Politecnico di Torino

III FACOLTÀ DI INGEGNERIA
Corso di Laurea Magistrale in Ingegneria Informatica

Project Presentation

Eternity II

A Tabu Search Approach

Students:
Davide Sanapo
Matricola S186760

Antonio Verardi
Matricola S184524

Anno Accademico 2011–2012

1 Algortihm Overview

1.1 Solution Representation, Feasibility and Creation

The simplified version of Eternity II we had to deal with presents both a small set of simple rules and constraints and an enormous solution space, that grows even bigger if also unfeasible solutions are considered alongside the feasible ones (on a 10x10 matrix all the possible tiles dispositions are over $3 \cdot 10^{160}$). Therefore, in order to avoid being lost in this huge unfeasible ocean and the risk of being able to reach only few feasibility islands, we decided to start from and generate only feasible solutions during our tabu search algorithm. To obtain this, we enforced the constraints related to border and corner tiles grey colours collocation deeply into the neighborhood generation process and the starting phase as well, completely removing the need of a feasibility check phase.

The structure of a general solution is separated from the tiles list (containing the information about colours and pieces category) and consists in a matrix storing only the index of the tile placed in the considered position and the rotation value of the tile itself. The number of matches of the solution is calculated completely scanning the matrix by rows and by columns at the time when the solution is created.

The starting solution needed by the algorithm is built-up in polynomial time $O(n^2)$ (where n represents the size of the matrix) simply by reading the list of the coloured squares compounding the game, dividing them into three categories (*corner*, *edge* and *inner*) and by placing each tile in the first free position available for its category. After that, the tile will be rotated to guarantee the satisfaction of the grey coloured triangles constraints. The initial solution is consequently constructed in a simple and fast deterministic way, without applying any heuristics to maximize the number of matches. Thus, in order to evaluate the quality of our solver on a set of random generated starting points, however maintaining reproducible the observations, we introduced a seed based randomized shuffling of the initial solution (which preserves feasibility, of course).

1.2 Neighborhood Generation

During each step of the algorithm a neighborhood is generated from the current solution, swapping and rotating two homogeneous tiles, or two tiles belonging to the same category. In each iteration, an initial position in the matrix is chosen according to a given politics (see section 3 for more details). The tile currently placed in the position just selected is swapped with every other tile of the same category, taking also into account all the rotation offsets that lead to feasible solutions. In order to enforce the constraints also in the neighborhood generation process, as stated before, we created three different generation procedures, one for each pieces category. If only moves as the ones described above are applied, the corner neighborhood size is a constant equal to 3, whereas the edge and the inner ones become polynomial functions of n, respectively $(4 \cdot (n-2) - 1)$ and $(((n-2) \cdot (n-2) \cdot 4 \cdot 4) - 13)$. All the three kinds of neighborhoods are generated alternatively in a cycle of three steps, as showed in the pseudocode 1 available in appendix.

A neighbor presents a lighter scheme with respect to the solution structure: each of them is composed only by the move applied to the current solution in order to build-up the neighbor itself and by the new number of matches. Matches are not calculated in $O(n^2)$ time as at the creation of the starting solution, but in a differential way from the matches of the base one. The complexity of this action is constant for each neighbor and in the worst case consists in 32 sums and differences plus the evaluation of few conditional expressions to check if the swapped pieces were adjacent. A single move is instead identified by the indexes of the tiles swapped, their position in the matrix and their new rotation offset. When a move is not tabu or meets the aspiration criteria, the neighbor that implements it becomes a complete solution and the search goes one step forward.

1.3 Three Different Tabu Lists

Having separated the pieces in the categories described before and considering that each of them corresponds to a set of possible moves with a very different cardinality from the others, creating three different tabu lists (one for each category) with different sizes seemed us to be the most reasonable choice. Although we maintained the possibility to choose between a common single tabu list implementation and the triple one, our initial intuition was verified by the experimental data obtained in the tuning phase of the software, when the triple tabu list implementation produced better results in all our simulations (see section 3).

During the test phase of the program we noticed that that the tabu lists presented several clones of the same tabu move. This was not an error, but a side effect of our representation of moves and neighbors. For example, all the 16 different moves generated by the swap of two tiles in the inner part of the matrix (each tile can have four different rotation offsets) correspond to the same forbidden reverse move, that, if it were applied, would restore the same solution before the swap had been performed. This issue leads to the creation of clones, which we have later transformed in a simple refresh of the expiration time of the tabu move already present in the list.

In addition, it is necessary to point out that this kind of neighborhood generation and tabu moves representation does not mark any group of solutions as tabu, but only simple moves. Paradoxically, in the worst case, the solution just left could be generated again in only two further moves, regardless the size of the tabu list. However, from the analysis of several detailed step-by-step running logs, our implementation seems not to be ever caught in any solutions generation cycle or local maximum. Furthermore, even if the neighbors generation rule we adopted in the solver presents so many weak points, it had reasonably produced good quality results, while other politics cons seem to fairly overcome their pros. For example, forbidding the placement of a tile in a certain position and consequently all the solutions sharing this attribute seem to us a too harsh criteria to move in the solution space, for the reason that it may forbid a great portion of good quality solutions which are actually very far from the one just discarded, having in common with that only one tile collocation. In fact, we tried to implement this rule, but we obtained average results worse of even 5 points with respect to the swap and rotate neighborhood outcomes. Also marking a whole solution as tabu, instead of a feature or a swap would be a worse choice, because not only the memory needed to store the tabu list, but also the complexity of the list checking phase would increase from $O(s)$ to $O(n^2 \cdot s)$, where s is the size of the tabu list.

The remaining part of the algorithm follows the basic structure of the tabu search, as available in pseudocode 1.

2 Enhancement Attempts

After the whole algorithm was set-up and implemented, we tried to apply simple changes to the base tabu search heuristics in order to improve the global results, meaning average score, best score and variance.

2.1 A Preliminary Search

The first enhancement we tried to apply (and the only one that still remains as default behaviour in the final version of the software, as showed in pseudocode 1) consists in performing a preliminary local search on the initial solution before the start of tabu search. Basing our attempt on the empiric principle that "a good solution may become a very good starting point", the local search we performed actually led to a real improvement in the outcomes, although we noticed that it often got stuck in a local minimum (frequently with the same number of matches) in few steps. For this reason we transformed it in a shorter and simpler version of the main tabu search, able of reaching better and various results. Thanks to this change, we were able to raise our average score of about 4 points and to increase our best of even 10 points.

2.2 Tableaux and Hills

After that, we decided to implement a strategy to be sure to avoid tableaux: when a tableau of a given length was met, a diversification of the current solution would provide a cheap way to escape from it. Nevertheless, our experimental data did not showed any new best solution, but actually an average worse from before. Analyzing some detailed logs of the test, we noticed that the diversification procedure was called especially at the very beginning of the tabu search and its frequency harshly decreased going further with the iterations. For this reason we consider our tabu search smart enough to escape from tableaux without any other enhancement. In fact, the chart in picture 1 and his detail in picture 2 suggest that the original tabu search has already a well diversified way to explore the solution space (also local maxima seem to be escaped in few iterations, too).

Another change we tested was to diversify the current solution if the number of non improving steps was growing larger than a given threshold (hills), but the results were poor as well, making us more confident in our previous implementation and its exploration strategy.

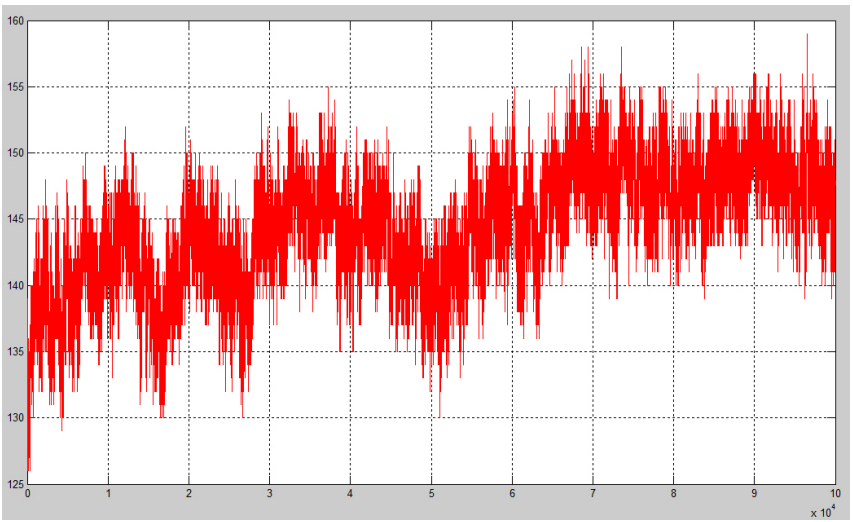

Figure 1: Step by step behaviour of the algorithm during a test

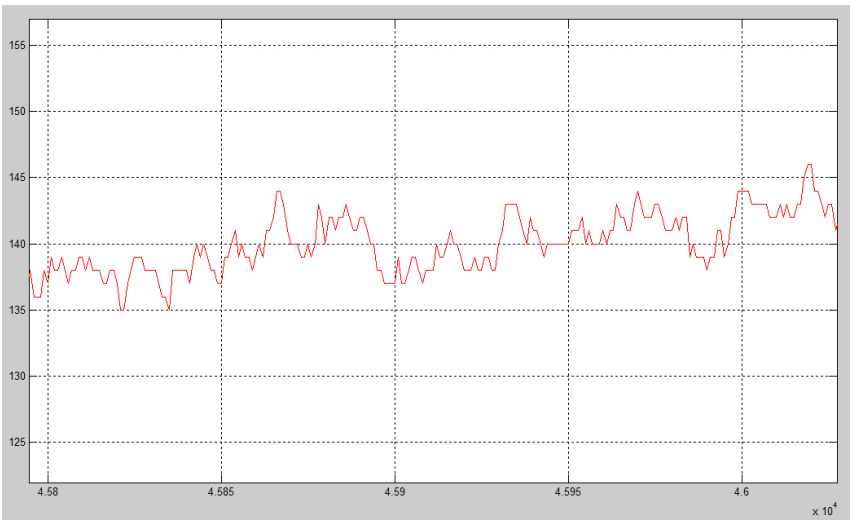

Figure 2: Detail of the algorithm behaviour

2.3 Multi-Start

Finally we tried to restart the tabu search from a diversification of the best solution when this one was not updated for a given number of steps, but the method did not provide any improvement. So we tried to reproduce the beneficial effect of the initial shorter tabu search flushing the current tabu lists when the best solution was not improved for a given number of iterations. We managed to reach an average score better than 1 point circa with respect to the original one, but we didn't reach any best result as good as 165 and the same happened combining the two actions. For this reason we did not considered it in the software data analysis in sections 3 and 4, although we decided to leave the choice to use or not, and to what extent, the multi-start mechanism to the user, by the parameter *Not Improving Steps Threshold*: it allows to customize the frequency of the multi-start and to completely disable it using the same value given for the parameter *Max Steps Threshold*.

3 Tuning

Maybe the hardest issue we faced during the whole development of our Eternity II solver was the tuning of the several parameters offered by the tabu search heuristics. Having ten parameters to set and practically an infinite set of possible values combinations, we adopted a greedy procedure to define in several refinement steps a group of parameters after the other. We left the threshold values for the maximum number of normal and not improving steps as parameters decided by the user, to best fit his execution time requirements. All the test were performed on the 10x10 instance of the problem, in order to optimize it for larger scale instances (as a 12x12 matrix).

The first parameters group we defined was made-up of the choice between the single or the triple tabu list implementation (see section 1) and the three neighborhood generation politics, one for each kind of piece. These politics are related to the choice of the first element of the couple involved in the swap (the second one is the element in the whole matrix that gives the higher number of matches after the swap, taking into account the tabu tenure and the acceptance criteria).

For the corner neighborhood the politics were: to swap always the first corner (by now called *First*), to jump from the first to the last one every step (*Opposite*) and to consider all the four corner tiles iteratively (*Next*). For the edge they consist in considering only the first and the last edge tile too (*Opposite*), in jumping to an edge for each side of the matrix in clockwise sense (*Next*) and in doing the same in anticlockwise sense (*Next Anti*). For the inner neighborhood, alongside the *Opposite* and the *Next* politics, were added the selection of the opposite piece of the second element involved in the previous swap (*Opposite Swap*) and a random generator (*Random*). These last two politics were aiming to enhance diversification in the biggest neighborhood of the three.

In the test we prepared, we chose a wide range of values internal to the domain of the remaining numerical parameters (the tabu list size for each kind of move and the expiration time) and set the value of steps threshold to 100000 (we noticed that in most cases the algorithm converged to the best solution before than this value, for info see 4) and the not improvement steps threshold, too (no multi-start, as said in section 2). We divided each range into 6 or 5 parts and took a representative value for each of them. Finally we tested all the possible combinations of politics and values on a random set of initial solutions, we grouped the results calculating the average score for each combination of *all* the eight parameters and selected the one with the highest average value.

The resulting politics were the *Opposite* for edges and the *Next* for the inner neighborhood, while for the corners we noticed that between the *First* and the *Next* politic there was no difference, maybe due to the small size of this neighborhood compared to the others, and so we arbitrarily selected the last one. As anticipated in section 1, the triple tabu list surpassed for each combination (without considering the ones with the same tabu lists size, of course) the single list implementation.

To set the remaining four parameters (tabu list sizes for corner, edge and inner moves

and the expiration time) we adopted a strategy similar to the one used in the previous test: we chose a wide interval for each parameter domain, we divided it into several parts and we took a representative value for each part. We tested the algorithm on a initial solutions set larger than before and then we calculated the average and the best score, alongside the variance, for every possible combination. After that, we selected the subranges with the best scores (absolute or average), taking into account also the variance previously calculated, and got them through a fine tuning test, that consists in dividing them in finer subranges and in increasing the size of the initial solutions set. Finally we selected the most promising combinations and we single tested them on an even larger instances set in order to provide statistical results as accurate as possible, that led us to the choice of the final values for the parameters.

The whole tuning process lasted more than 300 hours of computation, parallelized on our three personal multi-core laptops and led to the following set of parameters, marked as default values for algorithm (although they can be freely set by the user passing the whole set of parameters instead of only the threshold values).

- Tabu List Implementation: TRIPLE
- Corner Choice Politics: NEXT
- Edge Choice Politics: OPPOSITE
- Inner Choice Politics: NEXT
- Tabu List Size - Corner: 3
- Tabu List Size - Edge: 10
- Tabu List Size - Inner: 10
- Expiration Time: 30

4 Performance

The solver was written in C and tested on a personal computer mounting an Intel® Core™i7. A special setting was used during the compiling phase to speed up the program.
In order to collect a good number of data for statistics the program ran for 10000 times for each kind of matrix starting from a 05x05 one up to a 10x10 one.

4.1 Best and mean result

For each kind of problem a set of different data was gathered as shown in the summarizing table below.

Matrix	Abs Mean	Mean (%)	σ	Abs Max	Execution Time
05x05	36.455	91.137	0.770	38	1.620
06x06	54.280	90.466	1.044	57	2.852
07x07	75.037	83.330	1.306	79	4.442
08x08	97.666	87.202	2.101	104	6.280
09x09	124.739	86.624	1.893	131	8.406
10x10	156.018	86.677	4.219	165	10.898

Table 1: Summarizing table

The mean score per matrix was plotted with its standard deviation in order to understand how the dimension of the matrix modified the result.
The graph turned out to be different from the one we had imagined, in fact it had an unpredictable behaviour: we expected a decreasing progress, whereas we had a non monotonic one. Of course the parameters of the program were optimized for the 10x10 version. Obviously the standard deviation increased with the dimension.

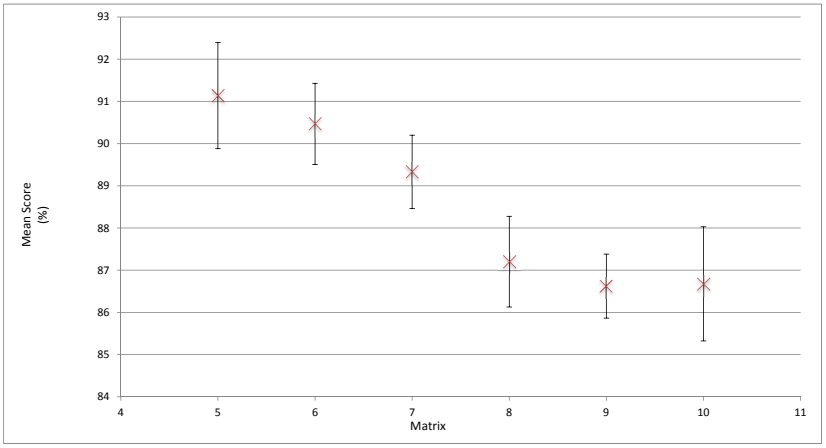

Figure 3: Mean score per matrix

4.2 Program convergence and step

An important issue for the program was its fast convergence as shown in the graph below. We can observe that the algorithm can converge in a small number of steps, but it reached an horizontal asymptote; after this the solution cannot be improved even if the number of iterations increases.

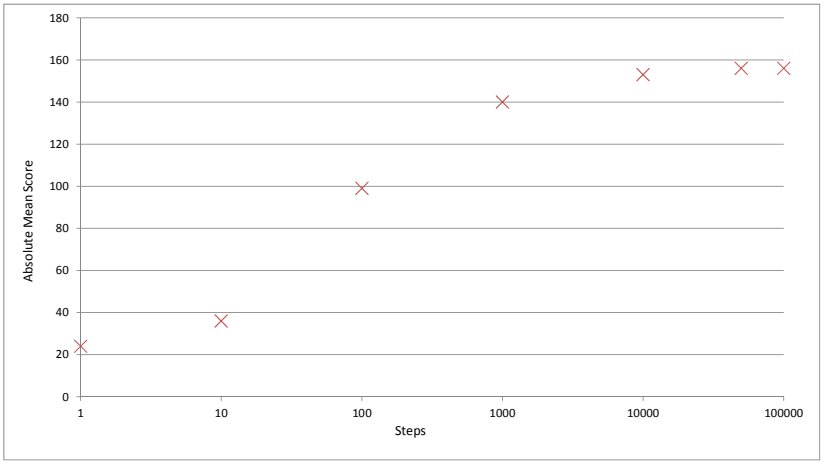

Figure 4: Program convergence per matrix

4.3 Time and memory allocation

The program was developed in such a way that it kept the memory utilization uniform.[1] It occupied the memory in a polynomial way with respect to the dimension of the matrix.
As shown in the following graph, the memory allocated seems to be like a parabola. It is trivial to understand the reason why this happens: the possible neighborhood generated increased quadratically.
Also the execution time seems to increase in the same way.

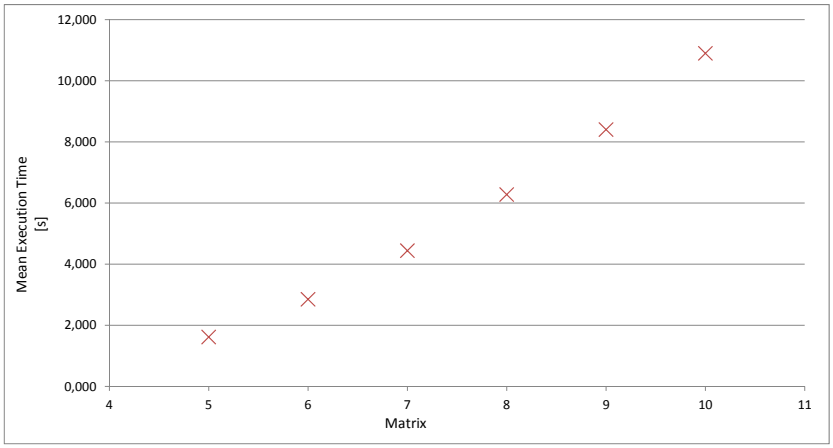

Figure 5: Program duration per matrix

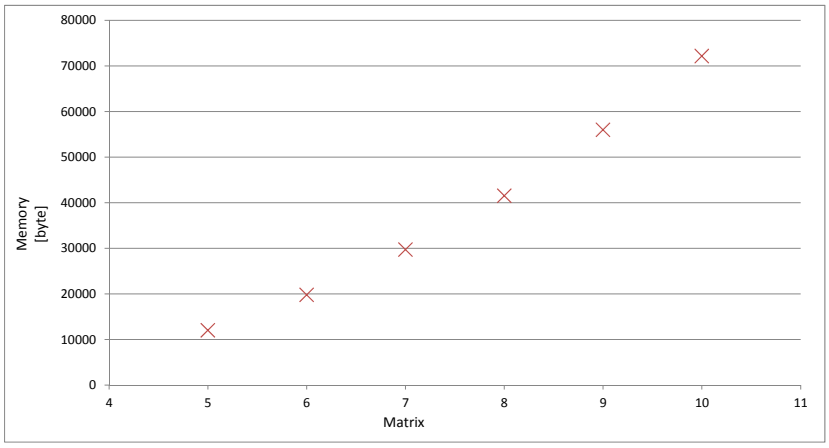

Figure 6: Memory allocation per matrix

We are proud of our results in terms of execution time and memory allocation, especially by the fact that the execution time for a 10x10 problem was around 11 seconds. Moreover, our best result was 165 over 180 in the 10x10 matrix. All the material and the source code of our implementation of the Eternity II solver can be found in a repository on the web at: http://code.google.com/p/eternity-tabu/

[1] Memory measurements were taken using Valgrind and Massif tools

5 Ideas for Future Works

The goal of our work was to implement an Eternity II solver based on Tabu Search and that is what we tried to do, although we are confident that realizing something more complex than a simple tabu search would certainly lead to serious improvements. The idea that during the software design phase intrigued us most was to implement an hybridization of our Tabu Search with a Variable Neighbor Search heuristics. This mixed approach may guide the Tabu Search to more promising areas of the solution space, in case the best solution was not updated for a lot of iterations. This can be achieved thanks to the iteratively increasing size of the VNS generated neighborhood, which consists in a chain of swap and rotate moves. Therefore, the VNS is used as an evolved version of a plain diversification of the solution. Another idea that may be worth a try is to find some greedy rules able to maximize the number of matches in a certain area of the matrix and to enforce them in the neighborhood generation phase.

It would also be interesting to further investigate the tuning phase of the Tabu Search heuristics, maybe applying an heuristic method in order to face the huge amount of computation time required by this issue. On the base of our limited but intense experience, we are quite sure that it would be at least as engaging as solving the Eternity II problem, or even more.

A Pseudocode

Algorithm 1 Tabu Search Pseudocode

```
startSolution ← feasible solution
currSolution ← rand(start,seed)
Perform a shorter version of tabu search(currSolution)

tabuListCorner ← tabuListInit(CornerSize,ExpireTime)
tabuListEdge ← tabuListInit(EdgeSize,ExpireTime)
tabuListInner ← tabuListInit(InnerSize,ExpireTime)
bestSolution ← currSolution

while currStep<maxStepsThreshold do

  if moveIsTabu==false then
    if currStep%3==0 then
      neighborhood ← neighborhoodGenerationCorner(currSolution)
      tabuList ← tabuListCorner
    else if currStep%3==1 then
      neighborhood ← neighborhoodGenerationEdge(currSolution)
      tabuList ← tabuListEdge
    else
      neighborhood ← neighborhoodGenerationInner(currSolution)
      tabuList ← tabuListInner
    end if
  end if

  candidate ← bestNeighbor(neighborhood)
  moveIsTabu ← false

  checkMove(neighbor,tabuList)
  if move is tabu and aspiration criteria is not satisfied then
    moveIsTabu ← true
    discard(candidate)
  else
    currSolution ← neighborIntoSolution(candidate)

    if currSolution > bestSolution then
      bestSolution ← currSolution
    else

      noImp++
      if noImp==noImprovementThreshold then
        noImp = 0
        diversification(currSolution)
        tabuListFlush(tabuListCorner,tabuListEdge,tabuListInner)
      end if

    end if
    currStep++
  end if

end while
```