GRASP and Path Relinking in Python

Final project: Modelos de Investigación Operativa

Sofia Friedsam
Lara Frost

Handed it on January 6, 2024.

Contents

1 Objective of the project **2**

2 GRASP in python **4**

3 Path relinking in python **7**

4 Results **10**

Appendix **17**

1 Objective of the project

In optimization the maximum diversity problem (MDP) describes the problem of choosing a subset of elements from a set so that the pairwise distances between the elements are maximized. Finding a maximum diverse set finds applications in ecology to establish diverse and therefore viable systems or to build diverse groups as representative samples in product design and testing. [2] There exist multiple algorithms to solve MDPs, in this project work we focus on GRASP and path relinking.

All our code can be found in our Git Repository [1].

1.1 Introduction to GRASP and path relinking

1.1.1 GRASP

GRASP stands for Greedy Randomized Adaptive Search Procedure and describes a multi-start method. It consists of two parts. At first a greedy algorithm is applied to construct a first solution. Then a local search is performed to improve the solution. In the first step, the construction part, candidate elements are defined and then a greedy function is applied. The aim is to find elements with the largest distance. In a pure greedy approach we would simply choose the one element to maximize the distance. However, we use a mix of greediness and randomization using a so-called Restricted Candidate List (RCL) with potential elements to enter the solution. We calculate the RCL of the set of candidates C as

$$RCL = \{i \in C : c^{(e)} \geq c^{max} - \alpha(c^{max} \breve{} c^{min})\},$$

where $c^{(e)}$ is the greedy function describing the increment associated with including an element e and c^{max} and c^{min} the largest resp. smallest incremental gain. The parameter $\alpha \in [0, 1]$ is a randomization parameter, where 0 equals a pure greedy approach and 1 a pure randomized construction. From the RCL a random element is chosen to enter the solution.

In the second step a local search (LS) is performed in order to find the local optimum in the current neighborhood.

Given an initial solution x_0 stemming from the GRASP algorithm, a neighborhood $N(x)$ and a function $f(x)$ to be minimized we want to find x according to

$$\begin{aligned} &x = x_0; \\ &\text{while } (\exists y \in N(x) | f(y) < f(x)) \\ &\quad x = y. \end{aligned}$$

Then the set x is the local minimum of $f(x)$ and therefore the result of the local search.

The effectiveness of the local search depends on the neighborhood structure and the function to be minimized, which are usually fixed, as well as the starting solution, which can be altered to get better results.

1.1.2 Path relinking

The basic idea of path relinking (PR) is to explore new solutions along a path that connects two high quality solutions, that is, nodes are successively exchanged in order to get from one solution from the other. In PR the starting solution is called initial solution and the target solution is the so-called guiding solution. In each step two nodes are exchanged against each other so that the objective function is maximized. The resulting solution is called intermediate solution and basically functions as the new initial solution for the next step. The resulting path is a path in the neighborhood of the already good solutions and might therefore uncover more high quality solutions. For PR to work effectively it is important that initial and guiding solution differ from each other sufficiently.

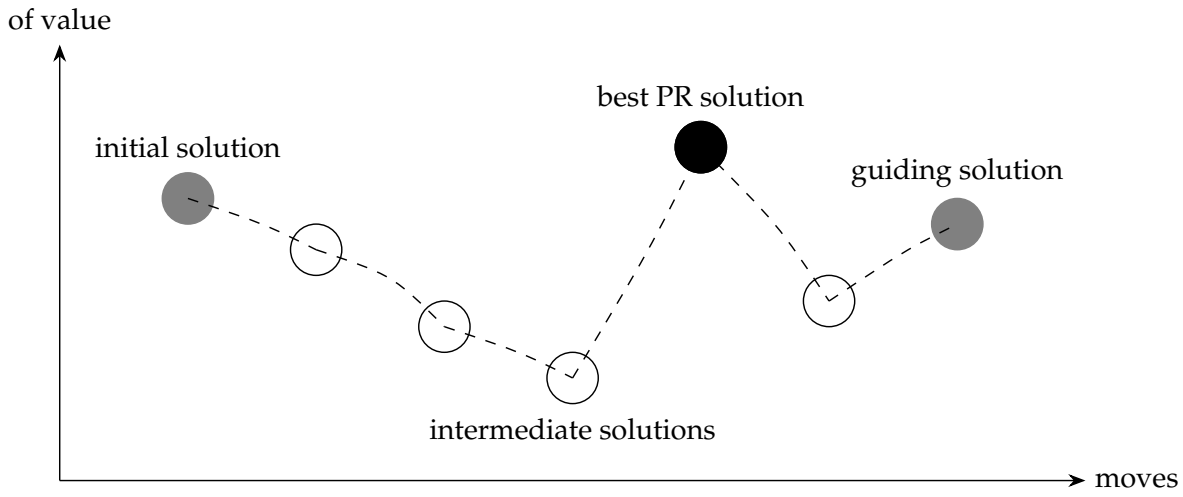

Figure 1: Illustration of path relinking.

1.2 Objective of the project

In this project we will use the already implemented GRASP algorithm with local search to obtain a set good diverse solutions, a so-called elite set. We will then use this elite set to choose pairwise solutions as the initial and guiding solution for PR.

The goal of this project is to find the best elite set and to implement an efficient algorithm for PR. We aim to find the solutions with the highest objective function values in order to maximize diversity.

The data provided consists of 20 instances with 2000 nodes each. Out of these 2000 nodes 500 are to be selected for the best solution. To quantify the quality of our different algorithms we will run all algorithms for each data set and select the best objective function value. Then we will calculate the deviation of the objective function value of the different algorithms to the best performing one. Lastly we can then sum over all instances to see the algorithmic approach with the least deviation to the best solution overall as well as a relative frequency of a given approach being the best one.

2 GRASP in python

Our first idea is to modify the given GRASP algorithm to obtain a set of solutions, so that we can apply the PR method on them. The modified algorithm consists of two parts: Calculating more solutions with the given GRASP algorithm and then applying the GRASP method again to this set of solutions. That way we can obtain a set of solutions that are in some kind distant from each other. We hope to get better results this way since there are more varied PR steps between the obtained solutions.

2.1 Calculating various solutions

At first, we wanted to calculate some solutions with the given GRASP algorithm and a precalculated α and save these solutions in an array. However the results we obtained did not vary too much from the solution obtained with the original GRASP with random αs. So we thought about letting the α improve itself. We start with a discrete set of values for α and use random numbers for the set of αs. The number of αs is given as a parameter in the function of GRASP. As we found out by experimenting with α, that there are even important differences between an α and an α+0.05, it would only be logical to inspect a small area around a good α to see if we can improve it. Therefore in a given interval around our so far best α we calculate some random numbers and compare the values of their solutions. Every in this process calculated solution can be saved in our array of solutions as they are calculated anyway. Since this new version of GRASP still did not hold significant improvements in the values of the objective functions, we wanted to find a better way to determine α.

After some investigation, we found some papers explaining Reactive GRASP. [3] [4] We decided to try the Reactive GRASP on our instances. We started again with a discrete set of values for α. First, each α is selected by the algorithm with the same probability. Reactive GRASP then adaptively changes the probabilities to which each α is picked in a way to favor values that produce good solutions. To start the learning process with some solutions for every α we calculate a predetermined amount of solutions for each of them. Then during the learning process, we need to define our value of the incumbent (i.e. best so far) solution and the average value of the solutions obtained with each of the αs. With this, we can calculate new probabilities for the αs. The problem we see with this method is the limited amount of α parameters used. For the whole range from 0 to 1, there is just a fixed set of αs. And also there were just minor differences in the probabilities for the αs which means the solutions did not differ a lot. We believe that maybe we should use more αs and more iterations or that the instances we are using are too small. Later we will see that with larger instances we obtain better values compared to the given algorithm.

2.2 Calculating an initial set for PR

Now that we have a lot of solutions for our instance we can use the given GRASP method on them to obtain a set of sufficiently different solutions. Therefore we need

to create an instance out of them. Obviously, our new nodes are the solutions but we need to calculate the distance between solutions efficiently. We decided to define the distance between two solutions as the sum of the distances between unequal nodes of those solutions. So now that we have a new instance with nodes and the distances between every node we can use GRASP on this instance to obtain a set of solutions that are different enough from each other.

2.3 Adjustments

Since GRASP does not consider the values of the solutions, but just the distances between the solutions, we want to modify the resulting set. Although we want solutions to differ from each other, we also want to keep our so far best solution. So if GRASP chooses a set of solutions, that does not contain the best so far, we will remove the worst solutions from the set and add our best. This way we can guarantee that this application of GRASP will not make us lose value in the result.

To improve our run time we decided to not keep every solution calculated in GRASP since we do not need too many solutions to start PR.

2.4 Changeable parameter

We want to point out that there are plenty of parameters that can be changed. For example the number of αs we use but also the number of iterations used to learn the α. Because of the small size of the instances, there were no significant changes in the obtained values. That is why we can not declare an optimal size of the set of αs. Another parameter is the size of the initial set. This parameter gives space to do a parameter analysis to compare the size of the initial set with the value obtained and the run time. However we will not cover this analysis in this work.

2.5 Comparison with normal GRASP

In this chapter, we want to compare our modified GRASP algorithm with the GRASP given in the lessons. To compare results we generate the mean value of the following values and use these definitions:

GRASP:	The GRASP algorithm as explained in the lecture.
GRASPMOD:	Our modified GRASP algorithm as explained above.
of:	Objective function value.
run time (s):	Running time in seconds.
dev:	Deviation of the of value of the algorithm to the best of value found.
% best of:	Relative frequency that the algorithm found the highest objective function value.

The following table 1 shows the results with respect to the number of iterations done.

algorithm	iter	of	run time (s)	dev	% best of
GRASPMOD	40	4753.35	3.83	0.31	0.53
GRASP	40	4765.22	3.98	0.07	0.60
GRASPMOD	100	4771.42	9.55	0.04	0.80
GRASP	100	4761.95	10.62	0.16	0.53
GRASPMOD	500	4782.00	45.85	0.07	0.73
GRASP	500	4782.51	51.68	0.06	0.67

Table 1: Comparison of GRASP and our modified GRASP with small instances.

Even though we compute % *best of* as a relative frequency it does not necessarily sum up to 1 in our results as we decided to count all algorithms that found the highest solution even if several algorithms found the same solution.

Unfortunately, we have to recognize that there is not a big difference between the modified GRASP and the normal one and this is even worse with a small amount of iterations. We believe that we do not have significant improvements because of the small size of the instances. Therefore we want to analyze the algorithms using large instances with n=2000. It is to say that it was almost impossible to use GRASP on these instances since the local search in the given GRASP would run way too long. So the following table 2 is calculated without the local search of GRASP.

algorithm	iter	of	run time (s)	dev	% best of
GRASPMOD	16	662434.20	8.34	0.02	0.75
GRASP	16	661829.50	8.29	0.11	0.25
GRASPMOD	32	662969.55	16.33	0.01	0.70
GRASP	32	662545.45	15.87	0.07	0.30
GRASPMOD	53	662994.45	26.68	0.02	0.80
GRASP	53	662534.80	27.34	0.09	0.20
GRASPMOD	100	662938.45	47.68	0.02	0.65
GRASP	100	662813.50	47.54	0.03	0.35

Table 2: Comparison of GRASP and our modified GRASP with large instances.

Now we can see some improvement in the values of our modified algorithm. On average our algorithm finds a better solution and there is not even a significant difference in the run time.

3 Path relinking in python

3.1 Implementations of path relinking

First of all we implemented the basic path relinking (PR) algorithm as shown in the lecture with diverse initial and guiding solutions stemming from our modified GRASP algorithm from section 2. We determined the nodes that we could keep since they were already present in both the initial and the guiding solution. Further we determined the nodes we had to enter into the guiding solution and the necessary exchanges to be made in order to do that and saved all these nodes in new sets. For each node in the initial solution we checked all possible exchange nodes and exchanged the nodes in order to maximize the objective function. This implementation is equivalent to the approach in the lecture, obtaining the best node to leave the solution set and the best node to enter in each step. Obviously this requires more computational time as this requires two for-loops and therefore a run time of n^2 with n the number of nodes to be exchanged. We therefore also implemented a simplified version of the PR where we omitted one loop of the optimization and set the node to exit the solution as a fixed random node. We then only searched for the best node to enter the immediate set.

However, using any of basic PR approaches on their own we saw for single data sets that PR does not actually significantly improve the solution as the initial or guiding solution often had a higher value of the objective function than even the best solution along the PR.

3.2 Path relinking with local search

We know that PR aims to find a better solution by moving from one already good solution to another. To get a graphical understanding we can imagine our solution space as a landscape of mountains and valleys where local optimums equal mountains. The PR algorithm then finds a path from one initial mountain to another guiding mountain. We hope that this path will go over other mountains as well so that we can find better solutions. However, with this imagination it is quite likely that our path does not actually lead us across another mountain top and instead leads us through valleys or along the side of mountains. To combat this we decided to implement a local search (LS) after a certain amount of steps. This allows us to not unintentionally move alongside a mountain while missing the top. We implemented two versions of this added local search algorithm. For a simpler solution we simply added the local search after a fixed number of steps. For a more advanced approach we determined the local search solution and checked whether this solution was better than our initial solution. If so we set the local search solution as our new intermediate solution for the next step of the PR.

We defined the number of steps by a frequency parameter in a way that a frequency of q equals a local search every $q * n$ steps where n is the number of nodes to be exchanged in order to get from the initial solution to the guiding solution, that is, the amount of steps in the PR algorithm. For example with 20 steps to be performed in the

PR and a frequency of 0.2 we will perform a local search every 4 steps in the PR and 5 in total. A frequency parameter equal to 0 means that only PR with no additional local search is performed.

The determination of the optimal amount of steps for our purposes is further explained below.

3.2.1 Determining the parameter for the local search

We expected the local search to improve the results compared to PR without local search and also expected that a lower frequency would result in higher objective function values as the local search is executed more often and therefore there are more possibilities to find a close local maximum. At the same time additional local searches mean an increased run time. We ran an analysis to find a frequency parameter with a good trade-off between these two variables.

To determine the optimal number of steps between each local search we tested five different values: 0, 0.05, 0.1, 0.2 and 0.3. We did not test the parameter for the whole data set but only for one data set with $n = 500$ were we applied our modified GRASP to find a good elite set and then ran the PR for all possible combinations of initial and guiding solution in the elite set.

As can be seen in table 3 our assumptions about the local search parameter proved to be correct. First we saw that including the local search as opposed to not including it lead to a significant increase of the objective function value. Also, as expected, increasing the frequency and thereby lowering the amount of local search iterations did decrease the objective function value but the decrease was not that significant. The run time was highest for the lowest frequency, again as expected, but did not decrease further after setting the frequency to 0.1 and above. (The run time for frequency equal to 0, that is the basic PR, is not representative here since our computer accidentally went into standby mode which prolonged the recorded time.)

frequency	of	run time (s)
0.00	7586.99	33.93
0.05	7658.11	31.90
0.10	7650.69	22.14
0.20	7638.70	21.11
0.30	7629.74	23.07

Table 3: Comparison of objective function value and run time for different frequency parameters for PR.

Given these results we decided to test our algorithms with frequency values of 0.1 and 0.2. We chose 0.1 as it is the lowest parameter with a similar run time to the higher ones. We wanted to see if there is a big difference compared to setting the parameter to 0.2. Additionally, we executed our algorithms with a parameter of 0.5. While we

did not test this here we wanted to check whether the additional increase in frequency could have substantial influence on the running time and how this would affect the objective function value.

3.2.2 Problems with the local search

We encountered some problems with the local search as we found that there were certain edge cases where our algorithm did not find a solution and we had to interrupt the program.

One problem we found was that the guiding solution can possibly be very close to a local maximum. For certain frequency parameters this lead to the local search always finding the same local optimum after some iterations and then going back to that local optimum in each local search step. This way the initial solution was always re-set to the local optimum and the algorithm never reached the guiding solution. We combated this by breaking the loop in our algorithm if the initial set for the next iteration stayed the same for more than one loop.

However, we could not identify all possible problems with our approach and therefore decided to also break our loop for the PR after the total amount of steps of the local search, defined by the frequency parameter, were performed and only keep the best result we obtained until that point.

4 Results

While the project description states to use the data set with instances of $n = 2000$ we found that running times for those instances far exceeded the given time limit of 1 or 15 minutes and therefore had to opt for the old data sets with instances of $n = 100$ or $n = 500$ respectively.

First we compared how the number of iterations for the GRASP affected the results. Since we used GRASP in our modified GRASP algorithm as well we set the number of iterations in the modified GRASP always equal to the ones for GRASP. Additionally we compared four different approaches to the PR algorithm with the original GRASP algorithm provided:

PR1: Basic PR algorithm as seen in the lecture.
PR2: Simplified PR algorithm where element that leaves the intermediate set is selected at random.
PR3: PR algorithm with local search after each set number of steps according to frequency parameter.
PR4: PR3 where the LS solution is used as initial solution for the next step.

We divided our analysis into the small instances with $n = 100$ and the larger ones with $n = 500$. To compare results we generated the mean value of objective function value, run time in seconds, deviation and relative frequency of best of value found. The running time for the PR algorithms is computed separately from GRASPMOD, meaning that the GRASPMOD has to be added to the run time of the PR for the total run time of each PR solution. Keep in mind that *% best of* does not necessarily sum up to 1. One could argue that only GRASP should be counted for this relative frequency here since performing PR is an extra step on top of the GRASP that only causes more computational cost if it does not enhance the solution and therefore is inferior in these cases.

4.1 Results from different frequency parameters

First we wanted to verify our findings about the best frequency parameter from section 3.2.1 . We did not see any differences in performance for the small data sets as these are likely too small and therefore concentrated on the big data sets with n=500. We compared the frequencies of 0.1, 0.2 and 0.5 against each other for 100 iterations and an elite set of size 3. As can be seen in table 4 for PR3 we had the same result as in

algorithm	freq	of	run time (s)
PR3	0.1	7737.40	40.76
PR3	0.2	7737.89	32.26
PR3	0.5	7726.90	32.97
PR4	0.1	7731.38	34.30
PR4	0.2	7743.05	59.82
PR4	0.5	7728.34	35.63

Table 4: Comparison of PR3 and PR4 for different frequency parameters for n=500.

our preliminary analysis of the frequency parameter: Increased frequency parameters lead to lower of values and lower run time. For PR4 we obtained contradictory results where a frequency of 0.2 produced a run time almost twice as long as for 0.1 but also generated a higher of value. For PR3 the of value for a frequency of 0.2 is also slightly higher than the one for 0.1 but this does not seem significant. The PR4 results did not seem reliable to us due to the increased run time and the other problems we encountered with PR4 in general. In the end we decided to perform the following analyses with a parameter of 0.1 in line with our preliminary analysis since the run time seemed manageable to us.

4.2 Results from instances n=100

We tested our instances for iterations of 25, 50, 80 and 300. For all analyses below we set the frequency parameter to 0.1 and the size of the elite set to 3. Table 5 shows exemplary results from the small data sets with 50 iterations. Our first finding is that the of value for all PR approaches is identical to GRASPMOD. We observed this for all small data sets except the small data set with 80 iterations where PR3 outperformed GRASPMOD by 0.1.

While GRASPMOD returns a slightly higher of value than GRASP here this is not true for all iteration values that we checked as explained in 2.5. We therefore attribute the better performance here to randomness due to the changes in random numbers as generated by the random seed.

Further we observed that GRASP and GRASPMOD results were identical for 300 iterations with a value of 355.16. Even though we cannot make definite statements

algorithm	of	run time (s)	dev	% best of
GRASP	353.53	0.08	0.33	0.67
GRASPMOD	354.23	0.10	0.13	0.83
PR1	354.23	0.02	0.13	0.83
PR2	354.23	0.00	0.13	0.83
PR3	354.23	0.03	0.13	0.83
PR4	354.23	0.01	0.13	0.83

Table 5: Results with 50 iterations for n=100.

about the optimal solution of the single instances, we strongly assume that both algorithms found the optimal solution for each data set within 300 iterations since a same mean value over the instances seems very unlikely.

A look at the of values and running times of GRASP in table 6 shows that as expected higher iterations lead to higher of values but we already find close results with only 25 iterations which is likely due to the fact that the data sets only contain 100 nodes. While the running time increased substantially with increasing iterations it is still below one second for all numbers of iterations. Overall we can say that employ-

algorithm	iter	of	run time (s)	dev	% best of
GRASP	25	352.41	0.05	0.65	0.50
GRASP	50	353.53	0.08	0.33	0.67
GRASP	80	355.06	0.15	0.03	0.83
GRASP	300	355.16	0.54	0.00	1.00

Table 6: Comparison of GRASP for different numbers of iterations for n=100.

ing only the basic GRASP is sufficient for small data sets and that high iterations most probably find the best of value for the small sets.

4.3 Results from instances n=500

We repeated the analyses above for the large data sets with 500 nodes. Table 7 shows exemplary results for 50 iterations. In general and as table 7 shows we found that for the larger iterations GRASPMOD outperformed GRASP as shown in section 2.5.

Additionally we found that out of the different PR approaches PR3 performs better or as good as the other algorithms each time across all iterations. We thought that setting the local search solution as new initial solution in PR4 would enhance the solution but that does not seem to be the case. We assume that focusing on the local solution in this way is detrimental to the solution as the algorithm is not able to move from one local optimum to another. However, the run time for PR4 is lower than PR3 which

indicates that the guiding solution might be reached faster. This observation is also in line with our results from checking different frequency parameters in table 4.

algorithm	of	run time (s)	dev	% best of
GRASP	7693.59	6.63	0.43	0.22
GRASPMOD	7699.81	6.19	0.35	0.00
PR1	7699.90	31.28	0.35	0.00
PR2	7699.83	1.05	0.36	0.00
PR3	7722.13	32.99	0.06	0.78
PR4	7711.35	25.29	0.20	0.11

Table 7: Results with 50 iterations for n=500.

Interestingly, PR2, the simplified PR, performs almost equal to PR1 in regard to the of value. For all numbers of iterations the absolute of value difference between PR1 and PR2 was constantly around 1. However, we can see that PR2 has an extremely low running time while PR1 takes about 30 seconds for each data set. This inspired us to combine PR2+PR3 to a new algorithm PR5 hoping to achieve a result combining both: good performance while maintained a low run time. The results of this analysis are shown in table 9.

PR5: Simplified PR algorithm as in PR2 with normal local search as in PR3.

Comparing performance and run time in dependence of the number of iterations for GRASP as done for n=100 we again can verify with table 8 that higher iterations lead to higher run times and higher of values. The last fact is more significant here since we deal with larger data sets. GRASP or respectively GRASPMOD with a similar run time are the strongest determinants for our overall run time since the PR algorithms are then independent of the number of iterations. Therefore one of our goals is to use PR approaches on top of GRASPMOD for low iterations but to obtain of values similar to 300 iterations. Another goal is of course to find the absolute best of value as we did for the n=100 instances, although that probably is only achievable with higher iterations. Indeed table 9 shows that our new algorithm PR5 reaches our first goal. PR5 achieves

algorithm	iter	of	run time (s)	dev	% best of
GRASP	25	7678.60	3.52	0.60	0.11
GRASP	50	7693.59	6.63	0.43	0.22
GRASP	80	7701.03	10.77	0.40	0.44
GRASP	300	7715.12	45.46	0.29	0.11

Table 8: Comparison of GRASP for different numbers of iterations for n=500.

of values like PR3, which was the best algorithm before, but in much less run time. We

compared PR5 results for 50 iterations to the GRASP solutions in table 10. Here we added the GRASPMOD run time to PR5 to compare the total run times. We see that PR5 with only 25 iterations already outperforms GRASP even for 300 iterations with a sixth of the run time. Note that the nan value is due to the fact that we only ran PR5 and did not compare to all algorithms for 25 iterations. For 50 iterations PR5 achieves even better results in terms of of value. We therefore consider PR5 our overall best algorithm even if PR3 has better performance in regard to the percentage that the best of value was found. Our second goal is to find the absolute best value using PR5. To

algorithm	of	run time (s)	dev	% best of
GRASP	7693.59	6.65	0.44	0.22
GRASPMOD	7699.81	6.22	0.36	0.00
PR1	7699.90	32.39	0.36	0.00
PR2	7699.83	1.05	0.36	0.00
PR3	7722.13	35.62	0.07	0.56
PR4	7711.35	26.30	0.21	0.11
PR5	7722.58	5.06	0.07	0.33

Table 9: Results with PR5 with 50 iterations for n=500.

algorithm	iter	of	run time (s)	dev	% best alg
GRASP	25	7678.60	3.52	0.60	0.11
GRASP	50	7693.59	6.63	0.43	0.22
GRASP	80	7701.03	10.77	0.40	0.44
GRASP	300	7715.12	45.46	0.29	0.11
PR5	25	7715.98	7.65	0.00	nan
PR5	50	7722.58	11.27	0.07	0.33

Table 10: Comparison of GRASP and PR5 with low iterations for n=500.

this end we let PR5 run with a frequency parameter of 0.1 and 300 iterations with an extended elite set of size 10. We checked 3 different random seeds to account for the choice of random numbers by Python. The results are shown in table 11. With this the overall best result we obtained in this project was 7757.54 with a total average run time for PR5 of 111.58s.

algorithm	seed	of	run time (s)
PR5	1	7757.54	111.58
PR5	2	7745.15	112.44
PR5	3	7751.46	162.83

Table 11: Results of PR5 with 300 iterations for different random seeds for n=500.

All our code can be found in our Git Repository [1]. We want to mention that the tables with our best obtained values can be found in the appendix. These tables show the best results found for every instance using GRASPMOD and PR5 for the small instances and just GRASPMOD for the large instances like explained in this paper.

Literature

[1] Sofia Friedsam and Lara Frost. *GRASP and Path Relinking in Python*. https://github.com/ge96cok/MIO-Project/. Jan. 2024.

[2] Rafael Martí, Anna Martínez-Gavara, Sergio Pérez-Peló, and Jesús Sánchez-Oro. "A review on discrete diversity and dispersion maximization from an OR perspective". In: *European Journal of Operational Research* 299.3 (2022), pages 795–813. ISSN: 0377-2217. DOI: https://doi.org/10.1016/j.ejor.2021.07.044. URL: https://www.sciencedirect.com/science/article/pii/S0377221721006548.

[3] Mauricio G. C. Resende and Celso C. Ribeiro. "Greedy Randomized Adaptive Search Procedures: Advances and Extensions". In: Springer, 2019. Chapter Chapter 6, pages 169–220. ISBN: 978-3-319-91086-4.

[4] Mauricio G. C. Resende and Ricardo M. A. Silva. "Grasp: Greedy Randomized Adaptive Search Procedures". In: *Wiley Encyclopedia of Operations Research and Management Science*. John Wiley Sons, Ltd, 2011. ISBN: 978-0-470-40053-1.

Appendix

file	GMOD of	GMOD time (s)	GRASP of	GRASP time (s)
Type1_22.1.txt	662763.00	28.26	662557.00	28.06
Type1_22.10.txt	662600.00	27.78	661597.00	27.22
Type1_22.11.txt	663115.00	26.13	662500.00	25.64
Type1_22.12.txt	662645.00	28.45	662334.00	28.31
Type1_22.13.txt	662492.00	28.61	662273.00	28.76
Type1_22.14.txt	662982.00	27.88	662433.00	28.19
Type1_22.15.txt	663587.00	27.97	661936.00	26.06
Type1_22.16.txt	662969.00	25.24	662159.00	27.57
Type1_22.17.txt	663440.00	27.71	663106.00	27.50
Type1_22.18.txt	663316.00	27.33	663105.00	28.01
Type1_22.19.txt	662488.00	28.11	661091.00	28.23
Type1_22.2.txt	663508.00	24.50	662362.00	26.96
Type1_22.20.txt	663794.00	26.30	663095.00	27.97
Type1_22.3.txt	662319.00	28.13	662713.00	26.89
Type1_22.4.txt	662884.00	28.63	661744.00	24.23
Type1_22.5.txt	662802.00	19.25	663221.00	27.49
Type1_22.6.txt	663717.00	28.15	662883.00	27.81
Type1_22.7.txt	662366.00	23.66	662892.00	25.65
Type1_22.8.txt	663472.00	23.75	663043.00	27.98
Type1_22.9.txt	662630.00	27.86	663652.00	28.21

Table 12: Detailed results for all n=2000 files with GRASPMOD and GRASP.

file	GMOD of	GMOD time (s)	PR5 of	PR5 time (s)
MDG-a_10_100_m10.txt	355.50	0.52	355.50	0.53
MDG-a_12_100_m10.txt	354.25	0.52	354.25	0.58
MDG-a_13_n500_m50.txt	7736.60	36.39	7778.30	74.56
MDG-a_14_100_m10.txt	356.06	0.61	356.06	0.51
MDG-a_16_n500_m50.txt	7749.92	44.92	7787.41	69.43
MDG-a_17_n500_m50.txt	7785.36	38.64	7787.20	76.83
MDG-a_19_n500_m50.txt	7713.42	35.27	7713.42	71.73
MDG-a_1_100_m10.txt	360.15	0.59	360.15	0.46
MDG-a_20_100_m10.txt	349.31	0.61	349.31	0.46
MDG-a_20_n500_m50.txt	7707.95	37.26	7723.31	73.20
MDG-a_2_n500_m50.txt	7742.31	37.95	7746.23	68.35
MDG-a_4_100_m10.txt	355.72	0.64	355.72	0.57
MDG-a_5_n500_m50.txt	7691.09	38.91	7748.51	76.76
MDG-a_6_n500_m50.txt	7713.84	39.45	7768.91	73.46
MDG-a_9_n500_m50.txt	7739.14	39.58	7764.58	71.54

Table 13: Detailed results for all n=100 and n=500 files with GRASPMOD and PR5.