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=FALSE))
```

```
##    user  system elapsed
##  30.299   1.315  31.823
```

Non-parallel

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

```
##    user  system elapsed
##  19.109   1.474  20.663
```

Dynamic knapsack solution

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

```
##    user  system elapsed
##   5.584   0.234   5.843
```

Greedy knapsack solution

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

```
##    user  system elapsed
##   1.371   0.196   1.573
```

Profiling

Bruteforce knapsack solution

Parallel

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

```
## Reducing depth to 2 (from 16)

##     time alloc release dups                                         ref
## 1  0.005 2.647       0  161                        c("matrix", "unlist")
## 2  0.001 0.191       0    0                                     "matrix"
## 3  0.004 1.070       0  467                  c("::", "getExportedValue")
## 4  0.001 0.133       0  191                c("parallel::mclapply", "::")
## 5  0.001 0.087       0    5                         "parallel::mclapply"
## 6  0.003 0.051       0   49            c("parallel::mclapply", "lapply")
## 7  0.001 0.019       0   11                         "parallel::mclapply"
## 8  0.001 0.035       0    1 c("parallel::mclapply", "lazyLoadDBfetch")
## 9  0.001 0.017       0    8  c("parallel::mclapply", "selectChildren")
## 10 0.008 0.485       0   11           c("parallel::mclapply", "cleanup")
## 11 0.004 2.058       0   13                        c("lapply", "Filter")
## 12 0.001 0.000       0    9                           c("lapply", "FUN")
##                                    src
## 1  matrix/unlist
## 2  matrix
## 3  ::/getExportedValue
## 4  parallel::mclapply/::
## 5  parallel::mclapply
## 6  parallel::mclapply/lapply
## 7  parallel::mclapply
## 8  parallel::mclapply/lazyLoadDBfetch
## 9  parallel::mclapply/selectChildren
## 10 parallel::mclapply/cleanup
## 11 lapply/Filter
## 12 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

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

```
##    time alloc release dups                   ref           src
## 1 0.004 2.694       0  161 c("matrix", "unlist") matrix/unlist
## 2 0.001 1.709       0    0              "matrix" matrix
## 3 0.003 2.292       0 3458     c("apply", "FUN") apply/FUN
## 4 0.001 0.772       0  849               "apply" apply
## 5 0.009 6.435       0 7837     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

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

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

##      time   alloc release dups                                  ref
## 1  0.011   9.748       0 4879 c("compiler:::tryCmpfun", "tryCatch")
## 2  0.206 170.571       0  245              c("matrix", "replicate")
## 3  0.002   0.002       0    0                            "matrix"
## 4  0.007   3.931       0 6629                        character(0)
## 5  0.002   0.715       0 2395                               "max"
## 6  0.002   0.701       0 1025                        character(0)
## 7  0.001   0.337       0  997                               "max"
## 8  0.001   0.251       0  694                        character(0)
## 9  0.001   0.618       0  517                               "max"
## 10 0.002   1.035       0 2478                        character(0)
## 11 0.001   0.345       0  939                               "max"
## 12 0.002   0.869       0 1029                        character(0)
## 13 0.001   0.598       0 1478                               "max"
## 14 0.002   0.979       0 2367                        character(0)
## 15 0.001   0.408       0  891                               "max"
## 16 0.001   0.686       0  843                        character(0)
## 17 0.002   0.737       0 2814                               "max"
## 18 0.002   1.412       0 1608                        character(0)
## 19 0.002   1.030       0 2554                               "max"
## 20 0.001   0.690       0 1014                        character(0)
## 21 0.002   0.869       0 1790                               "max"
## 22 0.003   1.410       0 3256                        character(0)
## 23 0.005   3.017       0 6000                               "max"
## 24 0.001   0.601       0 1322                        character(0)
## 25 0.001   0.558       0 1243                               "max"
## 26 0.001   0.503       0 1152                        character(0)
## 27 0.002   0.432       0 1705                               "max"
## 28 0.003   1.173       0 2276                        character(0)
## 29 0.001   0.380       0  374                               "max"
## 30 0.002   1.083       0 1864                        character(0)
## 31 0.001   0.589       0 1158                               "max"
## 32 0.001   0.581       0 1218                        character(0)
## 33 0.002   0.776       0 1510                               "max"
## 34 0.002   0.528       0 1732                        character(0)
## 35 0.001   0.167       0  652                               "max"
## 36 0.001   0.497       0  343                        character(0)
## 37 0.001   0.638       0 1026                               "max"
## 38 0.003   1.411       0 2956                        character(0)
## 39 0.002   1.072       0 2677                               "max"
## 40 0.001   0.714       0  815                        character(0)
## 41 0.003   1.559       0 3754                               "max"
## 42 0.001   0.687       0  940                        character(0)
## 43 0.001   0.628       0 1421                               "max"
## 44 0.001   0.554       0 1299                        character(0)
## 45 0.002   1.193       0 2545                               "max"
## 46 0.001   0.477       0 1064                        character(0)
## 47 0.002   0.593       0 1212                               "max"
## 48 0.001   0.364       0  996                        character(0)
## 49 0.002   1.038       0 2040                               "max"
## 50 0.002   1.224       0 1910                        character(0)
## 51 0.001   0.336       0 1478                               "max"
```

```
## 52 0.004   2.463       0 4428                              character(0)
## 53 0.002   1.292       0 2783                                     "max"
## 54 0.001   0.667       0 1241                              character(0)
## 55 0.004   2.284       0 4834                                     "max"
## 56 0.001   0.702       0 1264                              character(0)
## 57 0.002   1.208       0 2605                                     "max"
## 58 0.001   0.671       0 1342                              character(0)
## 59 0.002   1.134       0 2649                                     "max"
## 60 0.001   0.497       0 1083                              character(0)
## 61 0.005   2.700       0 5633                                     "max"
## 62 0.001   0.592       0  974                              character(0)
## 63 0.002   1.200       0 2559                                     "max"
## 64 0.002   1.150       0 2530                              character(0)
## 65 0.003   1.479       0 2951                                     "max"
## 66 0.002   0.807       0 2386                              character(0)
## 67 0.005   2.989       0 5599                                     "max"
## 68 0.002   1.138       0 2381                              character(0)
## 69 0.001   0.731       0  925                                     "max"
## 70 0.001   0.626       0 1509                              character(0)
## 71 0.003   1.605       0 3295                                     "max"
## 72 0.001   0.341       0 1317                              character(0)
## 73 0.004   2.171       0 4245                                     "max"
## 74 0.001   0.443       0  943                              character(0)
## 75 0.001   0.000       0  915                                     "max"
##                                  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
## 45 max
## 46
## 47 max
## 48
## 49 max
## 50
## 51 max
## 52
## 53 max
## 54
## 55 max
## 56
## 57 max
## 58
## 59 max
## 60
## 61 max
## 62
## 63 max
## 64
## 65 max
## 66
## 67 max
## 68
## 69 max
## 70
## 71 max
## 72
## 73 max
## 74
## 75 max
```

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

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

```
## Reducing depth to 2 (from 37)
```

```
##    time alloc release dups                                  ref
## 1 0.002 2.902       0 3038 c("compiler:::tryCmpfun", "tryCatch")
## 2 0.008 1.406       0  218       c("stopifnot", "is.data.frame")
## 3 0.004 9.748       0   44              c("replicate", "sapply")
## 4 0.001 0.336       0    5                               "order"
## 5 0.003 0.622       0   48                          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.