A Tabu Search approach for the Circle Packing Problem

Carrabs Francesco
Department of Mathematics
University of Salerno
Salerno, Italy
Email: fcarrabs@unisa.it

Carmine Cerrone
Department of Mathematics
University of Salerno
Salerno, Italy
Email: ccerrone@unisa.it

Raffaele Cerulli
Department of Mathematics
University of Salerno
Salerno, Italy
Email: raffaele@unisa.it

***Abstract*—This paper concerns the problem to place N non overlapping circles in a circular container with minimum radius. This is a well known and widely studied problem with applications in manufacturing and logistics and, in particular, to problems related to cutting and packing. In this paper we propose an algorithm that by applying a strength along a selected direction on each circle, simulates the shifting of circles on the plane and tries to reduce the radius of the circular container during this movements. The algorithm is based on a multistart technique where the starting solutions are produced by a tabu search heuristic that uses also the current best solution. The algorithm takes part in a public international contest in order to find optimal solutions to a special case in circle packing. The contest saw the participation of 155 teams and our algorithm achieved the tenth position.**

***Keywords*-Circle Packing, Tabu Search, MultiStart**

I. Introduction

The problem we study in this paper has important applications in activity related to cutting and packing, wireless communication networks, container loading, manufacturing, logistics, see [1] for a recent survey. In this context the most widely studied case is the packing of equal circles in the unit square. Packing different disks in a smallest container is a NP-complete combinatorial optimization problem and are been developed for it exact approaches, like branch-and-bound algorithm [3], and different heuristics and metaheuristics procedure [2][4][5].
The particular problem we address is the problem of packing n different circles (disk) in a circumference with minimum radius. We suppose that each disk has a different radius and we have to put them in a circumference of minimum radius. For this particular problem we propose several algorithms that can be classified in three main categories: constructive algorithms, shrinking algorithms and improvement algorithms. Each kind of algorithm is developed for a specific aim and we embedded these algorithms in a tabu search framework in order to obtain an effective algorithm for the circle packing problem.

In this paper we propose several algorithms that can be classified in three main categories: constructive algorithms, shrinking algorithms and improvement algorithms. Each kind of algorithm is developed for a specific aim and we embedded these algorithms in a tabu search framework in order to obtain an effective algorithm for the circle packing problem. The remainder of the paper is organized as follows. Section II introduces definitions and notations that are used in the paper. In sections III-V the algorithms of the three categories above mentioned are introduced and in section VI our tabu search is described. Finally, the computational results are presented in Section VII and some concluding remarks are given in Section VIII.

II. Definitions and Notations

Given a set $N = \{c_1, c_2, \ldots, c_n\}$ of circles, a circumference C is a *feasible solution* if it contains all the n circles of N without overlapping. The circle packing problem consists of finding a feasible circumference C with minimum radius. Let us define the function $r : N \to \mathbb{R}$ that, given a circle in input, returns the length of its radius. Moreover, let $r(C)$ be the radius length of circumference C. The position of each circle is identified by the coordinates of its center. Moreover, a position in C is *available*, for a circle c_i, if this circle can be placed in this position without crossing the perimeter of the circumference and overlapping other circles. Finally, let us define the *starting sequence* $S = \{c_{s_1}, \ldots, c_{s_n}\}$ as any permutation of the n circles of N.

III. The Constructive Algorithms

The constructive algorithms are used to quickly generate feasible solutions. In this section we introduce two constructive algorithms: the Greedy and the Spiral. The Greedy builds a circumference, step by step, by adding at each iteration a new circle that touches at least one of the circles already inserted. Among all the available positions, this algorithm selects the best in that moment, i.e. the position that minimizes the growing of circumference radius. The Spiral algorithm starts from a fixed circumference and tries to place circles, first along the C perimeter and, when it is not possible anymore, inside the circumference by invoking the Greedy algorithm. In the next subsections further details about these two algorithms are given.

A. The Greedy Algorithm

The first replacement strategy is a greedy algorithm that tries to build a circumference as small as possible by

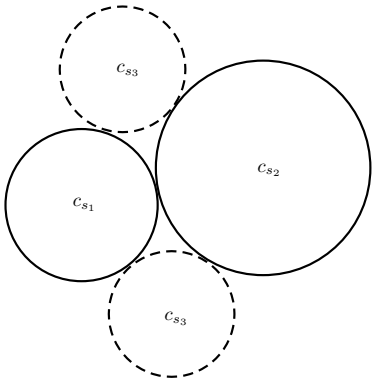

Figure 1. Greedy Algorithm example. There are two available positions where the circle c_{s_3} can be placed. The algorithm will select the position that minimizes the radius of circumference containing the three circles.

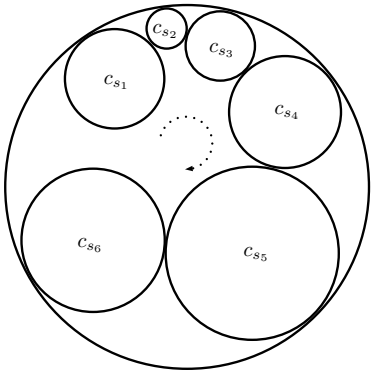

Figure 2. Spiral Algorithm. The first six cicles are placed along the circumference. The remaining circles have to be placed on the other available positions.

placing the circles in the best position on the plane. At the first iteration, the algorithm selects the first circle c_{s_1} of the starting sequence S and the current circumference C coincides with c_{s_1}, i.e. $r(C) = r(c_{s_1})$. At each iteration i, the algorithm selects the i-th circle c_{s_i} and places it in $C = \{c_{s_1}, \ldots, c_{s_{i-1}}\}$ assuring that c_{s_i} touches at least one of the circles already placed. Among all the available positions the algorithm selects the best one which is the position that increases as less as possibile the C radius. Figure 1 shows how the Greedy algorithm works.

The circles c_{s_1} and c_{s_2} are already placed and have a single contact point. There are two available positions where to place c_{s_3}. The former is over c_{s_1} and c_{s_2} and the latter under these two circles. Notice that every time a new circle is placed the number of available positions increases by one. For instance, let us suppose to place c_{s_3} over c_{s_1} and c_{s_2} so that an available position is lost but two new positions are generated: one between c_{s_1} and c_{s_3} and another one between c_{s_2} and c_{s_3}.

During the development of the algorithm we discovered that better results can be obtained by placing the smallest circles at the end of the procedure. Indeed, most of the time, the insertion of a large circle at the end of the procedure increases $r(C)$. On the contrary, by inserting first the biggest circles and later the smallest ones, there will be more chances to find available positions where the smallest circles can be placed without increasing $r(C)$. For this reason, we fix a threshold equal to 20% of n and when the algorithm reads the starting sequence S, it jump all the circles with a radius under this threshold to place them at the end of the procedure.

B. The Spiral Strategy

This strategy takes in input the starting sequence S and an empty circumference C with a fixed radius R and tries to insert all the circles in C. In this case, it is forbidden to change the radius R of C. At each iteration the algorithm places in C the current circle c_{s_i} assuring that this circle touches both the circumference and the circle $c_{s_{i-1}}$ (Figure 2).

If the radius R is enough large, all the circles will be placed along the circumference C and a lot of free space is wasted particularly in the center of C. For instance, the result of the Spiral strategy is shown in Figure 2 when $n = 6$. It is evident that the chosen radius R is too large and a lot of space is wasted in the center of circumference but also between c_{s_4} and c_{s_5}, c_{s_5} and c_{s_6}, c_{s_6} and c_{s_1}. In these positions could be possible to place c_{s_2} and c_{s_3} to obtain a smaller circumference. Since our aim is to find the smallest circumference, the value of R is usually small and the case described above rarely occurs. A more realistic case happens when the radius R is so small that not all the circles can be placed along the perimeter of circumference. In this case, the Spiral algorithm invokes a modified version of the Greedy algorithm, described in section III-A, to place the remaining circles in C.

Since R cannot be changed, the modified Greedy algorithm chooses the best position for a circle c_{s_i} by evaluating the distance between its center and the center of C. The greater this distance is the better the position chosen is. With this polity, we try to place the circles in the cavities between the perimeter of circumference and the circles already placed, leaving, in this way, freer space in the center of C where to place the remaining largest circles. Since the results of this algorithm depends on the starting sequence S and the radius R, it can happen that not all the circles are placed in C. In this case, the algorithm fails and the partial solution produced is rejected.

IV. The Shrinking Algorithm

Given a starting sequence S and a radius R, the Shrinking algorithm tries to generate a new circumference as small as possible and with a radius lower than R. The idea behind

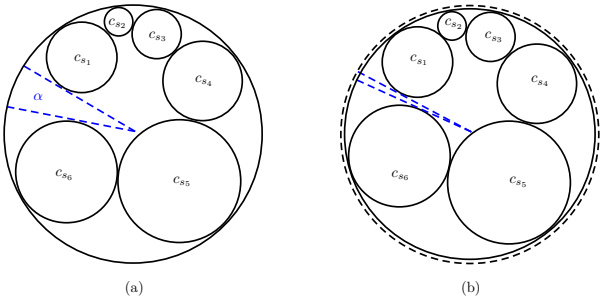

Figure 3. Shrinking Algorithm. (a) The angle α obtained after the positioning of circles along the circumference. If this angle is enough width, there are good chances to shrink the current circumference leaving all the circles in it. (b) The new better circumference obtained after the shrinking. The dotted circumference is the original one.

this algorithm is the following. Let us suppose to use the Spiral algorithm to place the circles along the perimeter of C, with radius R. If there is not enough space to do that, the remaining circles are placed in the available positions in C. It is easy to see that between the first and last circle placed along C there is an angle α (Figure 3(a)). The greater this angle is the greater the chances to "shrink" C are by leaving all the circles in it. In order to obtain a better circumference, the Shrinking algorithm reduces by 2% the current radius R and tries to insert all the circles in the new circumference by using the Spiral strategy (Figure 3(b)). This process is repeated until the Spiral strategy does not reach to insert all the circles in the smaller circumference.

Let us see in more details all the steps of the Shrinking algorithm. Given the starting sequence S and the radius R, in the first step the Shrinking algorithm invokes the Spiral algorithm with the previous parameters. There can be two possible cases to consider:

1) The Spiral algorithm does not find a circumference with radius R.
   Since no circumferences to improve are available, the Shrinking algorithm stops.
2) The Spiral algorithm returns a circumference with radius R.
   In this case the Shrinking algorithm define two parameters: $rMax = R$ and $rMin = R - (R * 2)/100$. In order to improve the current circumference, the Shrinking algorithm carried out a binary research within the range $[rMin, rMax]$ and, for each value selected $R' \in [rMin, rMax]$, it invokes the Spiral algorithm with the parameters S and R'. If a new better solution is found, the Shrinking algorithm restarts from the new radius R' and the same starting sequence S and so on until no more improvements are carried out or when the difference rMax-rMin is lower than a tolerance equal to $\epsilon = 10^{-5}$.

The Algorithm 1 shows the pseudocode of the Shrinking Algorithm.

Algorithm 1: Shrinking

Input: starting sequence S and the radius R;
Output: Circumference $\hat{C}$ or null;

```
1  C ← Spiral(S, R), Ĉ ← null;
2  if C = null then
3      return null;
4  rMax ← R, rMin ← R − R ∗ 2/100;
5  while rMin ≤ rMax do
6      R' ← (rMax − rMin)/2;
7      C ← Spiral(S, R');
8      if C ≠ null then
9          Ĉ ← C;
10         rMax ← R', rMin ← R' − R' ∗ 2/100;
11     else
12         rMin ← R';
13 return Ĉ;
```

V. The Improvement Strategies

The Greedy and the Spiral algorithms are able to quickly generate feasible solutions, for the circle packing problem, while the Shrinking algorithm requires more computational time but it often produces better solutions than the Greedy and the Spiral algorithms. However, in order to find solutions very close or equal to the optimal one, it is necessary to provide new strategies able to relocate the circles in the current circumference when it is very small and the movements are very hard to carry out. To this end, in the next two sections we will introduce two improvement strategies that are able to heavily improve the solutions generated by constructive algorithms.

A. The Bounce Strategy

The Bounce strategy is based on a physical simulation on circles movement in the circumference. We developed two versions of this strategy that will be used in two different contests. The only difference between these versions regards the starting feasible solution. In the first version, the strategy builds a starting solution by placing, at random, all the circles on the plane as a grid, and creating a circumference that contains all of them (Figure 4(1)). On the constrary, the second version takes in input a starting feasible solution. The other steps are the same for both versions.

The first version can be used as a constructive algorithm able to generate good solutions. The latter version is used to improve the good solutions obtained by Shrinking algorithm. Given a feasible solution C, the strategy applies on each circle the same force but along a direction chosen at random. Thanks to this force, the circles move in the circumference and every time that two circles collide they "bounce back" changing their direction and the same thing occurs when a circle collides with the circumference. The simultaneous movement of all the circles in different directions can create

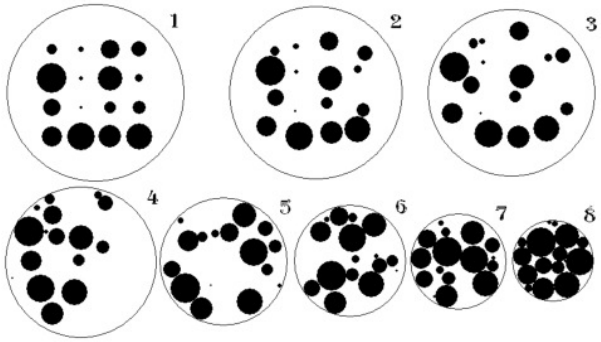

Figure 4. Bounce strategy example

new spaces where to relocate, essentially, the small and medium circles.

During this simulation, the strategy constantly tries to reduce the radius $r(C)$. This last operation can be carried out if, during the simulation, there is a moment in which all the circles do not touch the circumference. However, it is evident that this condition rarely occurs, in particular when the radius is already small. In order to solve this problem and obtain better circumferences, we create an "ideal" circumference C' with the same center as C and a radius $r(C')$ that is 0.1% smaller than $r(C)$. We force the circles in C' to remain in this last circumference. This means that if one of these circles touch C', during the simulation, it bounces back. On the contrary, the circles that are on the perimeter of C' or between C and C' move neglecting C' until they are not entirely placed in this last circumference. When this happens, we trap them in C'. During the simulation, if all the circles are trapped in C', we obtain a new feasible solution better than C. The algorithm stops if no circles are trapped into the ideal circumference C' within the time limit of 0.1 seconds.

In Figure 4 an example of how the Bounce Strategy works is shown. At the beginning all the circles are placed like in a grid (Figure 4(1)). Step by step the circumference is shrinked until it reaches the final dimension (Figure 4(8)) where the most of circles are blocked and no more improvements are possible.

The Bounce Strategy is slower than the constructive algorithms but, giving it enough time, it is able to find very good solutions. Moreover, we further improved the effectiveness of this strategy by this observation. When the circumference becomes too small, groups of circles, completely blocked, are formed in it. Since these circles remain blocked forever, they prevent a possible improvement of the current solution. We partially solve this standing off through a "slap" that consist of set, for a very short time, the same movement direction for all the circles. Obviously, this is a new type of movement for the circles, in respect to the movements generated by bounces and it simulate a

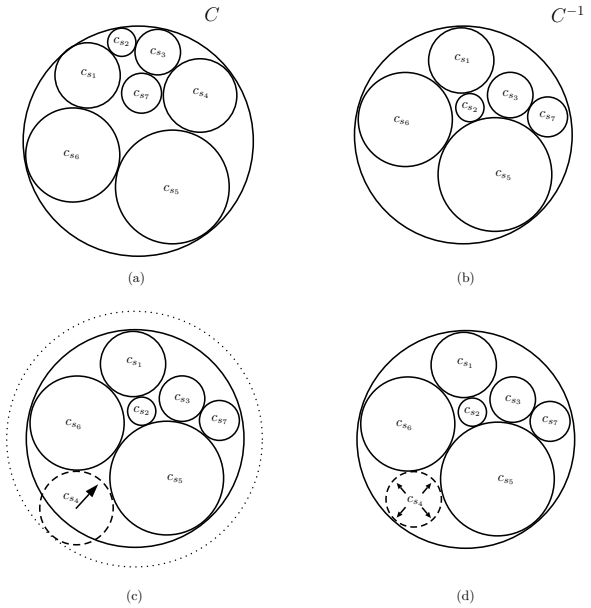

Figure 5. Blocked Circles strategy. (a) The current feasible circumference C. The circle $c_k = c_{s_4}$ is removed from C. (b) The resulting smaller circumference C^{-1} after the shrinking phase. (c) Perimetric Insertion policy. (d) Growing Insertion policy.

slap given to the circumference. Sometimes this operation can unblock at least a part of the blocked circles. However, a more effective strategy to tackle this problem is proposed in the next section.

B. Blocked Circles Strategy

By analyzing the simulation carried out by the Bounce Strategy, it is evident that the smaller is $r(C)$ the smaller are the movements carried out by circles in the circumference. In particular, we find out that several circles never change their position (or they change it but within a tolerance ϵ) from a certain point onward. We call these circles "blocked" circles, and due to their presence the improvement of C becomes hard to carry out or impossible when these circles are located along the circumference. For this reason, we tackle this problem by introducing a second improving strategy named *Blocked Circles*. The idea behind this strategy is to identify the blocked circles in the current circumference C and remove one of them, let us say c_k. In this way, we increase the free space in the resulting infeasible circumference C^{-1} and make the relocation of the remaining $n-1$ circles in it easier. Thanks to the free space left by removing c_k, we often reduce the radius of C^{-1} easily because this circumference has to contain one less circle (Figure 5(b)).

However, we are interested only in feasible solutions and then we have to develop some reinsertion policies to put c_k back in C^{-1}. The aim of the reinsertion policies is to generate a new circumference $C' = C^{-1} \cup \{c_k\}$ so that $r(C') < r(C)$. The two key aspects of the Blocked Circles

strategy are the selection of the circle c_k to remove and the reinsertion policies of c_k.

Regarding the selection of c_k, we identify the set B of blocked circles in C and introduce the smallest circles of B, in a candidates list ℓ. The size of ℓ is equal to 20% of $|B|$. The circle c_k will be selected at random from ℓ. We focus our attention on the smallest circles of B because on one hand our aim is to unblock the situation in C and, on the other hand, to reinsert c_k so that a new better feasible circumference C' is obtained. Obviously, the removal of large blocked circles assures freer space in C^{-1} and then it is easier to reduce its radius. However, it is very hard to reinsert these large circles in C^{-1} and obtain a new feasible solution C' with $r(C') < r(C)$.

After the construction of C^{-1}, the next step consists of defining the policies to use to reinsert the circle c_k in this circumference and obtain a better solution, when possible. To this end, we propose three different reinsertion policies. These policies are applied in sequence and the next one is invoked only if the current one does not produce a solution better than C. Let us see in detail how these policies work.

- **Direct Insertion**
  This is the easiest reinsertion policy. In practice, it checks if there is enough space in C^{-1} to reinsert the circle c_k. Since we use the free space generated by removing c_k, to shrink C^{-1} we will rarely find enough space in this circumference to directly insert c_k. Anyway, if c_k is enough small and $r(C^{-1})$ is enough large, this policy could be successful. In Figure 5(b) it is evident that there are no available places where directly reinsert the circle c_{s_4}. For this reason, we have to apply the next policy.
- **Perimetric Insertion**
  This policy places the circle c_k along the perimeter of C^{-1} assuring that the distance between the center of circumference and c_k is the minimal one. Obviously, whatever is the selected position, c_k cannot be entirely in C^{-1} because otherwise c_k would have been already inserted in C^{-1} by the previous Direct Insertion policy. The Figure 5(c) shows the situation after the placement of c_k along the perimeter of C^{-1}. In order to generate a feasible solution C' with $r(C') = r(C^{-1})$, we try to "pull" the circle c_k towards the center of the circumference assigning it that movement direction. At this point, we invoke the Bounce strategy where the perimeter of C^{-1} represents the ideal circumference in which the circle c_k have to be inserted. If Bounce fails then the infeasible solution is rejected otherwise a new feasible solution C' with $r(C') = r(C^{-1})$ is obtained.
- **Growing Insertion**
  The Growing Insertion policy finds in C^{-1} the largest position where to place the circle c_k without crossing the perimeter of the circumference. Obviously, we know that there is not enough space in C^{-1} to directly place c_k and then we insert in this position a smaller version of this circle (Figure 5(d)). However, to obtain a feasible solution we need to restore the correct size of c_k in C^{-1}. To this end, we invoke the Bounce strategy and, during the simulation, the policy tries to expand c_k within the limit of C^{-1}. As soon as the size of c_k is restored, a new feasible solution C' better than C is obtained. The policy stops after 10 iterations of the Bounce strategy without expansion of c_k.

VI. Tabu Search algorithm

The algorithms described in previous sections build their feasible solutions according to the starting sequence S and it is evident how this sequence heavily affect the solutions quality found. For this reason, our tabu search heuristic will use several starting sequences in order to widely explore the space solutions and makes more stable the final results. To this end, our heuristic uses a multistart strategy to generate several starting sequences. Moreover, after the individuation of the best solution for S, the tabu search selects a new starting sequence in the neighborhoods of S generated by the 2-Opt and 3-Opt operators.

The Algorithm 2 shows the pseudocode of our tabu search heuristic. The initialization step is carried out on the line 1. Here we fix to $\lceil n/10 \rceil$ the number of iterations carried by multistart strategy. The while loop (line 2) implements the multistart strategy in which a starting sequence S is generated at random and the radius $\hat{R}$ of the best circumference found so far is retrieved by invoking the `getBestSol` function. If there are not circumferences available yet, this function returns the radius $\hat{R} \leftarrow \frac{1}{2}\sum_{i \in N} r(i)$ of the trivial circumference obtained by placing on a line all the n circles side by side. At line 5 the initial radius R is set to the value of $\hat{R}$ increased by 5%.

The while loop of the line 6 is the core of tabu search heuristic. This loop is repeated until n minutes, without improvements of $\hat{C}$, have gone. The first operation of this loop is to invoke the Shrinking algorithm on the sequence S and the circumference with radius R. The resulting circumference C is further improved by invoking on it the two improvement strategies (line 8) described in section V. If the new circumference C is better than the incumbent one $\hat{C}$, then $\hat{C}$ is update and the `setBestSol` function is invoked. The application of the `setBestSol` function regards the parallelization of the algorithm and we will give more details about this in Section VII. The next step consist of finding the best solution in the neighborhood of S by invoking the `NextMove` function. The best solution found by `NextMove` is put into the tabu list. In our implementation the size of the tabu list is equal to $\lceil n/2 \rceil$.

The first step of `NextMove` function is to set $R' \leftarrow \frac{1}{2}\sum_{i \in N} r(i)$. At the end of this procedure, R' will represent the radius of the best circumference found. The `NextMove` is composed by the 2-opt and 3-opt operators that are used to

Algorithm 2: Tabu Search

Input: The set N;
Output: Circumference, as small as possible, containing all the circles of N;

1 $ms \leftarrow \lceil n/10 \rceil$;
2 **while** $ms \geq 0$ **do**
3 $ms \leftarrow ms - 1$;
4 Generate the starting sequence S and sets $\hat{R} \leftarrow$ `getBestSol()`;
5 $R \leftarrow \hat{R} + \hat{R} * 5/100$;
6 **while** *stopCriterion=false* **do**
7 $C \leftarrow Shrinking(S, R)$;
8 $C \leftarrow Improvements(C)$;
9 **if** $r(C) < r(\hat{C})$ **then**
10 $\hat{C} \leftarrow C$ and $\hat{R} \leftarrow r(C)$;
11 `setBestSol`$(\hat{C})$;
12 $(S, R) \leftarrow$ `NextMove(`*S,r(C)*`)`;
13 Save the move into the tabu list;
14 **return** $\hat{C}$;
15 **Function** `NextMove(`*S,R*`)`
16 $R' \leftarrow \frac{1}{2} \sum_{i \in N} r(i)$;
17 **forall the** $S_i \in 2 - Opt(S)$ **do**
18 **if** $S_i \notin TabuList$ **then**
19 $C \leftarrow Shrinking(S_i, R')$;
20 $C \leftarrow Improvements(C)$;
21 **if** $r(C) < R'$ **then**
22 $R' \leftarrow r(C)$, $S' \leftarrow S_i$;
23 **if** $r(C) < R$ **then**
24 $R' \leftarrow r(C)$, $S' \leftarrow S_i$;
25 break;
26 **forall the** $S_i \in 3 - Opt(S)$ **do**
27 **if** $S_i \notin TabuList$ **then**
28 $C \leftarrow Shrinking(S_i, R')$;
29 $C \leftarrow Improvements(C)$;
30 **if** $r(C) < R'$ **then**
31 $R' \leftarrow r(C)$, $S' \leftarrow S_i$;
32 **if** $r(C) < R$ **then**
33 $R' \leftarrow r(C)$, $S' \leftarrow S_i$;
34 break;
35 **return** (S', R');

generate the neighborhoods of the starting sequence S given in input. The 2-opt operator selects two circles of S and swap their positions. The Figure 6(a) shows how the 2-opt operator works. The circles selected are c_{s_2} and c_{s_4}. After the swap operation, we have the new starting sequence, on the right. The 3-opt operator selects three circles and carries out a circular right shifting on them. The Figure 6(b) shows how 3-opt operator works. The circles c_{s_2}, c_{s_4} and c_{s_5} are selected. The circular right shifting moves c_{s_5} in place of c_{s_2}, c_{s_2} in place of c_{s_4} and c_{s_4} in place of c_{s_5}. The other circles remain on their positions.

At the line 17 all the possible starting sequences derivable from S by the 2-Opt operation are generated. If the new sequence S_i is not tabu then the Shrinking algorithm is invoked on it and, later, the improving strategies are applied

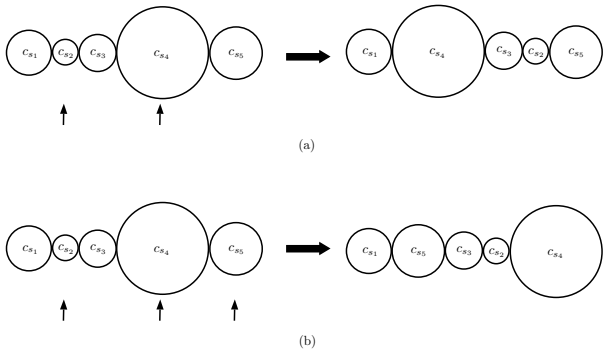

Figure 6. (a) The 2-Opt opetator and (b) the 3-opt operator.

on the resulting circumference C. If $r(C) < R'$ then we update the best radius R' found so far and the sequence S_i that generated it. If same operations are carried out if $r(C) < R$. In this last case, we stop (line 25) the generation of other sequences by 2-opt operator to speed up the procedure and because the solution found improves the incumbent one. The same operations are carried out in the for loop of lines 26-34. The only difference here is that the starting sequences are generated by using the 3-opt operator. At the end, the `NextMove` returns the best solution found. Notice that if the conditions of line 23 and 32 never hold, the solution returned is worst than the solution given in input. By accepting worst solutions, we allow tabu search to escape from local minimums.

VII. Computational Results

All the algorithms described in this paper, were developed during an international contest (http://www.recmath.org/contest/CirclePacking/index.php) based on the circle packing problem. The challenge was to pack N non-overlapping discs with radii from 1 to N into as small a circle as possible. 155 competitors from around the world, have posted the best solutions produced for 45 instances containing from 5 to 50 circles.

In the table VII we report our results. The first column n represents the number of circles in input while the second column reports the best known solution. The third and fourth column reports the solution of our tabu search and the gap (in percentage) from the best known solution, respectively. Our results are in bold when our solution coincides with the best known solution. The eventual mismatch on the last digit of the solutions depends on tolerance used by algorithms of the site that evaluate the solutions proposed. Finally, the last column shows what is the algorithm in our tabu search that found our final solution. The smallest two instances were so easy that the best known solution was found by Greedy algorithm while the next10 instances could be solved by using only the Shrinking algorithm. Starting from the instance with $n = 17$, it is necessary to use the whole

n	Best Solution	Our Solution	Gap %	Algorithm
5	9.00139775	**9.00139774**	–	Greedy
6	11.05704040	**11.05704039**	–	Greedy
7	13.46211068	**13.46211067**	–	Shrinking
8	16.22174668	**16.22174667**	–	Shrinking
9	19.23319391	**19.23319390**	–	Shrinking
10	22.00019301	**22.00019301**	–	Shrinking
11	24.96063429	**24.96063428**	–	Shrinking
12	28.37138944	**28.37138943**	–	Shrinking
13	31.54586702	**31.54586701**	–	Shrinking
14	35.09564714	35.09566220	0.0	Shrinking
15	38.83799551	**38.83799550**	–	Shrinking
16	42.45811644	**42.45811643**	–	Shrinking
17	46.29134212	**46.29134211**	–	Tabu search
18	50.11976262	50.11977669	0.0	Tabu search
19	54.24029359	54.24033451	0.0	Tabu search
20	58.40056748	58.43294249	0.1	Tabu search
21	62.55887709	62.56009885	0.0	Tabu search
22	66.76028624	66.76032481	0.0	Tabu search
23	71.19946161	71.30052907	0.1	Tabu search
24	75.75270412	75.86175714	0.1	Tabu search
25	80.28586444	80.30468209	0.0	Tabu search
26	85.07640122	85.30345567	0.3	Tabu search
27	89.79218157	89.91645449	0.1	Tabu search
28	94.54998647	95.00402128	0.5	Tabu search
29	99.51231790	99.93118117	0.4	Tabu search
30	104.57855509	104.82125659	0.2	Tabu search
31	109.77194699	110.21549729	0.4	Tabu search
32	114.86543833	115.67839664	0.7	Tabu search
33	120.21695715	120.88145711	0.6	Tabu search
34	125.43350176	126.14675162	0.6	Tabu search
35	131.15635463	131.73294513	0.4	Tabu search
36	136.53490083	137.47533684	0.7	Tabu search
37	142.17498054	142.94453249	0.5	Tabu search
38	147.85769136	148.46830833	0.4	Tabu search
39	153.55530120	154.44087433	0.6	Tabu search
40	159.48902487	160.35050100	0.5	Tabu search
41	165.29190969	166.23178297	0.6	Tabu search
42	170.92576162	172.34970548	0.8	Tabu search
43	177.07434007	178.37606878	0.7	Tabu search
44	183.17606157	184.75709672	0.9	Tabu search
45	189.63543911	190.80324992	0.6	Tabu search
46	195.91076340	197.18468348	0.7	Tabu search
47	202.18561174	203.56945416	0.7	Tabu search
48	208.63594673	209.74976500	0.5	Tabu search
49	214.66195201	216.25530807	0.7	Tabu search
50	221.08975259	222.97536695	0.9	Tabu search

Table I
DATA EXTRACTED FROM: AL ZIMMERMANN'S PROGRAMMING CONTESTS

tabu search to find good solutions. The results of the Gap column prove the effectiveness of our heuristic because the gap between our solutions and the best known is always lower than 1%. Moreover, the algorithms of the contest have ranked our solutions set tenth on 155 competitors.

The computational tests were carried out by running our algorithm for a week, in parallel mode, on 20 desktop computers, with a single-core 2.4GHz processor. The functions `getBestSol` and `setBestSol` were designed to communicate with a central repository to share the best solution R' to the 20 instances of the algorithm running on different PC.

VIII. CONCLUSION

In this paper we discussed a tabu search approach for the problem of packing unequal circles into a circular container with minimum radius. Our heuristic uses several algorithms and strategies that we developed to tackle the problems that arise during the shrinking of the circumference. We proposed the constructive and the Shrinking algorithms to quickly generate feasible solutions but the computational results shown that these algorithms are very effective on the small instances, up to n=16, where they find the best known solution, most of time. Moreover, we introduced some improvements strategies that, by simulating the movements of circles in the circumference, when they bounce with each other, try to free some space in the circumference to allow a reorganization of the circles and to produce better solutions.

The computational results prove the effectiveness of our heuristic with 12 instances solved in the best way, a gap on the other instances always lower than 1% and the tenth position gained in the final classification.

REFERENCES

[1] I. Castillo, F.J. Kampas, J.D. Pinter, *Solving circle packing problems by global optimization: Numerical results and industrial applications*, 191, Issue 3, 786802, 2008.

[2] P.G. Szab, M. C. Markt, T. Csendes, *Global optimization in geometrycircle packing into the square*, in: C. Audet, P. Hansen, G. Savard (Eds.), Essays and Surveys in Global Optimization, Kluwer, Dordrecht, 233266, 2005.

[3] M. Locatelli, U. Raber, *Packing equal circles in a square: a deterministic global optimization approach*, Discrete Appl. Math. 122, 39166, 2002.

[4] B. Addisa, M. Locatellib, F. Schoen, *Efficiently packing unequal disks in a circle*, Operations Research Letters 36, 3742, 2008.

[5] A. Grosso, A.R.M.J.U. Jamali,M. Locatelli,F. Schoen *Solving the problem of packing equal and unequal circles in a circular container*, J. Global Optimization, 47, 6381, 2010.