Algorithmic Methods for Mathematical Models

Lab Session 5 - GRASP, BRKGA

2022-2023

NIKOLAOS BELLOS

ANASTASIOS PAPAZAFEIROPOULOS

a) Pseudocode for GRASP algorithm

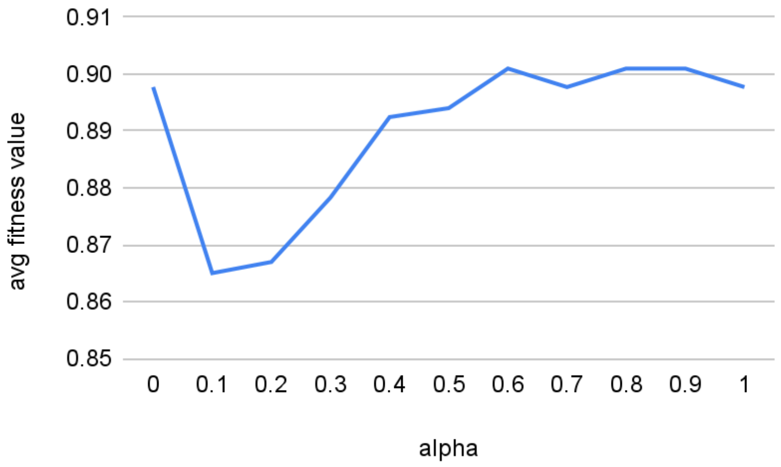
```
'''
GRASP
'''

# 1. Get the initial solution
# 2. Search the space for candidate feasible solutions (C)
# 3. Construct the RCL list from the candidate solutions (C)
# 4. Select a solution from the RCL list at random
# 5. If there are no new candidate solutions or time limit is exceeded,
stop iterating

# RCL algorithm (Restricted Candidate List) : select random

def grasp():
    solution = initialSolution() # from greedy
    fitness = solution.getFitness()
    while True:
        candidate_solutions = getCandidateSolutions(solution)
        if not candidate_solutions:
            break
        rcl = constructRCL(candidate_solutions, a)
        if not rcl:
            break
        solution = selectRandom(rcl)
        fitness = solution.getFitness()

    return solution

def constructRCL(candidate_solutions, a):
    rcl = []
    sort(candidate_solutions, key=fitness, asc)
    q_min = candidate_solutions[0].getFitness()
    q_max = candidate_solutions[-1].getFitness()
    rcl_max = q_min + a * (q_max - q_min)
    for candidate in candidate_solutions:
        if candidate.getFitness() <= rcl_max:
            rcl.append(candidate)
    return rcl
```

Finding the best value for parameter α

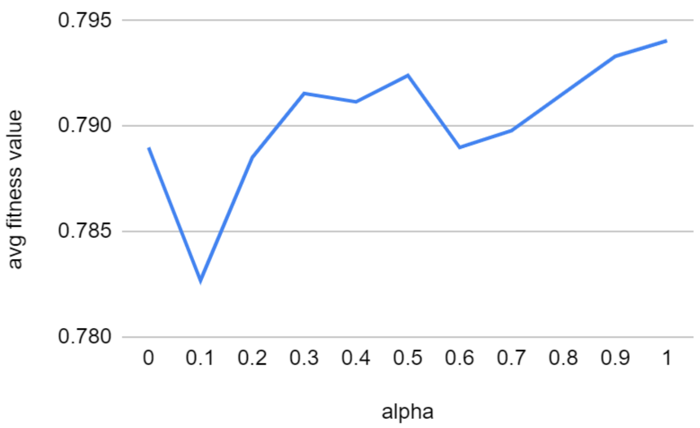
We generate 2 medium data instances for : 20 CPUs, 60 Tasks
and got the following results for the parameter **alpha**

Data Instance 1

parameter alpha	Iteration 1	Iteration 2	Iteration 3	Average
0 (greedy)	0.89769728	0.89769728	0.89769728	0.89769728
0.1	0.86308435	0.86482479	0.86698481	0.86503458
0.2	0.8678784	0.8698144	0.86610604	0.86699222
0.3	0.87570038	0.87798065	0.88090993	0.878305155
0.4	0.89769728	0.89316346	0.88716351	0.892430395
0.5	0.89196302	0.89020588	0.89603891	0.894000965
0.6	0.90094845	0.90094845	0.90094845	0.90094845
0.7	0.89769728	0.89769728	0.89769728	0.89769728
0.8	0.90094845	0.90094845	0.90094845	0.90094845
0.9	0.90094845	0.90094845	0.90094845	0.90094845
1 (random)	0.89769728	0.89769728	0.89769728	0.89769728



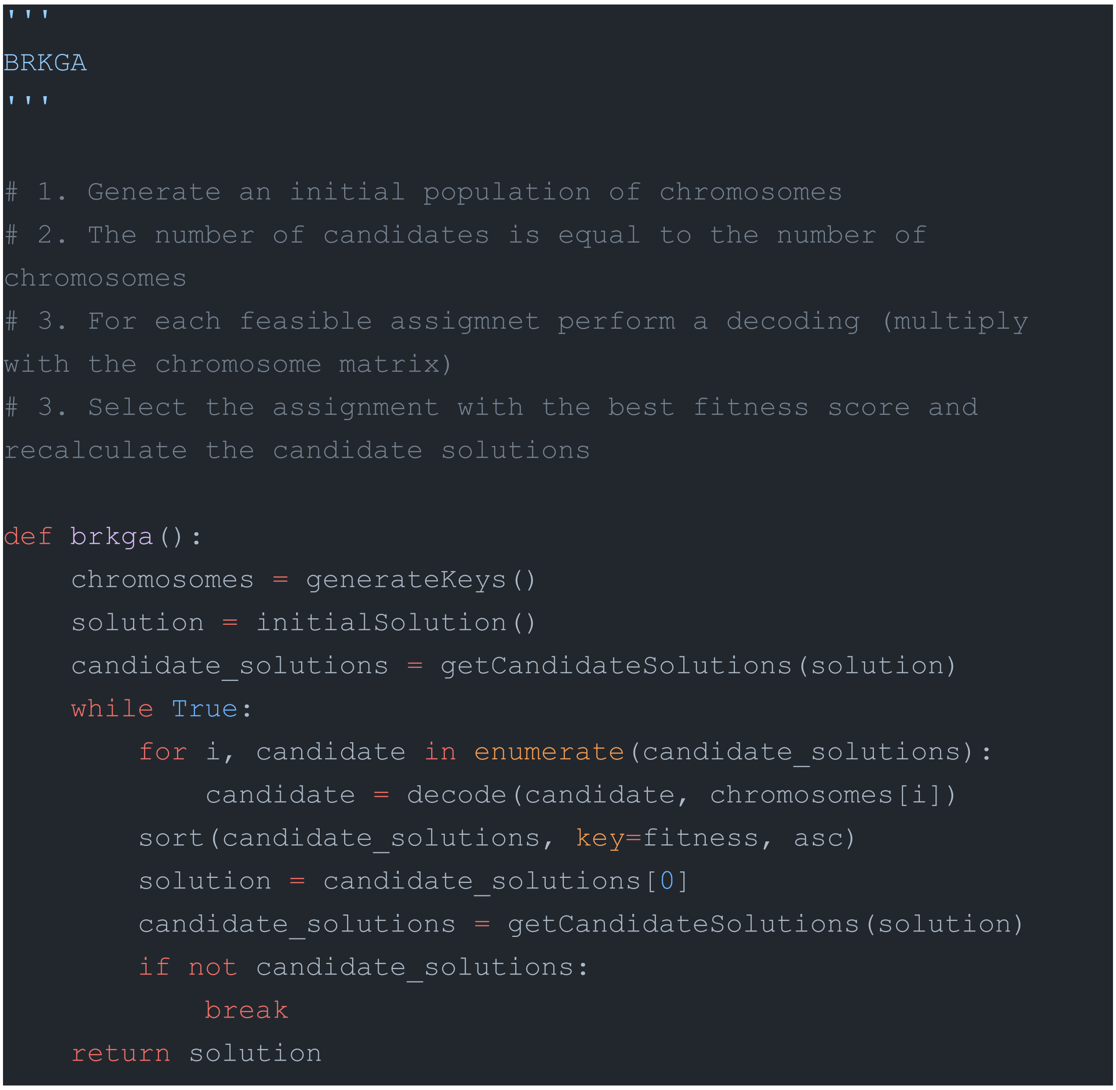
Data Instance 2				
parameter alpha	Iteration 1	Iteration 2	Iteration 3	Average
0 (greedy)	0.78897925	0.78897925	0.78897925	0.78897925
0.1	0.78239184	0.78338489	0.78294172	0.78266678
0.2	0.78978165	0.78801199	0.78723848	0.788510065
0.3	0.78978165	0.78897925	0.79331094	0.791546295
0.4	0.79331094	0.78897925	0.78897925	0.791145095
0.5	0.79581097	0.78897925	0.78897925	0.79239511
0.6	0.78897925	0.79331094	0.78897925	0.78897925
0.7	0.78978165	0.79331094	0.78978165	0.78978165
0.8	0.78978165	0.78978165	0.79331094	0.791546295
0.9	0.79331094	0.78978165	0.79331094	0.79331094
1 (random)	0.79478535	0.79331094	0.79331094	0.794048145



Conclusions

From the 2 plots we take the alpha value in which we observe the best fitness scores and take the average of those alpha values.
In both cases we observe that alpha = 0.1 give the best scores.
We will use this value in all of the following experiments.

b) Pseudocode for BRKGA algorithm



```
'''
BRKGA
'''

# 1. Generate an initial population of chromosomes
# 2. The number of candidates is equal to the number of
chromosomes
# 3. For each feasible assigmnet perform a decoding (multiply
with the chromosome matrix)
# 3. Select the assignment with the best fitness score and
recalculate the candidate solutions

def brkga():
    chromosomes = generateKeys()
    solution = initialSolution()
    candidate_solutions = getCandidateSolutions(solution)
    while True:
        for i, candidate in enumerate(candidate_solutions):
            candidate = decode(candidate, chromosomes[i])
        sort(candidate_solutions, key=fitness, asc)
        solution = candidate_solutions[0]
        candidate_solutions = getCandidateSolutions(solution)
        if not candidate_solutions:
            break
    return solution
```

The **chromosome** changes the actual weights / resources of the computers in the specific candidate solution.

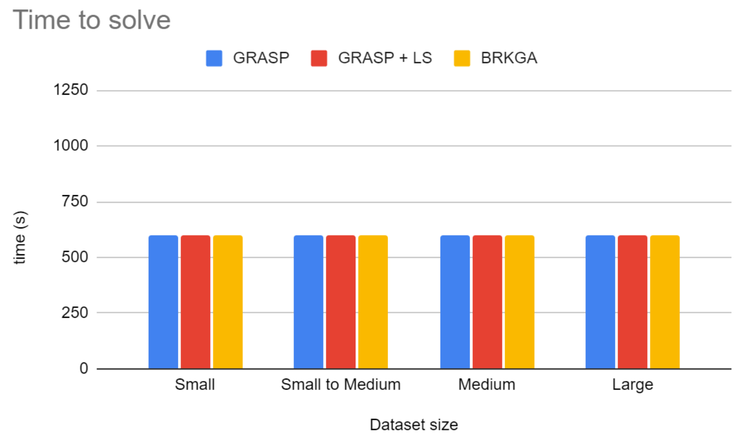
Combination of BRKGA parameters

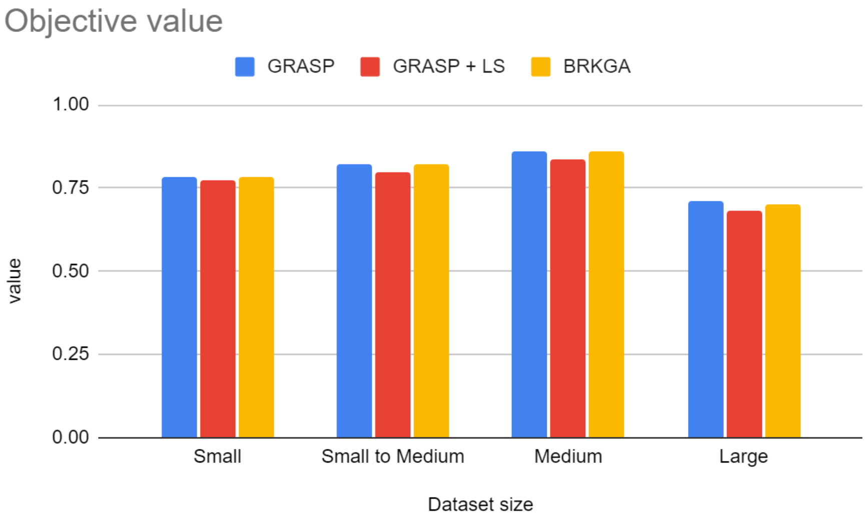
After testing, the parameters that worked best have been the following:

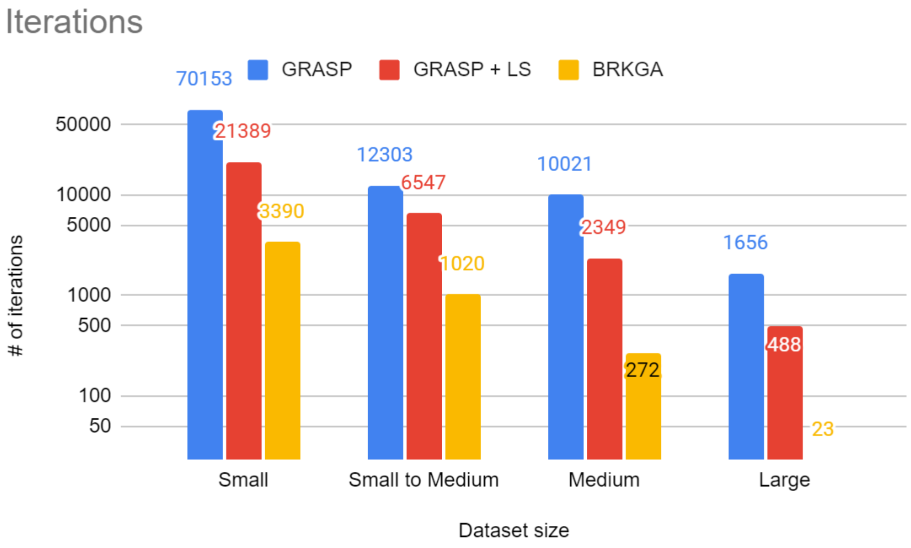
- Size of population	: 10
- Inheritance probability	: 0.7
- Elite set percentage	: 0.2
- Mutant set percentage	: 0.1

c) <u>Solve the previous instances with GRASP, GRASP + LS, GRKGA</u>

In the following plots we can see the results from running the 3 algorithms in for the instances of different sizes



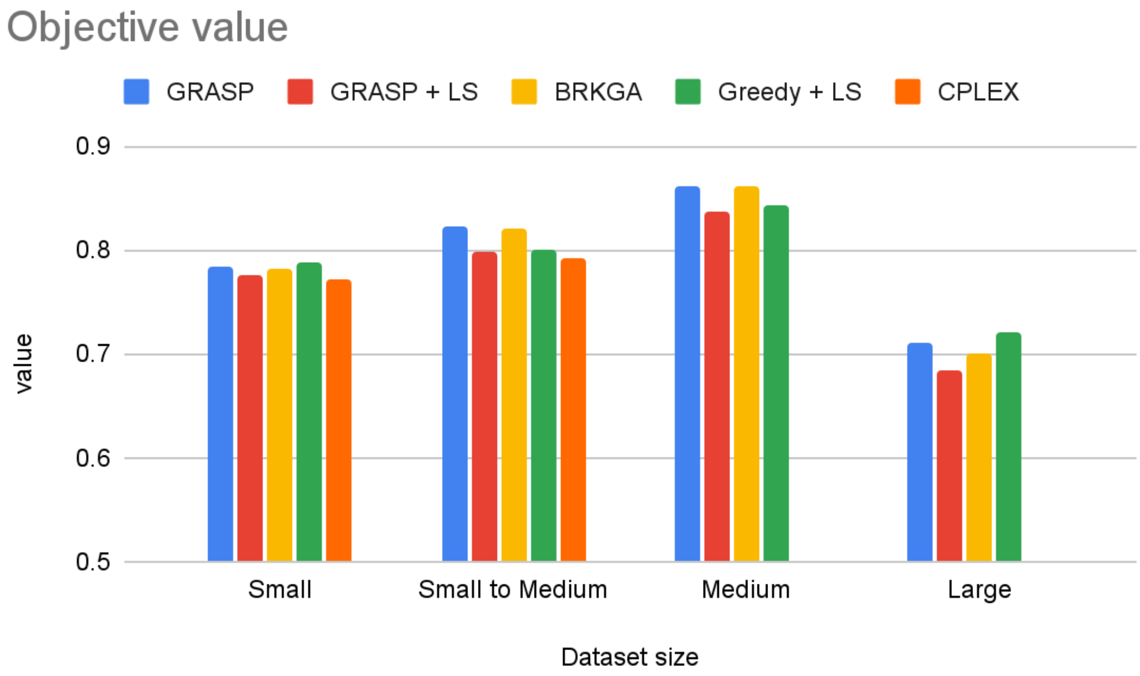




d) <u>Compare results with previous lab (Greedy, Local Search)</u>

We are only interested in comparing the objective value that these different algorithms produce (considering Greedy+LS and CPEX calculate the result in under 1 sec performing 1 iteration)

What we observe is that GRASP+LS and CPLEX usually produce the best results. GRASP with constructive phase only always gives inferior results. Also, BRKGA and Greedy+LS vary in the quality of their results, depending on the dataset, but they usually produce a good enough result (but never the best).


Objective value
GRASP
GRASP + LS
BRKGA
Greedy + LS
CPLEX
value
Small
Small to Medium
Medium
Large
Dataset size