TP3: Randonnée et optimisation

Evan Voyles

27 Mai

Code source disponible ici

Exercice 1 On considère un vecteur σ (qui contient 0 ou 1 pour l'entrée i selon que l'objet i ait été choisi pour remplir le sac à dos). On choisit initialement un vecteur σ tel que

- En partant d'une configuration σ, on propose une configuration σ' en modifiant aléatoirement l'un des σ_i par $1 - \sigma_i$;

On implémente notre lois de proposition pour l'algorithme de Metropolis avec la fonction `propose_sigma_prime`

```
# A partir d'une configuration \sigma, proposer \sigma' en modifiant aleatoirement l'un des
# \sigma_i par (1 - \sigma_i)
propose_sigma_prime <- function(sig) {

    n <- length(sig)
    i <- sample(1:n, 1)

    sig[i] <- 1 - sig[i]
    sig
}
```

- Pour une configuration proposée, on considère qu'elle n'est pas admissible si le poids du sac à dos dépasse P. Dans ce cas, elle est automatiquement refusée.

```
# Return the total weight of the items carried
total_weight <- function(sig, poids) { sum(sig * poids) }

# Return the total value of the items carried
total_valeur <- function(sig, valeur) { sum(sig * valeur) }
```

- Si la configuration est admissible, alors on poursuit l'algorithme de Metropolis en calculant le taux d'acceptation r. Pour cela, on considère la loi cible π qui est proportionnelle à $\exp(V_\sigma)$ où V_σ est la laveur du sac pour la configuration σ. $\frac{\pi(\sigma')}{\pi(\sigma)} = \exp(V_{\sigma'} - V_\sigma)$

```
# En sachant que \pi(\sigma') / \pi(\sigma) = exp(V_{\sigma'} - V_\sigma),
# On calcule le taux d'acceptation
#' @param
#' sig       \sigma actuel
#' sig_prime \sigma propose
#' valeurs   un vecteurs des valeurs correspondant aux objets signalés par \sigma
taux_accept <- function(sig, sig_prime, valeurs, k = 1) {
    Vsig  <- total_valeur(sig, valeurs)
    Vsigp <- total_valeur(sig_prime, valeurs)
    min(exp((Vsigp - Vsig) / k), 1)
}
```

Muni avec une lois de proposition et des conditions de rejet, on procède pour écrire l'algorithme de Metropolis qui renvoye un vectuer sigma pour la prochaine itération.

```
next_sigma <- function(sig, poids, valeurs, P, k = 1) {

    sig_prime <- propose_sigma_prime(sig)

    if (total_weight(sig_prime, poids) > P) {
        return(sig)
    }

    r <- taux_accept(sig, sig_prime, valeurs, k)

    if (runif(1) < r) {
        # Then we accept sig_prime
        return(sig_prime)
    } else {
        # We keep sig
        return(sig)
    }
}
```

En marchant aléatoirement dans l'espace de tous les permutations de σ, cette algorithme nous permet de générer une suite des $(\sigma_n)_n$ qui obéit deux conditions. La première c'est que la peseur total de σ_n ne dépasse jamais une valeur donnée P. Deuxièmement, la valeur de V_{σ_n} à la tendance à croire.

Exercice 2 Mettre en pratique avec les données suivantes :

```
item <- c("map", "water", "sandwich", "glucose", "tin", "banana", "apple", "cheese",
          "beer", "suntan_cream", "camera", "T-shirts", "trousers", "umbrella",
          "waterproof_trousers", "waterproof_overclothes", "note-case", "sunglasses", "towel",
          "socks", "book")
p <- c(0.05, 1, 0.7, 0.1, 0.5, 0.2, 0.3, 0.4, 0.5, 0.3, 1, 0.8, 0.4, 0.7, 0.4, 0.3, 0.4, 0.1, 0.3, 0.4,
v <- c(150, 300, 160, 60, 45, 60, 40, 30, 180, 70, 30, 100, 10, 40, 70, 75, 50, 80, 12, 50, 30)
P <- 7
n_it <- 1e3
n_items <- length(item)
```

On commence en écrivant une fonction qui simule une trajectoire de notre Chaîne de Markov qui part de $\sigma_0 = \mathbf{0}$.

```
sim_traj <- function(n, k = 1) {
    sigma0 <- rep(0, n_items)
    traj <- list(sigma0)

    for (i in 2:n) {
        traj[[i]] <- next_sigma(traj[[i - 1]], p, v, P, k)
    }

    traj
}
```

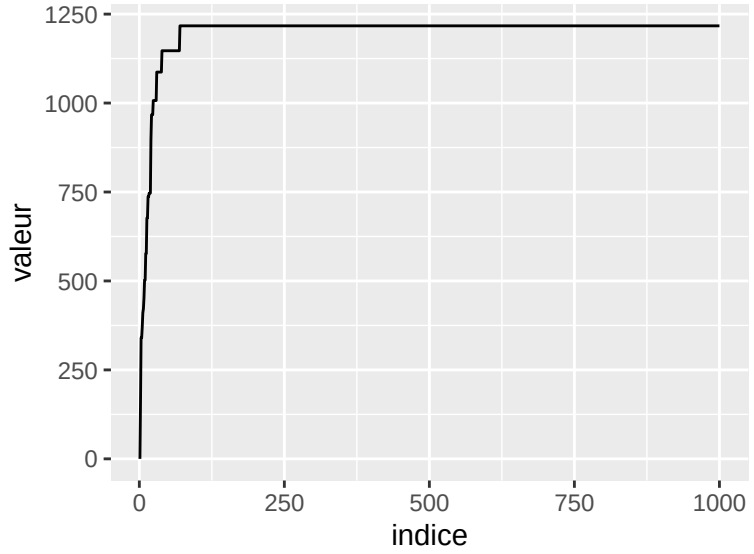
Ensuite on utilise la fonction `purrr::map_dbl` pour calculer la valeur de chaque configuration σ_n dans notre vecteur `traj`.

```
sim_traj_valeur <- function(n, k = 1) {
```

```
    traj <- sim_traj(n, k)
    val <- map_dbl(traj, total_valeur, v)

    list(t = traj, v = val)
}
```

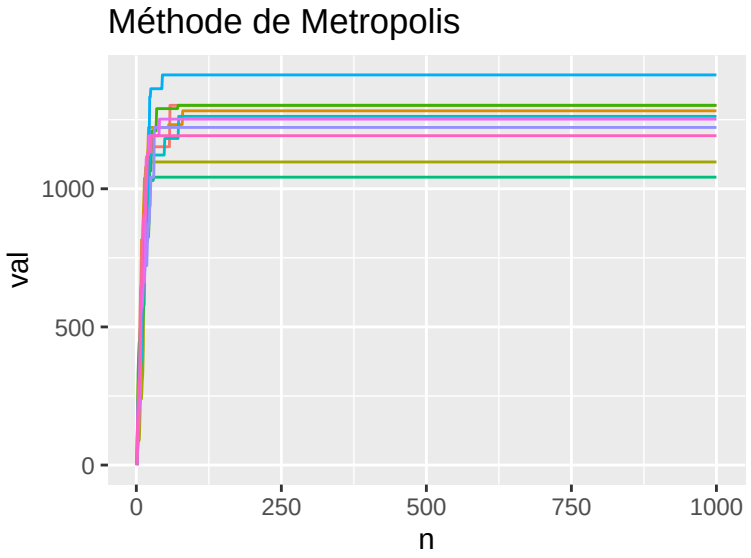
et on plotte une trajectoire de V_{σ_i}

```
n <- 1000
n_traj <- 20

val_traj <- sim_traj_valeur(n)

tibble(ind = 1:n, vals = val_traj$v) |>
    ggplot(aes(ind, vals)) +
    geom_line() +
    labs(x = "indice", y = "valeur")
```



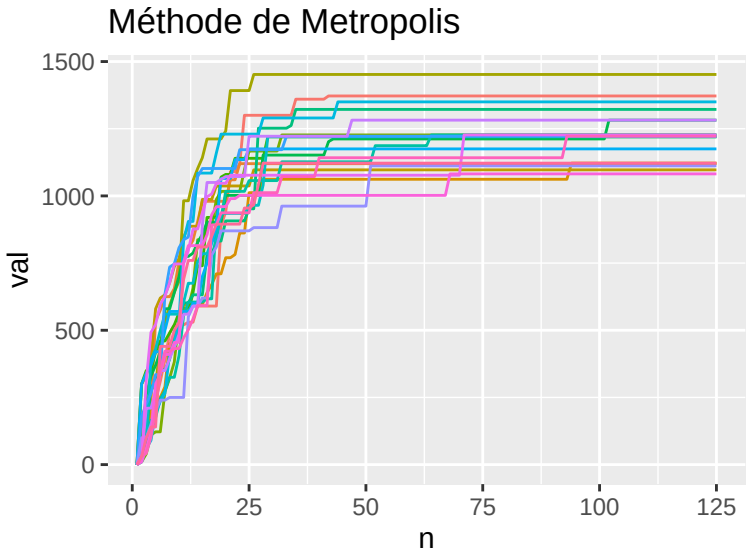
On pourrait simuler plusiers trajectoires sur le même graphique avec la fonction `plot_traj_val`.

```
plot_traj_val(1000, 10)
```



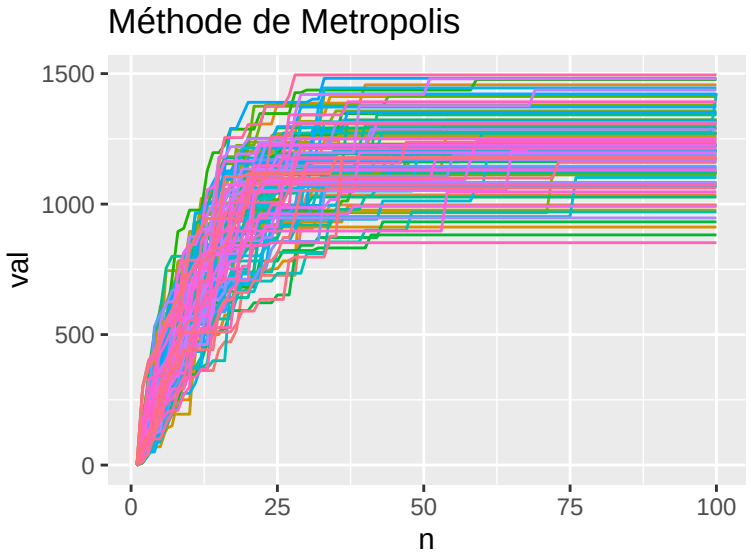
On remarque qu'il y a beaucoup de transitions lorsque $n < 125$ donc on lance encore des trajectoires mais cette fois-ci avec $n = 125$.

```
plot_traj_val(125, 20)
```



Si on aimerait trouver visuellement un maximum, on lance 100 trajectoires

```
set.seed(10)
plot_traj_val(100, 100)
```



Il nous semble qu'il existent plusiers configurations "stables" où c'est difficile d'accepter tout σ' proposé, et c'est pour cela que nous observons des trajectoires qui se stabilisent. On voit que certaines trajectoires s'immobilisent avec une valeur à peu près 800, tandis ce que des autres trajectoires dépassent à peine le seuil de 1500.

(b) Parmi les configurations visitées, laquelle correspond à la meilleure valeur totale du sac ? Pour quelle valeur de n est-elle obtenue? Renvoyer le contenu du sac correspondant.

On pourrait écrire une fonction pour trouver la valeur maximum d'une trajectoire et aussi son premier indice.

```
# Get the n indice of the max value in a val_traj
get_max_val <- function(n) {

    val_traj <- sim_traj_valeur(n)
    max_val <- max(val_traj$v)
    i_max <- purrr::detect_index(val_traj$v, function(x) { x == max_val })

    list(max = max_val, i = i_max, sigma = val_traj$t[[i_max]])
}
```

On utilise cette fonction pour une trajectoire quiconque de taille $n = 1000$ et on extrait l'indice maximum et la configuration correspondante du sac.

```
mv <- get_max_val(1000); mv
```

```
## $max
## [1] 1247
##
## $i
## [1] 37
##
## $sigma
##  [1] 1 1 1 1 1 1 1 1 0 1 1 1 1 1 0 0 0 1 1 1 1 1 0
```

Ce qui correspond à emmener les objets suivants

```
item[as.logical(mv$sigma)]
```

```
##  [1] "map"          "water"        "sandwich"     "glucose"      "tin"          "banana"       "appl
##  [9] "suntan_cream" "camera"       "T-shirts"     "trousers"     "note-case"    "sunglasses"   "towe
```

qui pèse

```
total_weight(mv$sigma, p)
```

```
## [1] 6.95
```

ce qui est admissible.

(c) Relancer 100 fois l'algorithme en allant jusqu'à $n = 10000$. Commenter les résultats.

```
# Now re run this algorithm 100 times with n = 1e5.
# to figure out the max of all max values

n_traj <- 1e3
n <- 1e2

max_val_traj <- list()

for (i in 1:n_traj) {
    max_val_traj[[i]] <- get_max_val(n)
}

# Now extract the max values
max_vals <- map_dbl(max_val_traj, function(traj) { traj$max } )
max_sigmas <- map(max_val_traj, function(traj) { traj$sigma } )

# The MAX value that I can get is 1532
# Let's find the index of the max max val

# Function factory
equals <- function(k) {
    function(x) {
        x == k
    }
}

i_max <- detect_index(max_vals, equals(max(max_vals)))
winning_configuration <- max_sigmas[[i_max]]

list(
        val = total_valeur(winning_configuration, v),
        p = total_weight(winning_configuration, p),
        items = item[as.logical(winning_configuration)]
    )
```

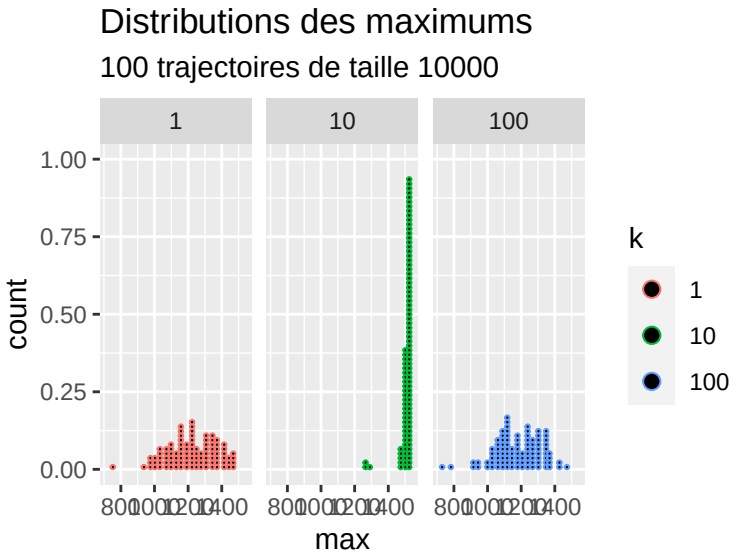
```
## $val
## [1] 1532
##
## $p
## [1] 6.75
##
## $items
##  [1] "map"                    "water"                  "sandwich"               "glucose"
##  [5] "tin"                    "banana"                 "apple"                  "cheese"
##  [9] "beer"                   "suntan_cream"           "T-shirts"               "waterproof_trousers"
## [13] "waterproof_overclothes" "note-case"              "sunglasses"             "towel"
## [17] "socks"
```

La valeur maximum que je trouve parmi les valeurs maximum d'un grand nombre de trajectoires est bien 1532.

(d) Réessayer en prenant cette fois une loi cible π qui est proportionnelle à $\exp(V_\sigma/10)$, puis $\exp(V_\sigma/100)$.

On introduit le paramètre k pour encoder une loi cible proportiennelle à $\exp(V_\sigma/k)$, et on écrit une fonction pour avoir le vecteur des valeurs maximums pour n_{traj} trajectoires de taille n.

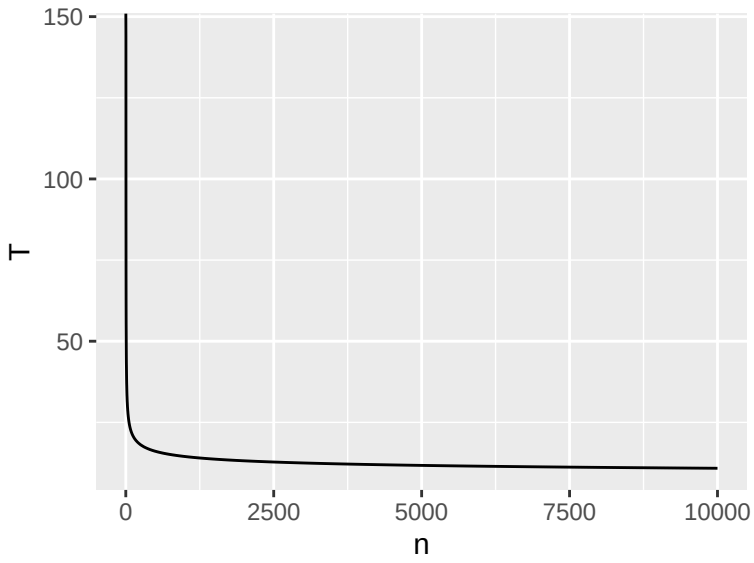
```
# Let's check out the distributions of the means

get_max_metropolis <- function(n_traj, n, k) {
    map_dbl(rep(n, n_traj), function(n) {
      traj <- sim_traj_valeur(n, k)
      traj$v[[n]]
    })
}
```



Il nous semble que diviser la valeur V_σ par 10 pendant le calcul du taux d'acceptation est optimale. Changer ce taux a l'effet de réduire les les σ' qui sont proposés si la différence de valeur entre $V_{\sigma'}$ et V_σ. Cependant, si on divise cet écart par 100, peut-être il rend le taux d'acceptation trop petit, et on n'accepte pas assez des nouveaux σ' proposés. On conclut que la valeur de 10 reste équilibré entre 1 et 100.

Exercice 3 Implémenter l'algorithme du recuit simulé avec un schéma de température logarithmique $T_n = 100/\log(n)$

```
temp_fn <- function(n) {
    100 / log(n)
}
```



(a) Quelle va être la différence par rapport à l'algorithme de Metropolis ?

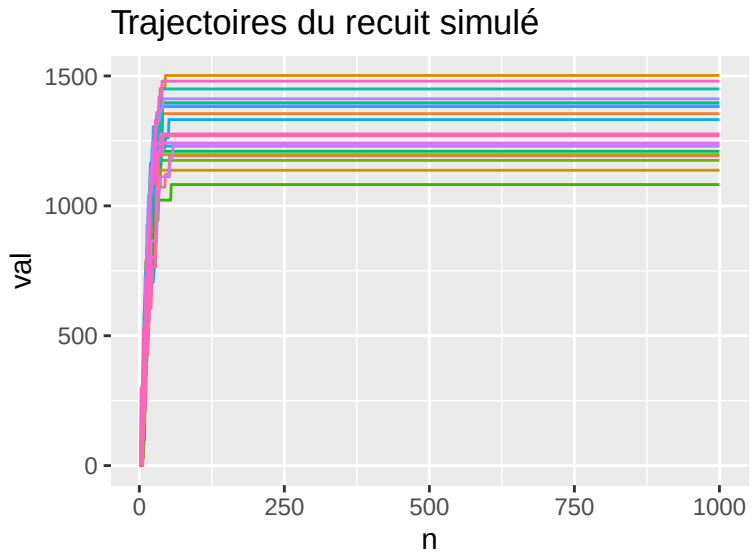
La différence principale apparait pendant le calcul du taux d'acceptation. Avec l'algorithme du recuit simulé on va influencer ce taux d'acceptation en divisant $V_{\sigma'} - V_{\sigma}$ par la température T_n. Contrairement à la partie (d) de l'exercice précedente, cette diviseur sera une valeur en fonction de n, l'itération actuelle.

```
taux_accept_recuit <- function(sig, sig_prime, valeur, T) {
    min(exp((total_valeur(sig, valeur) - total_valeur(sig_prime, valeur)) / T), 1)
}
```

On modifie alors les fonctions précedentes pour utiliser le nouveau taux d'acceptation.

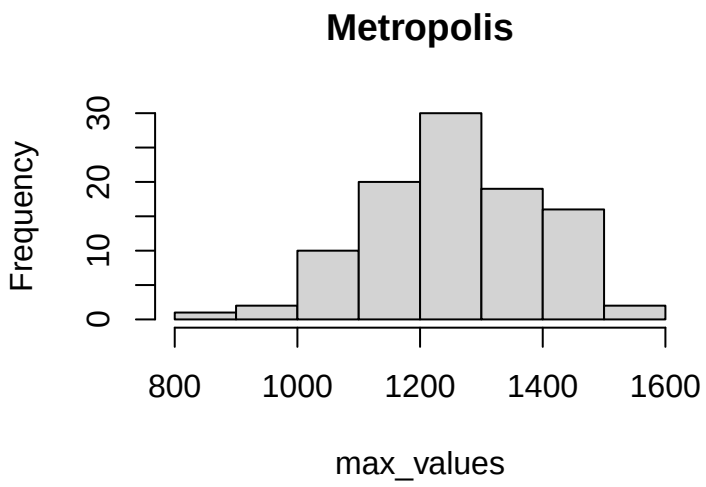
(b) Comparer les résultats avec ceux obtenus précédemment

```
plot_traj_val_recuit(1000, 20)
```



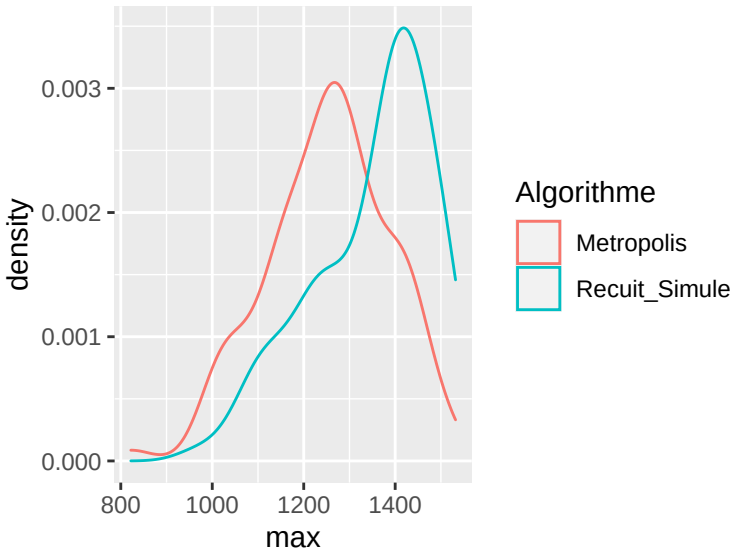
Visuellement on ne voit pas une grande différence. Pour comparer les deux algorithmes, on pourrait étudier la distribution des valeurs maximums des trajectoires générées selon l'algorithme de Metropolis et selon l'algorithme du recuit simulé.

```r
n_traj <- 100
n <- 10000

max_values <- get_max_metropolis(n_traj, n, 1)
max_values_recuit <- get_max_recuit(n_traj, n)
```




```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##     822    1169    1264    1252    1342    1520
```


```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##     970    1259    1382    1348    1440    1532
```



L'algorithme du recuit simulé non seulement a une moyenne de la valeur maximum d'une trajectoire qui est supérieure à celle des trajectoires générées par l'algorithme de Metropolis, il a aussi un minimum de 862 plus grand que 712 du Metropolis. On observe que pour l'algorithme du recuit simulé, il y a plus de masse pour les valeurs plus grandes; la distribution des maximums observés est penché à droite. Regardons les deux derniers bâtons dans les histograms pour voir à telle difference l'algorithme du recuit simulé génère des valeurs proches du maximum.

Il serait peut etre aussi intéressant de voir combien de trajectoires de taille n finit par trouver la valeur maximum, $V^* = 1532$.

On présente tout d'abord le nombre de trajectoires qui, après n itérations, sont tombés sur la valeur optimale $V^* = 1532$, suivi par le portion sur toutes les trajectoires.

```
n_met <- sum(max_values == 1532); n_met; n_met / n_traj
```

```
## [1] 0
```

```
## [1] 0
```

```
n_rec <- sum(max_values_recuit == 1532); n_rec; n_rec / n_traj
```

```
## [1] 2
```

```
## [1] 0.02
```

On finit la discussion en remarquant que l'on tombe sur la même configuration optimale avec l'algorithme du recuit simulé. De plus, il est plus probable que l'on tombe sur la configuration optimale en utilisant la méthode du recuit simulé.

```
n_traj <- 1e3
n <- 1e2

max_val_traj <- list()
for (i in 1:n_traj) {
    max_val_traj[[i]] <- get_max_val_recuit(n)
}

# Now extract the max values
max_vals <- map_dbl(max_val_traj, function(traj) { traj$max } )
max_sigmas <- map(max_val_traj, function(traj) { traj$sigma } )
```

```
i_max <- detect_index(max_vals, equals(max(max_vals)))
winning_configuration <- max_sigmas[[i_max]]

list(
        val = total_valeur(winning_configuration, v),
        p = total_weight(winning_configuration, p),
        items = item[as.logical(winning_configuration)]
    )
```

```
## $val
## [1] 1532
##
## $p
## [1] 6.75
##
## $items
##  [1] "map"                    "water"                  "sandwich"               "glucose"
##  [5] "tin"                    "banana"                 "apple"                  "cheese"
##  [9] "beer"                   "suntan_cream"           "T-shirts"               "waterproof_trousers"
## [13] "waterproof_overclothes" "note-case"              "sunglasses"             "towel"
## [17] "socks"
```

Exercice 4 On teste une troisième technique d'exploration, appelée algorithme génétique. Il s'agit ici de reproduire les principes du brassage génétique grâce à la sélection et la mutation.

- On simule m configurations

On produit m configurations en utilisant l'algorithme du recuit simulé avec n_{rec} itérations.

```
m <- 20
n_rec <- 10
n_gen <- 100


get_init_config <- function(n_rec) {
    traj <- sim_traj_recuit(n_rec)
    traj[[n_rec]] # extract the final configuration of a trajectory
}

# Generate m initial configurations running the simulated annealing algorithm
get_init_configs <- function(m, n_rec) {
    init_configs <- list()
    for (i in 1:m) {
        init_configs[[i]] <- get_init_config(n_rec)
    }
    init_configs
}
```

On pourrait avoir, par example :

```
get_init_configs(10, 100)
```

```
## [[1]]
##  [1] 1 0 1 1 0 1 1 1 1 1 1 1 1 1 1 0 1 1 1 0
##
## [[2]]
##  [1] 1 1 0 1 1 1 1 1 1 0 1 1 0 1 1 1 0 1 0 1 0
```

```
##
## [[3]]
##  [1] 1 1 1 1 1 0 1 0 1 0 1 1 0 1 0 1 1 1 0 1 0
##
## [[4]]
##  [1] 1 1 1 1 1 1 1 0 1 1 1 1 0 0 0 1 1 1 1 1 0
##
## [[5]]
##  [1] 1 1 1 1 1 1 0 0 0 0 1 1 0 1 1 1 1 1 1 1 0
##
## [[6]]
##  [1] 0 0 0 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1
##
## [[7]]
##  [1] 1 1 0 1 1 1 0 0 1 1 1 1 0 1 1 1 1 0 1 1 0
##
## [[8]]
##  [1] 1 1 0 1 0 1 1 1 1 1 1 0 1 1 0 1 1 1 0 1 1
##
## [[9]]
##  [1] 1 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 0
##
## [[10]]
##  [1] 1 1 1 1 1 1 1 1 1 1 0 1 0 1 0 1 1 1 0 1 0
```

- Pour chacune des m configurations, on définit l'état suivant selon ce principe :

1. Avec probabilité 1/2, l'état suivant est égal à la meilleure des m configurations actuelles
2. Avec probabilité 1/2, on utilise `propose_sigma_prime` pour générer un σ' admissible.

Pour procéder, on devrait définir une nouvelle fonction qui, parmi une liste de m configurations, nous renvoie la configuration avec la meilleure valeur.

```
# Given a list of m current configs, choose the one with the best value
get_best_config <- function(list_configs) {

    # Calculate the values of each config
    vals <- map_dbl(list_configs, total_valeur, v)
    # Compute the max value and find the first index of the max
    m_val <- max(vals)
    imax <- detect_index(vals, equals(m_val))

    list_configs[[imax]]
}
```

Ensuite, on écrit une fonction pour générer la nouvelle génération de m configurations, suivant la règle dans l'énoncé de l'algorithme.

```
get_next_sigma_gen <- function(sig, best, poids) {

    sig_prime <- propose_sigma_prime(sig)

    if (runif(1) < 0.5) {
        return(best)
    } else {
        if (total_weight(sig_prime, poids) > P) {
```

```
            return(sig)
        } else {
            return(sig_prime)
        }
    }
}
```

- Appliquer cette nouvelle méthode avec $m = 100$ à vos données et commenter.

On écrit tout d'abord la fonction pour simuler une trajectoire suivant l'algorithme génétique.

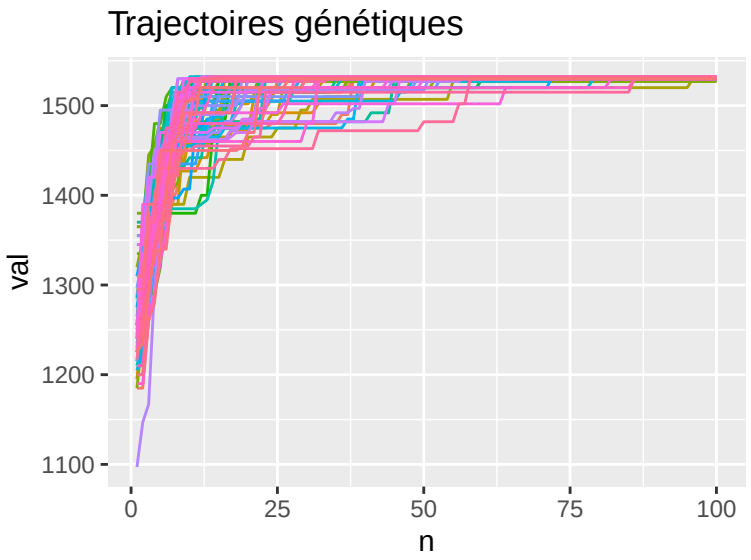
```
sim_traj_gen <- function(m, n_gen, n_rec_initial = 20) {

    M <- list(get_init_configs(m, n_rec_initial))

    # Now for each step, get the best configuration from the m current configurations
    for (i in (seq_len(n_gen - 1) + 1)) {

        best_config <- get_best_config(M[[i - 1]])

        current_gen <- M[[i - 1]]
        next_gen <- list()

        # Now for each specimen in the generation,
        for (j in 1:m) {
            next_gen[[j]] <- get_next_sigma_gen(current_gen[[j]], best_config, p)
        }

        M[[i]] <- next_gen
    }

    M
}
```

Puis une fonction qui, pour une famille de m configurations donné, renvoie la valeur maximum.

```
best_gen_vals <- function(m, n, n_rec = 20) {
    M = sim_traj_gen(m, n, n_rec)
    map_dbl(M, function(traj) {
      best <- get_best_config(traj)
      total_valeur(best, v)
    })
}
```

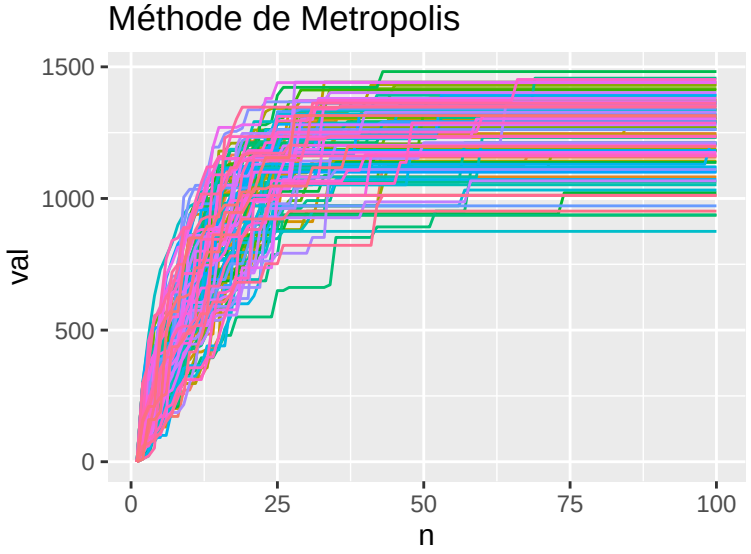
Plotons 100 trajectoires de 100 générations des familles de tailles $m = 100$:

```
plot_traj_val_gen(m = 100, n = 100, n_traj = 100)
```



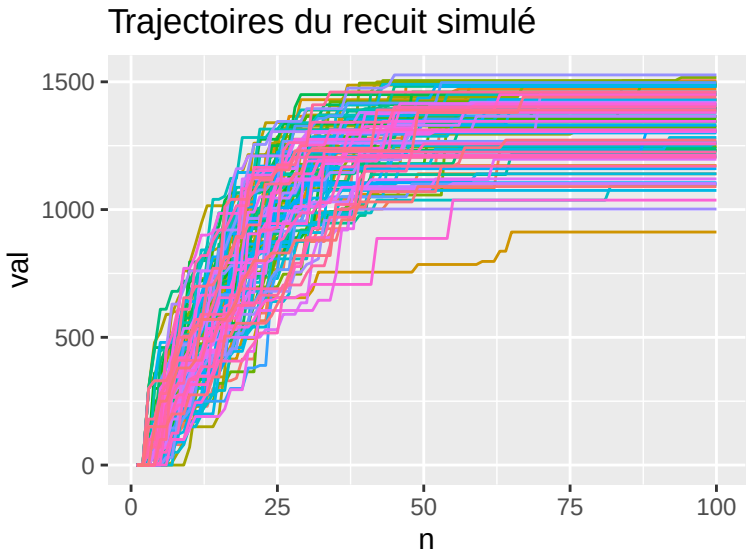
Comparons cela avec les trajectoires selon la méthode de Metropolis

```
plot_traj_val(n = 100, n_traj = 100)
```



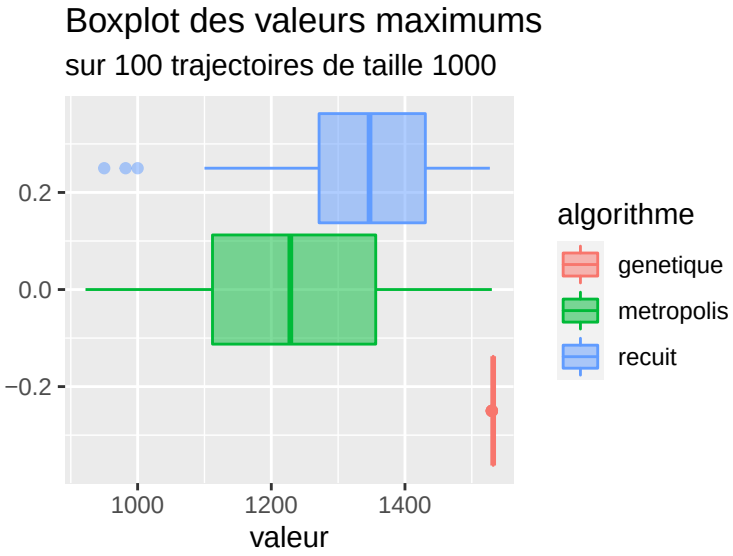
et l'algorithme du recuit simulé

```
plot_traj_val_recuit(n = 100, n_traj = 100)
```



Il est évident que les trajectoires converge vers le maximum $V = 1532$ le plus vite avec l'algorithme génétique. Les trajectoires sont beaucoup plus serrés que pour les autre méthodes.

```
max_vals_gen <- get_max_gen(m, n, n_traj)
max_vals_rec <- get_max_recuit(n_traj, n)
max_vals_met <- get_max_metropolis(n_traj, n, 1)

df <- tibble(metropolis = max_vals_met, recuit = max_vals_rec, genetique = max_vals_gen)

df <- df |> pivot_longer(1:3, names_to = "algorithme", values_to = "valeur")

df |> ggplot(aes(valeur, col = algorithme)) + geom_boxplot(aes(fill = algorithme), alpha = 0.5) +
  labs(title = "Boxplot des valeurs maximums", subtitle = "sur 100 trajectoires de taille 1000")
```



On remarque que presque toutes les trajectoires de l'algorithme génétique tombent sur la valeur maximum, déja dans 1000 itérations! On a dû simuler 10000 itérations pour que la méthode de métropolis avec scalaire de division $k = 10$ pourrait avoir une majorité de trajectoire qui finissent à 1532.

On peut conclure donc que l'algorithme génétique converge vers la vrai maximum beaucoup plus vite au nivea de nombre d'itérations. On s'explique cela en considérant qu'il y a une famille de $m = 100$ configurations qui

se mutent ensemble, choissisant de garder environ 50 configurations qui est la meilleure parmi toute la famille, et laisser que les autres se modifient aléatoirement. Choisir parfois la meilleure et parfois une modification nous aide à éviter de s'immobiliser sur une certaine valeur, comme on explore l'espace des états par plusiers trajectoires en même temps.

Il serait intéressant de voir la distribution des moyennes en fonction non de la taille d'une trajectoire, sinon le temps de calcul d'en réaliser. Cela est une considération parce que pour chaque itération dans une trajectoire de l'algorithme génétique, on effectue m propositions de σ'. Par conséquent, la compléxité de cet algorithme est plus important.