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
##   5.742   0.536   2.938
```

Non-parallel

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

```
##    user  system elapsed
##   5.570   0.598   6.180
```

Dynamic knapsack solution

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

```
##    user  system elapsed
##    5.37    0.05    5.43
```

Greedy knapsack solution

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

```
##    user  system elapsed
##   2.457   0.158   2.620
```

Profiling

Bruteforce knapsack solution

Parallel

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

```
## Reducing depth to 2 (from 6)
```

```
##    time alloc release dups                                      ref
## 1 0.005 0.052       0  127         c("parallel::mclapply", "lapply")
## 2 0.001 0.002       0   13 c("parallel::mclapply", "selectChildren")
## 3 0.001 0.026       0    1                      "parallel::mclapply"
## 4 0.001 0.036       0    5      c("parallel::mclapply", "readChild")
## 5 0.001 0.028       0   14        c("parallel::mclapply", "cleanup")
## 6 0.005 2.420       0    9                     c("lapply", "Filter")
## 7 0.001 0.000       0    1                        c("lapply", "FUN")
##                                 src
## 1 parallel::mclapply/lapply
## 2 parallel::mclapply/selectChildren
## 3 parallel::mclapply
## 4 parallel::mclapply/readChild
## 5 parallel::mclapply/cleanup
## 6 lapply/Filter
## 7 lapply/FUN
```

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 0.008 5.417       0 5192 c("lapply", "FUN") lapply/FUN
## 2 0.001 0.536       0  612           "lapply" lapply
## 3 0.011 6.031       0 6352 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 66)
```

```
##     time   alloc release  dups                                      ref
## 1  0.015   9.915   0.000  5057 c("compiler:::tryCmpfun", "tryCatch")
## 2  0.064 163.076 153.329    67              c("matrix", "replicate")
## 3  0.001   4.494   0.000     0                                  "matrix"
## 4  0.007  18.232   0.000 45646                           character(0)
## 5  0.001   0.012   0.000  1307                                  "max"
## 6  0.006   7.899   0.000 15840                           character(0)
## 7  0.001   0.632   0.000   505                                  "max"
## 8  0.001   0.386   0.000  1306                           character(0)
## 9  0.002   1.496   0.000  2316                                  "max"
## 10 0.001   0.373   0.000  1573                           character(0)
## 11 0.001   0.618   0.000   769                                  "max"
## 12 0.001   0.722   0.000  1277                           character(0)
```

```
## 13 0.001   0.631   0.000  1494                                "max"
## 14 0.001   0.787   0.000  1304                          character(0)
## 15 0.003   1.249   0.000  3809                                "max"
## 16 0.002   0.992   0.000  1594                          character(0)
## 17 0.001   0.799   0.000   849                                "max"
## 18 0.002   4.853   0.000  3064                          character(0)
## 19 0.001   3.196   0.000  8618                                "max"
## 20 0.007   7.160   0.000 20285                          character(0)
## 21 0.002   1.508   0.000  2609                                "max"
## 22 0.005   2.938   0.000  6337                          character(0)
## 23 0.001   0.748   0.000  1358                                "max"
## 24 0.002   2.337  79.393  6722                          character(0)
## 25 0.001   0.000   0.000  3765                                "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
```

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 46)
```

```
##    time alloc release dups                                   ref
## 1 0.002 4.149       0 3013 c("compiler:::tryCmpfun", "tryCatch")
## 2 0.008 1.349       0  243       c("stopifnot", "is.data.frame")
## 3 0.003 9.746       0   43             c("replicate", "sapply")
```

```
## 4 0.002 0.455       0    6                                "order"
## 5 0.003 0.454       0   45                              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.