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
##   6.727   0.429   7.182
```

Non-parallel

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

```
##    user  system elapsed
##   5.540   0.381   5.938
```

Dynamic knapsack solution

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

```
##    user  system elapsed
##   5.342   0.051   5.406
```

Greedy knapsack solution

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

```
##    user  system elapsed
##   2.383   0.158   2.550
```

Profiling

Bruteforce knapsack solution

Parallel

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

```
## Reducing depth to 2 (from 12)
```

```
##    time alloc release dups                                        ref
## 1 0.003 0.936       0  521                c("::", "getExportedValue")
## 2 0.001 0.227       0  191              c("parallel::mclapply", "::")
## 3 0.004 0.053       0   55          c("parallel::mclapply", "lapply")
## 4 0.001 0.027       0   11 c("parallel::mclapply", "lazyLoadDBfetch")
## 5 0.001 0.005       0    0                          "parallel::mclapply"
## 6 0.002 0.046       0    9  c("parallel::mclapply", "selectChildren")
## 7 0.005 0.253       0    7         c("parallel::mclapply", "cleanup")
## 8 0.004 2.273       0   17                       c("lapply", "Filter")
##                                  src
## 1 ::/getExportedValue
## 2 parallel::mclapply/::
## 3 parallel::mclapply/lapply
## 4 parallel::mclapply/lazyLoadDBfetch
## 5 parallel::mclapply
## 6 parallel::mclapply/selectChildren
## 7 parallel::mclapply/cleanup
## 8 lapply/Filter
```

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 0.019 11.59       0 11691 c("lapply", "FUN") lapply/FUN
```

The lapply 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 70)
```

```
##     time   alloc release  dups                                      ref
## 1  0.007   9.748   0.000  3626 c("compiler:::tryCmpfun", "tryCatch")
## 2  0.100 153.462 143.548  1498              c("matrix", "replicate")
## 3  0.001   2.399   0.000     0                                 "matrix"
## 4  0.003  11.952   0.000 16386                           character(0)
## 5  0.001   1.859   0.000 13264                                  "max"
## 6  0.001   4.092   0.000  3841                           character(0)
## 7  0.002   1.360   0.000  9985                                  "max"
## 8  0.004   2.475   0.000  4765                           character(0)
## 9  0.001   0.631   0.000  1627                                  "max"
## 10 0.003   1.901   0.000  4448                           character(0)
## 11 0.001   0.748   0.000   784                                  "max"
## 12 0.005   2.660   0.000  5683                           character(0)
```

```
## 13 0.001   0.618   0.000  1358                          "max"
## 14 0.002   0.786   0.000  2744                    character(0)
## 15 0.001   0.399   0.000   158                          "max"
## 16 0.004   1.882   0.000  3808                    character(0)
## 17 0.001   0.618   0.000   905                          "max"
## 18 0.004   2.282   0.000  5035                    character(0)
## 19 0.002   0.876   0.000  1969                          "max"
## 20 0.002   0.876   0.000  1807                    character(0)
## 21 0.001   0.761   0.000   796                          "max"
## 22 0.002   1.315   0.000  2797                    character(0)
## 23 0.001   0.438   0.000  1494                          "max"
## 24 0.002   1.224   0.000  1779                    character(0)
## 25 0.002   1.521   0.000  3225                          "max"
## 26 0.001   0.657   0.000  1573                    character(0)
## 27 0.002   1.405   0.000  2878                          "max"
## 28 0.001   0.748   0.000  1384                    character(0)
## 29 0.002   1.141   0.000  2678                          "max"
## 30 0.001   0.516   0.000  1226                    character(0)
## 31 0.001   0.561   0.000  1065                          "max"
## 32 0.002   0.593   0.000  1687                    character(0)
## 33 0.001   0.894   0.000   696                          "max"
## 34 0.006   3.320   0.000  7218                    character(0)
## 35 0.001   0.683   0.000  1494                          "max"
## 36 0.002   1.166   0.000  2504                    character(0)
## 37 0.001   0.709   0.000  1318                          "max"
## 38 0.003   1.630   0.000  3915                    character(0)
## 39 0.002   0.786   0.000  1410                          "max"
## 40 0.001   0.619   0.000  1132                    character(0)
## 41 0.001   0.709   0.000  1278                          "max"
## 42 0.011   8.508  80.594 20678                    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
```

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 18)
```

```
##    time alloc release dups                                  ref
## 1 0.002 5.828       0 3215 c("compiler:::tryCmpfun", "tryCatch")
## 2 0.006 1.219       0   41       c("stopifnot", "is.data.frame")
## 3 0.003 9.768       0   44              c("replicate", "sapply")
## 4 0.002 0.079       0    5                                "order"
## 5 0.005 0.977       0   53                           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.