Rapport du TP IOC

Partie 2 et 3 - Résolution de problèmes de sac à dos

Shuangrui CHEN
Prof: Nicolas Dupin
6 mars 2021

ReadMe

Compile with cmake:

Cmake minimum required version 3.17

Required modules: OpenMP

Run the following commands under ./TP_IOC:

```
mkdir build
cd ./build
cmake .. -j6
make -j6
```

Note: some execute files need to be run under the folder ./TP_IOC

Pictures are draw by matplotlib and pandas in python.

File: ./other/Draw.ipynb

1 Partie analyse numérique

Configuration matérielle et logicielle:

Hardware Overview:

Model Name:	MacBook Pro
Processor Name:	Intel Core i5-1038NG7 @ 2.00 GHz
Number of Processors:	1
Total Number of Cores:	4
L2 Cache (per Core):	512 KB
L3 Cache:	6 MB
Hyper-Threading Technology:	Enabled
Memory:	16 GB LPDDR4X
Os:	MacOS 11.0.1

Cmake: 3.19.6

- Question 15

1. Sur la fonction *solveUpperBound* de la classe *nodeBB*, j'utilise une variable intermédiaire *sum_value* qui stocke la somme de valeurs des items et met à jour le *localUpperBound* à la fin de calcule. Mais pour le cas normal (un truc n'est pas fixé ni supprimé et n'accède pas *kpBound*), cette somme est calculé après la mise à jour de *localUpperBound*, donc cela pose une erreur que le dernier truc est jamais calculé dans le *localUpperBound* car la dernier somme n'est pas mise à jour. La corrigé est de remplacer la variable *sum_value* par la variable de classe *localUpperBound*.

```
void NodeBB::solveUpperBound(int kpBound, int nbItems, vector<int> &weights, vector<int> &values) {
    int sum_value = 0;
    int sum_weight = 0;

    for (int i = 0; i < nbItems; ++i) {
        sum_weight += isFixed[i] * weights[i];
        sum_value += isFixed[i] * values[i];
    }

    if (sum_weight > kpBound) {
        overCapacitated = true;
    } else if (sum_weight == kpBound) {
        return;
    } else {
        for (int i = 0; i < nbItems; ++i) {
            if (!isFixed[i] && !isRemoved[i]) {
                sum_weight += weights[i];
                if (sum_weight == kpBound) {
                    localUpperBound = sum_value + values[i];
                    return;
                }
```

```
                if (sum_weight > kpBound) {
                    sum_weight -= weights[i];
                    int dif = kpBound - sum_weight;
                    criticalIndex = i;
                    fractionalVariable = (double) dif / weights[i];
                    localUpperBound = sum_value + fractionalVariable * values[i];
                    break;
                }
                sum_value += values[i];
            }
        }
    }
}
```

2. Dans la fonction *solve* de la classe *KpSolverBB*, quand le dernier noeud qui exagère le *upperBoundOPT* est tiré, le *costSolution* est égal à *upperBoundOPT* donc on sort la boucle while. Mais le *upperBoundOPT* n'est pas mise à jour donc quand à l'affichage du résultat, la solution cost est correct mais le « proven upper bound » reste une borne un peu grand que réel.

```
void KpSolverBB::solve() {
    init();
    int count = 0;

    while (!nodes.empty() && nbNodes < nbMaxNodeBB) {
        nbNodes ++;
        // Mise à jour au début mais pas à la fin
        int best = floor(getUpperBound());
        NodeBB *node = selectNode();
        node->solveUpperBound(knapsackBound, nbItems, weights, values);

        if (withPrimalHeuristics || node->getFractionalVariable() == 0) {
            node->primalHeuristic(knapsackBound, nbItems, weights, values);
            if (costSolution < node->getNodePrimalBound()) {
                costSolution = node->getNodePrimalBound();
                node->copyPrimalSolution(&solution);
            }
            if (node->getFractionalVariable() == 0) continue;
        }

        if (costSolution == best) break;

        NodeBB *nod1 = new NodeBB(*node);
        NodeBB *nod2 = new NodeBB(*node);
        nod1->fixVariable(node->getCriticalIndex(), false);
        nod1->solveUpperBound(knapsackBound, nbItems, weights, values);
        nod2->fixVariable(node->getCriticalIndex(), true);
        nod2->solveUpperBound(knapsackBound, nbItems, weights, values);
        insertNodes(nod1, nod2);

        getUpperBound(); // Corrigé
        // printStatus();
    }
    …
}
```

Voici 2 erreurs qui influence le résultat. Il y a aussi des imperfections qui influence le temps de calcul et la mémoire utilisé.

Test - NB d'instance	Version corrigé	Version personnelle
Heur - 100	0.0267317	0.0446288
Heur - 1000	0.135012	0.227716
Heur - 10000	0.113056	0.211654
Heur - 100000	0.11313	0.215883
BB - BestBound - 1000	0.441294	0.559441
BB - DP - DFS10 - 1000	0.254374	1.18351

Donc il y a aussi des améliorations possibles sur les détails. Par exemple:

Sur l'implémentations de l'algorithme B&B, dans la fonction solve, si on trouve une solution optimale ou excède le nombre maximal de noeud, on sort la boucle while. Mais j'ai ajouté une autre boucle superflue qui calcule la reste de noeuds dans la file. Cela n'est pas nécessaire si la solution est déjà optimale, aussi pas d'amélioration remarquable si on excède le nombre maximal de noeud.

```
// Code après la calcule normale de boucle while
//    while (!nodes.empty()) {
//        int best = floor(getUpperBound());
//        NodeBB *node = selectNode();
//        node->solveUpperBound(knapsackBound, nbItems, weights, values);
//
//        if (withPrimalHeuristics || node->getFractionalVariable() == 0) {
//            node->primalHeuristic(knapsackBound, nbItems, weights, values);
//            if (costSolution < node->getNodePrimalBound()) {
//                costSolution = node->getNodePrimalBound();
//                node->copyPrimalSolution(&solution);
//            }
//            if (node->getFractionalVariable() == 0) continue;
//        }
//
//        if (costSolution == best) break;
//    }
```

- Question 16

Selon le tableau ci-dessous, pour l'algorithme B&B, n'import quelle stratégie de branching est choisi, si on n'ajoute pas l'heuristique, il y a une grande possibilité que la stratégie directe vers une branche mauvaise et ne sort pas une solution optimale. Par exemple, même si pour une instance de 100 trucs (la ligne rouge), la stratégie BFS utilise tous 10000 noeuds et ne retourne pas une solution optimale. Pour les instances différents, on ne sais pas à l'avance quel stratégie de branching est la meilleur, donc si on ajoute une heuristique, cela améliore beaucoup la solution et le nombre de noeuds utilisés. L'heuristique DP (Blocs verts)semble très prometteuse et n'est pas influencé par la stratégie de branching. Cela est bien expliqué par son principe (Avec une DP qui résout le voisinage du point critique, cela est très peu influencé si on fixe le point critique). Mais cela m'étonne encore que pour l'instance de 100000 il ne prends qu'un noeud(encore possible).

testStrategiesBB

File	Algo	Strategy	Time	Nb Noeuds	gap
kp_100_2.in	Greedy		1.5979e-05		
kp_100_2.in	B&B	BestBound	0.00527021	465	0
kp_100_2.in	B&B	DFS10	0.0062074	509	0
kp_100_2.in	B&B	DFS01	0.010534	1046	0
kp_100_2.in	B&B	BFS	0.0619173	10000	0.349842
kp_100_2.in	B&B	Random	0.0568354	10000	0.352462
kp_100_2.in	B&B - Heur	BestBound	0.00200997	221	0
kp_100_2.in	B&B - Heur	DFS10	0.00403069	464	0
kp_100_2.in	B&B - Heur	DFS01	0.00575556	554	0
kp_100_2.in	B&B - Heur	BFS	0.00199897	238	0
kp_100_2.in	B&B - Heur	Random	0.0020533	238	0
kp_100_2.in	B&B - HeurDP	BestBound	0.0345479	220	0
kp_100_2.in	B&B - HeurDP	DFS10	0.0340457	220	0
kp_100_2.in	B&B - HeurDP	DFS01	0.0336526	220	0

kp_100_2.in	B&B - HeurDP	BFS	0.036137	220	0
kp_100_2.in	B&B - HeurDP	Random	0.0371226	220	0
kp_1000_2.in	Greedy		3.8836e-05		
kp_1000_2.in	B&B	BestBound	0.0254646	519	0
kp_1000_2.in	B&B	DFS10	0.0357956	799	0
kp_1000_2.in	B&B	DFS01	0.463291	10000	0.00465782
kp_1000_2.in	B&B	BFS	0.0447619	2036	0
kp_1000_2.in	B&B	Random	0.155558	6841	0
kp_1000_2.in	B&B - Heur	BestBound	0.0204405	279	0
kp_1000_2.in	B&B - Heur	DFS10	0.0270283	357	0
kp_1000_2.in	B&B - Heur	DFS01	0.0455913	603	0
kp_1000_2.in	B&B - Heur	BFS	0.0163411	225	0
kp_1000_2.in	B&B - Heur	Random	0.0166965	227	0
kp_1000_2.in	B&B - HeurDP	BestBound	0.0526017	134	0
kp_1000_2.in	B&B - HeurDP	DFS10	0.0556346	134	0
kp_1000_2.in	B&B - HeurDP	DFS01	0.0633728	134	0
kp_1000_2.in	B&B - HeurDP	BFS	0.057247	134	0
kp_1000_2.in	B&B - HeurDP	Random	0.0547372	134	0
kp_10000_2.in	Greedy		0.000357782		
kp_10000_2.in	B&B	BestBound	8.98718	10000	inf
kp_10000_2.in	B&B	DFS10	3.49432	10000	0.000391602
kp_10000_2.in	B&B	DFS01	2.67211	10000	9.79003e-05
kp_10000_2.in	B&B	BFS	0.478553	2137	0
kp_10000_2.in	B&B	Random	1.10066	5297	0
kp_10000_2.in	B&B - Heur	BestBound	0.352877	548	0
kp_10000_2.in	B&B - Heur	DFS10	7.07551	10000	0.000342652
kp_10000_2.in	B&B - Heur	DFS01	2.46661	3661	0
kp_10000_2.in	B&B - Heur	BFS	0.645393	1205	0
kp_10000_2.in	B&B - Heur	Random	0.690555	1300	0
kp_10000_2.in	B&B - HeurDP	BestBound	0.466908	497	0
kp_10000_2.in	B&B - HeurDP	DFS10	0.470138	497	0
kp_10000_2.in	B&B - HeurDP	DFS01	0.467642	497	0
kp_10000_2.in	B&B - HeurDP	BFS	0.47256	497	0
kp_10000_2.in	B&B - HeurDP	Random	0.464191	497	0

kp_100000_2.in	Greedy		0.00186504		
kp_100000_2.in	B&B	BestBound	42.6763	10000	inf
kp_100000_2.in	B&B	DFS10	40.5538	10000	inf
kp_100000_2.in	B&B	DFS01	43.4309	10000	inf
kp_100000_2.in	B&B	BFS	23.5839	10000	0
kp_100000_2.in	B&B	Random	6.41691	2977	0
kp_100000_2.in	B&B - Heur	BestBound	0.533048	131	0
kp_100000_2.in	B&B - Heur	DFS10	78.3633	10000	2.46654e-06
kp_100000_2.in	B&B - Heur	DFS01	0.144618	37	0
kp_100000_2.in	B&B - Heur	BFS	0.821749	219	0
kp_100000_2.in	B&B - Heur	Random	0.860631	243	0
kp_100000_2.in	B&B - HeurDP	BestBound	0.0410145	1	0
kp_100000_2.in	B&B - HeurDP	DFS10	0.0473219	1	0
kp_100000_2.in	B&B - HeurDP	DFS01	0.0479241	1	0
kp_100000_2.in	B&B - HeurDP	BFS	0.0487336	1	0
kp_100000_2.in	B&B - HeurDP	Random	0.0474729	1	0

- Question 17

Par la conclusion de question 16, la meilleur configuration des paramètres est de choisir heuristique DP n'import quel stratégie de branching (ici on prend BestBound pour l'expérimentation). Voici le résultat de toutes les instances:

testSolution

File	Nb Noeuds	Upperbound	Solution	gap	Time
kp_100_1.in	91	41700	41700	0	0.0394174
kp_100_2.in	220	38400	38400	0	0.0423227
kp_1000_1.in	1693	396688	396688	0	0.294811
kp_1000_2.in	134	407933	407933	0	0.0558726
kp_10000_1.in	1	4.09264e+06	4092641	0	0.0374066
kp_10000_1_0.1.in	1441	1.29668e+06	1296678	0	0.843554
kp_10000_1_0.01.in	99	417231	417231	0	0.0775613
kp_10000_1_0.02.in	490	585031	585031	0	0.302036
kp_10000_1_0.03.in	1995	714788	714788	0	0.974675

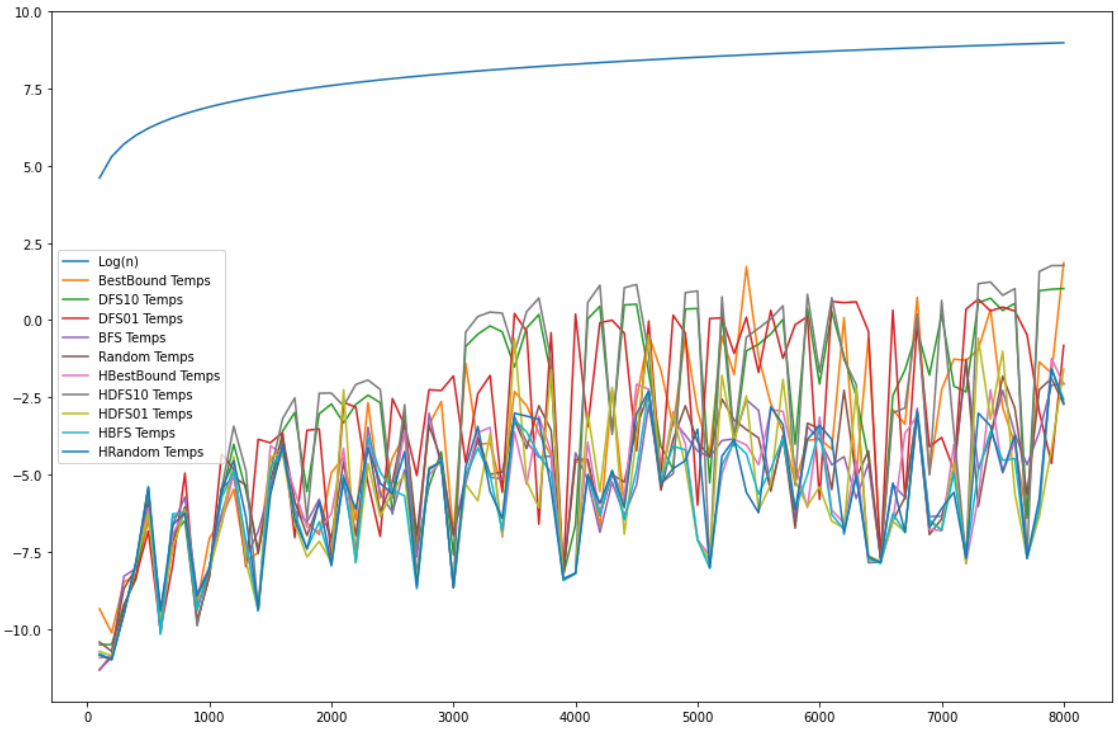
kp_10000_1_0.04.in	1	823225	823225	0	0.00889838
kp_10000_1_0.05.in	26	919169	919169	0	0.0321018
kp_10000_2.in	497	4.08579e+06	4085792	0	0.5501
kp_100000_1.in	1	4.06866e+07	40686622	0	0.0433814
kp_100000_2.in	1	4.05425e+07	40542543	0	0.0514366

Le gap est bien 0 pour tous. Le temps de calcules et le nombre de noeuds utilisés sont satisfiables.

- Question 18

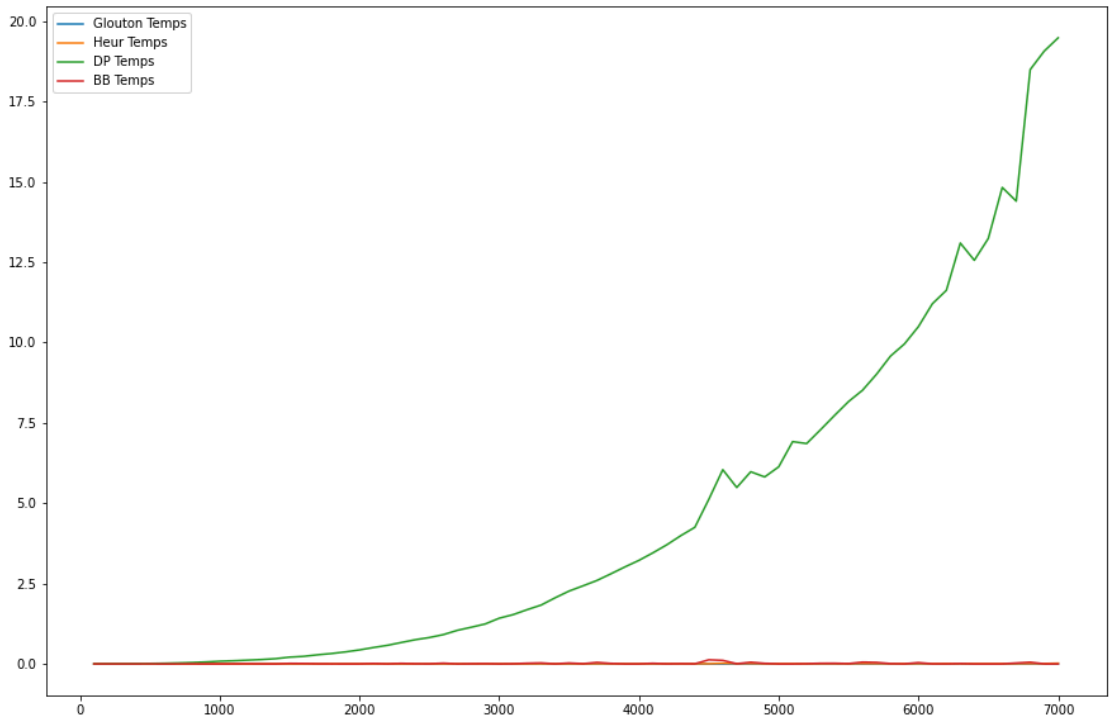
Après nombreux expérimentations sur l'influence de la taille de l'instance sur l'algorithme B&B, je ne trouve pas une claire conclusion que certaine paramétrisation du B&B est cohérent de la complexité O(N Log N). Mais je trouve qu'il y a des liens, voici deux graphes sur le nombre de noeuds utilisé et le temps de calculs en changeant la taille de l'instance. La donnée est produit par *testComplexite.cpp* et *testComplexiteNBNoeud.cpp* et est dessiné par python *Draw.ipynb*.

Temps de calculs de B&B en second



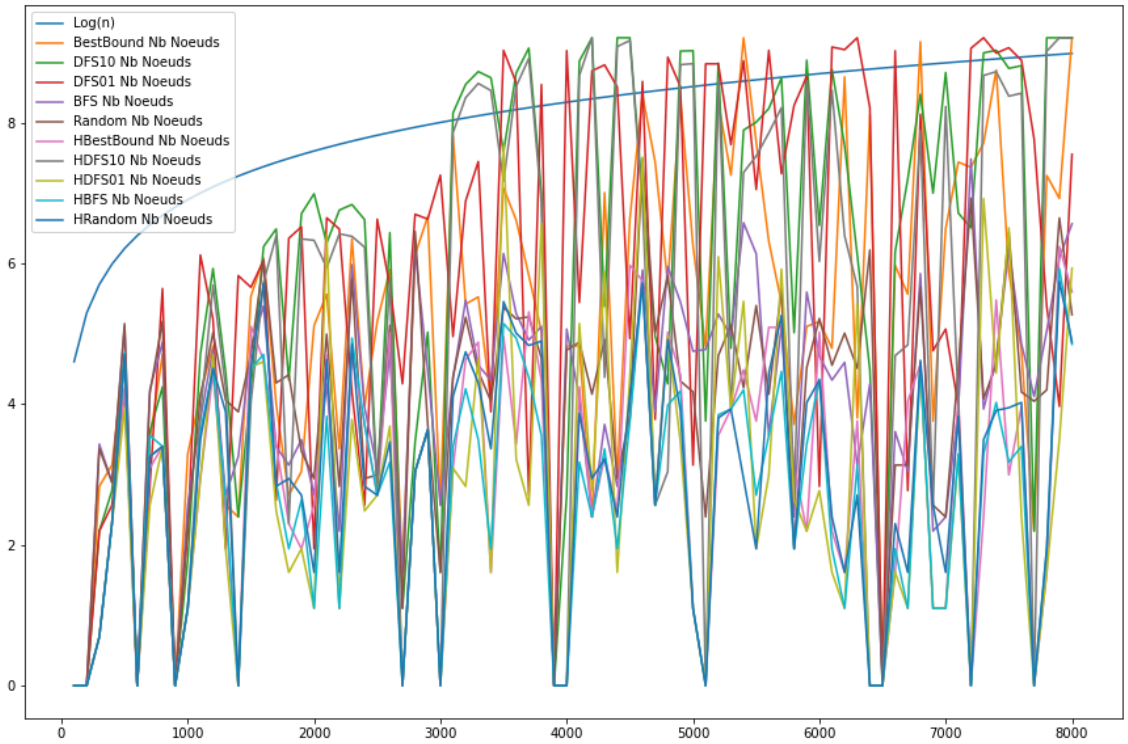
La taille de l'instance

Temps de calcules des différents algos



La taille de l'instance

Nombre de noeuds utilisés de B&B



La taille de l'instance

Sur l'aspect des algorithmes, voici une comparaison du temps de calcules:

Donc le seul algorithme qui est près de O(N^2) ou O(N^3) est la Programmation dynamique.

- Question 18bis

Greedy

File	Upperbound	Solution	gap	Time
kp_100_1.in	41752	41700	0.001247	2.8903e-05
kp_100_2.in	38441	38367	0.00192874	5.097e-06
kp_1000_1.in	396700	396644	0.000141185	2.3877e-05
kp_1000_2.in	407935	407930	1.2257e-05	2.4815e-05
kp_10000_1.in	4092641	4092640	2.44341e-07	0.000229979
kp_10000_1_0.1.in	1296681	1296663	1.38818e-05	0.000147471
kp_10000_1_0.01.in	417235	417210	5.99219e-05	0.000136089
kp_10000_1_0.02.in	585034	584993	7.00863e-05	0.000176912
kp_10000_1_0.03.in	714797	714764	4.61691e-05	0.000127502
kp_10000_1_0.04.in	823225	823208	2.06509e-05	0.000123702
kp_10000_1_0.05.in	919171	919134	4.02553e-05	0.00014272
kp_10000_2.in	4085793	4085773	4.89503e-06	0.000212613
kp_100000_1.in	40686915	40686621	7.22596e-06	0.00212569
kp_100000_2.in	40542690	40542542	3.65049e-06	0.00202093

HeurDP

File	Upperbound	Solution	gap	Time
kp_100_1.in	41700	41700	0	0.0712656
kp_100_2.in	38400	38400	0	0.0726921
kp_1000_1.in	396688	396688	0	7.2408
kp_1000_2.in	407933	407933	0	6.59421

B&B - Heur DP

File	Upperbound	Solution	gap	Time
kp_100_1.in	41700	41700	0	0.0394174
kp_100_2.in	38400	38400	0	0.0423227
kp_1000_1.in	396688	396688	0	0.294811
kp_1000_2.in	407933	407933	0	0.0558726
kp_10000_1.in	4.09264e+06	4092641	0	0.0374066

kp_10000_1_0.1.in	1.29668e+06	1296678	0	0.843554
kp_10000_1_0.01.in	417231	417231	0	0.0775613
kp_10000_1_0.02.in	585031	585031	0	0.302036
kp_10000_1_0.03.in	714788	714788	0	0.974675
kp_10000_1_0.04.in	823225	823225	0	0.00889838
kp_10000_1_0.05.in	919169	919169	0	0.0321018
kp_10000_2.in	4.08579e+06	4085792	0	0.5501
kp_100000_1.in	4.06866e+07	40686622	0	0.0433814
kp_100000_2.in	4.05425e+07	40542543	0	0.0514366

Sur nos instances, les heuristique arrive toujours à trouver la solution optimale. Cela n'est pas forcément la solution relative. Mais sur le temps de calculs, la programmation dynamique n'est pas très acceptable mais un très bon outil d'heuristique de B&B.

- Question 19

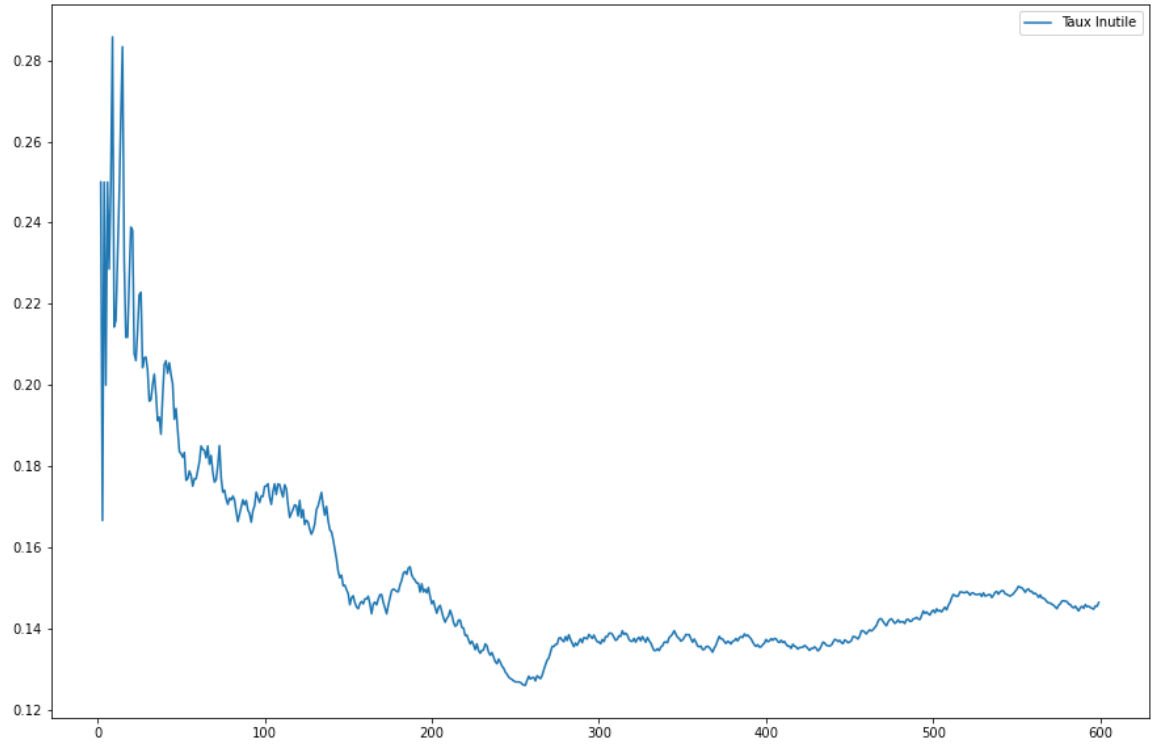
Selon la question 18 et 18bis, à la limite configuration matérielle, l'instance de taille 10000 est déjà incalculable. C'est mieux d'utiliser B&B et ajoute une heuristique DP.

- Question 20 (Question répondu et réponse inchangé de O&A)

Selon la graphe, le taux de case inutile est approximativement dans la borne de [30%, 10%]. Lorsque la taille des instances argumente, le taux de case inutile est stable autour de 14%.

- Question 21

Taux de case inutile

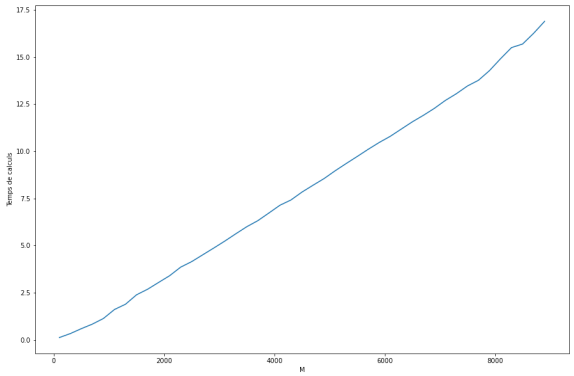


- Question 22 (Question répondu et réponse inchangé de O&A)

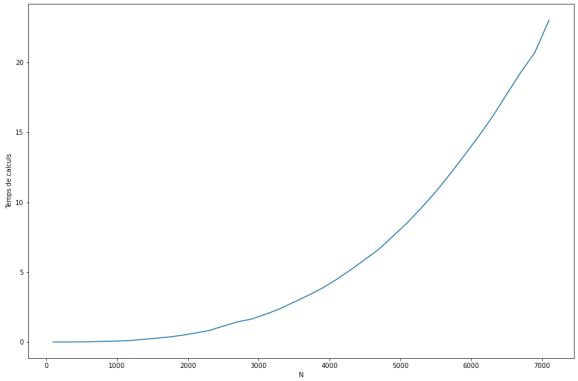
En comparant ces deux graphes, on peut conclure que l'impact de M sur la PD est linéaire, au contraire, l'impact de N est plutôt exponentiel.

La donnée vient de *testMimpact.csv* et *testNimpact.csv* qui sont produit par *testMimpact.cpp* et *testNimpact.cpp*.

Impact de M sur la programmation dynamique



Impact de N sur la programmation dynamique



2 Partie 3 Approfondissements implémentation C++

3.1 Démarrage à chaude

En train de faire