Lab Report Knapsack

Simon Jönsson, Fanny Karelius

2017-10-07

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=TRUE))
```

```
##    user  system elapsed
##   1.042   0.068   1.647
```

Non-parallel

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

```
##    user  system elapsed
##   0.793   0.038   0.832
```

Dynamic knapsack solution

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

```
##    user  system elapsed
##   5.434   0.081   5.522
```

Greedy knapsack solution

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

```
##    user  system elapsed
##   1.988   0.156   2.149
```

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 0.003 2.647       0  167           c("matrix", "unlist")
## 2 0.001 0.128       0    0                         "matrix"
## 3 0.003 0.053       0   65         c("mclapply", "lapply")
## 4 0.001 0.040       0   13 c("mclapply", "selectChildren")
## 5 0.002 0.056       0   19        c("mclapply", "cleanup")
## 6 0.004 2.293       0    9           c("lapply", "Filter")
##                         src
## 1 matrix/unlist
## 2 matrix
## 3 mclapply/lapply
## 4 mclapply/selectChildren
## 5 mclapply/cleanup
## 6 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))
```

```
##    time alloc release dups                   ref           src
## 1 0.006 2.671       0  167 c("matrix", "unlist") matrix/unlist
## 2 0.001 3.572       0    0              "matrix" matrix
## 3 0.001 1.278       0 3813     c("apply", "FUN") apply/FUN
## 4 0.001 1.860       0 1391               "apply" apply
## 5 0.002 4.358       0 3694     c("apply", "FUN") apply/FUN
## 6 0.001 0.000       0 3068               "apply" apply
```

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 70)
```

```
##     time   alloc release  dups                                    ref
## 1  0.007   9.915   0.000  5059 c("compiler:::tryCmpfun", "tryCatch")
## 2  0.080 162.500 147.113    65              c("matrix", "replicate")
## 3  0.001   1.228   0.000     0                                 "matrix"
## 4  0.001   2.656   0.000  2535                            character(0)
## 5  0.002   4.049   0.000  7299                                   "max"
## 6  0.006   8.765   0.000 23657                            character(0)
## 7  0.001   1.026   0.000  1003                                   "max"
## 8  0.001   0.791   0.000  2121                            character(0)
## 9  0.002   1.635   0.000  4019                                   "max"
## 10 0.002   1.431   0.000  2595                            character(0)
## 11 0.002   3.048   0.000  6970                                   "max"
## 12 0.006   8.009   0.000 14581                            character(0)
```

```
## 13 0.001   0.638   0.000  2654                                  "max"
## 14 0.005   3.299   0.000  6937                             character(0)
## 15 0.002   1.108   0.000  3169                                  "max"
## 16 0.003   2.448   0.000  4746                             character(0)
## 17 0.001   0.400   0.000   628                                  "max"
## 18 0.004   1.940   0.000  3446                             character(0)
## 19 0.001   0.657   0.000  1387                                  "max"
## 20 0.007   6.580   0.000 13402                             character(0)
## 21 0.001   0.761   0.000  1555                                  "max"
## 22 0.009  15.841  77.575 36283                             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
```

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 44)
```

```
##    time alloc release dups                                  ref
## 1 0.007 5.902       0 3083 c("compiler:::tryCmpfun", "tryCatch")
## 2 0.007 3.349       0  173       c("stopifnot", "is.data.frame")
## 3 0.004 7.767       0   44              c("replicate", "sapply")
## 4 0.001 0.395       0    5                             "order"
## 5 0.002 0.217       0   31                          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.