UE Intelligence Artificielle :

Le problème des seaux

Serigne Falilou BOP & Melvin CERBA

Sommaire

Manuel d'utilisation du programme

1) Lancer le programme

2) Entrer dans le champ prévu à cet effet le chemin depuis la racine d'un fichier représentant une instance du problème des seaux, ou d'un dossier contenant uniquement de tels fichiers.

3) Cliquer sur « résoudre ».

4) Après quelques secondes, les résultats apparaissent dans le tableau. Si vous avez analysé un fichier seul, une solution apparait en haut de la page. Vous pouvez alors recommencer.

Implémentation des algorithmes de parcours

Comme recommandé, nous avons implémenté un unique algorithme, puis nous avons greffé les différents parcours dessus en faisant varier la structure de « Ouverts ». Pour cela, nous avons choisi d'utiliser la structure de liste chainée en java :

Dans le cas du parcours en **profondeur**, on utilise la liste chainée comme *une pile* : on explore toujours le dernier élément de la liste, et on ajoute ses fils à la fin de la liste chainée.

Dans le cas du parcours en **largeur**, on utilise la liste chainée comme *une file* : on explore toujours le premier élément de la liste et on ajoute ses fils à la fin de la liste chainée.

Pour le parcours **« meilleur d'abord »**, on utilise la liste chainée comme *une liste triée* : on explore le premier élément et on insère chacun de ses fils dans « Ouverts » selon l'ordre croissant de des valeurs de l'heuristique des éléments. Ainsi la liste est toujours triée, et le premier élément est celui qui a l'heuristique de plus petite valeur.

Heuristiques proposées

Pour un problème à n seaux, soient :
- $C = [c1, c2, \dots, cn]$ avec ci la capacité maximale du seau i.
- $F = [f1, f2, \dots, fn]$ avec fi le contenu du seau i à l'état final.
- $E = [e1, e2, \dots, en]$ avec ei le contenu du seau i dans l'état évalué.
- *nbrR*, *nbrV* et *nbrT* le nombre de fois qu'on a utilisé respectivement la règle Remplir, la règle Vider et la règle Transvaser pour parvenir à l'état évalué.

Nous avons implémenté trois heuristiques :

L'heuristique **« basique »** utilisée en TD : $|\ somme(E)-somme(F)\ |$
Il s'agit simplement d'étudier la différence entre le contenu total des seaux à un instant donné et le contenu total des seaux à l'état finale

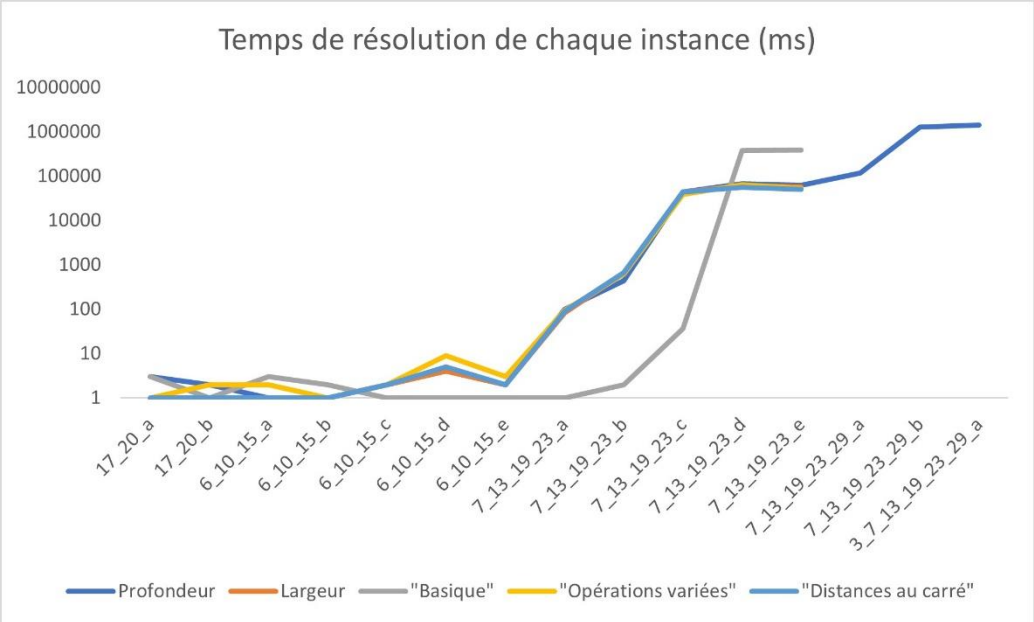
L'heuristique **« opérations variées »** : *| nbrR+nbrV-2*nbrT |*
L'idée est ici qu'on n'arrive à rien sans transvaser fréquemment le contenu des seaux. Cette heuristique privilégie donc un certain équilibre entre les règles de production.

L'heuristique **« distances au carré » :** *| somme(E)²-somme(F)² |*
Notre volonté avec cette heuristique était d'améliorer l'heuristique basique qui semblait donner de bons résultats. En élevant les paramètres au carré, nous voulions limiter les cas d'égalité entre deux états. (Par exemple *|2-4|* **=** *|6-4|*, mais *|2²-4²|* = *|4-16|* = *12* **=/=** *20* = *|36-16|* = *|6²-4²|*)

Analyse des performances

Voici une analyse du temps d'exécution des différents parcours sur chaque instance du problème des seaux, ainsi qu'un tableau récapitulant nos résultats quant à la solvabilité de chaque problème.

Solvabilité	INSTANCE
solvable	td
solvable	17_20_a
impossible	17_20_b
solvable	6_10_15_a
solvable	6_10_15_b
solvable	6_10_15_c
solvable	6_10_15_d
solvable	6_10_15_e
solvable	7_13_19_23_a
solvable	7_13_19_23_b
solvable	7_13_19_23_c
impossible	7_13_19_23_d
impossible	7_13_19_23_e
solvable	7_13_19_23_29_a
solvable	7_13_19_23_29_b
solvable	3_7_13_19_23_29_a



On remarque que les parcours en profondeur et les parcours en largeur ont sensiblement les mêmes performances et que, mise à part celle présentée en TD (« Basique » sur le graphe), nos heuristiques ne fournissent pas de meilleurs résultats.

On constate sans surprise que le temps d'exécution est fonction exponentielle du nombre de seaux compris dans l'instance évaluée (l'échelle du graphe ci-dessus est logarithmique), mais également que la complexité de l'état final est un facteur important. Par exemple, le problème « 7_13_19_23_e » est presque 1000 fois plus coûteux à résoudre que « 7_13_19_23_a », alors qu'ils ont le même nombre de seaux. On imputera cette différence aux fait que l'état final du premier problème ([5 7 17 21]) impose une contrainte à chaque seau, tandis que celle du second ([0 0 0 21]) n'en impose qu'à un seul. En réalité, qu'un seau soit vide est également une contrainte. Mais contrainte bien moins difficile à satisfaire qu'une autre, puisqu'elle est satisfiable par l'application d'une seule règle (Vider), et ce quelle que soit la quantité d'eau contenue dans le seau.

Nous n'avons malheureusement pas pu résoudre toutes les instances. Notre programme est tel que lorsqu'on le lance sur un dossier, on ne dispose des résultats qu'une fois tous les fichiers traités. Nous avons naïvement lancé l'analyse sur le dossier « ressources » contenant toutes les instances, et après plus de 24h, elle n'avait traité que deux instances à 2 seaux et deux à 6 seaux. Nous avons donc recommencé en segmentant le dossier « ressources » de sorte à analyser en parallèle, sur trois instances différentes du programme, les problèmes à moins de 5 seaux, les problèmes à 5 seaux et les problèmes à 6 seaux. Nous avons là aussi perdu beaucoup de temps, car après plus de 12h, les problèmes à 5 et 6 seaux n'avaient pas tous été résolus (nous n'avions donc aucun résultat pour ces problèmes, l'algorithme n'étant pas terminé). L'échéance de la soutenance approchant, nous avons décidé d'interrompre le programme et de recommencer, en analysant cette fois chaque problème séparément et en n'utilisant qu'un parcours plutôt que les sept implémentés. Voici donc pourquoi nous n'avons pas tous les résultats, et pourquoi le graphe n'affiche que le temps d'exécution du parcours en profondeur pour les problèmes à 5 et 6 seaux résolus.