Fast Solutions for Maximum Diversity Problem

Implementation of Metaheuristic Methods towards MDP fast solutions.

Antonio Pariente Granero
Universidad de Valencia, Valencia, España

anpagra@alumni.uv.es

15 of December, 2021

Abstract

The Maximum Diversity Problem, is a state-of-the-art challenge, which takes an important role on numerous of the actual problems related with diversity, like the staff hiring or genetic diversity breeding can be. Althoug there are techniques that provide optimal solutions for this problem, usually lead to prohibitive computing times as the dimension of the instance increases due to the Np-Hard nature of the problem. However the Metaheuristic and Heuristic fields, provide a wide-range of techniques which, in spite of the solution lack of optimality ensurance, provide the computation times demanded by the real problems they aim to solve.

In this document we will implement the GRASP and Tabu algorithms, two metaheuristic methods, and compare their performance over eight instances of the MDP problem with N=500 and a subset of 25 elements, restricted to 1 and 15 min of execution. The implementation will take place in C++, due to the speed when compared with other general-purpose programming languages, a later parameter optimization will provide better results among the given instances. A final analysis and conclusion about the comparision will follow.

Contents

1 Introduction

The Maximum Diversity Problem (MDP), consists in selecting a subset of m elements from a wider set of size N while maximizing the diversity between them, the mathematical problem can be expressed as follows:

Let $N = \{1, 2, .., n\} : n \in \mathbf{N}$ be indexes, and $S = \{s_i : i \in N\}$ the initial set. Let $d_{i,j} i, j \in N$ be the distance measuring diversity between s_i and s_j, if $m < n$ is the desired size of the varied set the problem is now:

$$\text{Maximize } y = \sum_{i=1}^{n} \sum_{j=i+1}^{n} d_{i,j} x_i x_j;$$

$$\text{Subject to} \sum_{i=1}^{m} x_i \quad x_i \in \{0, 1\} \ i \in N$$

This straightforward formulation was introduced in Kuo et. al. (1993) [1], where also is shown that the nature of the MDP is of a Non-deterministic Polynomial acceptable problem. As a consequence of this, the computing time for a solution becomes intractable as the dimension increases.

The size of the given instances, $n = 500$, is enough to endow the MDP with a prohibitive computation time, to overcome this issue, the Metaheuristic field renounces to the optimality condition in favor of speed. Two methods satisfying this properties are GRASP and Tabu, both state-of-the-art algorithms in the MDP treatment.

The language chosen to implement both algorithms is C++, it is because of the speed provided by the memory usage and the compilation performed naturally by this language, that the results are expected to be better that would have been if another language was considered. Then, both methods are computed several times over each instance, with a final spline fit, to determine an enough good value of their respective parameters. This study is done in Matlab due to the facilities provided, focused on manipulation and visualization of data. The final result is compared to determine if one method is superior among these instances with the established times.

2 Heuristics

The *meta* prefix inherited from Greek meaning "after", and so we say that a method is metaheuristic if it constructs, finds or select an heuristic algorithm that provides sufficiently good solution for an optimization problem. This methods are one step beyond the Heuristics in complexity. With this on mind, we introduce some essential structures for the GRASP and Tabu method imple-

mentation.

- **D**: This variable is the symmetric square matrix of size 500x500, common to all given instances. This matrix is meant only for reading purposes, even more, to be read several times and as fast as possible. Since the dimensions do not vary between instances, we can pre-allocate an *int array* structure in memory. In C++, it is a natural choice, because it stores data contiguously in the memory and consequently the reading is enough fast for our goals.

  As an implementation detail, it is worth mentioning that, using an array of 500x500 size in the stack is neither safe nor effective, therefore the allocation of the array is carried out in the Heap.

- **Candidate List (CL)**: As the name suggests, this variable contains the elements that could be labeled as part of the solution but haven't been yet, by this we can recognize them and avoid repeating those already selected. In relation to the structure, the variable size and a fast way of finding items in the CL prevails, so it may seem more sensible to use the *list<int>* object, nevertheless, some sources [4], show that the performance of the *vector<int>* structure is superior finding elements and changing the size, when the dimensions are affordable, 500 elements seem to fit this requirements.

- **M**: The list of already chosen elements in the solution, at the end of the algorithm will contain the feasible solution with, hopefully, enough good value of the Objective Function. Continuing the preceding reasoning, the *vector<int>* type is a good option to represent this variable, it is worth considering that will be manipulated in the Local Search phase.

- **Contribution of CL (cont())**: The main role of this item is to prevent re-evaluating the Of every time an element is added to M, also it contains the sum of distances from each element of the CL, to every element in M. As it is attached to CL, and its size is variable, the structure *pair<vector<int>,vector<int»* is the one used to contain, at *pair.first()* the CL and at *pair.second()* the cont() vector, allowing an easy but attached use of the two variables.

- **Objective Function (Of)**: The variable storing the current Objective Function value, when finishing, this will contain the amount to be compared with, that means the quality of a solution. The integer variable *int*, is enough for the method.

- **Seed**: It is the pseudo-random behaviour of the random functions in machines, that a seed is needed. To generate it, the class *steady_ clock* from the *chrono* library is used. Although this class contains a high precision

clock, it requires a continued computation that would slow down the performance, instead of this, the *steady_ clock* is a monotonic clock since time epoch, with enough precision for out purposes.

2.1 GRASP

The acronym GRASP, stands for Greedy Randomize Adaptive Problem. The idea is to build a feasible and reasonable good solution and guide it through the solution space while improving it with a local search. Two main steps are performed, the Pseudo-Greedy Construction and the Local Search improvement. In addition to the variables mentioned above, we define the RLC:

- **Restricted Candidate List (RLC)**: This variable is made from elements in the CL whose cont() is within $con\_max - \alpha * (cont\_max - cont\_min)$ and $cont\_max$, where naturally $cont\_max$ and $cont\_min$ are the respective max and min contributions of the elements in CL, and α is a parameter controling the greed of the construction algorithm. As a consequence of its frequently changing nature, the $vector < int >$ structure seems enough accurate to represent the *RCL*.

2.1.1 Pseudo-Greedy Construction

The construction phase is ruled by a greedy algorithm, that considers, at each step, the element with greatest distance and incorporate it to the solution. However, this could result in a poor objective function value, and therefore the parameter α is added and a random element from *RCL* is chosen.

In the beginning, the selection of the two first elements with maximum distances in D is performed, those elements are included in M and the Of is updated by adding that one distance value between them. After this, the *cont()* of each element from *CL* is updated by adding the distance from this element to the ones already included in *M*. By doing this after every addition to M of an element x_i, spares several operations . This whole process is accomplished by a function named *update*(), considers the element to be included x_i from *CL*, the actual M and the *Of* value.

The next step consists in computing the *RCL* and uniformly selecting an index of it by random. Usually the *rand*() function would have been the one to consider, however when doing $rand()\ \%\ RCL.size()$, the election is not uniform since the multiples of *RCL* size, have more chance of being selected. To overcome this issue, the *random* package is used, in a more precise way, the generator based on the seed, is introduced with an instance of *uniform_ int_ distribution<int>*.

From here the process is reduced to mere repetition, selecting an element from *RCL*, using *update()* and recomputing *RCL*.

2.1.2 Local Search

This part of the algorithm focuses on improving the solution. For each element in M, the distance from it to all the other elements in the solution is stored in the variable *val*. After this, the CL is routed and if the difference between *cont()* and the distance to the selected element in M is greater than *val*, it is considered an improvement and swapped for the element in the solution, which now is part of the CL, when this happens, the process is applied again at the first element of M, to consider as much elements as possible.

The subsequent *update()* function is used to even the remaining contributions. If no candidate in CL is considered as an improvement, the process is performed in the next element from M, until the end is reached.

2.1.3 Performance

In the below graphics, a Pseudo-Greedy-Construction is performed and the the improvement of the Local Search becomes visible.

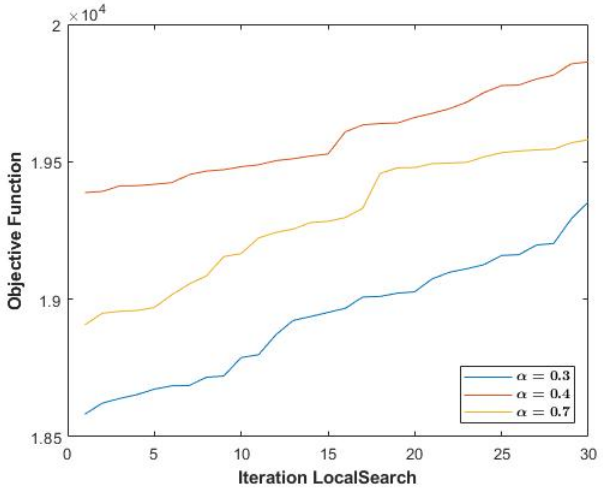

Figure 1: Performance of Local Search in Amparo instance over 30 iterations.

The above figure, shows the evolution of the Objective Function value over 30 iterations of Local Search. The initial solutions are generated with the Pseudo-Greedy Construction method and indicated α values. It becomes evident the forward behaviour of the Local Search, since the values are monotonically increasing.

2.2 Calibration

In the following graphics, the calibration of the α takes place. To accomplish this, the GRASP method is applied over all instances during 1 minute generating 30 results for each α value between 0.1 and 0.9. The extreme α values are discarded since this would result in a complete greedy or random process, respectively. To afford enough iterations multi-thread paradigm is used.

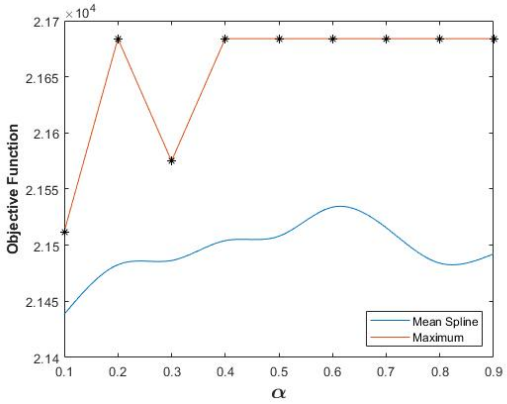

Figure 2: Mean and maximum Values.

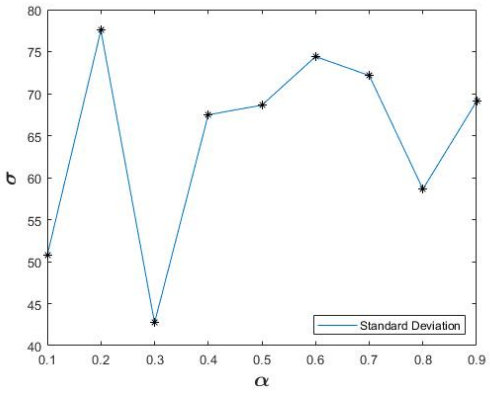

Figure 3: Standard Deviation of results.

Although the maximum values seem to be coincident (fig 2), the mean is enough diverse, therefore, a spline fit is applied with an r-square value of 0.9998, the result leads to an approximated value of α maximizing the mean of 0.6141. Despite of not providing the lowest *sd*, is an acceptable amount when compared with the alternatives (fig 3). These two elements, mean and deviation, indicate good performance among all the given instances.

2.3 Tabu

The Tabu method also consists in two steps, the first one constructs the solution random and the second one "walks" over the solution space, but considers worsening the Of, in order to later reach better solutions. To perform this task we define an additional structure:

- **Tabu List (TL)**: This variable stores, for each element in the problem, the iteration in which it was subtracted from M. If that iteration is lower than the parameter *tabu_ tenure*, the element is not considered when modifying the solution. To represent this resource, the *int array* of size 500 is used because the size is invariant since every element has a tabu value. Once again, this array is hosted by the Heap.

2.3.1 Random Construction

The process of constructing a random feasible solution, is done by randomly shuffling the entire instance and then introducing the first m elements in M. This may seem computationally expensive approach, but considering that the repetition is not allowed, to generate random values until none is duplicated, is a larger waste of time when both values are compared. To commit random shuffle, once more, the *Seed* value is introduced in a *uniform_ int_ distribution*.

After this, the *cont()* of each element that is not in M is calculated and the element is added to the *CL*.

2.3.2 Tabu Search

When considered, the solution space, as a surface with regions, the GRASP reaches a local maximum that could not be the best one. As an alternative, the Tabu, stores this local maximum and forces the solution to move, even if it implies a worse Of, with the intention to guide it towards another region with hopefully a better local maximum than the last one recorded.

When forcing the solution to move to the best possible value, if no restrictions are demanded, the natural path to follow is backwards, so no advance is accomplished. The solution is to introduce the list of forbidden values called Tabu List, defined above. If an element is taken out of M, that iteration is recorded in *TL*, avoiding it in the next iterations. The *tabu tenure* parameter controls the length of the list, and therefore, the amount of distance that is disposed to travel from one region to another. At the end the best of all local maximums is the one chosen as output.

2.3.3 Performance

To illustrate the Tabu Search performance, the method has been applied to Amparo instance. The result below shows the solution change for three different tabu tenures starting at close values.

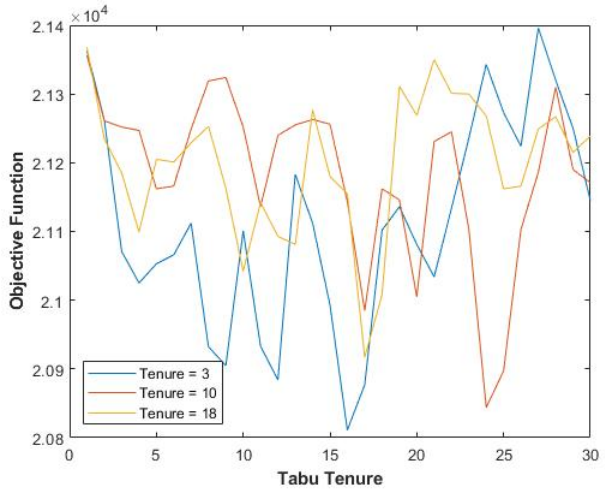

Figure 4: Performance of Local Search in Amparo instance over 30 iterations.

The figure (fig 4) exposes the nature of tabu strategy, because the solution in all the three cases gets worse and improves while traveling through regions of the solution space. The different tenure values change the behaviour of the search even for close initial solutions.

2.3.4 Calibration

Following the GRASP calibration strategy, this time without spline fit due to the integer nature of the *tabu tenure* tenure, the results showed below are obtained. The *tabu tenure* tenure values considered are lower than 500, avoiding the prohibition of all possible movements, i.e no elements in CL with allowed values. It is due to experimentation that usually values near half m perform good. Once again the iterations are executed in different threads to afford good calibration range.

This time the maximum values are plainly differentiated together with the mean (fig 5). The deviation values (fig 6) also seem to indicate that the value 5 for *tabu tenure* performs better among others.

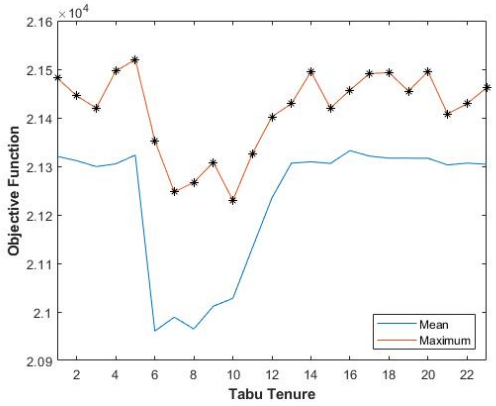

Figure 5: Mean and maximum Values.

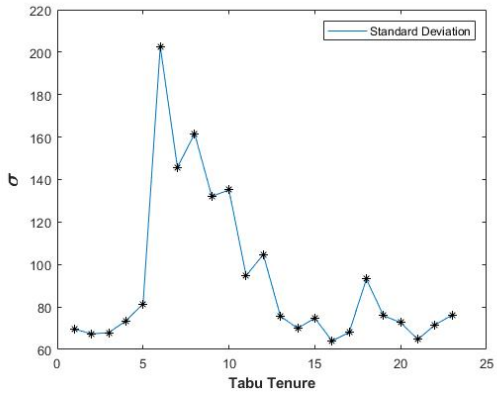

Figure 6: Standard Deviation of results.

3 Results

In this section the performance of both GRASP and Tabu methods is tested and exposed to the reader. The parameters used for this are the ones calculated in the previous section and the times of execution are restricted to 1 and 15 minutes. The instances used are eight MDP with size 500x500. The variables of the problem instead of binary 0 or 1 for each element are $m = 25$ elements within 0 and 499 composing the solution. Due to the size of m and the instances involved, only the *Of* value is going to be showed, however, the variable values together with the data used in the process and the code are available in the GitHub repository.

Instances\Algorithm	GRASP (1min)	GRASP (15min)	Tabu (1min)	Tabu (15min)	M_Tabu (1min)
Amparo	21519	21519	21189	21126	21376
Borja	21506	21506	21294	21023	21408
Daniel	21473	21473	21199	21032	21324
Emilio	21684	21684	21405	21273	21520
Jose	21506	21506	21321	21241	21325
Maria Jesus	21513	21513	21321	20932	21322
Raquel	21550	21550	21331	20960	21427
Virginia	21631	21631	21190	21015	21385

Table 1: Objective Function Values, of the solutions generated by GRASP and Tabu, when applied to eight 500x500 instances of MDP.

From the above table a few conclusions can be inferred. First the GRASP generates the same best results for 1 and 15 minutes. The media time of completing an iteration, measured over 400 of them, with the Intel(R) Core(TM) i5-7200U CPU with 2.5 GHz, and 8 GB of RAM used to generate the results, is 48,5419 milliseconds. If all the Pseudo-Greedy processes locate the initial solution in a reasonably good region, it is feasible that the solution doesn't improve when considering large computation times, because those regions are already explored.

As a counterpart, the Tabu algorithm behaves different in order to explore other regions, however, the calibrated *tabu tenure* is 5, this will make the solution travel to close places. In the case that a high peak containing the global maximum would exist far from the good regions explored by GRASP and Tabu, we could be missing the optimal solution.

To overcome this issue, a modification of the Tabu Search is made, selecting high *tabu tenure* values and clearing out the tabu list when completed. By doing this, the random-generated solution of Tabu, would be able to travel to further regions, exploring local optimum when arriving. The *M_ Tabu* column contains the solution of this modified Tabu with a *tabu tenure* of 50 and 1 min of execution. Although this modification generates better results than the standard Tabu, it continues performing worse than GRASP over the instances. This could mean that there is a region with concentrated peaks containing the optimal solution for this kind of data.

Considering all of this, it is clear that the performance over this instances and with the restricted times of computation, the GRASP performs over the Tabu, giving better solutions.

Variable\Instances	Amparo	Borja	Daniel	Emilio	Jose	Maria Jesus	Raquel	Virginia
Of	21519	21506	21473	21684	21506	21513	21550	21631
x1	77	127	104	15	399	19	110	31
x2	346	450	219	49	138	30	259	470
x3	79	220	253	45	478	186	264	80
x4	114	435	445	369	345	352	15	317
x5	54	21	172	447	392	280	12	381
x6	277	386	269	98	57	171	95	426
x7	460	131	66	149	430	58	219	463
x8	209	99	49	54	254	436	291	205
x9	201	421	116	224	45	212	359	104
x10	376	233	301	320	150	349	235	164
x11	342	406	478	28	123	336	17	408
x12	180	445	430	471	365	152	73	286
x13	361	143	494	285	194	458	415	52
x14	185	168	211	154	397	271	205	313
x15	378	476	453	73	208	361	445	84
x16	482	206	443	84	323	53	138	124
x17	478	34	186	321	491	40	86	400
x18	255	183	382	76	71	296	330	419
x19	302	442	389	42	439	119	228	78
x20	72	275	380	100	119	182	78	294
x21	259	418	329	229	327	206	302	210
x22	69	464	62	110	213	98	223	209
x23	67	93	336	244	423	394	134	345
x24	166	105	25	417	237	29	380	167
x25	430	470	67	214	318	140	459	41

Table 2: Solution values of the best results generated for the instances by GRASP and Tabu.

4 Conclusions

While the exact methods for hard problems like the MDP have been widely studied, in this paper we find the Metaheuristics providing enough good solutions with restricted small times. To fathom this approach, we have implemented the GRASP and Tabu Search methods to find solutions for eight instances from MDP with large size like $N = 500$. Between these two methods with 1 and 15 minutes of execution, the GRASP algorithm appears to give better results than Tabu in both of the times, even more with 1 minute of use, GRASP has enough computation time to reach his best solution.

Due to the obtained results, it seems a useful tool to overcome prohibitive computation times for Np-Hard problems like the MDP. Further research exploring different kind of instances and adapting the method to the solution space shape, could improve even more the results. In matter of computation, an accurate programming combined with optimized matrix operations, and multithread paradigm programmed in native language of graphic cards like CUDA, could contribute with significant improvement of computation time.

All the code and resources used in the paper are available in the GitHub repository.

References

[1] Dhir K.S. Kuo C.C Glover F. "Analyzing and modeling the maximum diversity problem by zero-one programming." In: *Decision Sciences 24 (6) 1171–1185.* (1993). DOI: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.418.1678&rep=rep1&type=pdf.

[2] Rafael Martí Abraham Duarte. "Tabu search and GRASP for the maximum diversity problem". In: *European Journal of Operational Research 178 (2007) 71–84* (2006). DOI: \url{https://www.researchgate.net/publication/222817653_Tabu_search_and_GRASP_for_the_maximum_diversity_problem}.

[3] Kenneth Sörensen Donald E. Knuth. "Encyclopedia of Operations Research and Management Science". In: University of Antwerp, 2013.

[4] Baptiste-Witch. *C++ benchmark – std::vector VS std::list VS std::deque.* URL: https://baptiste-wicht.com/posts/2012/12/cpp-benchmark-vector-list-deque.html.