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=TRUE))
```

```
##    user  system elapsed 
##   1.305   0.251   1.567
```

Non-parallel

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

```
##    user  system elapsed 
##   0.917   0.042   0.961
```

Dynamic knapsack solution

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

```
##    user  system elapsed 
##   5.399   0.109   5.513
```

Greedy knapsack solution

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

```
##    user  system elapsed 
##   2.006   0.133   2.142
```

Profiling

Bruteforce knapsack solution

Parallel

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

```
## Reducing depth to 2 (from 9)
```

```
##    time alloc release dups                              ref
## 1 0.005 2.686       0  167           c("matrix", "unlist")
## 2 0.001 0.152       0    0                        "matrix"
## 3 0.003 0.061       0   69         c("mclapply", "lapply")
## 4 0.001 0.009       0   15                      "mclapply"
## 5 0.001 0.019       0    2 c("mclapply", "selectChildren")
## 6 0.003 0.418       0   18        c("mclapply", "cleanup")
## 7 0.004 2.038       0    6           c("lapply", "Filter")
##                         src
## 1 matrix/unlist
## 2 matrix
## 3 mclapply/lapply
## 4 mclapply
## 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 0.003 2.647       0  167 c("matrix", "unlist") matrix/unlist
## 2 0.001 2.946       0    0              "matrix" matrix
## 3 0.003 1.859       0 4471     c("apply", "FUN") apply/FUN
## 4 0.001 0.934       0  685               "apply" apply
## 5 0.003 4.563       0 5162     c("apply", "FUN") apply/FUN
## 6 0.001 0.768       0  817               "apply" apply
## 7 0.001 0.000       0  834     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  0.009   9.734   0.000  4742 c("compiler:::tryCmpfun", "tryCatch")
## 2  0.001   0.514   0.000   371                c("$", "$.data.frame")
## 3  0.071 169.476 154.518    11              c("matrix", "replicate")
## 4  0.002   3.909   0.000     0                              "matrix"
## 5  0.001   5.515   0.000  7384                          character(0)
## 6  0.001   6.990   0.000 11394                                 "max"
## 7  0.007  19.561   0.000 52688                          character(0)
## 8  0.002   4.959   0.000 10428                                 "max"
## 9  0.004   8.771   0.000 14036                          character(0)
## 10 0.002   3.838  77.326 14356                                 "max"
## 11 0.004  14.993   0.000 32873                          character(0)
##                               src
## 1  compiler:::tryCmpfun/tryCatch
## 2  $/$.data.frame
## 3  matrix/replicate
## 4  matrix
## 5
## 6  max
## 7
## 8  max
## 9
## 10 max
## 11
```

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

```r
lineprof(greedy_knapsack(x = knapsack_objects[1:20000,], W = 3500))
```

```
## Reducing depth to 2 (from 29)
```

```
##    time alloc release dups                                   ref
## 1 0.002 1.070       0 3081 c("compiler:::tryCmpfun", "tryCatch")
## 2 0.007 6.024       0  175       c("stopifnot", "is.data.frame")
## 3 0.002 5.093       0   44              c("replicate", "sapply")
## 4 0.001 0.701       0    5                               "order"
## 5 0.001 0.000       0   35                          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.