I. Préliminaires

Implémentation des algorithmes de flot maximum

L'algorithme d'Edmonds-Karp est central dans ce projet. Le pseudo-code proposé dans le polycopié ne nous convenait pas car il ne marche que pour les graphes *strictement orientés*, à savoir les graphes orientés $G = (V, E)$ tels que $\forall (u, v) \in E$, $(v, u) \notin E$. En effet, si on applique l'algorithme sur un graphe $G = (V, E)$ tel qu'il existe $e = (u, v) \in E$ tel que $e^T = (v, u) \in E$, et si à une étape donnée de l'algorithme $f(e) < c(e)$ et $0 < f(e^T)$, alors l'algorithme du polycopié veut que dans le graphe résiduel on ait en même temps $c'(e) = c(e) - f(e)$ (car $f(e) < c(e)$) et $c'(e) = f(e^T)$ (car $f(e^T) > 0$). Cela ne poserait pas de problème si à chaque itération on avait un invariant de boucle du type $f(e)+f(e^T) = c(e)$, mais cela n'est même pas vérifié au début de l'algorithme.

La solution consiste à "dédoubler" les sommets du graphe de départ, en construisant le *digraphe* associé à ce graphe. Plus précisément soit $G = (V, E)$ le graphe orienté de départ, et G' le graphe orienté obtenu à partir de G de la manière suivante : on va "dédoubler" chaque $v \in V$ en deux nouveaux nœuds v_1 et v_2. v_1 va être un nœud d'entrée vers lequel vont pointer toutes les arêtes de G arrivant vers v (avec la même capacité que dans G), et v_2 sera un nœud de sortie, duquel vont sortir toutes les arêtes de G sortant de v (avec les mêmes capacités que dans G). On rajoutera une arête de v_1 vers v_2 avec une capacité *égale à* $+\infty$. Voici un dessin illustrant ce principe :

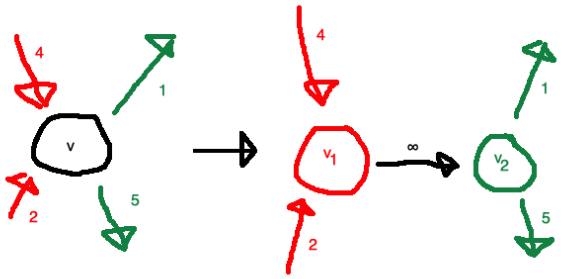

FIGURE 1 – Construction du digraphe

Il est clair que ce procédé permet de toujours se ramener à des graphes *strictement orientés*, tout en conservant la valeur du flux maximal. Par ailleurs, on peut retrouver les valeurs du flot sur chaque arête en allant chercher les valeurs du flot obtenu sur les arêtes qui existaient déjà dans le graphe de départ.

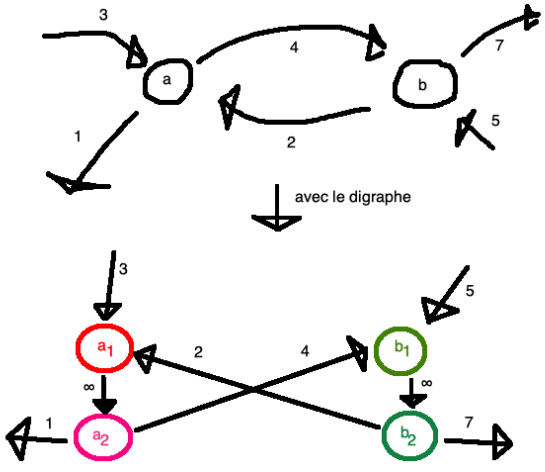

FIGURE 2 – Le graphe devient strictement orienté

Voici le pseudo-code de l'algorithme de Ford-Fulkerson, tel qu'exposé sur Wikipédia en anglais :

code 1 – Algorithme d'Edmonds-Karp

```
Edmonds-Karp(G){
    NOTATIONS: Pour un flot donné, G_f est le graphe résiduel
               et c_f la capacité résiduelle (∀(u,v) ∈ V², c_f(u,v) = c(u,v) − f(u,v)) .
    ENTREE: un graphe G=(V,E) avec capacités c, source s et puits t
    SORTIE: un flot de valeur maximale
    int[][] f est intialisé à 0;
    Tant qu'il existe un chemin augmentant P de s vers t dans G_f (BFS){
        c_f(P) = min{c_f(u,v) ; (u,v) ∈ P};
        Pour chaque (u,v) ∈ P{
            1. f(u,v) ⟵ f(u,v) + c_f(P) //on envoie du flot.
            2. f(v,u) ⟵ f(v,u) − c_f(P) //le flot peut "faire marche arrière"
                                        //ultérieurement.
        }
    }

}
```

Ce pseudo-code produit des matrices antisymétriques, ce qui peut s'interpréter en considérant qu'envoyer un flux de x de u à v revient à envoyer un flot de $-x$ de v à u. Cette interprétation n'est possible que dans le cas des graphes *strictement orientés*, car sinon on aurait affaire à des flots négatifs sur de vraies arêtes et car cela reviendrait à considérer que l'on fait "revenir" les flots qu'on a "envoyés", d'où un flot total nul. Mais le *digraphe* a permis de contourner cette difficulté, car désormais on applique l'algorithme à des graphes *strictement orientés*. Ainsi, à la fin on peut juste garder les valeurs positives de la matrice de flot trouvée par l'algorithme.

Notons qu'au niveau de l'implémentation, nous avons utilisé des matrices d'adjacence pour représenter les flots et les capacités. Ainsi, pour un graphe *strictement orienté* avec n sommets et m arêtes, notre implémentation de BFS se fait en $O(n^2)$: lorsqu'on effectue cette recherche, on visite au plus les n sommets du graphe considéré, et pour chacun d'entre eux on parcourt toutes les colonnes (n au total) de la ligne correspondante pour savoir vers où on peut aller à partir de ce sommet.

Ainsi, le nombre d'itérations de l'algorithme étant en $O(mn)$ d'après le raisonnement d'Edmonds-Karp, la complexité de notre implémentation de l'algorithme du flot maximum est en $O(mn^3)$.

Or, on applique cet algorithme sur le digraphe, qui a $2n$ sommets et $m+n$ arêtes. Ainsi, la complexité est en $O((m+n)n^3$ et comme ici $m \geqslant \frac{n}{2}$ et donc $n = O(m)$, la complexité de l'algorithme reste en $O(mn^3)$.

Modélisation

Un aspect important de la modélisation doit être précisé : sur quel graphe va-t-on appliquer les algorithmes de flot maximum ? Ici, il s'agira du graphe de couverture ("coverage" graph) défini dans l'énoncé. Or, dans la définition donnée dans l'énoncé, on précise seulement à quelle condition il existe une arête entre deux sommets de ce graphe, sans préciser quelle est la valeur de la capacité correspondante. On définit les capacités du graphe de couverture de la manière suivante :

- Pour deux senseurs i et j tels que la distance entre les deux est inférieure à 2R, on définit la capacité de l'arête (i,j) comme le minimum des temps de vie des senseurs (i.e. $\min(c(i), c(j))$ avec les notations de l'énoncé). Cela a un sens, car ces deux capteurs seront en mesure d'assurer la couverture de l'espace entre les deux tant que *les deux* n'auront pas épuisé leur temps de vie.
- Pour un senseur i à une distance du bord de gauche inférieure à R, l'arête (s,i) aura une capacité égale au temps de vie de ce senseur ($c(i)$ avec les notations de l'énoncé).
- Pour un senseur i à une distance du bord de droite inférieure à R, l'arête (i,t) aura une capacité égale au temps de vie de ce senseur ($c(i)$ avec les notations de l'énoncé).

II. Questions

Q1. L'idée est la suivante : on va se ramener au problème du flot maximum, mais sur un graphe légèrement modifié : il s'agira d'une sorte de *digraphe unitaire*. Soit $G = (V, E)$ le graphe orienté de départ, et G' le graphe orienté obtenu à partir de G de la manière suivante : on va "dédoubler" chaque $v \in V$ en deux nouveaux nœuds v_1 et v_2. v_1 va être un nœud d'entrée vers lequel vont pointer toutes les arêtes de G arrivant vers v (avec la même capacité que dans G), et v_2 sera un nœud de sortie, duquel vont sortir toutes les arêtes de G sortant de v (avec les mêmes capacités que dans G). On rajoutera une arête de v_1 vers v_2 avec une capacité *égale à 1* (le digraphe défini précédemment avait des capacités infinies à ces endroits).

La capacité unitaire de l'arête (v_1, v_2) fera que passer par cette arête dans G' sera équivalent à passer une seule fois par le nœud v dans le graphe original. Ainsi, on garantit des chemins aux nœuds disjoints, et la résolution du problème du flot maximal sur G' fournit le nombre maximal de chemins aux nœuds disjoints dans G.

Q2. Notre implémentation d'Edmonds-Karp a une complexité en $O(mn^3)$ où $m = \text{Card}(E)$ et $n = \text{Card}(V)$. Or, on applique ici cet algorithme à un graphe qui a $2n$ sommets et $m + n$ arêtes. Par le même argument que précédemment, la complexité de notre algorithme est donc en $O(mn^3)$

Q3. On note $t = \dfrac{M}{K}$. Il suffit d'utiliser les K premiers chemins aux nœuds disjoints, puis les K suivants, puis les K d'après et cela t fois. Plus formellement, notons $(p_i)_{i \in [\![1,M]\!]}$ M chemins aux nœuds disjoints dans G. Pour chaque instant $i \in [\![1, t]\!]$, on va utiliser les K chemins $(p_{Ki}, ..., p_{K(i+1)-1})$.

Q4. Notre algorithme donne les résultats suivants (les senseurs sont numérotés de 0 à 29, 30 est la source et 31 le puits) :

Temps CPU : 5 ms

Chemins : [[30, 3, 1, 7, 31], [30, 4, 2, 9, 31], [30, 6, 11, 15, 31], [30, 17, 12, 26, 31], [30, 18, 10, 16, 31], [30, 19, 25, 29, 31], [30, 21, 13, 28, 31]]

Temps de vie total : 2

Or, ici, K = 3, et donc on pourra assurer une 3-couverture pendant uniquement 2 unités de temps. Ainsi, un chemin ne sera pas utilisé. Voici le "scheduling" des senseurs.

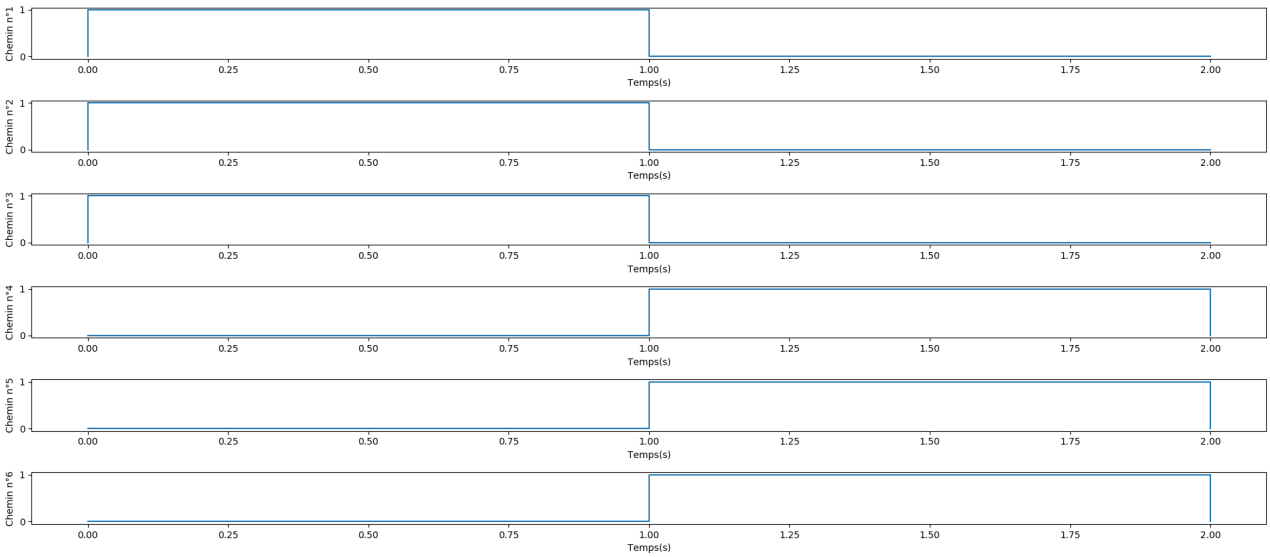

FIGURE 3 – "Scheduling" de la question 4

Q5. Notons $c_{\max} = \max_{e \in E} c(e)$. Il est clair que pour $p \geqslant c_{\max}$, le graphe G^p est égal à G et donc $F(p) = F(c_{max})$ et donc on peut se limiter à des valeurs de p inférieures ou égales à c_{max}. Ainsi, $\lim_{p \longrightarrow \infty} \Psi(p) = -\infty$. De plus, comme F est croissante en p, trouver $\max_{p \in \mathbb{N}} \Big(F(p); \Psi(p) \geqslant 0 \Big)$ revient à trouver le p maximal tel que $\Psi(p) \geqslant 0$, et à en calculer l'image par F. Comme pour $p \geqslant c_{\max}$, le graphe $G^p = G$, le p cherché est entre 0 et $c_{\max}$. On peut donc chercher p par dichotomie. Ce sera le p juste avant que Ψ bascule de signe.

code 2 – Obtention du K-flot maximal de G

```
MaxKrouteFlow(G){
    int gauche=0,centre=cMax/2,droite=cMax;
    int[][] gCentre=Gp(centre);
    int[][] f = null;
    int v=0;
    while(droite-gauche>1){ //Tant qu'on ne peut pas choisir
        f=maxFlow(gCentre);
        v=value(f)-K*centre;
        if(v>=0){    //Ψ(centre) est trop grande
            gauche=centre; //La valeur recherchée se trouve
                           //donc plus à droite : on se décale
            centre+=(droite-gauche)/2;
        }
        else{    //Ψ(centre) est trop faible
            droite=centre;  //La valeur recherchée se trouve
                            //donc plus à gauche : on se décale
            centre-=(droite-gauche)/2;
        }
        gCentre=Gp(centre);
    }
    int p;
    int vf=value(maxFlow(Gp(droite)));
    v=vf-K*droite;
    if(v>=0)p=droite;    //Si on est encore >= 0 à droite on prend le flot
                         //de droite car F(p) est croissant
    else p=gauche; //Sinon on doit prendre le flot de gauche
    return maxFlowWithBound(Gp(p),K*(vf/K));//On retourne le flot maximal,
    // mais il faut un flot multiple de K, d'où l'appel à
    // la fonction maxFlowWithBound, qui retourne un flot
    // de valeur K(v_f/K) où / désigne la division euclidienne.
}
```

Remarque :
MaxFlowWithBounds(G,bound) : renvoie un flot de valeur min(bound, f_{max}) où f_{max} est la valeur du flot maximal de G. Pour implémenter cette fonction, il suffit d'ajouter une nouvelle source et de relier cette nouvelle source à l'ancienne source avec une capacité égale au bound.

Q6. La recherche par dichotomie fait qu'on appliquera Edmonds-Karp un nombre de fois en $O(\log(c_{\max}))$. A chaque fois, on appliquera Edmonds-Karp sur un graphe avec n sommets et m arêtes. Ainsi, la complexité totale est en $O\Big(\log(c_{\max}) mn^3\Big)$.

Q7. Le lemme 1 nous invite à nous ramener à un problème de flot maximal dans un nouveau graphe G′ obtenu à partir du graphe original G. En effet, d'après le lemme 1, si f' est un flot saturant (et donc maximal) de G′, alors $f = f'_{|E} + d$ est un flot faisable de G. Ainsi, on a envie de chercher un flot saturant dans G′ pour en dégager un flot faisable dans G. Hélas, on ne sait pas si un flot faisable dans G est possible lorsque G′ n'admet pas de flot saturant. On va montrer dans le paragraphe qui suit un résultat qui rémédie à cela, à savoir que si G′ n'admet pas de flot saturant, alors G n'admet pas de flot faisable (on aura ainsi montré que trouver un flot faisable dans G équivaut à trouver un flot saturant dans G′. Il suffira ensuite de chercher un flot maximal dans G′ et de tester s'il est saturant (car dès qu'un flot saturant existe, tout flot maximal est saturant lui aussi).

Lemme 1

On reprend les notations de l'énoncé. On va montrer que si G admet un flot faisable, alors G′ admet un flot saturant (la contraposée de ce résultat correspond au résultat qu'on a annoncé).

Preuve :

Supposons que G admette un flot faisable f. Alors le flot :

$$\begin{cases} \forall e \in \mathrm{E}, & f'(e) = f(e) - d(e) \\ \forall v \in \mathrm{V} & f'(s', v) = \sum\limits_{u \in \mathrm{V}} d(u, v) \\ \forall v \in \mathrm{V} & f'(v, t') = \sum\limits_{u \in \mathrm{V}} d(v, u) \end{cases}$$

est saturant dans G′, et cela conclut la démonstration.
Ainsi, il suffit d'appliquer l'algorithme d'Edmonds-Karp au graphe G′, puis prendre la restriction du flot trouvé à G, et lui rajouter la demande d.
Edmonds-Karp appliqué à G′ a une complexité en $\mathrm{O}\big((m+2n+1)(n+2)^3\big) = \mathrm{O}\big(mn^3\big)$. Le calcul du nouveau flot se fera alors en m étapes. La complexité est donc en $\mathrm{O}\big(mn^3\big)$.

Q8. L'idée est de partir du flot faisable f et non pas du flot nul, puis d'appliquer Edmonds-Karp avec le graphe résiduel G_f défini par les capacités suivantes :

$$\forall (u, v) \in \mathrm{V}^2,\ c'_f(u, v) = \begin{cases} c(u, v) - f(u, v) & \text{si } (u, v) \in \mathrm{E} \\ f(v, u) - d(v, u) & \text{si } (v, u) \in \mathrm{E} \\ 0 & \text{sinon} \end{cases}.$$

Mais remarquons que pour que cette définition soit non-ambiguë, il nous faut avoir un graphe *strictement orienté*, donc on appliquera cet algorithme au digraphe associé à G.

Q9. Le lemme 5 suggère une fonction récursive dont les arguments sont un flot f et un majorant des capacités ν. Cette fonction utilise à chaque fois le lemme 4 pour dégager un K-flot élémentaire g, et le lemme 5 suggère qu'on peut l'extraire avec un facteur Δ, tel que défini dans ce lemme. Notre fonction se réexécute alors avec les arguments $(f - \Delta g, \nu - \Delta)$, et cela jusqu'à ce que ν devienne nul. À chaque fois, le K-flow élémentaire g est une solution au problème du flot faisable avec demandes défini dans la démonstration du lemme 4, avec la contrainte supplémentaire que ce flot soit égal à K. Pour le trouver, on part de l'algorithme de la question 7 et on implémente une fonction MaxFlowWithDemandsAndBound qui remplit vis-à-vis de la fonction MaxFlowWithDemands le même rôle que la fonction MaxFlowWithBound (définie à la question 5) remplit vis-à-vis de la fonction MaxFlow.

Il ne reste alors qu'à déterminer les arguments de base sur lesquels on va lancer notre fonction récursive. Soit f le K-Route -flot maximal de G. On écrit $f = \sum\limits_{i \in \mathrm{I}} w_i g_i$ où les g_i sont des K-flots élémentaires et les w_i sont leurs coefficients entiers. On va appliquer notre fonction récursive sur $\Big(f, \sum\limits_{i \in \mathrm{I}} w_i\Big)$. Voici le pseudo-code de l'algorithme.

code 3 – Décomposition du K-flot maximal de G

```
public DecompositionRecursive(G,f,ν,resu){
    if (ν == 0) return resu;
    int[][] g = MaxFlowWithDemandsAndBound(G,⌊f/ν⌋,⌈f/ν⌉);
    int Δ = ν;
    for (int i = 0;i<n;i++){
        for (int j = 0; j<n;j++){
            if (g[i][j]=0) Δ = min(ν−f[i][j],Δ);
            if (g[i][j]=1) Δ = min(f[i][j],Δ);
    }
    if (resu.get(g)==null) resu.put(g,Δ);
    else  resu.put(g,Δ+resu.get(g));
    return DecompositionRecursive(G,f−Δg,ν − Δ,resu);
}
public Decomposition(G,K){
    int[][] f = KRouteFlotMaximal(G,K);
    int nu = Val(f)/K;
    // Val(f) donne la valeur du flot f.
    // Val(f) = K∑_{i∈I} w_i
    return DecompositionRecursive(G,f,nu,[]) ;
}
```

Ici, la fonction renvoie un dictionaire dont les clés sont les flots élémentaires et les valeurs sont leurs coefficients dans la somme du K-route flow. Notons qu'on n'a pas de garantie que Δ soit le coefficient de g dans la somme déjà évoquée. C'est pour ça que la représentation de dictionnaire est utile : si au cours de l'algorithme on retombe sur le même flot élémentaire g, alors il suffit d'actualiser le coefficient qui lui est associé dans le dictionnaire resu.

Pour des raisons de commodité, on a implémenté cet algorithme de manière itérative (le principe restant le même). De même, plutôt que de renvoyer un dictionnaire de clés les K-flots élémentaires et de valeurs leurs coefficients dans la décomposition, on renvoie la liste des Δg tels que calculés dans l'algorithme. Certains K-flots élémentaires peuvent se répéter, mais ce problème est adressé dans notre implémentation de la question 11.

Q10. Il est clair que l'on fait au plus $\nu = \sum_{i\in I} w_i$ appels récursifs. Ou encore, en notant MKRF(G) le K-route flow maximal associé à G, on fait au plus $\frac{|\text{MKRF(G)}|}{\text{K}}$ appels récursifs. À chaque appel récursif, on fait une double boucle et un appel à FlotFaisable (de complexité en $O(mn^3)$), et donc, en supposant que l'accès aux valeurs du dictionnaire se fait en temps constant, on a une complexité en $O\left(\frac{|\text{MKRF(G)}|mn^3}{\text{K}}\right)$.

Q11. Il suffit d'appliquer l'algorithme précedent sur notre "coverage graph" G. Cela nous donne une liste de K-flots élémentaires g avec leurs coefficients w_g dans la décomposition du K-route flow maximal de G. Pour chaque K-flow élémentaire g, la durée de vie de la K-configuration couvrante associée (à savoir l'ensemble de senseurs par lesquels passent les K chemins aux sommets disjoints constituant g, et qui fournissent un "K-coverage") est précisément w_g. Ainsi, l'algorithme suivant répond à la question :

code 4 – Scheduling

```
Scheduling(N,R,c,K,A){
    time = 0;
    HashMap<LinkedList<Integer>,LinkedList<Integer>> map;
    // Un dictionnaire dont les clés sont des chemins disjoints de senseurs
    // et les valeurs sont les listes des instants où ces chemins sont actifs
    ArrayList<int[][]> decomposition=decomposition(MaxKrouteFlow(G),K);
    Pour chaque élément g de l'ArrayList decomposition{
        1. Pour chaque chemin constituant g, on ajoute
           (time,time+1,...,time+w_g−1) à la liste
           des dates pendant lesquelles il est actif.
        2. On actualise la valeur de time: time+=w_g;
    }
    On retourne map et time (time étant à la fin la durée de vie du réseau);
}
```

Q12. Chemins et temps d'allumage (le format est : chemin=>liste des temps d'allumage) :

[30, 4, 2, 29, 31]=>[0], [30, 17, 11, 5, 16, 31]=>[0, 1, 2], [30, 18, 12, 0, 16, 31]=>[7, 8, 9], [30, 17, 11, 14, 28, 31]=>[3], [30, 6, 3, 2, 29, 31]=>[1, 2], [30, 17, 11, 13, 16, 31]=>[3], [30, 18, 11, 1, 9, 31]=>[4, 5, 6], [30, 17, 11, 1, 9, 31]=>[0, 1, 2, 3], [30, 18, 11, 13, 16, 31]=>[4, 5, 6], [30, 17, 3, 2, 29, 31]=>[4, 5, 6, 7, 8, 9], [30, 19, 11, 5, 9, 31]=>[7, 8, 9]

Temps de vie total : 10

Temps CPU : 16 ms

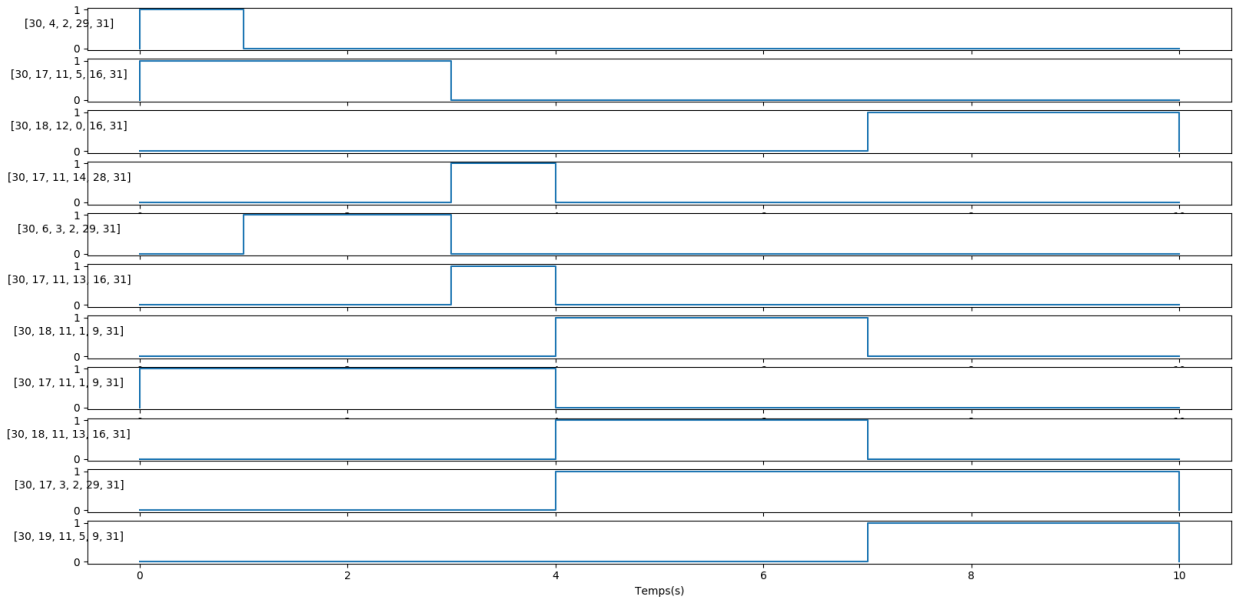

FIGURE 4 – "Scheduling" de la question 12