Lab Report Knapsack

Simon Jönsson, Fanny Karelius

2017-10-08

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
##   1.163   0.046   1.210
```

Non-parallel

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

```
##    user  system elapsed
##   0.878   0.028   0.909
```

Dynamic knapsack solution

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

```
##    user  system elapsed
##   5.370   0.078   5.455
```

Greedy knapsack solution

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

```
##    user  system elapsed
##   2.009   0.118   2.131
```

Profiling

Bruteforce knapsack solution

Parallel

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

```
## Reducing depth to 2 (from 8)
```

```
##    time alloc release dups                              ref
## 1 0.004 2.647       0  167           c("matrix", "unlist")
## 2 0.001 0.267       0    0                         "matrix"
## 3 0.005 0.077       0   78         c("mclapply", "lapply")
## 4 0.001 0.035       0    2 c("mclapply", "lazyLoadDBfetch")
## 5 0.001 0.017       0    8  c("mclapply", "selectChildren")
## 6 0.005 0.423       0    6        c("mclapply", "cleanup")
## 7 0.003 1.866       0   17           c("lapply", "Filter")
##                        src
## 1 matrix/unlist
## 2 matrix
## 3 mclapply/lapply
## 4 mclapply/lazyLoadDBfetch
## 5 mclapply/selectChildren
## 6 mclapply/cleanup
## 7 lapply/Filter
```

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 0.006 2.684       0  167 c("matrix", "unlist") matrix/unlist
## 2 0.001 1.559       0    0              "matrix" matrix
## 3 0.004 3.208       0 4254     c("apply", "FUN") apply/FUN
## 4 0.001 0.784       0  860               "apply" apply
## 5 0.003 2.503       0 2704     c("apply", "FUN") apply/FUN
## 6 0.001 0.798       0  872               "apply" apply
## 7 0.002 1.564       0 1696     c("apply", "FUN") apply/FUN
## 8 0.001 0.786       0  872               "apply" apply
## 9 0.001 0.000       0  854     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  0.011   9.914   0.000  4997 c("compiler:::tryCmpfun", "tryCatch")
## 2  0.088 162.834 147.201   127             c("matrix", "replicate")
## 3  0.001   0.676   0.000     0                           "matrix"
## 4  0.001   6.027   0.000  1396                              "max"
## 5  0.002   6.064   0.000 23529                       character(0)
```

```
## 6  0.001   2.700   0.000  1451                          "max"
## 7  0.005  10.313   0.000 23615                    character(0)
## 8  0.001   0.786   0.000  3274                          "max"
## 9  0.001   0.283   0.000  1626                    character(0)
## 10 0.003   1.790   0.000  2923                          "max"
## 11 0.006   4.878   0.000 10681                    character(0)
## 12 0.001   0.760   0.000   756                          "max"
## 13 0.002   1.535   0.000  3277                    character(0)
## 14 0.002   1.340   0.000  2796                          "max"
## 15 0.005   2.938   0.000  6040                    character(0)
## 16 0.001   0.780   0.000  1465                          "max"
## 17 0.001   0.180   0.000  1613                    character(0)
## 18 0.001   0.747   0.000   369                          "max"
## 19 0.001   0.812   0.000  1544                    character(0)
## 20 0.002   1.579   0.000  3398                          "max"
## 21 0.001   0.806   0.000  1544                    character(0)
## 22 0.002   1.547   0.000  3158                          "max"
## 23 0.001   0.632   0.000  1706                    character(0)
## 24 0.001   0.612   0.000  1305                          "max"
## 25 0.001   0.825   0.000  1264                    character(0)
## 26 0.002   1.334   0.000  2758                          "max"
## 27 0.004   2.913   0.000  6046                    character(0)
## 28 0.004   5.338  72.305 15335                          "max"
## 29 0.002   2.969   0.000 16118                    character(0)
##                                src
## 1  compiler:::tryCmpfun/tryCatch
## 2  matrix/replicate
## 3  matrix
## 4  max
## 5
## 6  max
## 7
## 8  max
## 9
## 10 max
## 11
## 12 max
## 13
## 14 max
## 15
## 16 max
## 17
## 18 max
## 19
## 20 max
## 21
## 22 max
## 23
## 24 max
## 25
## 26 max
## 27
## 28 max
## 29
```

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 0.003 4.842       0 2736 c("compiler:::tryCmpfun", "tryCatch")
## 2 0.006 1.339       0  520       c("stopifnot", "is.data.frame")
## 3 0.003 9.777       0   43              c("replicate", "sapply")
## 4 0.001 0.030       0    6                                "order"
## 5 0.004 0.879       0   46                           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.