Séparation et évaluation pour la planification de visites aux urgences

Mathias LOMMEL

Lucas OFFROY

4MA

I/ INTRODUCTION

Dans le cadre de ce projet, comme annoncé ci-dessus, nous allons considérer le problème de planification de visites aux urgences. Nous pourrons ainsi observer 3 majeures parties :

- L'implémentation de 4 algorithmes permettant la résolution de ce problème
- L'analyse et la comparaison des performances de ces algorithmes
- L'analyse d'exemples particuliers

Ce projet nous permettra ainsi de mettre en application les notions théoriques étudiées en cours, mais aussi de prendre du recul sur nos implémentations, et se questionner sur l'aspect éthique des résultats.

```
import Pkg;
Pkg.add("GLPK")
Pkg.add("Cbc")
Pkg.add("HiGHS")
Pkg.add("JuMP")
Pkg.add("Plots")
using GLPK
using Cbc
using HiGHS
using JuMP
using Plots
```

```
    Updating registry at `C:\Users\lucas\.julia\registries\
General.toml`
   Resolving package versions...
  No Changes to `C:\Users\lucas\.julia\environments\v1.9\Project.toml`
  No Changes to `C:\Users\lucas\.julia\environments\v1.9\
Manifest.toml`
   Resolving package versions...
  No Changes to `C:\Users\lucas\.julia\environments\v1.9\Project.toml`
  No Changes to `C:\Users\lucas\.julia\environments\v1.9\
Manifest.toml`
   Resolving package versions...
```

```
  No Changes to `C:\Users\lucas\.julia\environments\v1.9\Project.toml`
  No Changes to `C:\Users\lucas\.julia\environments\v1.9\
Manifest.toml`
   Resolving package versions...
  No Changes to `C:\Users\lucas\.julia\environments\v1.9\Project.toml`
  No Changes to `C:\Users\lucas\.julia\environments\v1.9\
Manifest.toml`
   Resolving package versions...
  No Changes to `C:\Users\lucas\.julia\environments\v1.9\Project.toml`
  No Changes to `C:\Users\lucas\.julia\environments\v1.9\
Manifest.toml`
```

II / Implémentation des algorithmes

1 - Algorithmes de Branch and Bound

Dans cette première partie, nous allons nous pencher sur les deux premiers algorithmes de Branch and Bound, vus en TD. A cet effet, nous allons commencer par définir quelques fonctions qui nous seront utiles pour ces deux algorithmes : *estimate_sup_init* et *compute_penalties*.

```
function estimate_sup_init(P,D,H)
    """
    Inputs :
        P : vecteur (nx1) des pénalités associées au retard de chaque
patient
        D : vecteur (nx1) des durées de rdv de chaque patient
        H : vecteur (nx1) des horaires de passage au plus tard de
chaque patient
    Outputs :
        z_sup : borne supérieure du problème
        order : ordre de passage associé à cette borne (vecteur nx1)
    """

    # Stratégie choisie : prendre les ratios H/P croissants

    # Tri des temps en fonction du critère des ratios H/P croissants
    ratios=H./P
    order=sortperm(ratios)

    # On décale les valeurs aux temps de commencement (ajout de 0 au
debut et on retire la dernière valeur)
    temps_début_traitement_apres_patient=[0;D[order]]
    pop!(temps_début_traitement_apres_patient)

    # On fait la somme cumulée pour avoir les temps de début de
passage
```

```
temps_début_traitement=cumsum(temps_début_traitement_apres_patient)

    # Première borne supérieure estimée
    z_sup=compute_penalties(P,H,temps_début_traitement,order)

    return [z_sup,order]
end

estimate_sup_init (generic function with 1 method)
```

La fonction **estimate_sup_init** estime la première borne primale du problème selon le critère des ratios (Temps maximal avant pénalité)/(Pénalités) croissants.

Ce raisonnment fait donc passer en priorité les patients qui ont une forte pénalité et un faible temps d'attente avant d'être pénalisé. Ces patients semblent en effet être à l'origine d'une forte pénalité si ils ne sont pas traités dans les premiers, ce qui justifie ce choix.

Cette première estimation est très importante notamment dans la version 2 de l'algorithme, car elle va permettre d'économiser un temps précieux de calcul. En effet, la conaissance d'une bonne borne primale permet de ne pas descendre jusqu'au bout de l'arbre, grâce à la stratégie d'élaguage.

1.1 : Première version

→ Fonction *compute_penalties*

Cette fonction permet de calculer les pénalités en prenant en paramètre les vecteurs de pénalités *P*, de délais *H*, de durées de début de traitement *t*, et d'ordre provisoire *order*.

Cette fonction doit permettre de calculer les pénalités même si nous ne sommes pas dans la profondeur n de l'arbre, c'est-à-dire que nous n'avons pas encore nécessairement considéré la totalité des patients dans notre vecteur *order*. Pour la suite, on normalisera le fait de mettre 0 dans les cases du tableau indicés par des nombres supérieurs au noeud en cours de traitement.

Par exemple, dans un cas de 4 patients, si on regarde le noeud de profondeur 2 où le patient 3 est traité en premier et le patient 4 est traité en 2ème, on aura $order := [3\,4\,0\,0]$. Notre fonction ne prendra donc pas en compte les zéros finaux pour calculer la pénalité (z_tilde) à ce noeud (avec la commande *order = trunc.(Int,order[order.!=0])*).

```
function compute_penalties(P,H,t,order)
    """
    Inputs :
        P : vecteur (nx1) des pénalités associées au retard de chaque
patient
        H : vecteur (nx1) des horaires de passage au plus tard de
chaque patient
        t : vecteur (nx1) des durées de traitement
        order : vecteur (nx1) donnant l'ordre de passage des patients
    Output :
        sum(res) : pénalité de l'ordre donné en paramètre
```

```
    """
    # On trie les zeros finaux si profondeur != n
    order = trunc.(Int,order[order.!=0])

    # Vecteur t associé
    t=trunc.(Int,t[1:length(order)])
    # Vecteur des pénalités associé
    P1=P[order]
    # Vecteur des délais associé
    H1=H[order]

    # Calcul du vecteur des pénalités
    res=P1.*(t-H1)
    # On retire les valeurs négatives
    res[res.<0].=0

    # On retourne la somme de toutes les pénalités
    return sum(res)
end
```

```
compute_penalties (generic function with 1 method)
```

Le vecteur *t* sera composé des durées de début de traitement, soit des durées cumulées des délais des patients traités avant. Dans l'exemple précédent, en appelant D_3 et D_4 les temps de traitement des patients 3 et 4 respectivement, *t*:=[0, D_3, D_3+D_4, D_3+D_4]. Ici, les informations intéressantes pour calculer les pénalités avec l'ordre donné sont les 2 premières cellules. Pour calculer les pénalités vectoriellement et optimiser le temps de calcul, on ne considèrera seulement ces premières cellules (avec la commande *t=trunc.(Int,t[1:length(order)])*).

La taille du vecteur *order* (resp *t*) sera donc fixe de dimension *n*. Ce choix de non-recours à l'allocation dynamique est justifié par le fait que les dimensions de ce vecteur resteront petites. En effet, le nombre de patients considérés ne nécessite pas recours à l'allocation dynamique, (et serait peut être même une perte de temps au vu du nombre d'opérations nécessaires sur le vecteur *order* (resp *t*).

Ensuite, on filtre P et H avec *order* pour récupérer vectoriellement les pénalités et délais dans l'ordre, et calculons vectoriellement la formule de pénalités donnée dans l'énoncé : $P.(t-H)$. Enfin, il ne faut pas oublier de considérer seulement les valeurs positives avec la commande *res[res.<0].=0*.

Nous avons donc, à cette étape, le vecteur de pénalités associé à chaque patient passé dans l'ordre donné. Nous retournons donc la somme de ces pénalités pour avoir la pénalité totale calculée au noeud actuellement considéré (z_tilde).

→ Algorithme *Branch_Bound_v1*

Nous avons construit les deux fonctions **compute_penalties** et **estimate_sup_init**, nous pouvons donc désormais construire le premier algorithme de Branch and Bound, basé sur le premier algorithme de B&B vu en TD.

```
function Branch_Bound_v1(P,D,H)
    """
    Inputs :
        P : vecteur (nx1) des pénalités associées au retard de chaque
patient
        D : vecteur (nx1) des durées de rdv de chaque patient
        H : vecteur (nx1) des horaires de passage au plus tard de
chaque patient
    Outputs :
        z_barre : solution optimale du problème (pénalité minimale)
        ordre : ordre de passage associé (vecteur nx1)
    """

# Estimation initiale d'une borne primale, qui est renvoyée si égale à
0 car on serait dans le cas d'une optimalité
    n = length(D)
    res = estimate_sup_init(P,D,H)
    z_barre = res[1]
    ordre = res[2]
    if z_barre == 0
        return estimate_sup_init(P,D,H)
    end

# Initialisation des objet nécessaires
    penalties=0                             # Pas de pénalités au début
    temps_traitement=zeros(n)               # Pas de patients traités
au début
    ordre_actuel=zeros(n)
    profondeur_actuelle=1                   # On se place à la
profondeur 1 au début

# Boucle pour avancer dans l'arbre
    while true

        # On fait passer le patient suivant à la profondeur courante

ordre_actuel[profondeur_actuelle]=ordre_actuel[profondeur_actuelle]+1

# Vérifications de placement dans l'arbre et de bon fonctionnement :

        # Si un patient est traité 2 fois (présent 2 fois dans le
vecteur ordre_actuel)
        if sum(ordre_actuel.==ordre_actuel[profondeur_actuelle])>1
            # On continue la boucle pour passer au patient suivant
dans la profondeur actuelle
            # (--> on élague)
            continue
        end
```

```
        # Si tous les patients ont été traités dans la profondeur
actuelle
        ## (tous les cas de figures ont été traités à ce niveau)

        # ordre_actuel[profondeur_actuelle]>n signifie qu'on est passé
au patient n+1 dans la profondeur actuelle
        ## --> on sort du cadre de l'arbre
        if ordre_actuel[profondeur_actuelle]>n

            if profondeur_actuelle==1
                # Si en plus, on est revenu à la profondeur de 1, on a
fini de boucler sur tout notre arbre
                ## --> on renvoie notre solution optimale
                return [z_barre,ordre]

            # Si nous ne sommes pas encore revenus à la profondeur 1
            else
                # On élague en remontant d'une profondeur (car toutes
les branches à ce niveau ont été parcourues)
                ## On remet le compteur des patients à 0 pour la
profondeur actuelle
                ordre_actuel[profondeur_actuelle]=0
                temps_traitement[profondeur_actuelle]=0
                # On remet le calcul des pénalités à 0
                penalties=0
                # On remonte d'une profondeur
                profondeur_actuelle=profondeur_actuelle-1
                continue
            end
        end


# Vérification de l'optimalité

        # GESTION DES TEMPS DE TRAITEMENT

        # Si on est en profondeur 1
        if profondeur_actuelle == 1
            # Le premier patient est traité sans délai
            temps_traitement[profondeur_actuelle]=0

        # Si on est plus profond dans l'arbre
        else
            # Le patient n sera traité une durée d après le patient n-
1, avec d le temps de traitement du patient n-1
            ## D[trunc(Int,ordre_actuel[profondeur_actuelle-1])] donne
le temps de traitement du patient précédent

temps_traitement[profondeur_actuelle]=D[trunc(Int,ordre_actuel[profond
eur_actuelle-1])]
```

```
        end

        # ESTIMATION DE LA BORNE DUALE

        # On utilise la somme cumulée des temps pour avoir le temps de
passage globaux
        ## --> conversion des temps relatifs en temps globaux

penalties=compute_penalties(P,H,cumsum(temps_traitement),ordre_actuel)


        # Si la borne duale est moins bonne que notre solution
actuelle
        if penalties >= z_barre
            # On continue la boucle pour passer au patient suivant
dans la profondeur actuelle
            ## --> on élague
            continue

        # Si au contraire, la borne duale est meilleure
        else

            # Si nous ne sommes pas encore arrivé au bout (tous les
patients pas encore traités)
            if profondeur_actuelle != n
                # On continue sur cette branche de l'arbre (ajout
d'une unité de profondeur)
                profondeur_actuelle=profondeur_actuelle+1
                continue

            # Si nous sommes arrivés au bout de l'arbre
            else
                # Nous avons trouvé une meilleure solution que la
précédente !
                # On remplace donc l'ancienne solution par cette
meilleure solution
                z_barre=penalties
                ordre=copy(ordre_actuel)

                # Ici, continue va nous mener à un autre tour de
boucle.
                # Forcément, vu que tous les patients ont été traités,
passer au patient suivant dans la profondeur n
                # va mener à une duplication de patients dans le
vecteur. Ceci va encore mener à d'autres tours
                # de boucle jusqu'à passer au patient n+1 dans la
profondeur actuelle
                ## --> on sort du cadre de l'arbre

                # Ensuite, il remontera donc d'une profondeur et
```

```
continuera sa course le long d'une autre branche de
                # profondeur n-1, etc...

                continue
            end
        end
    end
end
```

```
Branch_Bound_v1 (generic function with 1 method)
```

La fonction *Branch_Bound_v1* met en oeuvre l'algorithme de branchement et séparation de la première manière étudiée en cours. L'arbre est parcouru en profondeur.

On stocke la profondeur courante dans une variable nommée *profondeur_actuelle* et l'ordre des patients dans un vecteur nommé *ordre_actuel*. Si *profondeur_actuelle < n*, ce vecteur sera complété de 0, pour signifier que des patients dans les profondeurs *profondeur_actuelle+1, ..., n-1, n* ne sont pas encore considérés. Ainsi, au début du programme, le vecteur *ordre_actuel* sera uniquement constitué de 0.

A chaque tour de boucle, on avance d'une unité de profondeur dans l'arbre, en considérant, à la profondeur *profondeur_actuelle +1*, le patient à l'indice suivant (+1) de celui déjà considéré à la *profondeur_actuelle*. Si ce patient figure déjà dans la liste, on n'avance pas dans l'arbre (on n'incrémente pas la profondeur).

Par exemple, au début du programme, on aura (pour n=4 patients) :

- 1 - ordre_actuel = [0, 0, 0, 0], profondeur_actuelle=0
- 2 - ordre_actuel = [1, 0, 0, 0], profondeur_actuelle=1
- 3 - ordre_actuel = [1, 1, 0, 0], profondeur_actuelle=2 → Doublons donc on n'avance pas de profondeur
- 4 - ordre_actuel = [1, 2, 0, 0], profondeur_actuelle=2 ...etc

D'autres objets tels que : *temps_traitement* permettent de stocker les informations des temps de traitement des patients rangés dans *ordre_actuel*, dans le même ordre et décalés de 1 vers la droite (première valeur à 0 et dernière valeur à D[ordre_actuel[length(ordre_actuel)-1]]).

Ainsi, dans la fonction de calcul des pénalités, nous utiliserons la somme cummulée de ces temps, afin de connaitre les temps de passages des patients. Ces pénalités permettront ensuite de trouver une borne duale et/ou primale ou de juger qu'il n'est pas nécessaire d'aller plus loin dans l'arbre.

Le programme se divise donc en 2 parties :

- La première partie s'occupe du bon fonctionnement de la boucle décrite plus haut en vérifant que :
  - Si un patient est traité 2 fois (présent 2 fois dans le vecteur ordre_actuel), on continue la boucle pour passer au patient suivant dans la profondeur actuelle

- Si tous les patients ont été traités dans la profondeur actuelle (signifie qu'on est passé au patient n+1 dans la profondeur actuelle, donc on n'est plus dans le cadre de l'arbre ⇔ ordre_actuel[profondeur_actuelle]>n), alors :
    - Si en plus, on est revenu à la profondeur de 1, on a fini de boucler sur tout notre arbre et en renvoie notre solution optimale
    - Sinon, on élague en remontant d'une profondeur (car toutes les branches à ce niveau ont été parcourues)

- La deuxième partie qui s'occupe de vérifier les conditions d'optimalité et qui va :
  - Mettre à jour les temps de traitement en concordance avec *ordre_actuel*
  - Calculer les pénalités
  - Décider si :
    - On élague (cas où $\tilde{z} > \overline{z}$)
    - On continue d'explorer l'arbre en profondeur : cas où $\tilde{z} < \overline{z}$ mais $\tilde{x}$ pas réalisable (tous les patients n'ont pas été traités).
    - On met à jour notre solution optimale précédente : cas où $\tilde{z} < \overline{z}$ et $\tilde{x}$ réalisable (tous les patients ont été traités).

Le programme se termine donc en descente d'arbre comme suit (exemple avec n=4) :

- 1 - ordre_actuel = [4, 3, 2, 1], profondeur_actuelle=4 → Calcul des pénalités
- 2 - ordre_actuel = [4, 3, 2, 2], profondeur_actuelle=4 → Doublons donc on continue
- 3 - ordre_actuel = [4, 3, 2, 3], profondeur_actuelle=4 → Doublons donc on continue
- 4 - ordre_actuel = [4, 3, 2, 4], profondeur_actuelle=4 → Doublons donc on continue
- 5 - ordre_actuel = [4, 3, 2, 5], profondeur_actuelle=4 → Sortie du cadre de l'arbre donc on remonte d'une profondeur
- 6 - ordre_actuel = [4, 3, 3, 0], profondeur_actuelle=3 → Doublons donc on continue
- 7 - ordre_actuel = [4, 3, 4, 0], profondeur_actuelle=3 → Doublons donc on continue
- 8 - ordre_actuel = [4, 3, 5, 0], profondeur_actuelle=3 → Sortie du cadre de l'arbre donc on remonte d'une profondeur
- 9 - ordre_actuel = [4, 4, 0, 0], profondeur_actuelle=2 → Doublons donc on continue
- 10 - ordre_actuel = [4, 5, 0, 0], profondeur_actuelle=2 → Sortie du cadre de l'arbre donc on remonte d'une profondeur
- 11 - ordre_actuel = [5, 0, 0, 0], profondeur_actuelle=1 → Sortie du cadre de l'arbre donc on remonte d'une profondeur

--- Fin du programme car profondeur revenue à 0 ---

1.2 : Seconde version

→ Fonction *compute_penalties_v2*

Nous considérons cette fois un autre algorithme de B&B, basé sur le second raisonnement élaboré en TD. Nous devons donc repenser notre fonction *compute_penalties* : nous créons une nouvelle fonction, nommée *compute_penalties_v2*.

```
function compute_penalties_v2(P,H,D,t,order)
    """
    Inputs :
        P : vecteur (nx1) des pénalités associées au retard de chaque
patient
        H : vecteur (nx1) des horaires de passage au plus tard de
chaque patient
        D : vecteur (nx1) des durées de rdv de chaque patient
        t : vecteur (nx1) des durées de traitement
        order : vecteur (nx1) donnant l'ordre de passage des patients
(inversé)
    Output :
        sum(res) : pénalité de l'ordre donné en paramètre
    """
    # Estimation du temps total pour faire passer la totalité des
patients
    max_delay=sum(D)
    # Tri des zeros finaux si pas en profondeur n
    order = trunc.(Int,order[order.!=0])

    # Vecteur t associé
    t=trunc.(Int,t[1:length(order)])
    ###
    t= (-1).*t.+max_delay
    ###
    # Vecteur des pénalités associé
    P=P[order]
    # Vecteur des délais associés
    H=H[order]

    # Calcul du vecteur des pénalités
    res=P.*(t-H)
    # On enelève les valeurs négatives
    res[res.<0].=0

    # On retourne la somme de toutes les pénalités
    return sum(res)
end
```

```
compute_penalties_v2 (generic function with 1 method)
```

Cette fonction *compute_penalties_v2* permet de calculer les pénalités pour la deuxième manière de traiter notre problème : en considérant cette fois les patients traités dans l'ordre inverse de passage.

Le principe est le même que pour la version 1, seulement, ici, la commande *t= (-1).t.+max_delay* permet de calculer les temps de passages des patients en partant de la fin.

ATTENTION, ici, *t* sera égal aux sommes cumulées des temps de passages des patients stockés dans *ordre_actuel*.

De cette manière, pour le patient i, on soustrait à la durée totale (temps de passage de tous les patients), les durées de passages des patients qui passent après lui. On soustrait également la durée de passage du patient i. De cette manière, on obtient bien le temps auquel passe le patient i.

Autrement, c'est le même raisonnement que pour la version 1. La fonction permet de calculer les pénalités en prenant en paramètre les vecteurs de pénalités *P*, de délais *H*, de durées de traitement cumulées *t*, et d'ordre provisoire *order*.

→ Algorithme *Branch_Bound_v2*

Avec cette nouvelle méthode, nous allons pouvoir écrire le second algorithme de B&B. Celui-ci fonctionnera d'une manière similaire : la seule chose qui changera sera l'ordre donné, qui sera écrit dans le sens inverse de l'ordre de passage des patients; et l'évaluation des pénalités.

```
function Branch_Bound_v2(P,D,H)
    """
    Inputs :
        P : vecteur (nx1) des pénalités associées au retard de chaque
patient
        D : vecteur (nx1) des durées de rdv de chaque patient
        H : vecteur (nx1) des horaires de passage au plus tard de
chaque patient
    Outputs :
        z_barre : solution optimale du problème (pénalité minimale)
        reverse(ordre) : ordre de passage associé (vecteur nx1)
    """

# Estimation initiale d'une borne primale, qui est renvoyée si égale à
0 car on serait dans le cas d'une optimalité
    n=length(D)
    res=estimate_sup_init(P,D,H)
    z_barre=res[1]
    ordre=reverse(res[2]) # on reverse car on travaille à l'envers
    if z_barre == 0
        return estimate_sup_init(P,D,H)
    end

# Initialisation des objet nécéssaires
    penalties=0                             # Pas de pénalités au debut
```

```
    temps_traitement=zeros(n)             # Pas de patients traités au
début
    ordre_actuel=zeros(n)
    profondeur_actuelle=1                 # On se place en profondeur
de 1 au début

# Boucle pour avancer dans l'arbre
    while true

    # On fait passer le patient suivant à la profondeur courante

ordre_actuel[profondeur_actuelle]=ordre_actuel[profondeur_actuelle]+1

# Vérifications de placement dans l'arbre et de bon fonctionnement :

     # Si un patient est traité 2 fois (présent 2 fois dans le vecteur
ordre_actuel)
        if sum(ordre_actuel.==ordre_actuel[profondeur_actuelle])>1
            # On continue la boucle pour passer au patient suivant
dans la profondeur actuelle
            ## --> on élague
            continue
        end

        # Si tous les patients ont été traités dans la profondeur
actuelle
        ## --> tous les cas de figures ont été traités à ce niveau

        # ordre_actuel[profondeur_actuelle]>n signifie qu'on est passé
au patient n+1 dans la profondeur actuelle donc
        # on n'est plus dans le cadre de l'arbre
        if ordre_actuel[profondeur_actuelle]>n

            # Si en plus, on est revenu à la profondeur 1, on a fini
de boucler sur tout notre arbre
            ## --> on renvoie notre solution optimale
            if profondeur_actuelle==1
                return [z_barre,reverse(ordre)]

            # Si nous ne sommes pas encore revenus à la profondeur 1
            else
                # On élague en remontant d'une profondeur (car toutes
les branches à ce niveau ont été parcourues)
                # On remet le compteur des patients à 0 pour la
profondeur actuelle
                ordre_actuel[profondeur_actuelle]=0
                temps_traitement[profondeur_actuelle]=0
                # On remet le calcul des pénalités à 0
                penalties=0
                # On remonte d'une profondeur
```

```
                profondeur_actuelle=profondeur_actuelle-1
                continue
            end
        end

# Vérification de l'optimalité

        # GESTION DES TEMPS DE TRAITEMENT

        # Dans cette version, les temps_traitement sont égaux aux
temps de passages des patients stockés dans ordre_actuel
        ## --> On donne à la fonction compute_penalties_v2 les sommes
cumulées des temps_traitement


temps_traitement[profondeur_actuelle]=D[trunc(Int,ordre_actuel[profond
eur_actuelle])]

        # Estimation des pénalités au niveau de profondeur actuel
(borne duale)

penalties=compute_penalties_v2(P,H,D,cumsum(temps_traitement),ordre_ac
tuel)


        # Si la borne duale est moins bonne que notre solution
actuelle
        if penalties >= z_barre
            # On continue le bouclage pour passer au patient suivant
dans la profondeur actuelle
            ## --> on élague
            continue

        # Si, au contraire, la borne duale est meilleure
        else

            # Si nous ne sommes pas encore arrivé au bout (tous les
patients ne sont pas encore traités)
            if profondeur_actuelle != n
                # On continue sur cette branche de l'arbre (ajout
d'une unité de profondeur)
                profondeur_actuelle=profondeur_actuelle+1
                continue

            # Si nous sommes arrivés au bout de l'arbre
            else
                # Nous avons trouvé une meilleure solution que la
précédente !
                # On remplace donc l'ancienne solution par cette
meilleure solution
```

```
                z_barre=penalties
                ordre=copy(ordre_actuel)

                # Ici, "continue" va nous mener à un autre tour de
boucle.

                # Forcément, vu que tous les patients ont été traités,
passer au patient suivant dans la profondeur n
                ## va mener à une duplication de patients dans le
vecteur.

                # Ceci va encore mener à d'autres tours de boucle
jusqu'à passer au patient n+1 dans la profondeur
                ## actuelle (on ne sera plus dans le cadre de
l'arbre).

                # Ensuite, il remontera donc d'une profondeur et
continuera sa course le long d'une autre branche de
                # profondeur n-1, etc...

            end
        end
    end
end
```

```
Branch_Bound_v2 (generic function with 1 method)
```

Cette deuxième version fonctionne ainsi de la même manière que la première, à la différence que les pénalités sont calculées autrement (comme expliqué plus haut).

Cela implique donc que le vecteur *temps_traitement* est traité autrement, en stockant cette fois, à l'indice i, la durée de traitement du patient placé à l'indice i du vecteur *ordre_actuel*.

On fait également appel à la fonction *compute_penalties_v2* et non plus à la fonction *compute_penalties_v1*.

2 - Formulation PLNE

Dans le cadre de ce 3e algorithme, l'objectif va être d'écrire le problème considéré en PLNE. Nous pourrons ensuite utiliser une résolution de ce problème linéaire en nombres entiers, à l'aide d'un solveur tel que HiGHS.

Pour cette formulation, nous allons utiliser 3 variables différentes :

- $x_{i,j} \in \{0,1\} = 1 \Leftrightarrow$ le patient i passe avant le patient j, $\forall i, j \in \{1,2,\ldots,n\}$
- $r_i \geq 0$: correspond au retard du patient i par rapport à sa date prévue au pire
- $t_i \geq 0$: correspond à l'instant de passage du patient i

Les contraintes considérées pour ces variables sont les suivantes :

- la variable $x_{i,j}$ traduit un ordre total, nous devons donc considérer deux contraintes :

  Transitivité : si le patient i passe avant le patient j, et le patient j passe avant le patient k, alors le patient i passe avant le patient k

$$\forall i,j,k \in \{1,\dots,n\}, i \neq j, i \neq k, j \neq k, x_{i,k} \geq x_{i,j} + x_{j,k} - 1$$

```
Ordre total : soit le patient i passe avant le patient j, soit j passe avant i
```

$$\forall i,j \in \{1,\dots,n\}, i \neq j, x_{i,j} + x_{j,i} = 1, et \forall i \in \{1,\dots,n\}, x_{i,i} = 0$$

- la variable t_i correspond à l'instant de passage du patient i : cet instant correspond à la somme des durées de rdv de chaque patient passé avant lui

$$\forall i \in \{1,\dots,n\}, t_i = \sum_{j=1}^{n} x_{j,i}.D_j$$

- la variable r_i correspond au retard par rapport à l'instant de passage au pire du patient : $r_i = max(0, t_i - H_i)$. Cela se traduit avec deux contraintes (uniquement 2 car nous souhaitons ici minimiser ces retards) :

$$\forall i \in \{1,\dots,n\}, r_i \geq 0, et\ r_i \geq t_i - H_i$$

Bien entendu, dans ce problème, l'objectif est de minimiser la pénalité engendrée par l'ordre de passage des patients, la fonction objectif s'écrit donc :

$$min \sum_{i=1}^{n} r_i.P_i$$

Notre problème est donc défini, et nous allons pouvoir écrire une méthode *PLNE*, permettant de construire un tel modèle.

```
function PLNE(P,D,H)
    """
    Inputs :
        P : vecteur (nx1) des pénalités associées au retard de chaque patient
        D : vecteur (nx1) des durées de rdv de chaque patient
        H : vecteur (nx1) des horaires de passage au plus tard de chaque patient
    Output :
        model : modèle décrit ci-dessus
    """
    # HiGHS solver
    model = Model(HiGHS.Optimizer);
```

```
    # On décide de ne pas afficher les outputs du modèle
    set_optimizer_attribute(model, "output_flag", false)

    # Variables
    n=length(P)
    I=Array(1:n)

    ## r[i] = retard du patient i par rapport à sa date prévue au pire
    @variable(model, r[I]>=0);

    ## t[i] = instant de passage du patient i
    @variable(model, t[I]>=0);

    ## x[i,j] = 1 ssi le patient i passe avant le patient j
    @variable(model, x[I,I], Bin);

    # Fonction objectif : minimiser les pénalités liés aux retards
    @objective(model, Min, sum(r[i]*P[i] for i in I));

    # Contraintes

    ## Contraintes liées à l'ordre total x[i,j]
    ### Contrainte de transitivité
    @constraint(model, [i in I, j in I, k in I; i!=j && i!=k && j!=k],
x[i,k]>=x[i,j]+x[j,k]-1)
    ### Ordre total
    @constraint(model, [i in I, j in I; i!=j], x[i,j]==1-x[j,i])

    ## Contrainte liée à l'instant de passage
    @constraint(model, [i in I], t[i]==sum(D[j]*x[j,i] for j in I))

    ## Contraintes liées au retard mesuré
    @constraint(model, [i in I], r[i]>=t[i]-H[i])

    return model;
end
```

```
PLNE (generic function with 1 method)
```

Ensuite, nous pouvons écrire une seconde fonction, qui résout le modèle, et détermine l'ordre de passage des patients.

Pour déterminer cet ordre de passage, nous allons utiliser les instants de passages de chaque individu (variable *t*), et nous allons récupérer l'ordre correspondant à l'ordonnancement croissant de ces instants.

```
function solve_PLNE(model)
    """
    Input :
        model : modèle à résoudre
```

```
    Outputs :
        terminaton_status(model) : état de la solution
        objective_value(model) : valeur optimale de la solution
        indices : ordre associé à la solution
    """
    # On résout le PLNE
    optimize!(model);
    # On récupère l'ordre de passage des patients
    t=value.(model[:t])
    T=convert(Array, t)
    indices=sortperm(T)

    return ([termination_status(model),
objective_value(model),indices]);
end
```

```
solve_PLNE (generic function with 1 method)
```

3 - Branch & Bound basé sur la relaxation linéaire

Nous allons, dans cette partie, implémenter un 3e algorithme de B&B.

Celui-ci est assez différent des deux premières implémentations de type B&B. En effet, ici, nous allons nous baser sur la relaxation linéaire de la formulation en PLNE vue précédemment.

L'idée est donc la suivante :

- on calcule une borne supérieure $\overline{z}$ du problème, et on garde en mémoire l'ordre associé
- on construit le modèle global, sans contrainte supplémentaire : on résout alors sa relaxation linéaire pour obtenir une borne inférieure $\tilde{z}$.

Plusieurs cas de figure peuvent être rencontrés :

- $\tilde{z} \geq \overline{z}$: l'ordre trouvé est déjà optimal
- $\tilde{z} < \overline{z}$:
  - si la solution $\tilde{x}$ est réalisable (dans Z), on met à jour la borne supérieure ($\overline{z} \leftarrow \tilde{z}$ et on met à jour l'ordre associé, puis on élague)
  - sinon, on va descendre dans l'arbre :
    - on détermine la composante de $\tilde{x}_{i,j}$ la plus fractionnaire (la plus proche de 0.5)
    - on crée deux noeuds fils, avec deux contraintes supplémentaires disjointes : $x_{i,j} \geq 1$ et $x_{i,j} \leq 0$
    - on se déplace vers le noeud caractérisé par $x_{i,j} \leq 0$, on construit le modèle associé, avec cette nouvelle contrainte, et on met à jour $\tilde{z}$, $\tilde{x}$

Arrivé à ce noeud, on réalise un raisonnement analogue au précédent :

- $\tilde{z} \geq \overline{z}$: on élague (on remonte dans l'arbre)
- $\tilde{z} < \overline{z}$:

- si la solution $\tilde{x}$ est réalisable (dans Z), on met à jour la borne supérieure ($\overline{z} \leftarrow \tilde{z}$ et on met à jour l'ordre associé, puis on élague : on remonte dans l'arbre)
- sinon, on va descendre dans l'arbre :
  - on détermine la composante de $\tilde{x}_{i,j}$ la plus fractionnaire (la plus proche de 0.5)
  - on crée deux noeuds fils, avec une contrainte supplémentaire : $x_{i,j} \geq 1$ ou $x_{i,j} \leq 0$
  - on se déplace vers le noeud caractérisé par $x_{i,j} \leq 0$, on construit le modèle associé, avec cette nouvelle contrainte, et on met à jour $\tilde{z}$, $\tilde{x}$

Et ainsi de suite...

Ainsi, comme pour les deux B&B précédents, nous opterons pour un parcours en profondeur. Ce choix réside principalement dans le fait que ce type de parcours nous permet de se déplacer dans l'arbre sans devoir garder en mémoire chaque modèle construit à chaque niveau de l'arbre.

→ Fonction *generate_model*

Nous commençons par construire une méthode permettant de construire le modèle considéré à chaque noeud. Celle-ci prend 4 paramètres : P,D,H (utiles pour construire le modèle de base), et un vecteur *contraintes*, contenant les contraintes ajoutées au modèle initial.

Ces contraintes seront codées de la manière suivante :

- pour une contrainte $x_{i,j} \geq 1$, la contrainte vaudra [i ,j , 1]
- pour une contrainte $x_{i,j} \leq 0$, la contrainte vaudra [i ,j , 0]

Dans le nouveau B&B, nous créerons un vecteur de contraintes (nommé *contraintes_actuelles*), contenant toutes les contraintes supplémentaires du modèle initial. Nous ajouterons ainsi de manière itérative chaque contrainte supplémentaire à la fin de ce vecteur.

```
function generate_model(P,D,H,contraintes)
    """
    Inputs :
        P : vecteur (nx1) des pénalités associées au retard de chaque
patient
        D : vecteur (nx1) des durées de rdv de chaque patient
        H : vecteur (nx1) des horaires de passage au plus tard de
chaque patient
        contraintes : vecteur contenant les contraintes
supplémentaires
    Output :
        model : modèle construit
    """
    # HiGHS solver
    model = Model(HiGHS.Optimizer);
    # On décide de ne pas afficher les outputs du modèle
    set_optimizer_attribute(model, "output_flag", false)
```

```
    # Variables
    n=length(P)
    I=Array(1:n)

    ## r[i] = retard du patient i par rapport à sa date prévue au pire
    @variable(model, r[I]>=0);

    ## t[i] = instant de passage du patient i
    @variable(model, t[I]>=0);

    ## x[i,j] = 1 ssi le patient i passe avant le patient j
    @variable(model, x[I,I], Bin);

    # Fonction objectif : minimiser les pénalités liés aux retards
    @objective(model, Min, sum(r[i]*P[i] for i in I));

    # Contraintes

    ## Contraintes liées à l'ordre total x[i,j]
    ### Contrainte de transitivité
    @constraint(model, [i in I, j in I, k in I; i!=j && i!=k && j!=k],
x[i,k]>=x[i,j]+x[j,k]-1)
    ### Ordre total
    @constraint(model, [i in I, j in I; i!=j], x[i,j]==1-x[j,i])

    ## Contrainte liée à l'instant de passage
    @constraint(model, [i in I], t[i]==sum(D[j]*x[j,i] for j in I))

    ## Contraintes liées au retard mesuré
    @constraint(model, [i in I], r[i]>=t[i]-H[i])

    ## On ajoute les contraintes supplémentaires
    for c in contraintes
        @constraint(model, x[c[1],c[2]]==c[3])
    end

    return model;
end
```

```
generate_model (generic function with 1 method)
```

Nous construisons également une fonction *solve_relaxation*, qui sera utilisée à chaque noeud de l'arbre. Celle-ci résout la relaxation linéaire du modèle considéré.

```
function solve_relaxation(model)
    """
    Input :
        model : modèle dont on veut résoudre la relaxation
    Output :
```

```
        model : relaxation du modèle (résolue)
    """
    # On résout la relaxation
    undo = relax_integrality(model);
    set_optimizer_attribute(model, "output_flag", false) # mute
solvers' outputs
    optimize!(model);
    # On renvoie le modèle
    return model;
end
```

```
solve_relaxation (generic function with 1 method)
```

→ Fonctions *getOrdre* et *getX*

A chaque noeud de notre arbre, nous devons récupérer la matrice $\widetilde{X}$, solution de la relaxation du noeud considéré. C'est pourquoi nous créons une fonction *getX*, permettant de récupérer cette variable pour un modèle donné.

Dans cette fonction, nous testons également si la matrice obtenue est dans Z (si $\forall i,j \in (1,\ldots,n), x_{i,j} \in (0,1)$). Cette information sera contenue dans la variable *dans_Z*.

```
function getX(model)
    """
    Input :
        model : modèle relaxé résolu dont on souhaite récupérer la
solution
    Outputs :
        X : matrice X du modèle
        dans_Z : variable booléenne à 1 si toutes les composantes de X
sont binaires
    """
    # On récupère la variable X
    tab=value.(model[:x])
    X=convert(Array, tab)
    # On teste si la solution est réalisable (x_{i,j}=0 ou x_{i,j}=1
pour tout i,j)
    dans_Z=all(elements->isinteger(elements),X)
    return([X,dans_Z])
end
```

```
getX (generic function with 1 method)
```

Dans le cas où la solution de la relaxation est réalisable, nous devons récupérer l'ordre associé à cette solution. Nous devons donc déterminer l'ordre de passage des patients pour un modèle donné.

Comme nous avions pu faire dans la partie précédente, nous pouvons récupérer cet ordre en récupérant les instants de passage de chaque patient.

```
function getOrdre(model)
    """
    Input :
        model : modèle relaxé dont on souhaite déterminer l'ordre
    Output :
        indices : ordre de passage
    """
    # On récupère les instants de passage
    t=value.(model[:t])
    T=convert(Array, t)
    # On récupère l'ordre de passage
    indices=sortperm(T)
    return(indices)
end
```

```
getOrdre (generic function with 1 method)
```

→ Algorithme *Branch_Bound_v3*

```
function Branch_Bound_v3(P,D,H)
    """
    Inputs :
        P : vecteur (nx1) des pénalités associées au retard de chaque
patient
        D : vecteur (nx1) des durées de rdv de chaque patient
        H : vecteur (nx1) des horaires de passage au plus tard de
chaque patient
    Output :
        z_state : état de la solution (rendu uniquement si au moins
une relaxation est réalisée)
        z_barre : solution du problème
        ordre : ordre de passage de la solution
    """
    # Nombre de patients
    n=length(D)

    # Détermination d'une borne primale
    res=estimate_sup_init(P,D,H)
    z_barre=res[1]
    ordre=res[2]
    if z_barre==0
        # Si la borne supérieure est nulle, elle est optimale --> on
renvoie la solution
        return [z_barre,ordre]
    end
    # Détermination d'une borne duale
    modeleCourant=generate_model(P,D,H,[])         # On initialise le
modèle en PLNE sans contraintes supplémentaires
    solve_relaxation(modeleCourant)                # On résout la
```

```
relaxation linéaire de ce PLNE
    z_tilde=objective_value(modeleCourant)         # Récupération de
la borne inférieure
    z_state=termination_status(modeleCourant)      # Etat de la
solution
    getteur=getX(modeleCourant)                    # Récupération de
la matrice X associée
    x_tilde=getteur[1]
    dans_Z=getteur[2]                              # Indicateur
donnant si X est dans Z ou non


    elague=false    # Variable binaire : à un noeud donné, =0 si on
branche / =1 si on élague
    profondeur=1    # Profondeur de parcours de l'arbre

    # Tableau gardant en mémoire les contraintes
    contraintes_actuelles=[]         # Contraintes du modèle courant

    if z_barre==z_tilde
        # Cas où la borne supérieure est déjà optimale / borne primale
nulle
        return [z_barre,ordre]
    else
        # Dans le cas contraire, on commence le parcours de l'arbre
        while(true)
            if (z_barre<=z_tilde)
                # Cas où la relaxation donne une borne inférieure plus
grande que z_barre --> On élague
                elague=true
            end
            if (dans_Z)
                # Cas où le x_tilde a toutes ses composantes entières
(solution réalisable)
                if (z_tilde < z_barre)
                    #  si la valeur optimale est meilleure : on met à
jour z_barre
                    z_barre=copy(z_tilde)
                    x_barre=copy(x_tilde)
                    ordre=getOrdre(modeleCourant)
                end
                # Dans tous les cas, on élague
                elague = true
            end


            if (!elague)
                # Cas où x_tilde est non réalisable pour Pk
                ##  --> On recherche la composante la plus
```

```
fractionnaire
                prox=abs.(x_tilde.-0.5)
                indices=argmin(prox)
                i=indices[1]
                j=indices[2]

                # On détermine la contrainte à ajouter au modèle
                ## On code la contrainte avec 2 paramètres :
                ##          - (i,j) : indices de la composante la plus
fractionnaire
                ##          - k (=0 ou =1): contrainte x_ij == k
                nouvelleContrainte=[i,j,0]

                # On ajoute la contrainte à la liste des contraintes
traitées
                push!(contraintes_actuelles,nouvelleContrainte)

                # Modification du modèle

modeleCourant=generate_model(P,D,H,contraintes_actuelles)   # On
ajoute la contrainte au modèle

                # On a ici ajouté une contrainte : on descend d'un
rang dans l'arbre
                profondeur=profondeur+1

            else
                # On élague le noeud
                elague=false

                # On remonte jusqu'au premier noeud pas encore traité
selon un parcours en profondeur
                noeud_trouve=false          # Devient "false" lorsque
l'on se trouve à un nouveau noeud

                while(!noeud_trouve)
                    # Tant qu'un tel noeud n'est pas trouvé

                    # On remonte d'un niveau
                    profondeur=profondeur-1
                    # On supprime la dernière contrainte ajoutée

contrainte_a_supprimer=contraintes_actuelles[length(contraintes_actuel
les)]
                    deleteat!(contraintes_actuelles, findall(x-
>x==contrainte_a_supprimer,contraintes_actuelles))

                    # On détermine la contrainte complémentaire de
celle supprimée
                    contrainteC=copy(contrainte_a_supprimer)
```

```
                    contrainteC[3]=abs(1-contrainteC[3])

                    # Si la contrainte complémentaire n'a pas été
traitée
                    if (contrainteC[3]==1)
                        # On ajoute la contrainte complémentaire aux
contraintes traitées
                        push!(contraintes_actuelles,contrainteC)

                        # On l'ajoute au modèle

modeleCourant=generate_model(P,D,H,contraintes_actuelles)

                        # On est redescendu dans l'arbre : on sort de
la boucle while et on incrémente la profondeur
                        noeud_trouve=true
                        profondeur=profondeur+1
                    end

                    if !noeud_trouve && profondeur==1
                        # On est remonté au probleme initial
                        ## --> tous les cas ont été traités : on
renvoie z_barre et ordre
                        return [z_state, z_barre, ordre]
                    end

                end
            end
            # On résout la relaxation de Pk
            solve_relaxation(modeleCourant)
            # On met à jour la solution relaxée
            z_tilde=objective_value(modeleCourant)          #
Récupération de la borne inférieure
            z_state=termination_status(modeleCourant)       # Etat de
la solution
            getteur=getX(modeleCourant)                     #
Récupération de la matrice X associée
            x_tilde=getteur[1]
            dans_Z=getteur[2]                               #
Indicateur donnant si X est dans Z ou non
        end
    end
end
```

```
Branch_Bound_v3 (generic function with 1 method)
```

→ Fonction *printBB*

Définition d'un fonction d'affichage des solutions, qui permet de se concentrer sur l'ordre de passage de certains patients (en les affichant en rouge), ou d'afficher tous les patients d'une

différente couleur, dans l'ordre, afin de pouvoir comparer les résultats de différentes instances visuellement plus facilement.

```
function printBB(a, patients_remarquables=[])
    """
    Inputs :
        a : vecteur rendu par l'algorithme qui résout le problème
        patients_remarquables : patients que l'on souhaite étudier
    """

    if a[1] == OPTIMAL
        penalite = trunc(Int,a[2])
        ordre = a[3]
    else
        penalite = trunc(Int,a[1])
        ordre = a[2]
    end

    print("Pénalités : ")
    println(penalite)
    println("Ordre de passage :")

    if length(patients_remarquables)==0
        for i in 1:length(ordre)
            printstyled(i,"-->";color = :black)
            printstyled(" Patient ",trunc(Int,ordre[i]),"\n"; color =
trunc(Int,ordre[i]+1))
        end


    else
        for i in 1:length(ordre)
            if ordre[i] in patients_remarquables
                col = :red
            else
                col= :blue
            end
            printstyled(i," -->";color = :black)
            printstyled(" Patient ",trunc(Int,ordre[i]),"\n"; color =
col)
        end
    end
end

# Exemple :
P=[5,4,1,2]
D=[14,15,16,13]
H=[1,2,3,4]
## Impression de l'ordre
printBB(Branch_Bound_v2(P,D,H))
```

```
print("\n")
## Impression de l'ordre en se concentrant sur le patient 4
printBB(Branch_Bound_v2(P,D,H),[4])
```

```
Pénalités : 137
Ordre de passage :
1--> Patient 1
2--> Patient 2
3--> Patient 4
4--> Patient 3

Pénalités : 137
Ordre de passage :
1 --> Patient 1
2 --> Patient 2
3 --> Patient 4
4 --> Patient 3
```

Cette fonction sera très utile, notamment dans la dernière partie de ce projet.

4 - Quelques exemples...

Nous allons ici tester brièvement nos algorithmes, afin de s'assurer que ceux-ci fonctionnent bien et donnent la même valeur optimale. Nous allons donc créer des cas tests, "sans trop se poser de question", simplement pour s'assurer de la bonne implémentation de notre code.

On commence par exemple, avec l'exemple traité en TD, pour lequel nous connaissons la valeur optimale (z=12).

```
# Exemple du cours
P=[4,5,3,5]
D=[12,8,15,9]
H=[16,26,25,27]

print("\n")
@time printBB(solve_PLNE(PLNE(P,D,H)))
print("\n")
@time printBB(Branch_Bound_v1(P,D,H))
print("\n")
@time printBB(Branch_Bound_v2(P,D,H))
print("\n")
@time printBB(Branch_Bound_v3(P,D,H))
print("\n")
```

```
Pénalités : 12
Ordre de passage :
1--> Patient 4
2--> Patient 1
3--> Patient 2
```

```
4--> Patient 3
  3.255529 seconds (5.76 M allocations: 375.180 MiB, 3.35% gc time,
97.37% compilation time: 4% of which was recompilation)

Pénalités : 12
Ordre de passage :
1--> Patient 1
2--> Patient 2
3--> Patient 4
4--> Patient 3
  0.312921 seconds (270.60 k allocations: 17.270 MiB, 99.57%
compilation time)

Pénalités : 12
Ordre de passage :
1--> Patient 1
2--> Patient 2
3--> Patient 4
4--> Patient 3
  0.001119 seconds (444 allocations: 18.414 KiB)

Pénalités : 12
Ordre de passage :
1--> Patient 1
2--> Patient 2
3--> Patient 4
4--> Patient 3
  2.572664 seconds (3.19 M allocations: 211.336 MiB, 3.12% gc time,
97.72% compilation time)
```

On observe ainsi que les 4 algorithmes implémentés dans la partie précédente, convergent bien tous vers la même valeur optimale. Toutefois, nous pouvons voir que l'ordre rendu n'est, sur ce test, pas le même pour toutes les implémentations. Ce résultat était prévisible, puisqu'en TD nous avions vu qu'il existait plusieurs solutions optimales.

On se propose maintenant de tester sur une autre instance, de dimension plus grande.

```
# Nouvel exemple
P=[4,5,3,5,4,4,5,4,5,7]
D=[12,8,15,9,15,17,12,48,45,1]
H=[16,26,25,27,20,15,14,14,12,15]

print("\n")
@time printBB(solve_PLNE(PLNE(P,D,H)))
print("\n")
@time printBB(Branch_Bound_v1(P,D,H))
print("\n")
@time printBB(Branch_Bound_v2(P,D,H))
print("\n")
```

```julia
@time printBB(Branch_Bound_v3(P,D,H))
print("\n")
```

```
Pénalités : 1298
Ordre de passage :
1--> Patient 10
2--> Patient 7
3--> Patient 1
4--> Patient 2
5--> Patient 4
6--> Patient 5
7--> Patient 6
8--> Patient 3
9--> Patient 9
10--> Patient 8
  0.222253 seconds (39.12 k allocations: 2.797 MiB)

Pénalités : 1298
Ordre de passage :
1--> Patient 1
2--> Patient 10
3--> Patient 7
4--> Patient 2
5--> Patient 4
6--> Patient 5
7--> Patient 6
8--> Patient 3
9--> Patient 9
10--> Patient 8
  4.302429 seconds (61.28 M allocations: 4.436 GiB, 19.25% gc time,
0.27% compilation time)

Pénalités : 1298
Ordre de passage :
1--> Patient 10
2--> Patient 7
3--> Patient 1
4--> Patient 2
5--> Patient 4
6--> Patient 5
7--> Patient 6
8--> Patient 3
9--> Patient 9
10--> Patient 8
  0.009864 seconds (99.39 k allocations: 7.125 MiB)

Pénalités : 1298
Ordre de passage :
1--> Patient 1
```

```
2--> Patient 10
3--> Patient 7
4--> Patient 2
5--> Patient 4
6--> Patient 5
7--> Patient 6
8--> Patient 3
9--> Patient 9
10--> Patient 8
  1.488499 seconds (5.27 M allocations: 359.449 MiB, 5.97% gc time,
0.55% compilation time)
```

Avec ces deux tests, nous pouvons être rassurés vis-à-vis du bon fonctionnement de nos algorithmes.

Ces deux premiers tests nous permettent aussi d'observer brièvement la différence de temps d'exécution entre ces 4 méthodes de résolution. C'est justement ce que nous allons analyser dans la 3e partie.

III / Etude des performances temporelles

Nous allons à présent mener une étude sur les performances temporelles de nos implémentations. Nous découperons cette étude en 2 parties :

- 1) Etude sur des instances prises au hasard
- 2) Etude sur des instances choisies particulièrement

1) Dans les instances prises au hasard, nous étudierons l'efficacité temporelle moyenne des algorithmes en fonction de la taille des instances. Nous mettrons donc en avant les performances pures de la stratégie de réflexion derrière les implémentations.

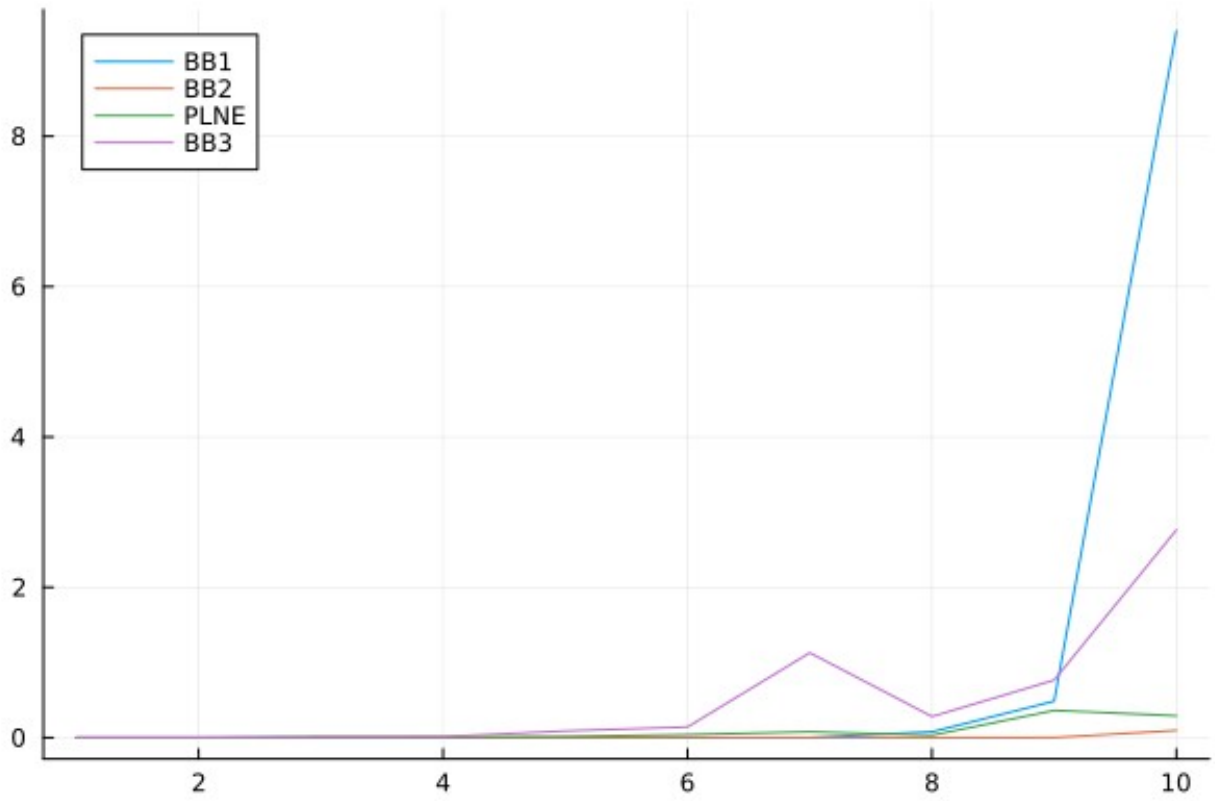
2) Dans les instances choisies, nous considèrerons des instances où la réponse est évidente, afin de mettre en avant que, dans certains cas, ces algorithmes seront plus longs que le cerveau humain et donc inutiles. Nous essaierons aussi, afin de vanter les mérites de nos implémentations, de mettre en avant des cas où les programmes font preuve d'une grande rapidité pour des problèmes assez complexes, pouvant être considérés comme impossibles à résoudre pour un cerveau humain

1 - Performances avec des instances prises au hasard

Nous allons commencer par tester les performances de nos algorithmes sur des instances obtenues avec la fonction *rand*, qui génère des valeurs aléatoirement, de manière uniforme, parmi un ensemble donné de valeurs possibles. Nous utiliserons donc cette méthode pour générer des vecteurs P,D et H.

Pour chaque instance aléatoire construite, nous calculerons le temps de calcul, et nous le conserverons dans un vecteur. Nous pourrons ensuite comparer l'évolution des temps de calculs en fonction de la dimension des instances.

```
# Taille maximale des instances
nmax=10
# On initialise les vecteurs qui récupèreront les temps de calcul
tempsv1=zeros((1,nmax))
tempsv2=zeros((1,nmax))
tempsPLNE=zeros((1,nmax))
tempsv3=zeros((1,nmax))

for npatients in 1:nmax
    # Pour chaque taille, on choisit aléatoirement une instance
    P=vec(rand(1:10, 1,npatients))
    D=vec(rand(5:40, 1,npatients))
    H=vec(rand(10:50, 1,npatients))
    # B&B v1
    tic=time()
    Branch_Bound_v1(P,D,H)
    tac=time()
    tempsv1[1,npatients]=tac-tic
    # B&B v2
    tic=time()
    Branch_Bound_v2(P,D,H)
    tac=time()
    tempsv2[1,npatients]=tac-tic
    # PLNE
    tic=time()
    solve_PLNE(PLNE(P,D,H))
    tac=time()
    tempsPLNE[1,npatients]=tac-tic
    # B&B v3
    tic=time()
    Branch_Bound_v3(P,D,H)
    tac=time()
    tempsv3[1,npatients]=tac-tic

end

plot(1:nmax,transpose(tempsv1),label="BB1")
plot!(1:nmax,transpose(tempsv2),label="BB2")
plot!(1:nmax,transpose(tempsPLNE),label="PLNE")
plot!(1:nmax,transpose(tempsv3),label="BB3")
```



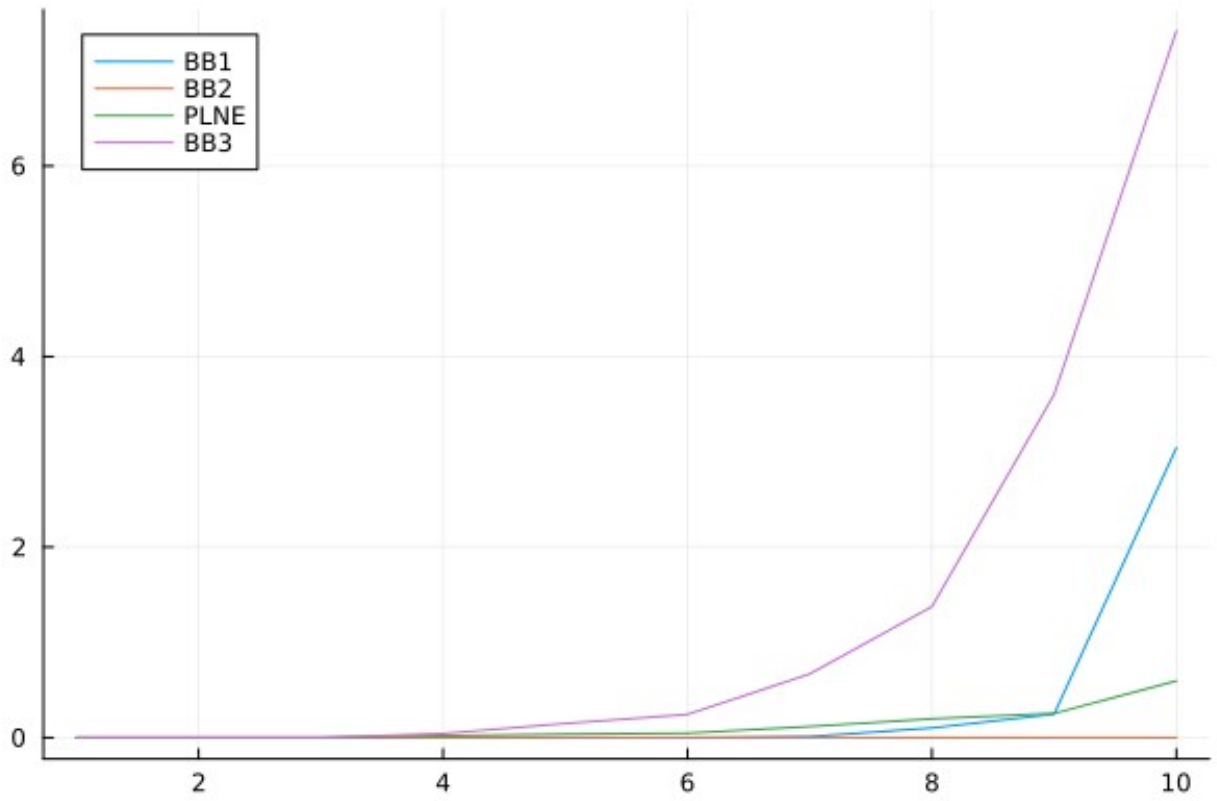
On observe ainsi qu'en moyenne, sur des échantillons générés aléatoirement, l'implémentation de type B&B n°2 reste la plus performante. Ce résultat était assez prévisible, puisque son fonctionnement est spécialement construit pour le problème : il est ciblé sur le cas d'étude.

A l'inverse, le fonctionnement des deux autres algorithmes de type B&B sont bien plus généraux : par exemple, la 3e version se base sur une règle de branchement complètement générale, qui fonctionne théoriquement avec toutes les formulations de type PLNE. D'ailleurs, ces deux autres formulations semblent avoir un temps d'exécution qui explose lorsque le nombre de patients croit.

Toutefois, nous avons ici sélectionné des durées de rdv et dates de passages au plus tard dans des ordres de grandeurs assez similaires. Nous pouvons tenter de modifier ces paramètres, et voir si les mêmes résultats sont obtenus. Nous allons ici seulement diminuer les dates de passages au plus tard. Ainsi, les pénalités seront plus importantes, et nécesairement, le problème, plus complexe.

```
# Taille maximale des instances
nmax=10
# On initialise les vecteurs qui récupèreront les temps de calcul
tempsv1=zeros((1,nmax))
tempsv2=zeros((1,nmax))
tempsPLNE=zeros((1,nmax))
tempsv3=zeros((1,nmax))

for npatients in 1:nmax
```

```
    # Pour chaque taille, on choisit aléatoirement une instance
    P=vec(rand(1:10, 1,npatients))
    D=vec(rand(5:40, 1,npatients))
    H=vec(rand(30:60, 1,npatients))
    # B&B v1
    tic=time()
    Branch_Bound_v1(P,D,H)
    tac=time()
    tempsv1[1,npatients]=tac-tic
    # B&B v2
    tic=time()
    Branch_Bound_v2(P,D,H)
    tac=time()
    tempsv2[1,npatients]=tac-tic
    # PLNE
    tic=time()
    solve_PLNE(PLNE(P,D,H))
    tac=time()
    tempsPLNE[1,npatients]=tac-tic
    # B&B v3
    tic=time()
    Branch_Bound_v3(P,D,H)
    tac=time()
    tempsv3[1,npatients]=tac-tic

end

plot(1:nmax,transpose(tempsv1),label="BB1")
plot!(1:nmax,transpose(tempsv2),label="BB2")
plot!(1:nmax,transpose(tempsPLNE),label="PLNE")
plot!(1:nmax,transpose(tempsv3),label="BB3")
```



Ce changement permet de mettre en évidence, de manière plus prononcée, l'explosion en temps de la version n°1 et n°3, lorsque le problème se complexifie.

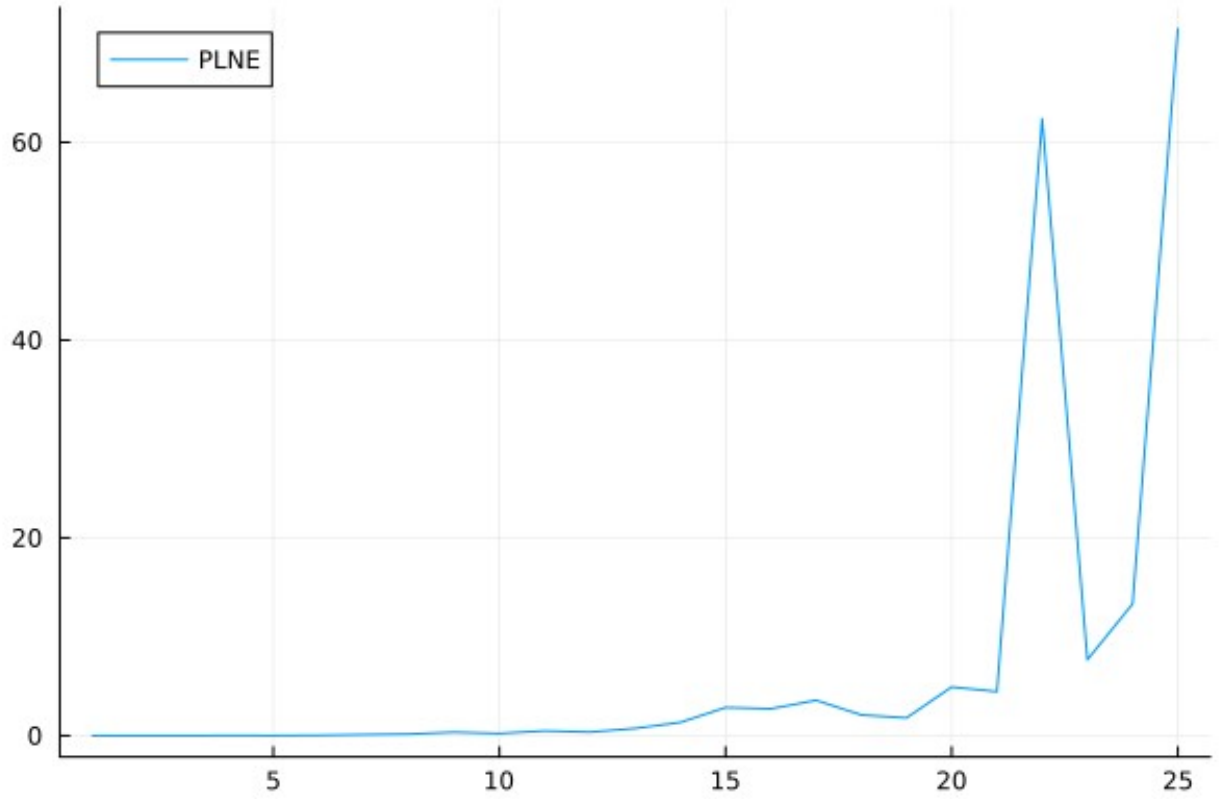
Nous pouvons aussi voir que la version n°2 reste la meilleure, et qu'en moyenne, la résolution par solveur, avec la formulation PLNE, est plus performante que la version B&B n°3. Ce résultat semblait assez évident au premier abord, puisque le solveur utilisé réalise lui aussi un B&B avec relaxations à chaque noeud, mais est bien plus optimisé que notre version.

Nous remarquons ainsi que nos deux meilleurs algorithmes (en termes de temps de calcul) sont le Branch and Bound n°2, et la résolution par solveur du PLNE. Nous allons donc tenter de comparer ces deux implémentations.

Pour cela, nous commençons par établir jusqu'à quelle taille d'instance la fonction PLNE converge-t-elle en temps raisonable.

```
# Taille maximale des instances
nmax=25
# On initialise les vecteurs qui récupèreront les temps de calcul
temps=zeros((1,nmax))

for npatients in 1:nmax
    # Pour chaque taille, on choisit aléatoirement une instance
    P=vec(rand(1:10, 1,npatients))
    D=vec(rand(5:40, 1,npatients))
    H=vec(rand(30:60, 1,npatients))
```

```
    # PLNE
    tic=time()
    solve_PLNE(PLNE(P,D,H))
    tac=time()
    temps[1,npatients]=tac-tic

end

plot(1:nmax,transpose(temps),label="PLNE")
```

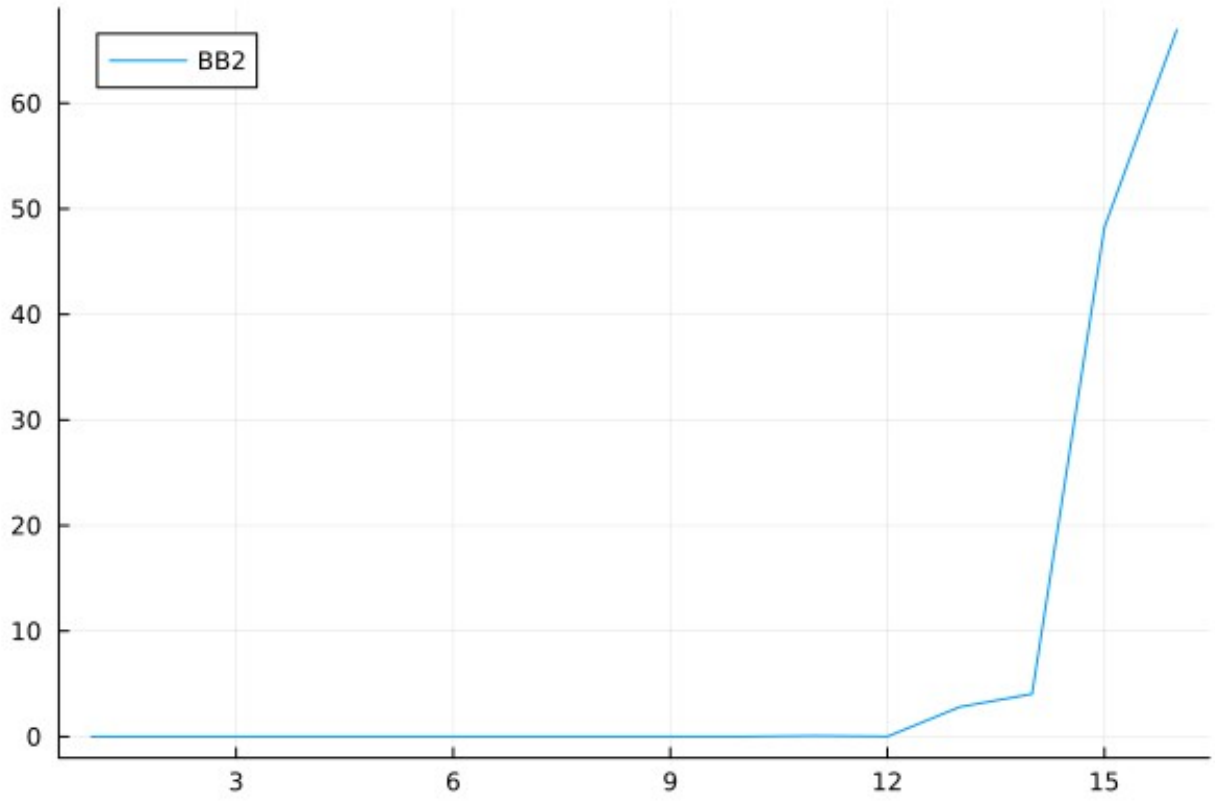


On observe ainsi que la résolution par solveur peut être utilisée en temps raisonable jusqu'à une taille d'environ 25.

Qu'en est-il de la 2e version de notre Branch and Bound ?

```
# Taille maximale des instances
nmax=16
# On initialise les vecteurs qui récupèreront les temps de calcul
temps=zeros((1,nmax))

for npatients in 1:nmax
    # Pour chaque taille, on choisit aléatoirement une instance
    P=vec(rand(1:10, 1,npatients))
```

```
    D=vec(rand(5:40, 1,npatients))
    H=vec(rand(30:60, 1,npatients))

    # B&B v2
    tic=time()
    Branch_Bound_v2(P,D,H)
    tac=time()
    temps[1,npatients]=tac-tic

end

plot(1:nmax,transpose(temps),label="BB2")
```



On observe qu'ici la 2e version du Branch and Bound donne la solution en un temps raisonnable pour des instances jusqu'à une dimension 16 environ.

Cette implémentation semble être la plus intéressante d'un point de vue pratique, pour un nombre de patients inférieur à 10. En effet, ici, la vie des patients est en jeu, et nous devons obtenir la solution le plus rapidement possible : chaque seconde compte ! En revanche, pour des patients plus nombreux, il serait plus cohérent d'utiliser la version PLNE.

Toutefois, ces observations ne sont réalisées que sur un certain nombre d'instances, et nous ne pouvons pas généraliser ces résultats à n'importe quelle instance existante.

2 - Performances avec des instances particulières

Comme annoncé précédemment, nous allons présenter des instances particulières pour lesquelles la solution peut sembler évidente. Pour autant, dans ce cas, certains algorithmes ne convergeront pas instantanément comme nous aurions pu le penser.

→ Instance A : Un cas simple

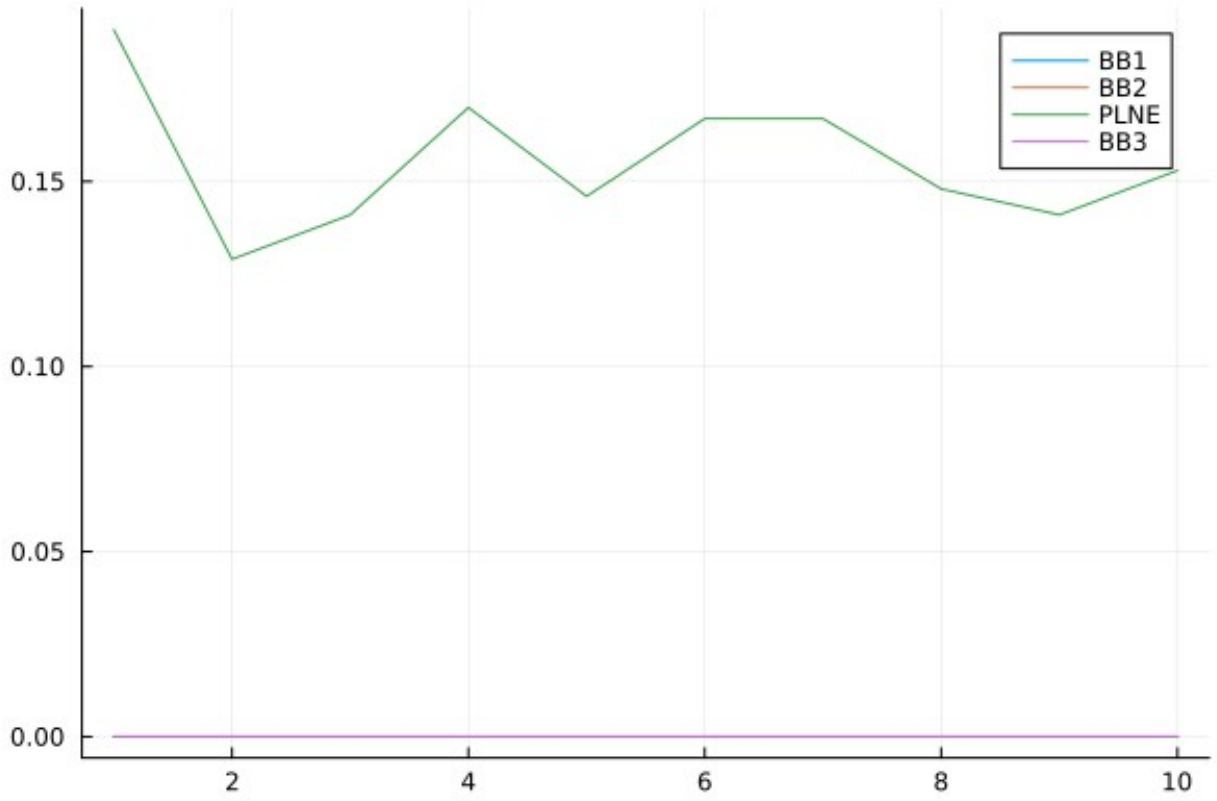
On considère des patients "égaux" : avec les mêmes pénalités et les mêmes temps de traitement, et qui arrivent aux urgences de manière "uniforme" (nécessité de les traiter à des temps réguliers).

1. Avec une possibilité d'avoir aucune pénalité

Cette première instance est un exemple que l'on pourrait assigner à un dentiste par exemple, avec des rendez-vous, peu d'urgence, et des patients qui arrivent chacun à leur tour. Dans ce cadre, nous pouvons réussir à trouver une solution avec une pénalité nulle (puisque chaque patient a déjà pris rendez-vous, et chaque rendez-vous est assez rapide).

```
P=[1,1,1,1,1,1,1,1,1,1]
D=[5,5,5,5,5,5,5,5,5,5]
H=cumsum([5,5,5,5,5,5,5,5,5,5])


tempsv1=zeros((1,length(P)))
tempsv2=zeros((1,length(P)))
tempsPLNE=zeros((1,length(P)))
tempsv3=zeros((1,length(P)))
nmax=length(P)

for npatients in 1:length(P)
    P1=P[1:npatients]
    D1=D[1:npatients]
    H1=H[1:npatients]
    # B&B v1
    tic=time()
    Branch_Bound_v1(P,D,H)
    tac=time()
    tempsv1[1,npatients]=tac-tic
    # B&B v2
    tic=time()
    Branch_Bound_v2(P,D,H)
    tac=time()
    tempsv2[1,npatients]=tac-tic
    # PLNE
    tic=time()
    solve_PLNE(PLNE(P,D,H))
    tac=time()
    tempsPLNE[1,npatients]=tac-tic
    # B&B v3
```

```
    tic=time()
    Branch_Bound_v3(P,D,H)
    tac=time()
    tempsv3[1,npatients]=tac-tic
end


plot(1:nmax,transpose(tempsv1),label="BB1")
plot!(1:nmax,transpose(tempsv2),label="BB2")
plot!(1:nmax,transpose(tempsPLNE),label="PLNE")
plot!(1:nmax,transpose(tempsv3),label="BB3")
```



On observe ici que, dans ce premier cas de figure, les Branch and Bound 1, 2 et 3 sont très performants, et trouvent une solution au problème en un temps presque nul. Ceci est dû au fait que la borne primale initiale est déjà nulle, et ainsi, aucune itération n'est réalisée au cours de l'algorithme.

A l'inverse, dans ce cas, la résolution du PLNE par solveur est aussi performante, mais bien plus lente : quelque soit la dimension du problème (pour des petite dimensions), la durée de résolution est similaire (et assez faible). En effet, cette implémentation est plus longue car elle doit construire le modèle, ce qui n'est pas instantané. En revanche, comme vu plus haut, sur des instances plus grande, elle sera plus efficace que les 3 autres.

Ainsi, dans ce premier cas, dont une solution peut très rapidement être trouvée à la main, on observe que tous les algorithmes ne sont pas aussi rapides que ce que nous pourrions penser.

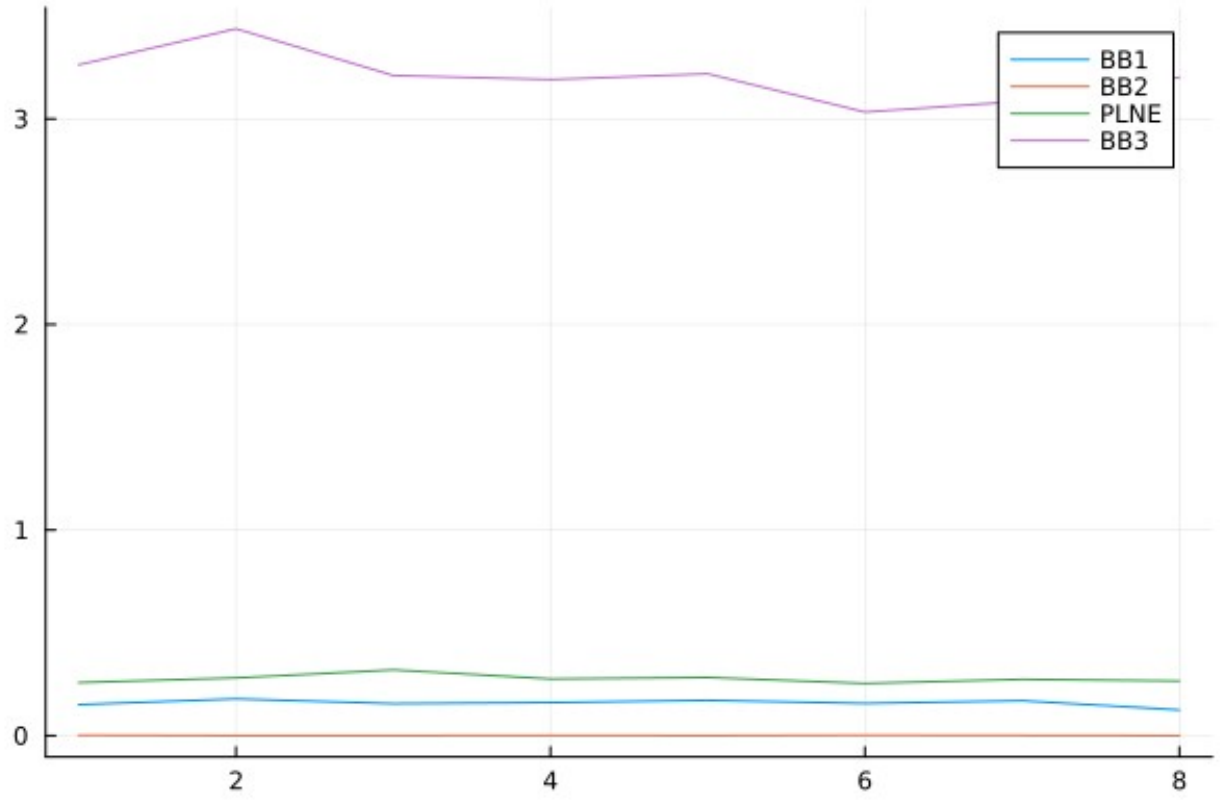
2. Avec des délais de traitement plus grands et donc une évidence d'avoir des pénalités

Dans le cas précédent, tout s'organisait plutôt bien : la pénalité d'une solution optimale était nulle. Désormais, nous allons considérer un cas similaire au précédent, mais dans lequel les choses se corsent : des pénalités vont apparaître car nous augmentons les durées de rdv des patients.

Ici, nous pouvons associer ce second cas à un médecin généraliste, pour lequel les durées de rendez-vous sont théoriquement fixées (en général, autour de 15min), mais, très souvent, les durées de prise en charge des patients est bien plus grande, et du retard s'accumule au fil de la journée.

```
P=[1,1,1,1,1,1,1,1]
D=[10,10,10,10,10,10,10,10]
H=cumsum([5,5,5,5,5,5,5,5])

tempsv1=zeros((1,length(P)))
tempsv2=zeros((1,length(P)))
tempsPLNE=zeros((1,length(P)))
tempsv3=zeros((1,length(P)))
nmax=length(P)

for npatients in 1:length(P)
    P1=P[1:npatients]
    D1=D[1:npatients]
    H1=H[1:npatients]
    # B&B v1
    tic=time()
    Branch_Bound_v1(P,D,H)
    tac=time()
    tempsv1[1,npatients]=tac-tic
    # B&B v2
    tic=time()
    Branch_Bound_v2(P,D,H)
    tac=time()
    tempsv2[1,npatients]=tac-tic
    # PLNE
    tic=time()
    solve_PLNE(PLNE(P,D,H))
    tac=time()
    tempsPLNE[1,npatients]=tac-tic
    # B&B v3
    tic=time()
    Branch_Bound_v3(P,D,H)
    tac=time()
    tempsv3[1,npatients]=tac-tic
end

plot(1:nmax,transpose(tempsv1),label="BB1")
plot!(1:nmax,transpose(tempsv2),label="BB2")
```

```
plot!(1:nmax,transpose(tempsPLNE),label="PLNE")
plot!(1:nmax,transpose(tempsv3),label="BB3")
```



Dans ce cas, si l'on trie les patients par H/P croissants, on observe que la borne primale utilisée en début d'algorithme est déjà optimale.

Les Branch and Bound n°1 et 2 tirent profit de cette observation, puisque l'on observe que la convergence est assez rapide :

- Pour la version n°2, le résultat est obtenu presque instantanément. Cela signifie que l'on élague très vite, et on renvoie la borne assez rapidement (on considère chaque patient comme le dernier, et à chaque fois, on élague $\rightarrow$ on ne parcourt que n noeuds).
- Pour la version n°1, le résultat est légèrement plus long. Ici, le fait de parcourir l'arbre du début vers la fin induit un plus grand nombre de noeuds à parcourir. Il est ainsi plus long de prouver que la borne primale est en réalité la solution.

On observe que la 3e version est bien moins performante. En effet, elle est environ 10 fois plus lente que les autres. Dans notre cas, nous démarrons bien avec la même borne primale (solution optimale), mais les relaxations rendent des solutions optimales bien plus faibles que celle-ci. Ainsi, bien que la solution optimale soit conservée depuis le début de l'algorithme, de nombreuses itérations sont nécessaires avant de pouvoir conclure que z_barre est bien la solution optimale.

→ Instance B : Des patients "égaux" qui arrivent tous en même temps aux urgence

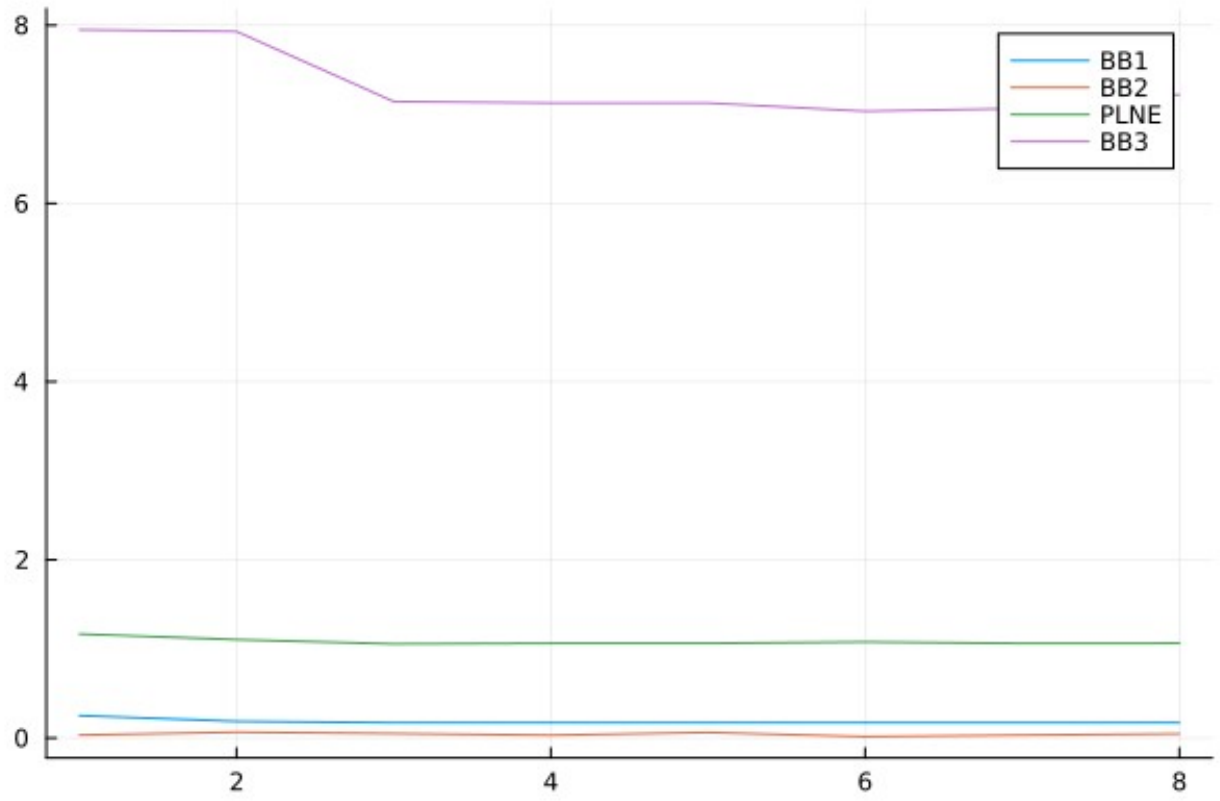
On considère ici des patients identiques (comme dans le cas précédent : avec les mêmes pénalités et les mêmes temps de traitement), mais qui arrivent en même temps. Nous modélisons ici un cas simple pouvant être obtenu aux urgences : un groupe d'amis a été victime d'une infection alimentaire par exemple.

Il y a donc ici une nécessité de traiter tous les patients en même temps.

Visuellement, il existe de nombreuses solutions optimales : n'importe quel ordre est acceptable. Ainsi, ce problème peut très facilement être résolu de tête. Qu'en est-il des algorithmes ? C'est ce que nous allons découvrir.

```
P=[1,1,1,1,1,1,1,1]
D=[5,5,5,5,5,5,5,5]
H=[5,5,5,5,5,5,5,5]

tempsv1=zeros((1,length(P)))
tempsv2=zeros((1,length(P)))
tempsPLNE=zeros((1,length(P)))
tempsv3=zeros((1,length(P)))
nmax=length(P)

for npatients in 1:length(P)
    P1=P[1:npatients]
    D1=D[1:npatients]
    H1=H[1:npatients]
    # B&B v1
    tic=time()
    Branch_Bound_v1(P,D,H)
    tac=time()
    tempsv1[1,npatients]=tac-tic
    # B&B v2
    tic=time()
    Branch_Bound_v2(P,D,H)
    tac=time()
    tempsv2[1,npatients]=tac-tic
    # PLNE
    tic=time()
    solve_PLNE(PLNE(P,D,H))
    tac=time()
    tempsPLNE[1,npatients]=tac-tic
    # B&B v3
    tic=time()
    Branch_Bound_v3(P,D,H)
    tac=time()
    tempsv3[1,npatients]=tac-tic
end
```

```
plot(1:nmax,transpose(tempsv1),label="BB1")
plot!(1:nmax,transpose(tempsv2),label="BB2")
plot!(1:nmax,transpose(tempsPLNE),label="PLNE")
plot!(1:nmax,transpose(tempsv3),label="BB3")
```



Comme pour le cas précédent, dès le début, la borne primale des 3 algorithmes de Branch and Bound est la solution optimale.

Ainsi, comme pour le cas précédent, la version n°2 ne parcourt que n noeuds (élague à profondeur 1 pour chacun d'entre eux).

Pour la version n°1, montrer que la borne supérieure est optimale nécessite de parcourir l'arbre dans son intégralité, d'où un temps d'exécution plus long.

Pour la 3e version, le fait de devoir parcourir l'arbre en entier, couplé à la création de modèles rend cette méthode très lente.

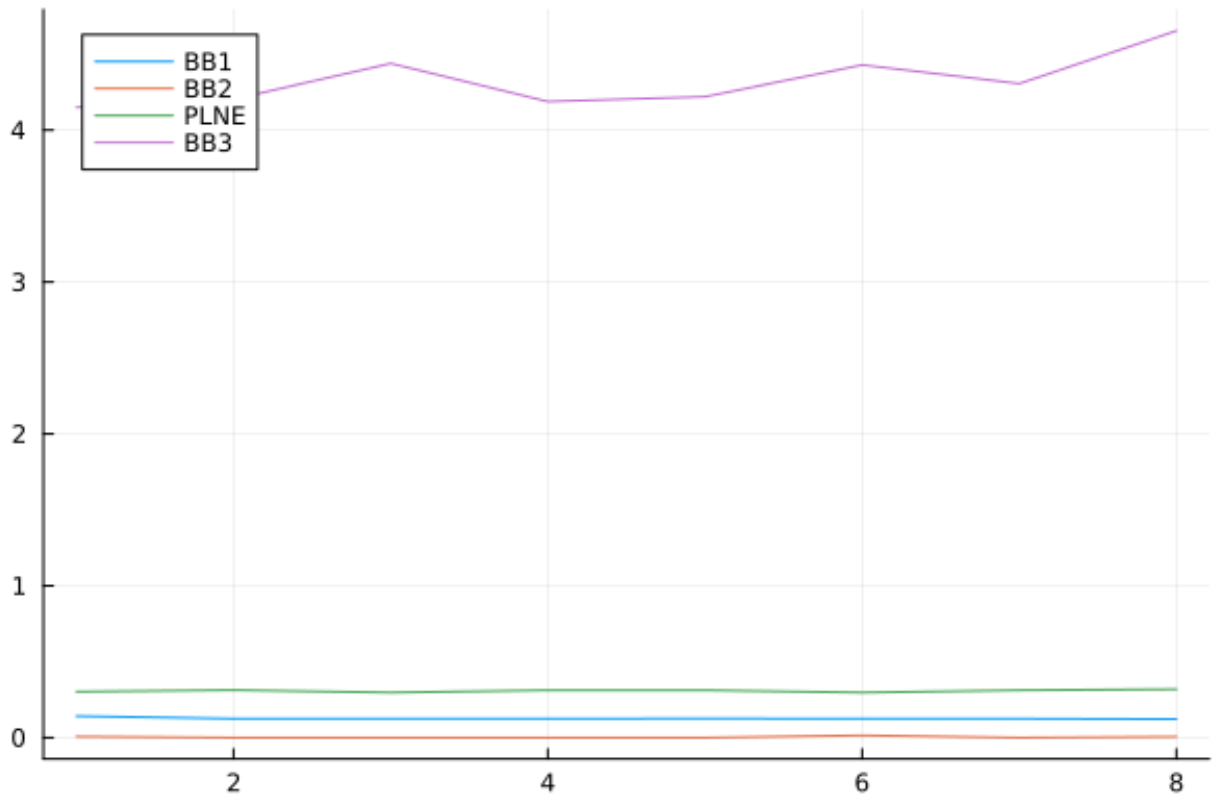
Le PLNE est quant à lui intermédiaire sur cette instance, bien que la symétrie du problème ne le rende pas spécialement performant.

Ici, nous avons montré une propriété assez intéressante du Branch and Bound utilisant la relaxtion du PLNE : il n'est pas performant dans des cas de symétries (nécessité de parcourir beaucoup de noeuds alors que la solution optimale est connue dès le début).

→ Instance C : Un patient très urgent et d'autres patients moins urgents "égaux"

On considère ici un autre cas typique des urgences : plusieurs patients similaires, avec un degré d'urgence assez faible, et un autre patient, plus gravement blessé. En réutilisant l'exemple précédent, on imagine que le groupe d'amis arrive aux urgences en même temps qu'une victime d'un accident de la route.

```
P=[5,1,1,1,1,1,1,1]
D=[50,5,5,5,5,5,5,5]
H=[0,5,5,5,5,5,5,5]


tempsv1=zeros((1,length(P)))
tempsv2=zeros((1,length(P)))
tempsPLNE=zeros((1,length(P)))
tempsv3=zeros((1,length(P)))
nmax=length(P)

for npatients in 1:length(P)
    P1=P[1:npatients]
    D1=D[1:npatients]
    H1=H[1:npatients]
    # B&B v1
    tic=time()
    Branch_Bound_v1(P,D,H)
    tac=time()
    tempsv1[1,npatients]=tac-tic
    # B&B v2
    tic=time()
    Branch_Bound_v2(P,D,H)
    tac=time()
    tempsv2[1,npatients]=tac-tic
    # PLNE
    tic=time()
    solve_PLNE(PLNE(P,D,H))
    tac=time()
    tempsPLNE[1,npatients]=tac-tic
    # B&B v3
    tic=time()
    Branch_Bound_v3(P,D,H)
    tac=time()
    tempsv3[1,npatients]=tac-tic
end

plot(1:nmax,transpose(tempsv1),label="BB1")
plot!(1:nmax,transpose(tempsv2),label="BB2")
plot!(1:nmax,transpose(tempsPLNE),label="PLNE")
plot!(1:nmax,transpose(tempsv3),label="BB3")
```



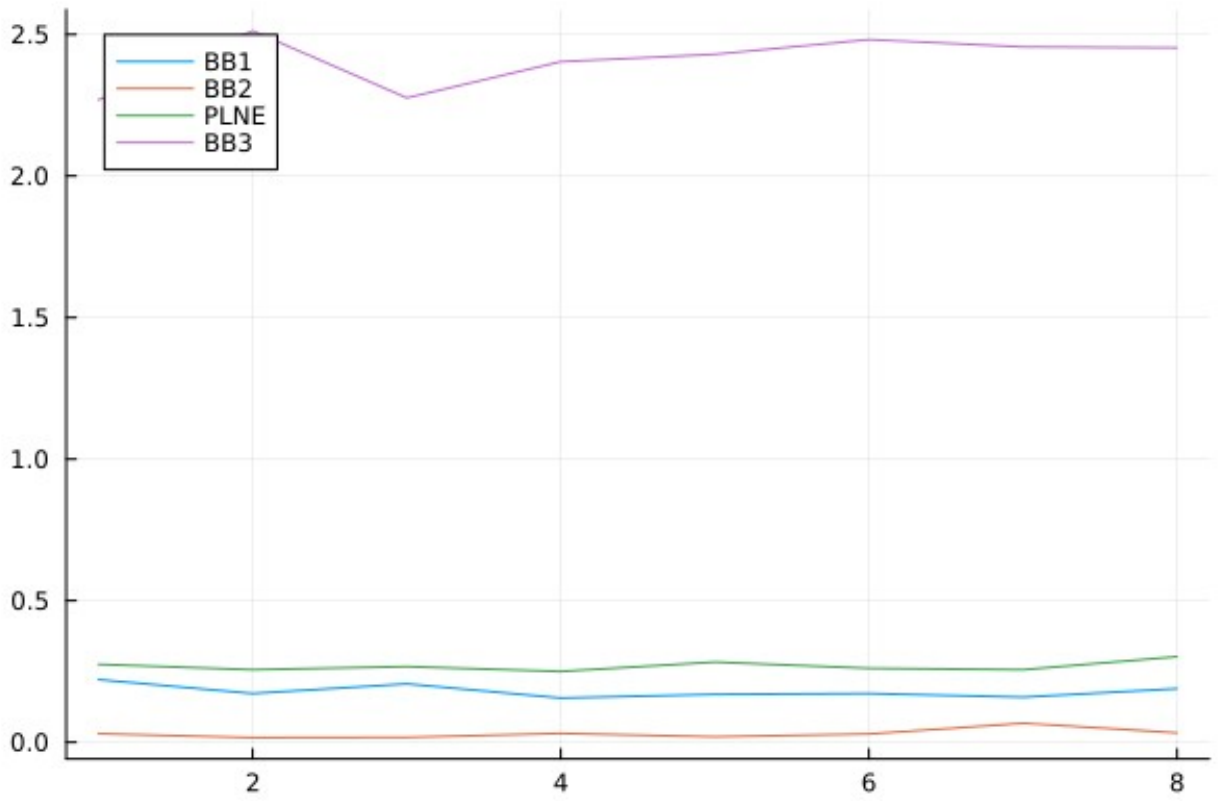
Encore une fois, la solution nous semble évidente : faire passer le patient urgent en premier, et faire passer ensuite les patients égaux (dans n'importe quel ordre). Toutefois, les algorithmes implémentés de sont pas pour autant instantanés : mis à part le B&B n°2, tous les algorithmes convergent en des durées supérieures à 0 (tout de même inférieures à la seconde pour B&B n°1 et PLNE, mais plusieurs secondes pour le 3e B&B...).

→ Instance D : Un seul patient moins urgent que d'autres patients "égaux"

Comme pour l'instance précédente, nous pouvons tout à fait considérer le cas inverse : une journée aux urgences, où la majorité des patients sont blessés à un degré d'urgence similaire, et un individu, lui, très légèrement blessé.

```
P=[0.1,1,1,1,1,1,1,1]
D=[1,5,5,5,5,5,5,5]
H=[50,5,5,5,5,5,5,5]


tempsv1=zeros((1,length(P)))
tempsv2=zeros((1,length(P)))
tempsPLNE=zeros((1,length(P)))
tempsv3=zeros((1,length(P)))
nmax=length(P)
```

```
for npatients in 1:length(P)
    P1=P[1:npatients]
    D1=D[1:npatients]
    H1=H[1:npatients]
    # B&B v1
    tic=time()
    Branch_Bound_v1(P,D,H)
    tac=time()
    tempsv1[1,npatients]=tac-tic
    # B&B v2
    tic=time()
    Branch_Bound_v2(P,D,H)
    tac=time()
    tempsv2[1,npatients]=tac-tic
    # PLNE
    tic=time()
    solve_PLNE(PLNE(P,D,H))
    tac=time()
    tempsPLNE[1,npatients]=tac-tic
    # B&B v3
    tic=time()
    Branch_Bound_v3(P,D,H)
    tac=time()
    tempsv3[1,npatients]=tac-tic
end

plot(1:nmax,transpose(tempsv1),label="BB1")
plot!(1:nmax,transpose(tempsv2),label="BB2")
plot!(1:nmax,transpose(tempsPLNE),label="PLNE")
plot!(1:nmax,transpose(tempsv3),label="BB3")
```



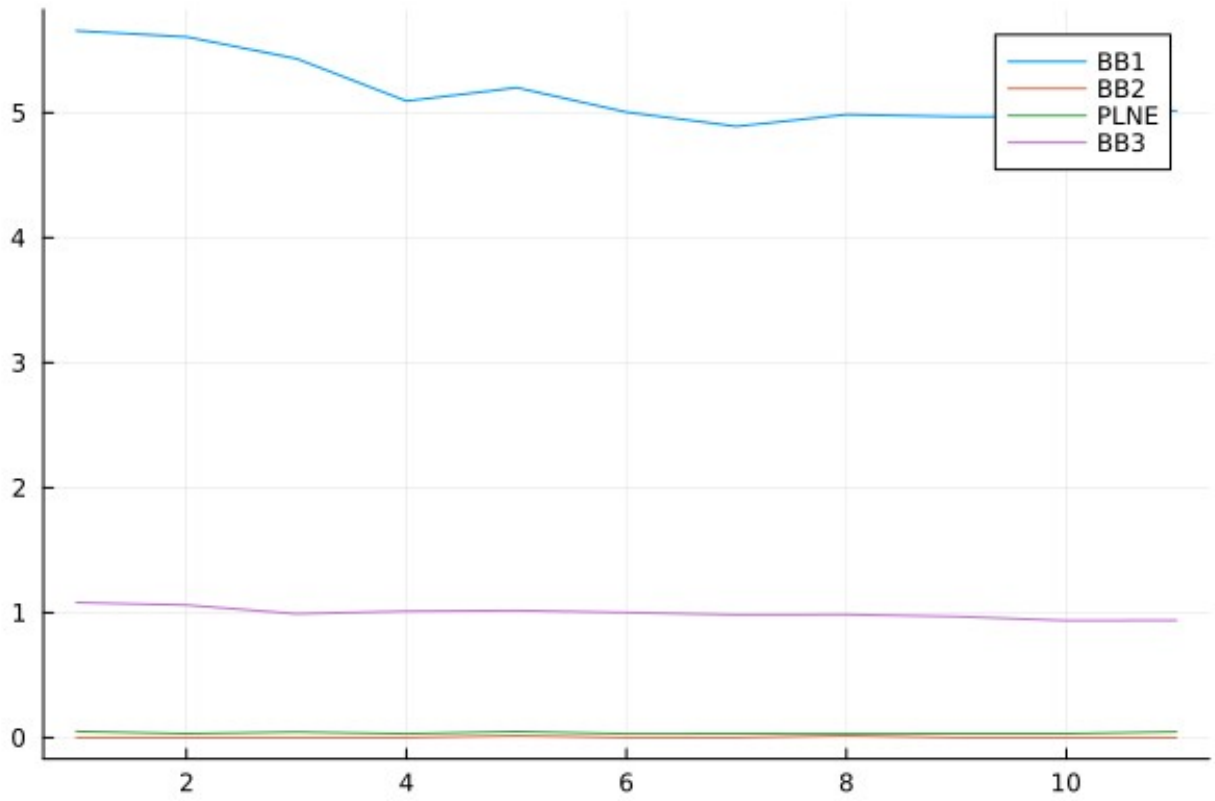
Ici, on observe globalement le même type de graphique. Toutefois, les durées de calculs sont nettement plus faibles : ceci est certainement dû au fait que la présence du nouvel individu n'ajoute pas une grande pénalité (tout se passe comme si il n'y avait que n-1 patients).

→ Instance E : problème infaisable à la main

On considère ici un problème où toutes les données s'entremêlent, empêchant de pouvoir trouver une solution à la main sans avoir recours à l'énumération de toutes les solutions possibles.

```
P=[4,7,10,1,0.1,18,5,2,7,12,1]
D=[10,12,25,23,5,15,3,35,39,10,4]
H=[25,19,48,27,32,23,3,41,80,11,18]


tempsv1=zeros((1,length(P)))
tempsv2=zeros((1,length(P)))
tempsPLNE=zeros((1,length(P)))
tempsv3=zeros((1,length(P)))
nmax=length(P)

for npatients in 1:length(P)
    P1=P[1:npatients]
    D1=D[1:npatients]
```

```
    H1=H[1:npatients]
    # B&B v1
    tic=time()
    Branch_Bound_v1(P,D,H)
    tac=time()
    tempsv1[1,npatients]=tac-tic
    # B&B v2
    tic=time()
    Branch_Bound_v2(P,D,H)
    tac=time()
    tempsv2[1,npatients]=tac-tic
    # PLNE
    tic=time()
    solve_PLNE(PLNE(P,D,H))
    tac=time()
    tempsPLNE[1,npatients]=tac-tic
    # B&B v3
    tic=time()
    Branch_Bound_v3(P,D,H)
    tac=time()
    tempsv3[1,npatients]=tac-tic
end

plot(1:nmax,transpose(tempsv1),label="BB1")
plot!(1:nmax,transpose(tempsv2),label="BB2")
plot!(1:nmax,transpose(tempsPLNE),label="PLNE")
plot!(1:nmax,transpose(tempsv3),label="BB3")
```



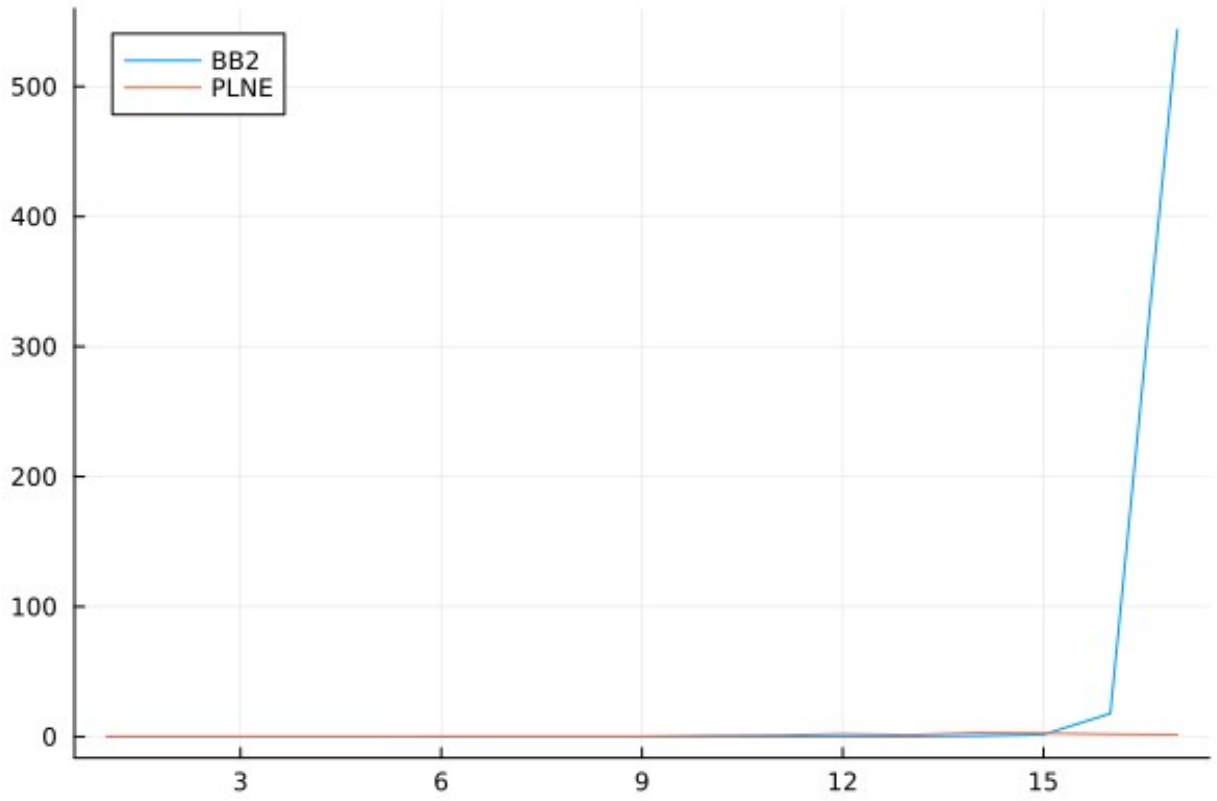
Ici, nous pouvons vanter les mérites de nos implémentations : pour un problème pouvant prendre des heures à résoudre à la main, nous pouvons ici obtenir une solution optimale en une fraction de secondes.

Tout particulièrement, le B&B n°2 et le PLNE permettent l'obtention d'une solution en un temps presque nul (obtention quasi immédiate).

Pour les simulations de cas pratiques, nous choisirons des exemples parlants avec peu de patients, et donc notre stratégie s'appuiera sur l'utilisation du B&B n°2, le plus efficace pour des instances de taille inférieure à 15 :

```
# Taille maximale des instances
nmax=17
# On initialise les vecteurs qui récupèreront les temps de calcul
tempsv2=zeros((1,nmax))
tempsPLNE=zeros((1,nmax))


for npatients in 1:nmax
    # Pour chaque taille, on choisit aléatoirement une instance
    P=vec(rand(1:10, 1,npatients))
    D=vec(rand(5:40, 1,npatients))
    H=vec(rand(30:60, 1,npatients))
    # B&B v2
    tic=time()
```

```
    Branch_Bound_v2(P,D,H)
    tac=time()
    tempsv2[1,npatients]=tac-tic
    # PLNE
    tic=time()
    solve_PLNE(PLNE(P,D,H))
    tac=time()
    tempsPLNE[1,npatients]=tac-tic
end

plot(1:nmax,transpose(tempsv2),label="BB2")
plot!(1:nmax,transpose(tempsPLNE),label="PLNE")
```



On remarque bien ici, que, pour des instances plus petites, la création du modèle fait perdre un peu de temps au PLNE tandis que le B&B n°2 trouve quasiment instantanément une solution optimale. En revanche, pour des tailles d'instances plus grandes (au dela de 15), le PLNE "rattrape" le B&B n°2 et est donc plus efficace. Nous choisirons, pour nos instances pratiques de petites tailles, l'algorithme du B&B n°2.

VI/ Etude de cas pratiques, remise en cause du modèle, éthique de modélisation

Nous allons maintenant mener une étude sur certaines instances que nous avons jugé intéressantes à étudier dans le cadre du module. Nous critiquerons les résultats fournis par les algorithmes, essaierons d'expliquer pourquoi ces résultats sont calculés, jusqu'à remonter à la remise en cause de la modélisation initiale du problème. Nous discuterons enfin du lien entre ce projet et l'éthique de modélisation dont on a parlé en cours.

Pour cette dernière partie, nous allons utiliser notre deuxième implémentation, le Branch and Bound n°2, puisqu'il fourni une solution optimale le plus rapidement.

Instance A : Un patient gravement blessé qui demande une mobilisation très longue, au milieu de patients moins graves.

Imaginons le cas d'un patient (1) très gravement blessé, qui se plaint de maux de tête et de vomissements. Les autres patients ont juste une petite coupure et ont seulement besoin d'un pansement.

Le diagnostique du patient (1) est fait par 4 practiciens, qui jugent donc différement la gravité du patient. Tous concluent qu'une mobilisation très longue dans le bloc sera nécessaire pour observer le sujet. En revanche, l'un pense à un simple malaise, l'autre pense à contusion cérébrale, le troisième à un potentiel infarctus, et le dernier craint une hémoragie cérébrale.

On pourrait se demander si même malgré le fait que le patient (1) reste au bloc longtemps, l'estimation de la gravité de ses blessures ait un impact sur son passage en premier.

Nous distinguerons ainsi 4 analyses différentes, faites par les 4 praticiens différents. Dans chaque cas, le patient (1) nécessite une longue observation, et ce, tout de suite : D=50 et H=0. Selon l'analyse faite par le praticien, la gravité de l'état du patient sera croissante : nous ne modifierons donc que P entre les 4 analyses.

Practicien 1 : Simple malaise (P=30)

```
P=[30,5,7,8] #Le patient (1) est assez grave
D=[50,5,5,5] #Le patient (1) devra être observé longtemps (10 fois plus longtemps que les autres, disons 5h contre 30 mins)
H=[0,5,4,3] # Le patient (1) doit passer maintenant

printBB(Branch_Bound_v2(P,D,H),[1])
```

```
Pénalités : 482
Ordre de passage :
1 --> Patient 4
2 --> Patient 3
3 --> Patient 2
4 --> Patient 1
```

On observe ici que le patient (1) passe en dernier.

Practicien 2 : Contusion cérébrale (P=50)

```
P=[50,5,7,8] #Le patient (1) est grave
D=[50,5,5,5] #Le patient (1) devra être observé longtemps (10 fois
plus longtemps que les autres, disons 5h contre 30 mins)
H=[0,5,4,3] # Le patient (1) doit passer maintenant

printBB(Branch_Bound_v2(P,D,H),[1])
```

```
Pénalités : 782
Ordre de passage :
1 --> Patient 4
2 --> Patient 3
3 --> Patient 2
4 --> Patient 1
```

Dans ce second cas, le patient (1) passe en dernier même si il semble nécessaire de la prendre en charge maintenant, d'autant plus que la gravité de son état devient critique (selon l'estimation du practicien 2).

Practicien 3 : Potentiel infarctus (P=70)

```
P=[70,5,7,8] #Le patient (1) est très grave
D=[50,5,5,5] #Le patient (1) devra être observé longtemps (10 fois
plus longtemps que les autres, disons 5h contre 30 mins)
H=[0,5,4,3] # Le patient (1) doit passer maintenant

printBB(Branch_Bound_v2(P,D,H),[1])
```

```
Pénalités : 982
Ordre de passage :
1 --> Patient 4
2 --> Patient 3
3 --> Patient 1
4 --> Patient 2
```

Dans le cas d'un diagnostic d'infarctus, le patient (1) passe en avant-dernier, alors que son état est vraiment critique, et que, moralement, il devrait passer en priorité.

Practicien 4 : Potentielle hémoragie cérébrale (P=80)

```
P=[80,5,7,8] #Le patient (1) est très très grave
D=[50,5,5,5] #Le patient (1) devra être observé longtemps (10 fois
plus longtemps que les autres, disons 5h contre 30 mins)
H=[0,5,4,3] # Le patient (1) doit passer maintenant

printBB(Branch_Bound_v2(P,D,H),[1])
```

```
Pénalités : 1008
Ordre de passage :
1 --> Patient 1
2 --> Patient 4
```

```
3 --> Patient 3
4 --> Patient 2
```

Ce n'est qu'en estimant la gravité de l'état du patient à 80 que l'algorithme fait passer le patient (1) en premier... On peut donc se demander comment quantifier la gravité d'une blessure.

Cette question soulève des problématiques de choix dans nos valeurs de pénalités, et donc des questionnements sur le regard humain derrière le modèle. Est-ce que réduire la gravité d'une blessure à un simple nombre est cohérent ?

De plus, on pourrait toujours contredire le choix de notre "nombre de gravité" en fonction des instances. De plus, une évaluation succinte de la gravité d'une blessure est plutôt imprécise. Et les différent practiciens n'évaluent pas de la même manière les blessures, ce qui casse toute notion de "normalité".

Quantifier la gravité des blessures par les pénalités est donc remis en cause, ce qui justifie la nécessité d'avoir un regard humain lors des choix de la modélisation et des résultats du modèle.

Nous pouvons ainsi mettre en lien cet exemple d'instances au texte étudié en cours: $Du bon usage des mod èles math é matiques$ de Ivar EKELAND. En effet, le modèle d'étude est simplifié, on résume un phénomène humain (gravité de la blessure) par des valeurs numériques. De plus, ces valeurs peuvent être estimées de différentes manières. Si ces hypothèses simplifiées sont fausses ou pas assez justes, le modèle donnera un ordre lui aussi erroné ("Garbage in, Garbage out"). Ceci montre bien la complexité de modéliser des phénomènes humains, d'autant plus lorsque la modélisation touche à la santé humaine.

On pourrait affiner le modèle en le complexifiant davantage, en définissant une règle quant aux pénalités, en adaptant les valeurs en fonction de la situation actuelle des urgences, du nombre de patients, de la gravité de chacun, ..etc.

Enfin, pour reprendre notre exemple, c'est seulement à partir d'un certain seuil que notre patient (1) passe d'avant-dernier à premier, donc l'ordre de passage est très sensible aux valeurs de pénalités dans cet exemple. Ceci montre une autre faille du modèle...

Instance B : Deux patients identiques gravement blessés qui demandent une mobilisation très longue, au milieu de patients moins graves.

Imaginons cette fois que deux patients arrivent aux urgences en même temps. Le patient (1) présente des plaies par arme blanche au niveau de l'abdomen, le patient (2) est sujet à une crise d'épilepsie.

Il doivent tous deux être pris en charge sur le champ.

On propose ici 3 modélisations différentes pour le problème considéré : à chaque fois, nous modifions la pénalité accordée à chaque patient.

```
P=[50,50,6,8,7] #Les patient 1 et 2 estimés à graves
D=[50,50,5,5,5] #Les patient (1) et (2) devront être observés
longtemps
H=[0,0,5,4,3] # Les patients (1) et (2) doivent passer maintenant
```

```
printBB(Branch_Bound_v2(P,D,H),[1,2])
```

```
Pénalités : 4038
Ordre de passage :
1 --> Patient 5
2 --> Patient 4
3 --> Patient 3
4 --> Patient 2
5 --> Patient 1
```

```
P=[70,70,6,8,7] #Les patient 1 et 2 estimés à très graves
D=[50,50,5,5,5] #Les patient (1) et (2) devront être observés
longtemps
H=[0,0,5,4,3] # Les patients (1) et (2) doivent passer maintenant

printBB(Branch_Bound_v2(P,D,H),[1,2])
```

```
Pénalités : 5538
Ordre de passage :
1 --> Patient 5
2 --> Patient 4
3 --> Patient 2
4 --> Patient 1
5 --> Patient 3
```

De la même manière que dans l'instance A, si la gravité estimée n'est pas assez grande, les deux patients passent en dernier ce qui est un choix immoral. De la même manière, augmenter légèrement la pénalité peut les faire remonter dans la file d'attente, bien que certains patients moins urgents passent tout de même en priorité.

```
P=[80,80,6,8,7] #Les patient 1 et 2 estimés à très très graves
D=[50,50,5,5,5] #Les patient (1) et (2) devront être observés
longtemps
H=[0,0,5,4,3] # Les patients (1) et (2) doivent passer maintenant

printBB(Branch_Bound_v2(P,D,H),[1,2])
```

```
Pénalités : 6112
Ordre de passage :
1 --> Patient 2
2 --> Patient 4
3 --> Patient 1
4 --> Patient 5
5 --> Patient 3
```

Pour cette dernière instance, bien qu'ils soient identiques, le patient (2) passe en 1er et le patient (1) en 3ème. Les deux patients sont bien identiques, mais pourquoi le patient (2) ne passerai pas en 3e, et le (1) et 1er ? Ceci est en réalité dû à la manière dont on a implémenté le code : nous parcourons l'arbre d'une certaine manière, et dans un certain ordre. Dès l'écriture du code, nous

avons déjà fait un choix. Le problème est ici que ce choix a des conséquences au niveau Humain : une personne va être privilégiée.

De plus, on observe que le patient (4) s'entremêle entre les 2 patients urgents, alors qu'il n'est pas si urgent, et ne devrait donc pas passer à cette position.

Nous observons ainsi que le problème observé précédemment se renforce lorsque nous considérons plusieurs patients urgents. Les urgences accueillant chaque minute de tels patients, il semble assez clair que l'utilisation de tels algorithmes dans la réalité nécessite une grande précaution.

De plus, un autre facteur est à prendre en compte. En effet, les patients ont beau être identiques, leur blessure n'est pas la même, et donc les soins administrés ne seront pas identiques. Ainsi, il est tout à fait imaginable que le bloc opératoire permettant de retirer le couteau de l'abdomen du patient (1) soit disponible avant celui nécessaire au traitement de la crise d'épilepsie du patient (2).

Dans ce cas, il serait tout à fait pertinent de traiter le patient (1) en premier. Or, cet aspect de blocs opératoires n'est pas traité dans le modèle : encore une fois, la réalité est simplifiée.

Nous pouvons d'ailleurs faire référence au texte de Viviane Despret, $Habiter en Oiseau_{¿}$. En effet, ce texte précise que les modèles "économiques" sont tous simplifiés et que l' "individualité indisciplinée" est éradiquée dans ces modèles. Cette instance en est un très bon exemple : les patients sont vus comme identiques par le modèle et sont traités de la même manière, alors qu'en réalité, la décision de l'ordre de passage se doit d'être plus complexe.

Instance C : Un patient grave et un patient plus grave au milieu de patients moins graves.

Imaginons, pour cette instance qu'un patient (1) vient d'être héliporté après une chute en ski. Il a plusieurs os de cassés. Le patient (2), quant à lui, est entre la vie et la mort après un incendie.

Moralement, on devrait croire que le patient (2) est dans un état plus grave que le patient (1). Nous lui affecterons donc une pénalité supérieure.

De plus, les deux patients devront rester longtemps au bloc opératoire. Le patient (2) devra cependant être pris en charge sur le champ, le patient (1), lui, ne devra pas tarder à être pris en compte.

```
P=[50,70,5,7,8] #Le patient 1 est estimé à assez grave et le patient 2
estimé à très grave
D=[50,50,5,5,5] # Les patients 1 et 2 devront rester longtemps au bloc
opératoire
H=[1,0,5,4,3] # Le patient 1 doit être traité sur le champs, le
patient 2 ne doit pas tarder à être traité

printBB(Branch_Bound_v2(P,D,H),[1,2])


P=[50,80,5,7,8] #Le patient 1 est estimé à assez grave et le patient 2
```

```
estimé à très très grave
D=[50,50,5,5,5] # Les patients 1 et 2 devront rester longtemps au bloc opératoire
H=[1,0,5,4,3] # Le patient 1 doit être traité sur le champs, le patient 2 ne doit pas tarder à être traité

printBB(Branch_Bound_v2(P,D,H),[1,2])


P=[60,80,5,7,8] #Le patient 1 est estimé à grave et le patient 2 estimé à très très grave
D=[50,50,5,5,5] # Les patients 1 et 2 devront rester longtemps au bloc opératoire
H=[1,0,5,4,3] # Le patient 1 doit être traité sur le champs, le patient 2 ne doit pas tarder à être traité

printBB(Branch_Bound_v2(P,D,H),[1,2])
```

```
Pénalités : 4182
Ordre de passage :
1 --> Patient 5
2 --> Patient 4
3 --> Patient 2
4 --> Patient 3
5 --> Patient 1
Pénalités : 4208
Ordre de passage :
1 --> Patient 2
2 --> Patient 5
3 --> Patient 4
4 --> Patient 3
5 --> Patient 1
Pénalités : 4798
Ordre de passage :
1 --> Patient 2
2 --> Patient 5
3 --> Patient 4
4 --> Patient 1
5 --> Patient 3
```

Nous avons distingué 3 cas différents. Dans chacun d'entre eux, nous modifions les pénalités des deux patients urgents considérés.

On observe que, pour chaque cas, le patient (2), plus urgent, passe bel et bien avant le patient (1). Nous pourrions donc être plutôt satisfaits du résultat. Toutefois, comme pour le test précédent, un patient peu urgent peut "s'intercaler" entre les deux, faisant perdre de précieuses minutes au patient (1). De la même manière, il est aussi possible qu'aucun des deux patients ne passe en premier, ce qui constitue un aberration compte tenu de la situation.

De plus, un simple accroissement de la pénalité du patient (2) l'envoie directement en haut de la liste. Malheureusement, ce changement fait de l'ombre au patient (1). En effet, celui-ci reste en dernière position, et passe derrière tous les patients non urgents.

Une augmentation de la pénalité de ce dernier le fait difficilement remonter dans l'ordre "optimal".

Cette extension du test précédent montre qu'il est aussi très important de faire attention aux écarts de pénalités entre plusieurs patients : un gap trop important peut favoriser l'un d'entre eux, au détriment des autres.

Instance D : Un patient moyennement urgent avec une durée de traitement très longue au milieu de patients moins graves.

Imaginons un patient atteint de l'appendicite (1). Son traitement n'est pas si urgent, mais devra prendre beaucoup de temps. Les autres patients ont des difficultés nettement moins graves : l'un s'est coupé (2), l'autre s'est cogné (3), le dernier s'est fait une entorse (4). On estime que l'urgence des patient est égale, mais que les pénalités ne sont pas égales.

```
P=[25,2,2,4] # Le patient 1 un est plus grave que les autres
D=[50,13,14,17] # Le patient 1 devra être opéré plus longtemps que les autres patients
H=[10,10,10,10] # Les 4 patients sont aussi urgents les uns que les autres
#(l'appendicite n'étant pas forcément urgent dans un premier temps)

printBB(Branch_Bound_v2(P,D,H),[1])
```

```
Pénalités : 414
Ordre de passage :
1 --> Patient 1
2 --> Patient 4
3 --> Patient 2
4 --> Patient 3
```

On observe ici que l'ordre auquel nous pourrions penser est bien celui rendu par l'algorithme. En effet, le patient souffrant de l'appendicite passe bel et bien en premier. Ainsi, la modélisation du problème semble globalement être correcte.

Désormais, modifions légèrement le problème : parmi les autres patients, celui qui s'est coupé s'est en fait coupé un bout de doigt (2), celui qui s'est cogné souffre en réalité d'un traumatisme cranien (3), et celui qui s'est fait une entorse est en fait un joueur de football connu et doit jouer le match de demain (4).

Ainsi, les gravités changent dans notre modèle :

```
P=[25,20,20,40] # Les pénalités augmentent, car les retard sont plus embettants que dans l'instance précédente
D=[50,13,14,17] # Les temps de traitement des patients ne changent pas
H=[10,10,10,5]  # Le joueur de foot doit être pris en charge plus tôt
```

```
pour pouvoir se rééduquer plus vite
# On considère aussi une éventuelle influence des médias

printBB(Branch_Bound_v2(P,D,H),[1])
```

```
Pénalités : 1390
Ordre de passage :
1 --> Patient 4
2 --> Patient 2
3 --> Patient 3
4 --> Patient 1
```

Cette légère modification, pouvant être considérée comme une précision du problème, change complètement le résultat, et réarange considérablement l'ordre : le patient souffrant de l'appendicite (1) passe cette fois en dernier.

Ainsi, au travers de cet exemple, plusieurs défauts sont soulevés :

- Le médecin estime grossièrement la blessure pour en donner une "note", mais, est-il assez précis ? Pour palier à cela, devrait-il mener des analyses plus approfondies, afin d'estimer la gravité de la blessure (auquel cas du temps précieux pourrait être perdu...) ?
- Les caractéristiques des patients ne sont pas connues dès leurs arrivées aux urgences. Ils seront inclus au modèle une fois qu'ils seront diagnostiqués. Mais, ce temps d'attente de diagnostic n'est pas inclu au modèle, et en fonction de la gravité d'une blessure ou d'une autre, ce temps d'attente peut jouer sur l'aggravement (ou l'inverse) de l'état du patient. Ainsi, quand bien même nous aurions un modèle exact vis-à-vis de la gravité des patients et des pénalités, il faudrait prendre en compte cet effet de "gonflement" (ou dégonflement) de la gravité lié à l'attente de diagnostic.

Nous allons traiter un exemple pour étudier ce phénomène.

Instance E : phénomène de gonflement/dégonflement de la gravité lié à l'attente de diagnostic

En reprenant l'exemple précédent, si les patients sont diagnostiqués dès leur arrivée :

```
P=[25,20,20,40] # Les pénalités augmentent, car les retard sont plus
embettants que dans l'instance précédente
D=[50,13,14,17] # Les temps de traitement des patients ne changent pas
H=[10,10,10,5]  # Le joueur de foot doit être pris en compte plus tôt
pour pouvoir se rééduquer plus vite

printBB(Branch_Bound_v2(P,D,H))
```

```
Pénalités : 1390
Ordre de passage :
1--> Patient 4
```

```
2--> Patient 2
3--> Patient 3
4--> Patient 1
```

Si les patients sont diagnostiqués après un certain temps, leur état respectif a pu changer :

- La crise d'appendicite du patient (1) devient plus grave et urgente (H : 10 → 5, D inchangé, P : 25 → 30)
- L'état du patient (2) ne change pas (H : 10 → 20, D, P inchangés)
- le patient ayant subit un traumatisme cranien (3) est tombé dans les pommes dans la salle d'éttente : il devient plus urgent, grave et demandera plus de temps au bloc (H : 10 → 5, D : 14 → 30, P : 20 → 35)
- le joueur de foot (4) voit son match de demain annulé : il devient moins urgent de le soigner, la pénalité diminue (H : 5 → 20, D inchangé, P : 40 → 20)

```
P=[30,20,35,20]
D=[50,13,30,17]
H=[5,20,5,20]

printBB(Branch_Bound_v2(P,D,H))
```

```
Pénalités : 2310
Ordre de passage :
1--> Patient 3
2--> Patient 2
3--> Patient 4
4--> Patient 1
```

En considérant ces modifications, nous observons que l'ordre des patients 3 et 4 ont été permutés, ce qui semble tout à fait légitime puisque le joueur de football (4) ne doit plus être soigné dans l'immédiat. Toutefois, celui-ci, tout comme le patient (2) qui s'est coupé un doigt, passent tout de même avant le patient (1) souffrant de l'appendicite.

Ainsi, l'évolution de l'urgence du patient (1) n'a pas été suffisante pour le faire passer plus tôt : il va finalement passer après un patient souffrant d'une simple entorse.

Dans le cas où nous souhaitons créer un modèle considérant ces fluctuations, nous devons être en capacité de modifier les valeurs attribuées rapidement, et de manière assez fiable, pour que les choix réalisé n'impactent pas négativement le nouvel ordre construit.

Evidemment, les instances choisies ont permis de montrer les défauts du modèle, mais d'autres défauts pourraient bien entendu être relevé en terme d'éthique.

A propos des patients dans un état actuel peu grave, mais succeptibles de chuter (ex : défaut de respiration potentiellement transformé en crise cardiaque), peut-on vraiment résumer cet état avec une "pénalité" ? Faut-il introduire de l'aléatoire ? Si oui, d'autres questions éthiques se soulèvent, car le choix de l'ordre des patients fait intervenir de l'aléatoire, qui, par essence ne sera pas nécessairement juste...

V/ Conclusion

Au final, ce projet a été une grande source d'enrichissements. En effet, selon nous, celui-ci eu deux principales utilités :

- un aspect mathématique : réussir à implémenter des algorithmes étudiés en cours, en traduisant nos pensées par des lignes de codes; mais aussi réussir à modéliser un problème concret et réel. La mise en place de tels algorithmes a été assez stimulante, puisqu'elle nous a offert une certaine liberté vis-à-vis de l'implémentation. De plus, être confronté à d'éventuels problèmes techniques nous a poussé à mettre à jour notre manière de penser. Ce projet nous a donc été, pour nous, une source de réflexions, d'adaptation, de discussions, et de comparaisons.

- un aspect éthique : nous avons pu voir, de manière plus concrète et visuelle, que le modèle d'étude peut être remis en cause. Nous avons ainsi tenté d'appliquer des raisonnement moraux similaires à ceux abordés en cours d'éthique, transformant ce projet en une illustration des textes étudiés. Ce projet illustre parfaitement les problèmes liés à la simplification de la réalité : décrire un phénomène social ou humain est très complexe et amène à se poser des questions.

En réalisant ce projet, nous avons donc appris à modéliser et à transmettre nos raisonnements à une machine, mais nous avons tout autant appris à prendre du recul sur notre travail, ce qui nous a amené à se questionner sur le modèle, qui nous paraissait pourtant plutôt bon et réaliste au premier abord.