INF8775 Lab2 Analyses Asymptotiques

Fabrice Charbonneau et Joël Poulin

Mars 2019

Analyses asymptotiques de la consommation de ressources pour les algorithmes glouton, programmation dynamique et recherche locale pour un problème d'emplacements de restaurants semblable à celui du problème de sac à dos.

1 Glouton

Le code c++ utilisé pour l'algorithme glouton, en ne gardant que les parties pertinentes pour l'analyse, est le suivant:

```
int getSumRatio(vector<Ratio> ratios) {
        int sumRatio = 0;
        for (unsigned int i = 0; i < ratios.size(); i++) {
                sumRatio += ratios[i].ratio;
        }
        return sumRatio;
}

Solution resolveGlouton(Problem problem) {
        Solution solution;
        vector<Resto> restos = problem.restos;
        // Compute the ratios of all restaurant emplacements
        vector<Ratio> ratios;
        double sumRatio = 0;
        for (unsigned int i = 0; i < restos.size(); i++) {
                Ratio Ri;
                Ri.ratio = (double) restos[i].r / (double) restos[i
                    ].q;
                Ri.resto = restos[i];
                sumRatio += Ri.ratio;
                ratios.push_back(Ri);
        }
        random_device rd;
        mt19937 generator(rd());
        uniform_real_distribution<double> uniform(0.0, 1.0);
        solution.totalChickens = 0;
        while ((ratios.size() > 0) && (solution.totalChickens <
            problem.capacity)) {
                // Choose a restaurant
                int tryIdx, chosenIndex;
                int sumRatio = getSumRatio(ratios);
                while (true) {
                        tryIdx = (int)(uniform(generator) * (ratios
                            .size()-1));
                        if (uniform(generator) < ((double) ratios[
                            tryIdx].ratio / (double) sumRatio)) {
                                chosenIndex = tryIdx;
                                break;
                        };
                }
                // If the selected resto surpasses the capacity
                if ((solution.totalChickens + ratios[chosenIndex].
                    resto.q) > problem.capacity) {
                        // Remove the restaurant from the
                            possibilities
                        ratios.erase(ratios.begin() + chosenIndex);
                }
                else {
                        // Add the id to the solution
                        solution.restos.push_back(ratios[
                            chosenIndex].resto);
                        // Add the chickens to the total
```

```
                        solution.totalChickens += ratios[
                            chosenIndex].resto.q;
                        // Remove the restaurant from the
                            possibilities
                        ratios.erase(ratios.begin() + chosenIndex);
                    }
                }
                return solution;
}
```

Tout d'abord, on peut découper le code en deux parties: la première boucle for qui calcule le ratio pour chacun des emplacements, puis la boucle while qui continue tant qu'il reste des emplacements possibles et qu'il y a une capacité restante positive. Il est à noter que dans nos analyses, nous allons supposer que la suppression d'un élément dans un vecteur se fait en $\Theta(1)$. La première boucle for se fait en $\Theta(n)$. La boucle while contient deux parties: une autre boucle qui trouve un index de façon probabiliste en fonction des ratios trouvés, puis un if qui teste si l'emplacement correspondant à l'index trouvé peut être accepté dans la solution ou non. Le reste des opérations sont élémentaires et se font en $\Theta(1)$, en temps linéaire. La boucle qui trouve l'index de façon probabiliste dépend de la taille des emplacements restants, et celle-ci diminue au fur et à mesure qu'on ajoute des éléments dans la solution. De plus, pour les premières itérations de la boucle while globale, on est presque garanti que l'index choisi sera retenu, car la capacité restante est maximale au départ. C'est seulement vers la fin que les index choisis risquent de ne pas être retenus, auquel cas on doit retirer ces emplacements de la liste, faisant en sorte que d'accepter un index devient plus rapide. Il est à noter que la fonction

```
getSumRatio(ratios)
```

doit se faire en $\Theta(n)$, mais que ce n décroit aussi au fur et à mesure que l'algorithme avance. On peut alors supposer que la complexité de la boucle while globale est d'environ de $\Theta(nlog(n))$. Cette complexité étant plus importante que celle de la première boucle for, qui est en $\Theta(n)$, on peut alors conclure que notre analyse théorique nous indique une complexité dans l'ordre de:

$$\Theta(nlog(n))$$

2 Programmation Dynamique

Le code c++ utilisé pour l'algorithme de programmation dynamique, en ne gardant que les parties pertinentes pour l'analyse, est le suivant:

```
Solution resolveDynProg(Problem problem) {
        Solution solution;
        vector<Resto> restos = problem.restos;
        const unsigned int N = restos.size();

        // Initialisation du tableau
        vector<vector<unsigned int>> D(N);
        for (auto&& x : D)
                x.resize(problem.capacity + 1);

        // Remplissage du tableau
        for (unsigned int i = 0; i < N; i++) {
                for (unsigned int j = 0; j < problem.capacity + 1;
                    j++) {
                        if (j == 0) {
                                D[i][j] = 0;
                        }
                        else if (i == 0) {
                                if (restos[i].q <= j)
                                        D[i][j] = restos[i].r;
                        }
                        else {
                                // Manage unexistant values
                                if (j < restos[i].q) {
                                        D[i][j] =D[i - 1][j];
                                }
                                else {
                                        D[i][j] = max(restos[i].r +
                                             D[i - 1][j - restos[i
                                            ].q], D[i - 1][j]);
                                }

                        }

                }
        }

        // Trouver la solution    partir du tableau
        int j = problem.capacity;
        solution.totalChickens = 0;

        for (int i = N - 1; i >= 0; i--) {
                if (i == 0) {
                        if (j >= restos[i].q) {
                                solution.restos.push_back(restos[i
                                    ]);
                        }
                }
                else {
                        if (D[i][j] != D[i - 1][j]) {
                                if (j - restos[i].q >= 0) {
```

```
                                        // Add the id to the
                                            solution
                                        solution.restos.push_back(
                                            restos[i]);
                                        // Add the chickens to the
                                            total
                                        solution.totalChickens +=
                                            restos[i].q;
                                        // Update j
                                        j -= restos[i].q;
                                }
                            }
                        }
                }
        return solution;
}
```

L'algorithme de la programmation dynamique contient deux parties principales: le remplissage du tableau et la recherche de la solution à partir du tableau. La partie trouver la solution se fait en seulement $\Theta(n)$, une seule boucle for parcourant chaque ligne du tableau avec pour chaque ligne un traitement en temps linéaire $\Theta(1)$, alors que le remplissage est en $\Theta(nm)$ où m est la capacité du problème. Autrement dit, la complexité du remplissage correspond à la taille du tableau, qui est effectivement de $n*m$. Trouver la valeur à chaque emplacement se fait à l'aide d'opération élémentaires en temps linéaire $\Theta(1)$. La complexité de l'algorithme de la programmation dynamique implémenté est donc de :

$$\Theta(nm)$$

.

3 Heuristique d'amélioration locale

Le code c++ utilisé pour l'algorithme d'heuristique d'amélioration locale est le suivant:

```
int findRevenue(const Solution & solution) {
        int revenue = 0;
        for (unsigned int i = 0; i < solution.restos.size(); i++) {
                revenue += solution.restos[i].r;
        }
        return revenue;
}

vector<Resto> findUnusedResto(const Problem & problem, const
    Solution & solution) {

        vector<int> sortedIds;
        for (unsigned int i = 0; i < solution.restos.size(); i++) {
                sortedIds.push_back(solution.restos[i].id);
        }

        // Sort the ids
        sort(sortedIds.begin(), sortedIds.end());
        // Parse ids backwards to delete the right resto in unused
            restos
        vector<Resto> unusedRestos = problem.restos;
        for (int i = sortedIds.size() - 1; i >= 0; i--) {
                unusedRestos.erase(unusedRestos.begin() + sortedIds
                    [i] - 1);
        }
        return unusedRestos;
}

Solution resolveHeu(Problem problem) {
chrono::high_resolution_clock::time_point start = std::chrono::
    high_resolution_clock::now();
chrono::high_resolution_clock::time_point finish;
std::chrono::duration<double> elapsed;

Solution solution = resolveGlouton(problem);
int currentRevenue = findRevenue(solution);

bool shouldContinue = true;
vector<Resto> unusedRestos = findUnusedResto(problem, solution);
while (shouldContinue) {
shouldContinue = false;
bool isTimerLow = true;
// Try 1-2 to 1-2 swaps for each possible combination
for (unsigned int i = 0; i < solution.restos.size() && isTimerLow;
    i++ ) {
for (unsigned int j = 0; j < unusedRestos.size() && isTimerLow; j++
     ) {
for (unsigned int k = 0; k < solution.restos.size() && isTimerLow;
    k++ ) {
if (k != i) {
for (unsigned int l = 0; l < unusedRestos.size() && isTimerLow; l
    ++) {
```

```
if (k >= solution.restos.size() || j >= unusedRestos.size()) {
break;
}
if (l != j) {

Resto newResto_j = unusedRestos[j];
Resto newResto_l = unusedRestos[l];
Resto curResto_i = solution.restos[i];
Resto curResto_k = solution.restos[k];

unsigned int oneToOneRevenue = currentRevenue - curResto_i.r +
    newResto_j.r;
unsigned int oneToTwoRevenue = currentRevenue - curResto_i.r +
    newResto_j.r + newResto_l.r;
unsigned int twoToOneRevenue = currentRevenue - curResto_i.r -
    curResto_k.r + newResto_j.r;
unsigned int twoToTwoRevenue = currentRevenue - curResto_i.r -
    curResto_k.r + newResto_j.r + newResto_l.r;

unsigned int oneToOneCapacity = solution.totalChickens - curResto_i
    .q + newResto_j.q;
unsigned int oneToTwoCapacity = solution.totalChickens - curResto_i
    .q + newResto_j.q + newResto_l.q;
unsigned int twoToOneCapacity = solution.totalChickens - curResto_i
    .q - curResto_k.q + newResto_j.q;
unsigned int twoToTwoCapacity = solution.totalChickens - curResto_i
    .q - curResto_k.q + newResto_j.q + newResto_l.q;

vector<unsigned int> potentialRevenues = { oneToOneRevenue,
    oneToTwoRevenue, twoToOneRevenue, twoToTwoRevenue };
sort(potentialRevenues.begin(), potentialRevenues.end());

for (int r = potentialRevenues.size() - 1; r >= 0; r--) {
if (potentialRevenues[r] <= currentRevenue) {
break;
}
if (potentialRevenues[r] == oneToOneRevenue && oneToOneCapacity <=
    problem.capacity) { //oneToOne
swap(solution.restos[i], unusedRestos[j]);
currentRevenue = oneToOneRevenue;
solution.totalChickens = oneToOneCapacity;
shouldContinue = true;
break;
}
else if (potentialRevenues[r] == oneToTwoRevenue &&
    oneToTwoCapacity <= problem.capacity) { //oneToTwo
swap(solution.restos[i], unusedRestos[j]);
solution.restos.push_back(newResto_l);
unusedRestos.erase(unusedRestos.begin() + l);
currentRevenue = oneToTwoRevenue;
solution.totalChickens = oneToTwoCapacity;
shouldContinue = true;
break;
}
else if (potentialRevenues[r] == twoToOneRevenue &&
    twoToOneCapacity <= problem.capacity) { //twoToOne
swap(solution.restos[i], unusedRestos[j]);
```

```
unusedRestos.push_back(curResto_k);
solution.restos.erase(solution.restos.begin() + k);
currentRevenue = twoToOneRevenue;
solution.totalChickens = twoToOneCapacity;
shouldContinue = true;
break;
}
else if (potentialRevenues[r] == twoToTwoRevenue &&
    twoToTwoCapacity <= problem.capacity) { // twoToTwo
swap(solution.restos[i], unusedRestos[j]);
swap(solution.restos[k], unusedRestos[l]);
currentRevenue = twoToTwoRevenue;
solution.totalChickens = twoToTwoCapacity;
shouldContinue = true;
break;
}
}
// Contrainte de temps
finish = std::chrono::high_resolution_clock::now();
elapsed = finish - start;
if (elapsed.count() * 1000 >
    MAXIMUM_LOCAL_HEURISTIC_SEARCH_TIME_IN_MS) {
shouldContinue = false;
isTimerLow = false;
break;
}
}
}
}
}
}
}

}

solution.elapsedTime = elapsed.count() * 1000;
return solution;
}
```

(Les indentations ont dû être supprimées pour des questions de visibilité. Pour une meilleure visualisation du code, aller voir le fichier main.cpp directement.)

D'abord, remarquons que dans l'implémentation présente, les fonctions

```
findRevenue(solution)
```

et

```
findUnusedResto(problem, solution)
```

sont appelées avant la recherche, en plus de l'algorithme glouton appelé au départ. On peut déjà conclure que l'appel à la fonction gloutonne se fait dans l'ordre de $\Theta(nlog(n))$ considérant l'analyse faite précédemment.

En premier lieu, analysons la complexité des deux fonctions appelées au départ. FindRevenue se fait dans l'ordre de $\Theta(n_0)$, avec n_0 étant le nombre d'emplacements dans une solution. FindUnusedResto contient une boucle en $\Theta(n_0)$, un sort

qu'on peut imaginer en $\Theta(n_0 log(n_0))$, et une dernière boucle en $\Theta(n_0)$ (en supposant que la suppression d'un élément dans un vecteur se fait en $\Theta(1)$). FindUnusedResto se fait donc en $\Theta(n_0 log(n_0))$.

Ensuite, analysons la complexité des boucles imbriquées qui constituent la recherche de solution dans un voisinage. La première et la troisième boucles for itèrent sur les emplacements retenus par la solution courante (n_0), et la deuxième et quatrième parcourent les emplacements qui ne sont pas dans la solution (n_1). Il est à noter que l'union de ces deux ensembles forment la totalité des emplacements possibles, soit $n_0 + n_1 = n$. Ces valeurs (n_0 et n_1) peuvent changer au fil de l'exécution de l'algorithme, en raison des échanges 1 à 2 et 2 à 1. En pire cas, on a que $n_0 = n_1$ donc une complexité de $n/2 * n/2 * n/2 * n/2 = n^4/2$ ce qui donne $\Theta(n^4)$. Cela dépend de la capacité du problème et des coûts des emplacements choisis. Ces 4 boucles se retrouvent dans une boucle while, qui continue tant qu'une meilleure solution a été trouvée précédemment par la recherche de solution. En pire cas, une nouvelle solution pourrait être trouvée pendant un grand nombre d'itérations, et il n'est pas évident de préciser cette valeur: cela dépend également de la solution de départ. Cela dépend de la grandeur du voisinage. En faisant des échanges de 1 à 1, 2 à 1, 1 à 2 et 2 à 2, on peut imaginer que la taille du voisinage est relativement grande, mais cela dépend ensuite de la valeur des revenus des emplacements à un problème spécifique. Il est alors difficile de bien cerner le nombre d'itérations faites par la boucle while. On peut représenter la valeur inconnue du voisinage par la variable V, qu'on assume relativement élevée en raison des 4 échanges possibles. V viendrait alors augmenter la complexité déjà obtenue de $\Theta(n^4)$.

Finalement, les complexités de l'appel à la fonction gloutonne et celles des fonctions de départ ne sont pas aussi importante que celle de la recherche locale. La complexité globale de cet algorithme sera donc autour de:

$$\Theta(Vn^4)$$