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
##   1.181   0.115   1.312
```

Non-parallel

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

```
##    user  system elapsed
##   0.901   0.037   0.945
```

Dynamic knapsack solution

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

```
##    user  system elapsed
##   5.403   0.109   5.531
```

Greedy knapsack solution

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

```
##    user  system elapsed
##   2.086   0.160   2.270
```

Profiling

Bruteforce knapsack solution

Parallel

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

```
## Reducing depth to 2 (from 7)
```

```
##     time alloc release dups                            ref
## 1  0.005 2.647       0  161           c("matrix", "unlist")
## 2  0.001 0.128       0    0                         "matrix"
## 3  0.003 0.042       0   65         c("mclapply", "lapply")
## 4  0.001 0.026       0   11 c("mclapply", "selectChildren")
## 5  0.001 0.006       0    5                      "mclapply"
## 6  0.001 0.014       0    3 c("mclapply", "selectChildren")
## 7  0.001 0.032       0    4                      "mclapply"
## 8  0.001 0.025       0   13        c("mclapply", "cleanup")
## 9  0.004 2.421       0    8           c("lapply", "Filter")
## 10 0.001 0.000       0    1              c("lapply", "FUN")
##                         src
## 1  matrix/unlist
## 2  matrix
## 3  mclapply/lapply
## 4  mclapply/selectChildren
## 5  mclapply
## 6  mclapply/selectChildren
## 7  mclapply
## 8  mclapply/cleanup
## 9  lapply/Filter
## 10 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

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

```
##    time alloc release dups                   ref           src
## 1 0.009 5.497       0  161 c("matrix", "unlist") matrix/unlist
## 2 0.001 3.541       0 3028     c("apply", "FUN") apply/FUN
## 3 0.001 1.234       0 3853              "apply" apply
## 4 0.002 3.761       0 2369     c("apply", "FUN") apply/FUN
## 5 0.001 0.000       0 3057              "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 66)
```

```
##     time   alloc release  dups                                  ref
## 1  0.007   9.748   0.000  4350 c("compiler:::tryCmpfun", "tryCatch")
## 2  0.057 161.430 147.559   774             c("matrix", "replicate")
## 3  0.001   4.071   0.000     0                          "matrix"
## 4  0.003  12.440   0.000 30725                        character(0)
## 5  0.001   1.266   0.000  3386                               "max"
```

```
## 6  0.002   6.395   0.000 12845                          character(0)
## 7  0.001   2.829   0.000  2985                                 "max"
## 8  0.005  10.599   0.000 23816                          character(0)
## 9  0.001   0.580   0.000  3926                                 "max"
## 10 0.001   2.741   0.000  1197                          character(0)
## 11 0.001   0.326   0.000  5662                                 "max"
## 12 0.004   7.680   0.000  9138                          character(0)
## 13 0.001   2.016   0.000  7409                                 "max"
## 14 0.001   2.485   0.000  4164                          character(0)
## 15 0.001   1.653   0.000  5137                                 "max"
## 16 0.001   1.024   0.000  3415                          character(0)
## 17 0.001   0.000  75.625  2463                                 "max"
## 18 0.001   1.096   0.000  7414                          character(0)
## 19 0.001   7.631   0.000  2266                                 "max"
## 20 0.002   0.697   0.000 17211                          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
```

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[1:2000,], W = 3500))
```

```
## Reducing depth to 2 (from 39)
```

```
##    time alloc release dups                                  ref
## 1 0.005 6.219       0 3201 c("compiler:::tryCmpfun", "tryCatch")
## 2 0.001 0.000       0  103                               "order"
##                            src
## 1 compiler:::tryCmpfun/tryCatch
## 2 order
```

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.