TP3: Randonnée et optimisation

Evan Voyles

May 25

```r
library(tidyverse)
library(purrr)
library(tibble)
library(ggplot2)
library(pracma)
```

Un randonneur se trouve face à un dilemme classique : il aimerait emmener avec lui un maximum d'affaires qui pourraient lui servir mais ne souhaite pas alourdir inutiliment son sac, qui ne doit pas dépasser un certain poids P. Il cherche donc à déterminer la combinaison d'affaire qui vérifierait le meilleur rapport utilité/légèreté. Pour ce faire, il associe à chacune de ses M affaires, son poids p_i et sa valeur v_i pour $i \in \{1, \dots, M\}$. L'objectif est de trouver la configuration qui maximise la valeur du sac à dos sans dépasser P.

1. On condidère 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

```r
# 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
}

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

total_valeur <- function(sig, valeur) {
    sum(sig * valeur)
}

# We'll use the transition matrix Q(. | V) = N(V, 1)
# En sachant que \pi(\sigma') / \pi(\sigma) = exp(V_{\sigma'} - V_\sigma),
# On implemente l'algorithme de metropolis hastings comme suivant:
taux_accept <- function(sig, sig_prime, valeur) {
    min(exp(total_valeur(sig, valeur) - total_valeur(sig_prime, valeur)), 1)
}
```

```r
item <- c("map", "water", "sandwich", "glucose", "tin", "banana", "apple", "cheese",
          "beer", "suntan_cream", "camera", "T-shirts", "trousers", "umbrella",
```

```r
          "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)


next_sigma <- function(sig, poids, valeur, P) {

    sig_prime <- propose_sigma_prime(sig)

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

    r <- taux_accept(sig, sig_prime, valeur)

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

}

sim_traj <- function(n) {
    sigma0 <- rep(0, n_items)
    traj <- list(sigma0)

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

    traj
}

sim_traj_valeur <- function(n) {

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

    list(t = traj, v = val)
}

val_traj <- sim_traj_valeur(1000)

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

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

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

# Now re run this algorithm 100 times with n = 1e5.
# Figure out the max of all 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 } )
max(max_vals)
```

```
## [1] 1532
```

```r
# Consistently, the MAX value that I can get is 1532
# Let's find the index of 1532

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

i_1532 <- detect_index(max_vals, equals(1532))

winning_configuration <- max_sigmas[[i_1532]]

total_valeur(winning_configuration, v)
```

```
## [1] 1532
```

```r
total_weight(winning_configuration, p)
```

```
## [1] 6.75
```

```r
item[as.logical(winning_configuration)]
```

```
##  [1] "map"                     "water"                   "sandwich"
##  [4] "glucose"                 "tin"                     "banana"
##  [7] "apple"                   "cheese"                  "beer"
## [10] "suntan_cream"            "T-shirts"                "waterproof_trousers"
## [13] "waterproof_overclothes" "note-case"                "sunglasses"
## [16] "towel"                   "socks"
```

We are going to go ahead and implement the simulated annealing algorithms

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

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

# We use the same method to sample the next value
# We reject if the total weight is larger than P
# The probability of acceptance will now be
# P(\sigma, \sigma', T) instead of just P(\sigma, \sigma')
#' @param
#' n index used to compute the temperature function
next_sigma_recuit <- function(sig, poids, valeur, n, P) {

    sig_prime <- propose_sigma_prime(sig)

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

    r <- taux_accept_recuit(sig, sig_prime, valeur, temp_fn(n))

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

}

sim_traj_recuit <- function(n) {
    sigma0 <- rep(0, n_items)
    traj <- list(sigma0)

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

    traj
}

sim_traj_valeur_recuit <- function(n) {

    traj <- sim_traj_recuit(n)
    val <- map_dbl(traj, total_valeur, v)

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

```r
val_traj <- sim_traj_valeur_recuit(1000)

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

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

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

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

```
## [1] 1532
```

```r
# Consistently, the MAX value that I can get is 1532
# Let's find the index of 1532

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

i_1532 <- detect_index(max_vals, equals(1532))

winning_configuration <- max_sigmas[[i_1532]]

total_valeur(winning_configuration, v)
```

```
## [1] 1532
```

```r
total_weight(winning_configuration, p)
```

```
## [1] 6.75
```

```r
item[as.logical(winning_configuration)]
```

```
##  [1] "map"                 "water"               "sandwich"
##  [4] "glucose"             "tin"                 "banana"
##  [7] "apple"               "cheese"              "beer"
## [10] "suntan_cream"        "T-shirts"            "waterproof_trousers"
```

```
## [13] "waterproof_overclothes" "note-case"              "sunglasses"
## [16] "towel"                  "socks"
```

Implementation of the genetic algorithm

```
# First thing I want to do is simulate m initial configurations.
# Let's simulate m initial configurations by using the metropolis
# algorithm n_met times

m <- 20
n_met <- 10
n_gen <- 100

get_init_config <- function(n_met) {
    traj <- sim_traj(n_met)
    traj[[n_met]]
}

get_init_configs <- function(m, n_met) {
    init_configs <- list()
    for (i in 1:m) {
        init_configs[[i]] <- get_init_config(n_met)
    }
    init_configs
}

# 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]]
}

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)
        }
    }
}
```

```r
n_gen <- 100

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

# Now for each step, get the best configuration from the m current configurations
for (i in 2:n_gen) {

    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
}

# Now for each list in M, compute the value of the best_config
map_dbl(M, function(traj) {
    best <- get_best_config(traj)
    total_valeur(best, v)
})
```

```
##   [1]  972  972 1047 1122 1122 1167 1222 1272 1322 1332 1362 1372 1372 1422 1447 1497 1497 1497
##  [19] 1497 1497 1497 1497 1497 1497 1497 1497 1497 1497 1497 1497 1497 1497 1497 1497 1497 1497
##  [37] 1497 1497 1497 1497 1497 1497 1497 1497 1497 1497 1517 1517 1517 1517 1517 1517 1517 1517
##  [55] 1517 1517 1517 1517 1517 1517 1517 1517 1517 1517 1517 1517 1517 1517 1517 1517 1517 1517
##  [73] 1517 1517 1517 1517 1517 1517 1517 1517 1517 1517 1517 1517 1517 1517 1517 1517 1517 1517
##  [91] 1517 1517 1517 1517 1517 1517 1517 1517 1517 1517
```