Lab Report Knapsack

Simon Jönsson, Fanny Karelius

2017-10-11

Introduction

In this lab we implement different solutions for the **0/1-knapsack**- and **unbounded knapsack** problem. We have implemented both a parallel and non-parallel brute force solution, a dynamic programming solution and a solution using the greedy heuristic. We have documented the runtimes and the profiling of each solution.

Runtime of codes

Bruteforce knapsack solution

Parallel

```
system.time(brute_force_knapsack(x = knapsack_objects[1:20,], W = 3500, parallel=TRUE))
```

```
##    user  system elapsed
##  44.335  20.586  48.147
```

Non-parallel

```
system.time(brute_force_knapsack(x = knapsack_objects[1:20,], W = 3500, parallel=FALSE))
```

```
##    user  system elapsed
##  18.982   1.082  20.088
```

Dynamic knapsack solution

```
system.time(knapsack_dynamic(x = knapsack_objects[1:500,], W = 3500))
```

```
##    user  system elapsed
##   5.601   0.215   5.824
```

Greedy knapsack solution

```
system.time(greedy_knapsack(x = knapsack_objects[1:1000000,], W = 3500))
```

```
##    user  system elapsed
##   1.253   0.262   1.516
```

Profiling

Bruteforce knapsack solution

Parallel

```
lineprof(brute_force_knapsack(x = knapsack_objects[1:12,], W = 3500, parallel=TRUE))
```

```
## Reducing depth to 2 (from 7)
```

```
##    time alloc release dups                                      ref
## 1 0.003 2.647       0  161                    c("matrix", "unlist")
## 2 0.001 0.161       0    0                                  "matrix"
## 3 0.005 0.059       0   82         c("parallel::mclapply", "lapply")
## 4 0.001 0.017       0    5 c("parallel::mclapply", "selectChildren")
## 5 0.001 0.052       0    6      c("parallel::mclapply", "readChild")
## 6 0.002 0.017       0   20        c("parallel::mclapply", "cleanup")
## 7 0.004 2.435       0    4                     c("lapply", "Filter")
## 8 0.001 0.000       0   19                        c("lapply", "FUN")
##                                src
## 1 matrix/unlist
## 2 matrix
## 3 parallel::mclapply/lapply
## 4 parallel::mclapply/selectChildren
## 5 parallel::mclapply/readChild
## 6 parallel::mclapply/cleanup
## 7 lapply/Filter
## 8 lapply/FUN
```

Here we see that apart from allocation, all segments of the code are in similar timesteps - quite tricky to identify bottlenecks. However one could look over using some primitive functions instead of using lapply to find the row with near-optimal value. A suggestions might be using **max()**.

Non-parallel

```
lineprof(brute_force_knapsack(x = knapsack_objects[1:12,], W = 3500, parallel=FALSE))
```

```
##    time alloc release  dups                   ref           src
## 1 0.005 2.671       0   161 c("matrix", "unlist") matrix/unlist
## 2 0.001 0.789       0     0              "matrix" matrix
## 3 0.004 9.624       0 11281     c("apply", "FUN") apply/FUN
```

Here we see that the apply function and the unlist function creates alot of allocation. Creating a solution which does not use unlist but instead without pre-allocation might help. Also the apply function might be exchanged with using a **max()** primitive. Generating the matrix with given parameters: weight and val one could use **max** to find the maximum val given that the weight $\leq$ W.

Dynamic knapsack solution

```
lineprof(knapsack_dynamic(x = knapsack_objects[1:100,], W = 3500))
```

```
## Reducing depth to 2 (from 66)
```

```
##     time   alloc release  dups                                    ref
## 1  0.006   9.748       0  3607 c("compiler:::tryCmpfun", "tryCatch")
## 2  0.064 170.571       0  1517              c("matrix", "replicate")
## 3  0.002   0.316       0     0                                "matrix"
## 4  0.002   0.827       0  1734                           character(0)
## 5  0.002   0.538       0  1217                                  "max"
## 6  0.006   5.161       0  8471                           character(0)
## 7  0.001   3.943       0  2708                                  "max"
## 8  0.001   1.096       0  8147                           character(0)
```

```
## 9  0.002   4.891       0  8387                                "max"
## 10 0.001   4.519       0  3981                          character(0)
## 11 0.001   1.437       0  9340                                "max"
## 12 0.002   6.096       0  3552                          character(0)
## 13 0.001   2.021       0 12014                                "max"
## 14 0.001   4.333       0  4176                          character(0)
## 15 0.001   3.669       0  8954                                "max"
## 16 0.002   8.004       0 11535                          character(0)
## 17 0.001   0.537       0 12586                                "max"
## 18 0.001   2.915       0  1108                          character(0)
## 19 0.001   4.475       0  6024                                "max"
## 20 0.003   8.081       0 24293                          character(0)
## 21 0.001   0.227       0  1655                                "max"
## 22 0.002   4.219       0  1803                          character(0)
## 23 0.001   1.282       0  7385                                "max"
## 24 0.002   1.276       0  5283                          character(0)
##                               src
## 1  compiler:::tryCmpfun/tryCatch
## 2  matrix/replicate
## 3  matrix
## 4
## 5  max
## 6
## 7  max
## 8
## 9  max
## 10
## 11 max
## 12
## 13 max
## 14
## 15 max
## 16
## 17 max
## 18
## 19 max
## 20
## 21 max
## 22
## 23 max
## 24
```

Here we identify that the segment in the code that takes most time to run is the replicate function. This could be handled by some other primitive, or maybe the pre-allocation can be circumvented by having dynamic size of the vector.

Greedy knapsack solution

```
lineprof(greedy_knapsack(x = knapsack_objects[1:20000,], W = 3500))
```

```
## Reducing depth to 2 (from 34)
```

```
##    time alloc release dups                                    ref
## 1 0.002 3.147       0 3154 c("compiler:::tryCmpfun", "tryCatch")
```

```
## 2 0.006 4.414       0  102       c("stopifnot", "is.data.frame")
## 3 0.002 6.562       0   44              c("replicate", "sapply")
## 4 0.001 0.721       0    5                                 "order"
## 5 0.001 0.000       0   36                           character(0)
##                             src
## 1 compiler:::tryCmpfun/tryCatch
## 2 stopifnot/is.data.frame
## 3 replicate/sapply
## 4 order
## 5
```

Not alot to improve on here.

Parallelizing brute force knapsack

The performance that could be gained is non-existent since the lapply used doesn't contain any calculations. So there is little to no sequential computations that are done. If we had a computationally heavy lapply segment then we could gain an decrease in computation time.