables avant de passer à un autre voisinage sans pour autant explorer tous les noeuds possibles.

**Recherche en largeur d’abord (BFS):**

Cette méthode consiste a étendre tous les noeuds possibles d’un état donné, donc on explore d’abord tout son voisinage avant de passer au noeud suivant, on parle alors d’une exploration par niveau.

cette méthode s’annonce clairement peu efficace car avant d’aboutir a une solution, nous devons exploser tous les noeud niveau par niveau ce qui accroit sa complexité de façon exponentielle.

de plus, l’ensemble **Open** n’est jamais vidé que lorsqu’on atteint le niveau le plus bas (***n***) et donc très gourmand en terme de mémoire et qui met autant de stresse sur le processeur.

**Complexité temporelle:**

**Recherche en profondeur d’abord (DFS):**

après avoir atteint le noeud final, on vérifie chaque clause si elle on contredit pas la solution trouvé, si le premier littéral est different alors on vérifie avec le deuxième puis le troisième. on parcours donc au pire cas les n variables du vecteur.

on le fait avec toutes les clauses m.

ce qui nous donne une complexité de asymptotique *O(m.n)*

On génère les branches pour le parcours de toutes les n variables. donc n fois

pour les 2n solutions possibles.

*( n + n.(n-1) + n.(n-2) + … + n ) x 2 - car on génère les deux valeurs (correspond pas avec ci-dessous)*

Donc le pire cas est lorsque l’ensemble n’est pas satisfiable et on a une complexité de Cn = *m.n.2n+1*

asymptotiquement *O(n2.2n), ou bien O(n.2n+1) (à valider)*

Mais ce résultat n’est pas indicatif du temps expérimentales réels.

le meilleur cas est *O(n2)*

**Recherche en largeur d’abord (DFS):**

La vérification de la solution est identique à la première méthode.

On a donc une complexité *O(m.n)*

Mais avant de générer les branches pour le parcours de toutes les variables, on doit d’abord explorer tout le voisinage et donc ( n + n.(n-1) + n.(n-2) + … + n ) x 2

pour les 2n solutions possibles.

donc le pire cas est lorsque l’ensemble n’est pas satisfiable et on a une complexité de Cn = *m.n.2n+1*

asymptotiquement *O(n2.2n), ou bien O(n.2n+1) (à valider)*

Ici, le meilleur cas est lorsque la première feuille atteinte st une solution, ce qui donne une complexité de *O(n.2n)*

**Complexité spatiale:**

**Recherche en profondeur d’abord (DFS):**

Il faut savoir que le plus taxant sur la mémoire dans ce problème est l’ensemble **Open**, car après chaque exploration d’un noeud, on ajoute l’état actuelle à **Open** et n’est pas supprimé de ce dernier que lorsque on explore tous ses fils. Alors, comme ici nous avons un parcours en profondeur, on assure l’exploration de toute une branche jusqu’à atteindre une feuille pour vérifier une solution, et donc, toute la branche devient visitée en on a aucun problème à la supprimer de l’ensemble. se qui nous donne une complexité spatiale de N-niveaux.N-variables et donc *Tn = O(n2)*

**Recherche en largeur d’abord (DFS):**

Dans cette méthode, l’ensemble **Open** n’est jamais vidé que lorsque on atteint une feuille. i.e après chaque exploration d’un noeud, on ajoute l’état actuelle à **Open** et celui-ci n’est pas supprimé de ce dernier que lorsque l’on explore tous ses fils. Ainsi, comme ici nous avons un parcours en largeur, on ne supprime aucune branche jusqu’a avoir atteint le dernier niveau pour tous les voisins, et donc on a une complexité spatiale de 2N-niveaux.N-variables et donc *Tn = O(n.2n)*

**Temps d’exécution:**

**Recherche en profondeur d’abord (DFS):**

Environs 0.200 ~ 600 seconde selon la machine.

**Recherche en largeur d’abord (DFS):**

Varie entre 2.12 et 5.30 minutes selon la machine

Comme on est toujours sur une exécution mono thread, on ne peut donc avoir de résultat bien meilleur si on augmente les performances d’une machine, il faudrait donc penser au calcul parallèle. **( à implémenter )**

**Consommation mémoire :**

**DFS :** L’espace occupé est assez négligeable en comparaison avec le reste de l’application.

**BFS :** L’espace occupé est très important dépassant les 600mo

**Stratégie de recherche guidée :**

**Algorithme A\* :**

Afin, d’appliquer cette méthode heuristique, nous devons savoir ce qu’on va utiliser comme attribut pour l’estimation, après réflexion, nous avons déterminé que le nombre de clauses satisfiables était notre choix pour guider la machine.

Il faut savoir que dans A\*, on commence toujours par un point de départ, et comme un état doit avoir ces variables définies, c.-à-d. pas de variables avec la valeur (-1), on génère donc d’abord une solution aléatoire qui va être notre point de départ, après, nous choisissons le nœud qui satisfait le plus grand nombre des clauses, et nous le développons jusqu’à aboutir à une solution positive.

Cette Méthode ressemble beaucoup plus à l’algorithme de recherche par glouton « Greedy algorithm » car chaque déplacement vers un nœud voisin a le même poids (1) et chaque nœud a exactement le nombre de déplacements vers celui-ci comme son poids, c.à.d. le prochain nœud à visiter a comme poids exactement N-nœuds-visités+1

On peut donc la considérer comme un cas particulier de la solution A\*, et on s’aperçoit donc que calculer F= G+H et comme une action redondante.

Nous devons certainement faire plus de recherche vis-à-vis notre solution qui suit le principe de l’algorithme glouton « Greedy algorithm » et donc à chaque étape nous obtenons un optimum local, et parfois on n’atteint pas un optimum global. Bien que rare, dans certain cas dans nos tests, l’algorithme passe beaucoup de temps dans un seul niveau (même distance). Mais se problème semble être résolu lorsque nous ne calculons pas la fonction G.

Algorithme de la fonction H(n):

// Chaque fois que un voisin est visité on vérifie les clauses satisfaites avec la fonction suivante:

Pour chaque clause Faire

initialiser verifClauses (liste de booléen)

Pour chaque élément de noeud Faire

Si Clause[i] <> -1 Alors

Si Clause[i] =1 Alors

verifClause.ajouter(vrai)

Sinon verifClause.ajouter(faux)

Fin Si;

Sinon Si Noeud[j] = 1 Alors

verifClause.ajouter(faux)

Sinon verifClause.ajouter(vrai)

Fin Si;

Fin Si;

Fait;

Si(non contient(verifClause, vrai) Alors

compteur++;

Fait;

Retourner compteur;

PS: Cette solution à été implémenté en se basant sur la fonction solution ou la seule différence est à la place du compteur, nous avons un retour en **faux.**

Algorithme Choix Du meilleur resultat

Trier Open.stat.H ordre croissant

Courant := Open(0)

**Temps d’exécution :**

**A\*:**

2 millisecondes à 32 millisecondes.

**Avec 1000 instances :**

32.599 secondes – un seul thread

3.5 seconds – multi thread (CPU a 6 cores - 12 threads)

Voici une petite explication de certaines méthodes dans notre programme :

RechercheExhaustive:

est une classe qui contient est méthodes et fonctions essentielles pour nous permettre la résolution du problème.

Elle contient aussi les attributs nécessaires.

Attributs:

DataClause: les clauses récupérées depuis le fichier CNF.

NbrClauses, NbrVariables: Autodescriptives. dépend du fichier choisi.

Solution: une chaine de caractère servant à imprimer la solution trouvée.

Méthodes:

IsSolution():

Algorithme AStar;

Entrée : Clauses à évaluer

Sortie : solution si existe, nil sinon

Var :

Open : liste vide de noeuds;

Closed : liste vide de noeuds;

Current : noeudA\*;

State : noeudA\*;

Début

pour i de 0 à NbrVariables faire // generer du premier noeud

current.enflier(Random(entre 0 et 1)); // he is a random dude!

}

State.etat =current;

State.H = CompteurClausesSat(current);

State.G = 0;

Open.enfiler(State); // le premier noeud

Tant que !vide(Open) faire

si EstSolution(State.etat()) alors

retourner State.etat();

sinon

Closed.add(State); // I mean it is pretty obvious

Open.defiler(); // on supprime le premier le la liste

GenererFils(State, Open, Closed);

// les nouveux fils vont dans open

Trier(Open,CompareHeuristic);

State = Open.defiler();

Fsi;

Fait;

retourner Nil;

Fin;

Algorithme GenererFils;

Entrée :

Open : liste vide de noeuds;

Closed : liste vide de noeuds;

State : noeudA\*;

Sortie : Liste de nouveaux Listes noeuds fils

Var :

etat : liste vide de noeuds;

index, val : entier

Début:

Integer val = 0;

int index = 0;

pour chaque i de State.etat() faire

etat.enfiler(State.etat());

val = (!(i == 1)) ? 1 : 0; // inverser la valeur booléenne

etat[index++] = val;

New.etat = etat;

New.Heuristique = CompterClausesSat(etat);

New.G = State.G+1;

Si (!contient(Closed, New.etat()) Alors

Si (!contientEtat(Open, New.etat() Alors

Open.enfiler(New);

sinon New.Heuristique(State.getG());

Fsi;

Fsi;

etat.vider();

fait

Fin;

est une fonction qui vérifie si la solution généré est une solution valide

i.e: qui ne produit pas de contradictions avec les clauses

Isleaf():

est une fonction qui vérifie si état actuel est une feuille donc toutes les variables ont étaies générées

getSons():

est une fonction choisi un littéral de l'ensemble de noeuds non explorées et qui génère ses deux fils (les deux valeurs possibles).