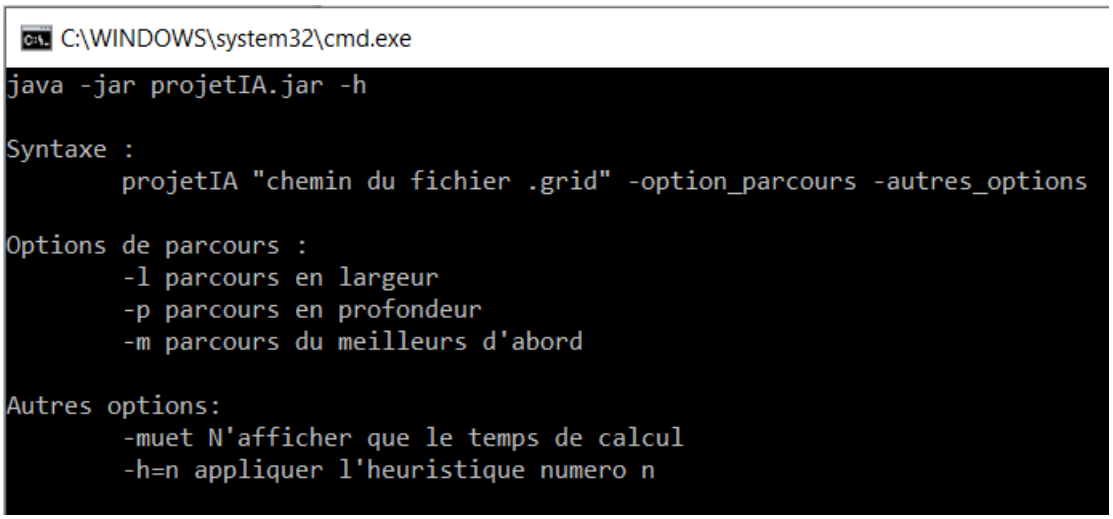
BARRY Jean-Noël

RAPPORT PROJET IA

1 - Utilisation et syntaxe

Le projet ayant été programmé en Java (version 8), il est nécessaire d'avoir une version de java installé sur la machine de test.

Le projet est fournit en version compilée (projetIA.jar), on peut obtenir la syntaxe avec l'option **-h** :



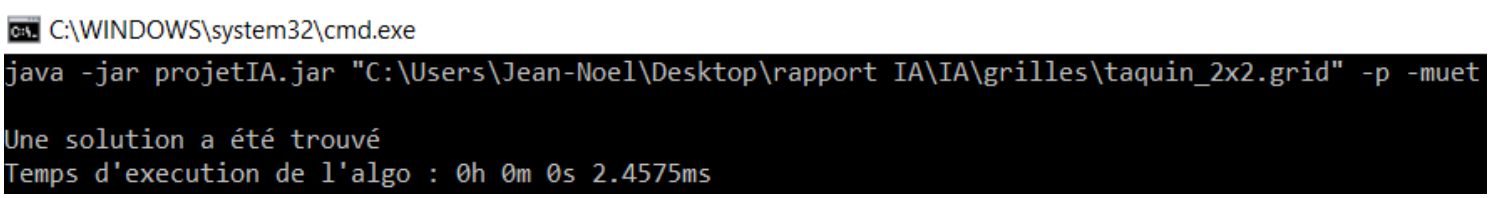
A noter :

- L'heuristique choisie avec l'option **-h=n** n'influence que le parcours du meilleur d'abord. Il y a 4 heuristiques disponibles, l'euristique 1 est appliquée par défaut.

- Si il n'y a pas d'option de parcours, le parcours par défaut est le parcours en largeur

Voir la page 2 pour un exemple de résolution de fichier. Comme on peut le voir, le contenu du fichier d'entré est affiché pour rappeler la grille initiale et la grille finale. Viens ensuite l'affichage (de haut en bas) des étapes de la résolution (si elle existe) et enfin du temps d'exécution de l'algo.[1]

Il est possible, comme indiqué dans l'aide, d'avoir le strict minimum affiché dans la console, il suffit d'utiliser l'option **-muet** .



1 Bien sur, le temps d'exécution ne prend en compte que le temps du calcul. Le temps de lecture du fichier et le temps d'affichage sont exclus du résultat.

C:\WINDOWS\system32\cmd.exe

```
java -jar projetIA.jar "C:\Users\Jean-Noel\Desktop\rapport IA\IA\grilles\taquin_2x2.grid" -p

  ___/ Grille lue : \___

___ Etat initial ___

> configuration :
```

e	l
f	

```
> coord case vide : (x=1, y=1)


___ Etat final ___

> configuration :
```

	f
l	e

```
> coord case vide : (x=0, y=0)

Résolution en cours...
  ___/ Résolution : \___
```

e	l
f	

```
    \/ (1)
```

e	
f	l

```
    \/ (2)
```

	e
f	l

```
    \/ (3)
```

f	e
	l

```
    \/ (4)
```

f	e
l	

```
    \/ (5)
```

f	
l	e

```
    \/ (6)
```

	f
l	e

```
Temps d'execution de l'algo : 0h 0m 0s 2.8994ms
```

2 - Implémentation

Le projet a été implémenté selon une approche orientée objet. Voici les différentes classes qui le compose :

- ***InteractionProb*** : (tout ce qui touche à l'interaction avec le problème) : lecture et mise en mémoire des grilles, affichage des grilles, affichage des états.
- ***ProjetIA :*** Gestion des arguments en ligne de commande, algo général qui appelle les différentes classes nécessaires.
- ***Etat :*** Objet représentant un état. Inclus aussi la génération des états sucesseurs.
- ***Initial :*** Objet statique[2] représentant l'état initial.
- ***Final :*** Objet statique représentant l'état final. Détermine aussi si un état est final.
- ***Ouvert :*** Interface qui sert de modèle aux 3 classes suivante. On se sert de cette interface pour pouvoir utiliser tous les types d'ouvertures correspondantes à tous les types de parcours sans avoir à réécrire l'algo 3 fois :
  - Ouverture_Largeur : Structure qui contiendra les états ouverts pour le parcours en largeur.
  - Ouverture_Profondeur : Structure qui contiendra les états ouverts pour le parcours en profondeur.
  - Ouverture_Meilleurs : Structure qui contiendra les états ouverts pour le parcours du meilleur.
- **Ferme :** Structure qui contiendra les états déjà parcourus. Détermine si un état a déjà été parcouru.
- **Heuristique :** Objet statique qui va contrôler l'ordre des états dans la structure Ouverture_Meilleurs en fonction d'une fonction d'heuristique donné.

Un état est composé de :
- une configuration : la grille qui représente l'état actuel sous forme de texte. *Pour l'affichage, pour générer des successeurs et pour comparer les états entre eux.*
- m et n (stockés une seule fois en mémoire) : la taille de la grille respectivement en largeur et hauteur.
- les coordonnées de la case vide (par rapport au point le plus en haut à gauche) (calculé automatiquement). *Pour générer des successeurs plus rapidement et plus simplement.*
- la référence vers l'état précédent. Pour pouvoir sauvegarder et afficher le chemin parcouru.

Le tri des états en fonction du parcours choisit se fait en utilisant des structures pré-existantes dans le langage (**pile**, **file** et **file ordonnée** pour les parcours respectifs : **profondeur**, **largeur** et **meilleur**).

Les files ordonnées (PriorityQueue) ont une fonction de classement automatique prédéfinie que l'on va redéfinir en se baisant sur les valeurs d'euristique renvoyés pour chaque état par la classe Euristique.java (méthode compareTo dans Etat.java).

La vérification de l'existence d'un état dans Ferme ce fait grâce à une table de hashage qui associe une configuration a un état. Ce qui permet de détecter si un état est dans fermé beaucoup plus rapidement qu'avec un parcours de tableau classique.

2 Qui ne peut pas être instancié, il sert ici comme un moyen efficace pour manipuler l'état initial et final de façon plus lisible qu'avec des variables temporaires.

3 - Euristique

Voici les 4 euristiques qui ont étés implémentés :

Idée 1 : Plus un symbole est proche de sa bonne place, plus on se rapproche de la solution finale.

Euristique 1 : On calcule, pour chaque symbole, le nombre de déplacement minimal (s'il n'y avait aucun autre symbole autours) qu'il doit faire pour être à la bonne place. L'euristique consiste à additionner pour tous les symboles ce nombre de déplacement minimal.

Défaut : Un symbole proche peut nécessiter de nombreux mouvements pour finir à la bonne place (le temps de déplacer un symbole qui est éloigné de sa bonne place par exemple).

Idée 2 : Plus il y a de symboles bien placés, plus on se rapproche de la solution finale.

Euristique 2 : On compare symbole contre symbole, et on additionne le nombre de symboles mal placés.

Défaut : Cette technique ne prend pas en compte l'éloignement. Un symbole peut être à une case de sa bonne place ou 10 cases, il sera traité de la même façon.

Idée 3 : Si on découpe la grille en une multitude de secteurs 2x2, on pourra facilement calculer le nombre de déplacement minimum d'un symbole pour être à la bonne place (si ce symbole existe dans le secteur en question dans la grille finale).

Euristique 3 : Dans le meilleur des cas, pour mettre un symbole à la bonne place, Il faut :

- 1 déplacement pour aller en haut, en bas, à droite ou à gauche
- 4 déplacements pour aller en diagonale

On va donc prendre à chaque fois un symbole et un bloc de 2x2 cases (dans la même zone) et si le symbole est dans ce bloc, on analyse sa position relative dans le bloc pour calculer le nombre de déplacement minimaux. S'il n'est pas dans ce bloc, on le compte comme un déplacement couteux (toute la grille). L'Euristique est la somme des déplacements.

Défaut : Si le symbole n'est pas dans la plage 2x2, encore une fois, sa proximité ne sera pas prise en compte dans l'euristique. De plus la plage 2x2 prise en compte peut ne pas comporter d'espace ce qui augmentera le nombre de déplacement minimum et faussera l'euristique.

Idée 4 : Si on possède les bons symboles sur une ligne mais pas dans le bon ordre, on se rapproche de la solution. De plus, chaque symbole est unique donc on peut avoir une idée approximative du « désordre » de la ligne en associant chaque symbole avec un nombre et en faisant des calculs avec.

Euristique 4 : On associe à chaque symbole son code ASCII et on calcule pour chaque ligne la soustraction de chaque symbole. On compare, pour chaque ligne, avec la soustraction faite dans l'état final et la valeur absolue de la différence entre ces 2 soustractions est rajouté à l'euristique.

Défaut : Parfois la soustraction donne une mauvaise idée de la différence entre 2 lignes.
Ex : On cherche à avoir **5 3 2 6**.
5 – 3 – 2 – 6 = -6
si on a **3 – 5 – 2 – 6** ça fait **-10** et **abs(-6 - (-10)) = 4** alors que
9 – 8 – 1 – 7 ça fait **-7** et **abs(-6 - (-7)) = 1**
Ici l'euristique privilégierait **9 8 1 7** alors qu'il est le moins proche...

4 – Résultats et analyse

Tout d'abord, voici les grilles qui possèdent une solution et celles qui ne possèdent pas de solution ainsi que le temps de calcul minimum pour trouver ces solutions (calcul sur pc portable avec processeur i5) :

taquin_2x2 (1,7726 ms)
taquin_2x4 (964,0274 ms)
taquin_2x4b (4,6353 ms)
taquin_2x4c (19,2131 ms)
taquin_2x4d (12,9673 ms)
taquin_3x3 (1,8258 ms)
taquin_3x3b (6,2282 ms)
taquin_3x3c (6,2684 ms)
taquin_3x3d (113914,7435 ms)
taquin_3x3e (11,0305 ms)
taquin_3x3f (2,9231 ms)
taquin_3x4b (15,2251 ms)
taquin_4x4 (63,9208 ms)
taquin_4x4e (48,5565 ms)
taquin_5x5 (4597679.1949 ms)

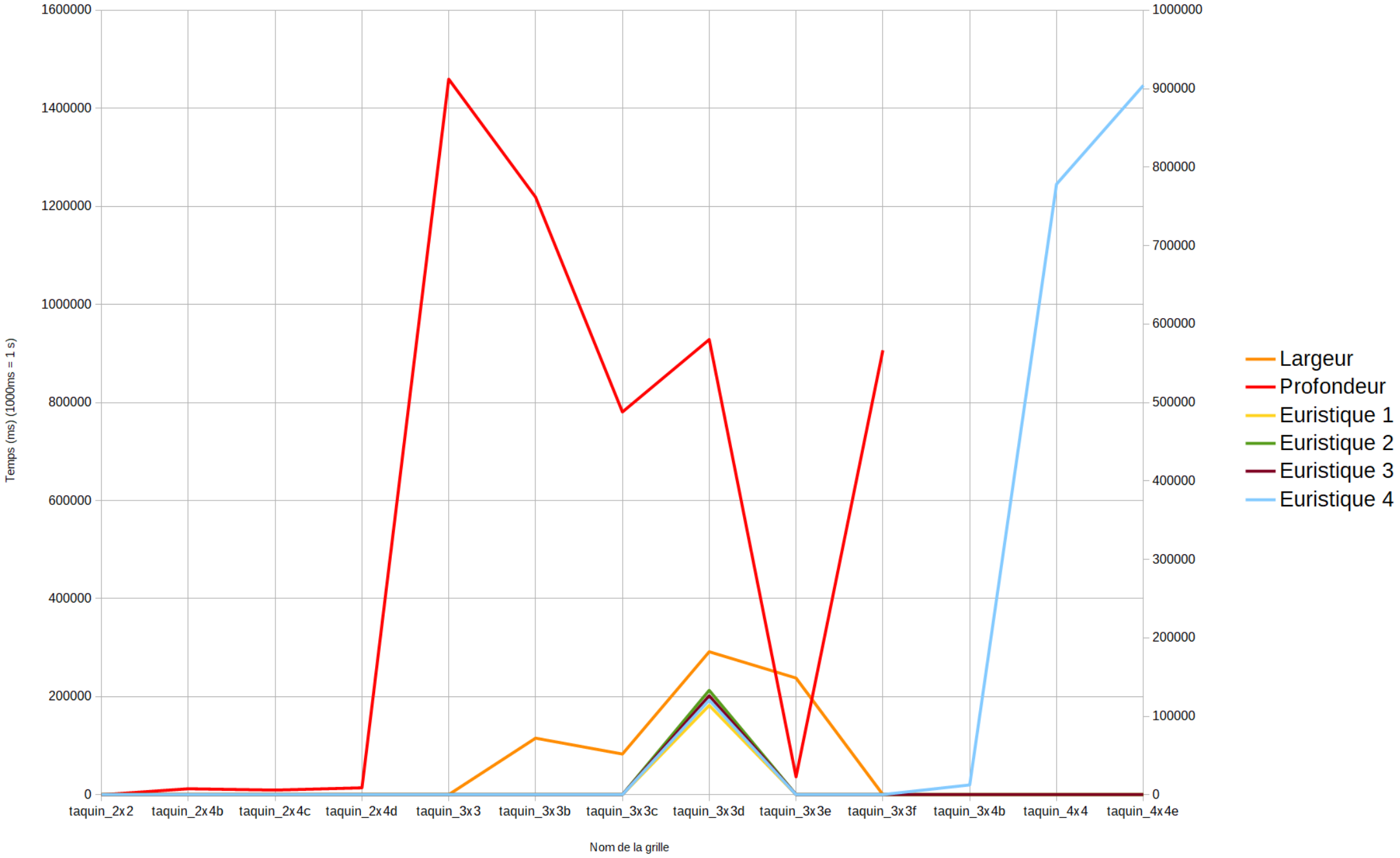
(les autres grilles n'ont pas été testés ou mettait trop de temps à répondre (>1H))

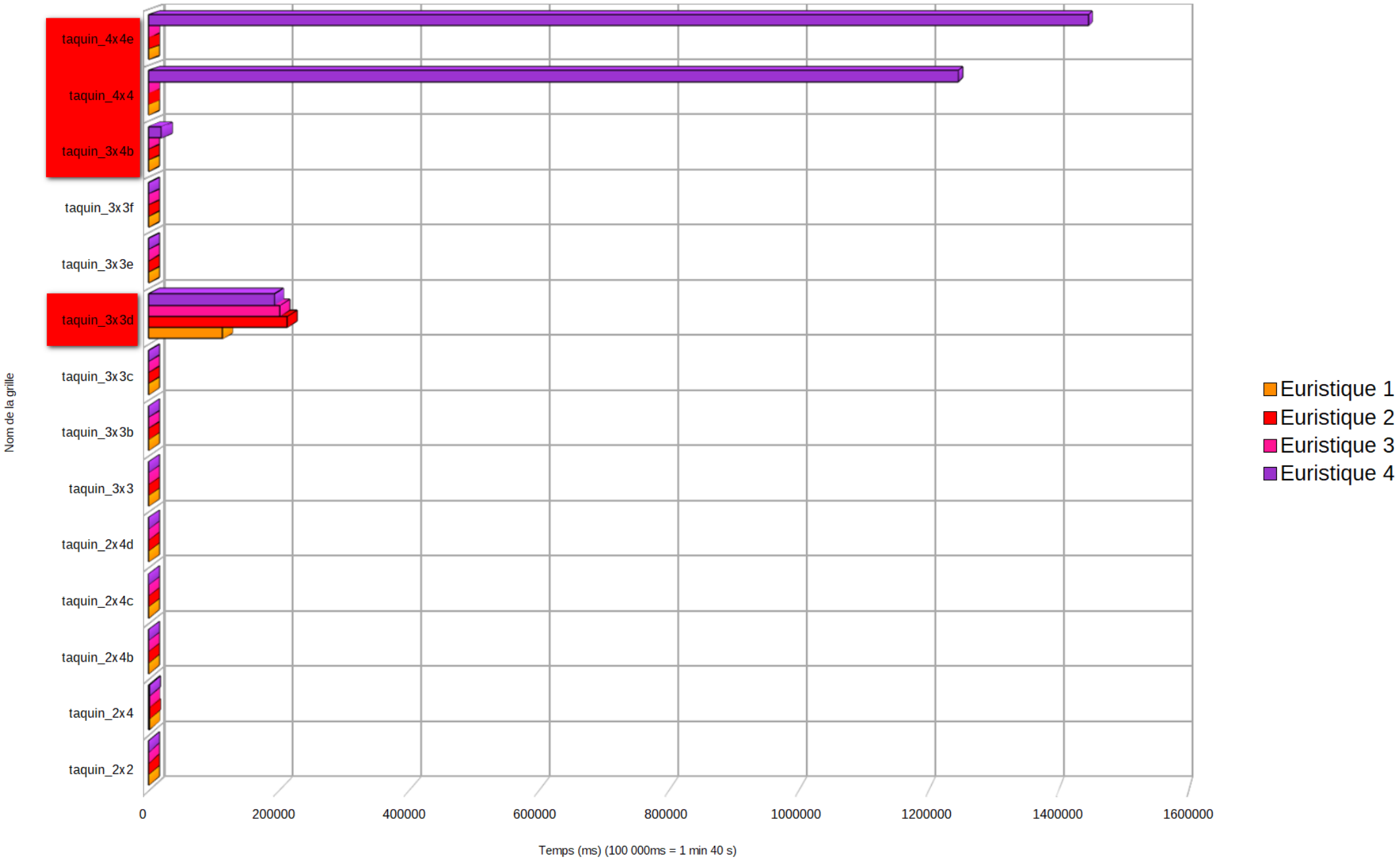
Ci-dessous, le tableau des temps de calcul en fonction du type de parcours choisit (temps en ms) :

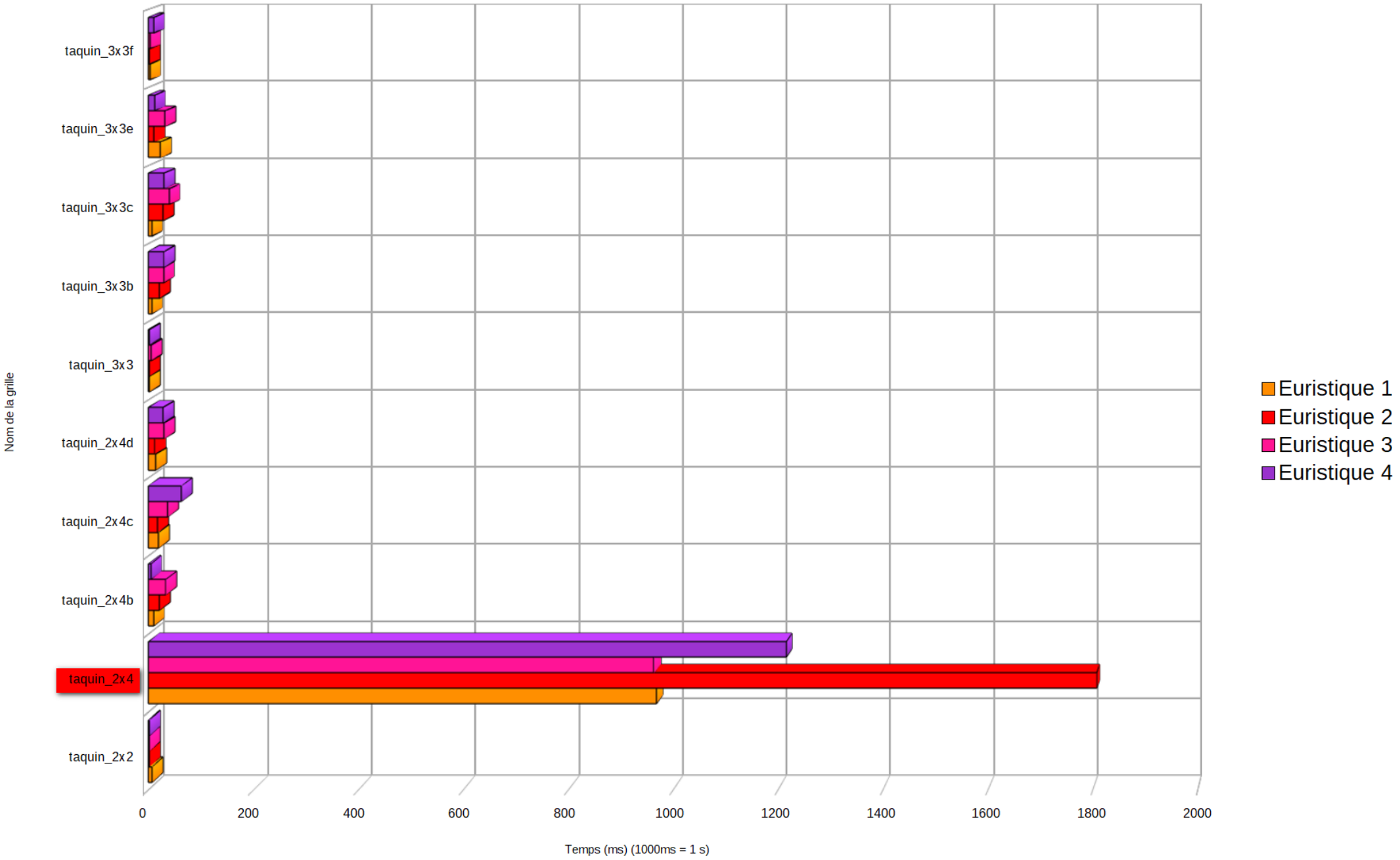
			Meilleur			
Nom grille	Largeur	Profondeur	Euristique 1	Euristique 2	Euristique 3	Euristique 4
taquin_2x2	18,0678	1,7726	7,101	2,2045	1,797	2,1947
taquin_2x4b	216,4074	7732,8081	9,8706	20,7944	33,2967	4,6353
taquin_2x4c	330,9456	6055,7375	19,2131	17,5599	36,9193	63,2213
taquin_2x4d	800,2564	8917,184	13,7638	12,9673	30,0814	27,9857
taquin_3x3	6,1996	912212,7689	1,8612	1,8258	5,1782	1,7078
taquin_3x3b	115555,7217	761881,9639	6,2282	20,9799	28,796	29,8589
taquin_3x3c	83160,2699	488049,7168	6,2684	28,258	39,8511	30,0521
taquin_3x3d	291782,0582	580352,9226	113914,7435	213249,8493	202058,2632	193271,6235
taquin_3x3e	238186,0926	22885,0417	23,1972	11,0305	31,7079	11,475
taquin_3x3f	3,8844	566685,104999	2,9231	1,6159	3,0084	9,7123
taquin_3x4b			15,2251	40,8747	21,9368	20226,2967
taquin_4x4		LONG	63,9208	614,2246	359,0521	1245543,104
taquin_4x4e			48,5565	394,723199	172,847099	1446826,261

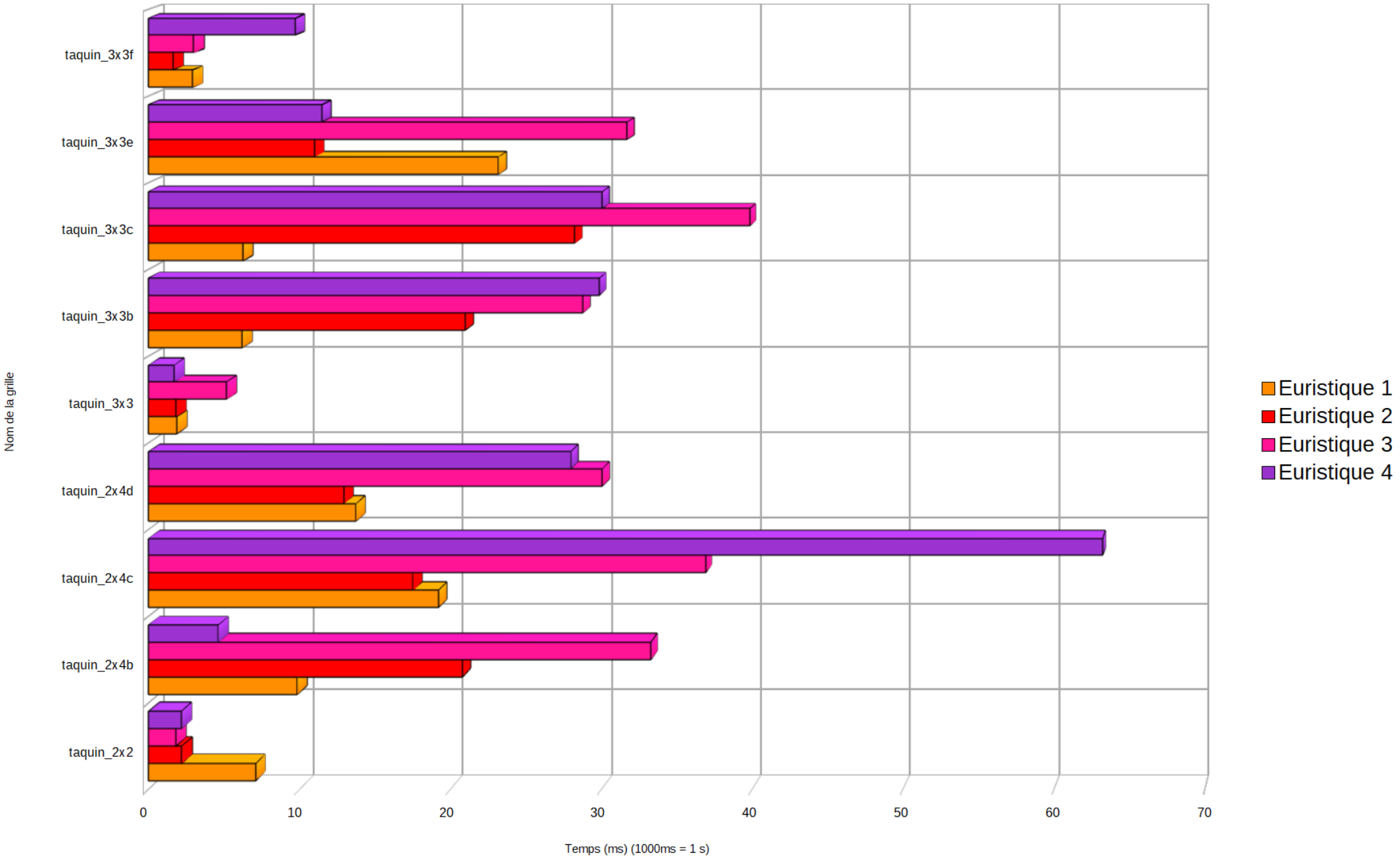
On peut dresser un premier graphe général (Page **6**) où l'on met en évidence que les parcours en largeur et en profondeur sans euristique particulières sont les plus longs sur les taquins de taille comprise entre 2x2 et 3x3.
On se concentre donc ensuite sur le parcours « meilleur d'abord » et des euristiques associés, en enlevant les grilles au fur et à mesure, si la durée du traitement de la grille écrase les autres valeurs du graphe (grilles surlignées en rouge). (Pages **7-9**).

L'Euristique 4 est efficace dans certains cas mais augmente considérablement le temps de calcul dès que la grille devient trop grande. L'Euristique 1 reste la plus polyvalente et globalement la plus rapide. L'Euristique 3 s'avère plus efficace avec des grilles de grande taille. L'Euristique 2 peut être intéressante pour des grilles de petite taille.


1600000
1400000
1200000
1000000
800000
600000
400000
200000
0
1000000
900000
800000
700000
600000
500000
400000
300000
200000
100000
0
Temps (ms) (1000ms = 1 s)
taquin_2x2
taquin_2x4b
taquin_2x4c
taquin_2x4d
taquin_3x3
taquin_3x3b
taquin_3x3c
taquin_3x3d
taquin_3x3e
taquin_3x3f
taquin_3x4b
taquin_4x4
taquin_4x4e
Nom de la grille
Largeur
Profondeur
Euristique 1
Euristique 2
Euristique 3
Euristique 4


taquin_4x4e
taquin_4x4
taquin_3x4b
taquin_3x3f
taquin_3x3e
taquin_3x3d
taquin_3x3c
taquin_3x3b
taquin_3x3
taquin_2x4d
taquin_2x4c
taquin_2x4b
taquin_2x4
taquin_2x2
Nom de la grille
0
200000
400000
600000
800000
1000000
1200000
1400000
1600000
Temps (ms) (100 000ms = 1 min 40 s)
Euristique 1
Euristique 2
Euristique 3
Euristique 4


taquin_3x3f
taquin_3x3e
taquin_3x3c
taquin_3x3b
taquin_3x3
taquin_2x4d
taquin_2x4c
taquin_2x4b
taquin_2x4
taquin_2x2
Nom de la grille
0
200
400
600
800
1000
1200
1400
1600
1800
2000
Temps (ms) (1000ms = 1 s)
Euristique 1
Euristique 2
Euristique 3
Euristique 4


taquin_3x3f
taquin_3x3e
taquin_3x3c
taquin_3x3b
taquin_3x3
taquin_2x4d
taquin_2x4c
taquin_2x4b
taquin_2x2
Nom de la grille
0
10
20
30
40
50
60
70
Temps (ms) (1000ms = 1 s)
Euristique 1
Euristique 2
Euristique 3
Euristique 4