Lab Report Knapsack

Simon Jönsson, Fanny Karelius

2017-10-09

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[1:16,], W = 3500, parallel=FALSE))
```

```
##    user  system elapsed
##   1.193   0.053   1.249
```

Non-parallel

```
system.time(brute_force_knapsack(x = knapsack_objects[1:16,], W = 3500, parallel=FALSE))
```

```
##    user  system elapsed
##   0.908   0.040   0.952
```

Dynamic knapsack solution

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

```
##    user  system elapsed
##   5.482   0.098   5.609
```

Greedy knapsack solution

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

```
##    user  system elapsed
##   2.156   0.175   2.369
```

Profiling

Bruteforce knapsack solution

Parallel

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

```
## Reducing depth to 2 (from 9)
```

```
##     time alloc release dups                                ref
## 1  0.008 2.671       0  167            c("matrix", "unlist")
## 2  0.001 0.129       0    0                          "matrix"
## 3  0.001 0.138       0   17     c("mclapply", "detectCores")
## 4  0.004 0.062       0   52          c("mclapply", "lapply")
## 5  0.001 0.027       0   11 c("mclapply", "lazyLoadDBfetch")
## 6  0.001 0.003       0    2     c("mclapply", "unserialize")
## 7  0.001 0.038       0    1                       "mclapply"
## 8  0.010 0.829       0   13         c("mclapply", "cleanup")
## 9  0.003 1.732       0   13            c("lapply", "Filter")
## 10 0.001 0.000       0   44                     character(0)
##                          src
## 1  matrix/unlist
## 2  matrix
## 3  mclapply/detectCores
## 4  mclapply/lapply
## 5  mclapply/lazyLoadDBfetch
## 6  mclapply/unserialize
## 7  mclapply
## 8  mclapply/cleanup
## 9  lapply/Filter
## 10
```

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))
```

```
## Reducing depth to 2 (from 3)
```

```
##    time alloc release dups                   ref           src
## 1 0.008 2.828       0  167 c("matrix", "unlist") matrix/unlist
## 2 0.001 1.678       0   16   c("apply", "aperm") apply/aperm
## 3 0.002 1.400       0 2575               "apply" apply
## 4 0.011 7.292       0 7967     c("apply", "FUN") apply/FUN
## 5 0.001 0.623       0  712               "apply" apply
## 6 0.001 0.000       0  676     c("apply", "FUN") apply/FUN
```

Similar to the parallel solution this code has very little overhead that could be optimized.

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  0.009   9.747   0.000  4489 c("compiler:::tryCmpfun", "tryCatch")
## 2  0.130 168.612 152.813   635             c("matrix", "replicate")
```

```
## 3  0.002   0.001   0.000     0                        "matrix"
## 4  0.001   0.004   0.000     0                      character(0)
## 5  0.001   0.605   0.000     7                             "max"
## 6  0.009   5.848   0.000 11189                      character(0)
## 7  0.003   1.474   0.000  3912                             "max"
## 8  0.003   2.055   0.000  4309                      character(0)
## 9  0.001   0.796   0.000  1216                             "max"
## 10 0.005   2.534   0.000  5231                      character(0)
## 11 0.003   2.002   0.000  4920                             "max"
## 12 0.007   2.664   0.000  5617                      character(0)
## 13 0.001   0.599   0.000   751                             "max"
## 14 0.003   1.933   0.000  3515                      character(0)
## 15 0.003   1.192   0.000  3358                             "max"
## 16 0.001   0.574   0.000   824                      character(0)
## 17 0.001   0.457   0.000  1187                             "max"
## 18 0.005   2.778   0.000  5365                      character(0)
## 19 0.001   0.709   0.000  1318                             "max"
## 20 0.010   3.324   0.000  7573                      character(0)
## 21 0.003   1.923   0.000  3249                             "max"
## 22 0.001   0.319   0.000  1485                      character(0)
## 23 0.002   1.117   0.000  1857                             "max"
## 24 0.003   1.067   0.000  2355                      character(0)
## 25 0.001   0.670   0.000   958                             "max"
## 26 0.005   2.500   0.000  5528                      character(0)
## 27 0.001   0.386   0.000  1025                             "max"
## 28 0.003   1.688   0.000  2656                      character(0)
## 29 0.002   1.218   0.000  2958                             "max"
## 30 0.003   1.572   0.000  3059                      character(0)
## 31 0.001   0.676   0.000  1374                             "max"
## 32 0.001   0.761   0.000  1397                      character(0)
## 33 0.001   0.722   0.000  1574                             "max"
## 34 0.001   0.483   0.000  1493                      character(0)
## 35 0.002   1.366   0.000  2518                             "max"
## 36 0.010   6.026   0.000 12892                      character(0)
## 37 0.001   0.554   0.000   865                             "max"
## 38 0.002   0.768   0.000  2329                      character(0)
## 39 0.002   0.000  75.874   404                             "max"
## 40 0.001   1.043   0.000  5231                      character(0)
## 41 0.001   0.305   0.000  2155                             "max"
## 42 0.002   0.600   0.000  1417                      character(0)
## 43 0.001   1.174   0.000   451                             "max"
## 44 0.015  10.818   0.000 24781                      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
## 25 max
## 26
## 27 max
## 28
## 29 max
## 30
## 31 max
## 32
## 33 max
## 34
## 35 max
## 36
## 37 max
## 38
## 39 max
## 40
## 41 max
## 42
## 43 max
## 44
```

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 47)
```

```
##    time alloc release dups                                  ref
## 1 0.006 6.079       0 2256 c("compiler:::tryCmpfun", "tryCatch")
## 2 0.006 6.256       0 1000       c("stopifnot", "is.data.frame")
## 3 0.003 4.898       0   44             c("replicate", "sapply")
## 4 0.001 0.266       0    5                          "order"
## 5 0.004 0.820       0   55                       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.