Mathematical and Computational Applications, Vol. 16, No. 1, pp. 279-289, 2011.
© Association for Scientific Research

AN EXTENDED HEURISTIC ALGORITHM TO SETTLE REACTING OBJECTS ON A PLANAR SURFACE

Gorkem Tokatli[1], Pinar Dundar[2], Moharram Challenger[1, 3], Tufan Turaci[4]
[1]Ege University, International Computer Institute, Izmir, Turkey
[2]Ege University, Mathematics Department, Applied Mathematics, Izmir, Turkey
[3] Islamic Azad University, Shabestar Branch, Iran
[4] Ege University, Mathematics Department, Computer Science, Izmir, Turkey
gorkem.tokatli@ege.edu.tr, pinar.dundar@ege.edu.tr, challenger@engineer.com, tufanturaci@gmail.com

Abstract- Graph theory is a key subject for both mathematics and computer science. It is used for modelling many problems such as maximal independent set, minimum covering and matching. In our study, we have extended the previous work on placing materials that may react with each other on a 2-D warehouse. We have modelled the problem using graph theory. Then, we have developed extensions on the heuristic algorithm which is using Paull-Unger method that finds Maximal Independent Sets. First two of these extensions include finding solutions with gaps for specific graphs, and meanwhile capability of performing replacement in any desired rectangle surface. The last and most effective extension is pruning unnecessary backtracking steps with the help of smarter heuristics in the algorithm.

Key Words- Graph Theory, Independent Set Problem, Storage Problem, Heuristic Algorithms.

1. INTRODUCTION

Placing the chemically reacting and interfering objects in an appropriate way is an important issue for many sectors. Finding a compact placement solution can be hard for materials that mostly react with others. We have modelled this problem with the graph data model. In this model, materials are mapped to vertices and the reaction interferences are mapped to edges. A graph $G = (V(G), E(G))$ is composed of non-empty $V(G)$ set of vertices and $E(G)$ set of edges that connects unordered pair of vertices.

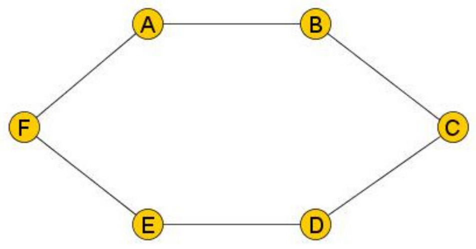
In a graph G, if there is an edge between vertices u and v, this shows that these two materials react with each other. In a graph G, if there is always a path between any pair of vertices, then this graph is called *Connected Graph* [1, 2]. Let $S \subseteq V(G)$ be a subset of a set of vertices in a graph G. For any pair of vertices, if there is no edge existing between them, then this subset is called *Independent Set* [1, 2]. If an independent set is not a subset of any other independent sets, then this set is called *Maximal Independent Set* [1, 3, 4]. A graph G may have more than one maximal independent sets. The number of elements in the independent set that has the biggest number of vertices, is called *Independence Number*. It is denoted by $\beta(G)$ [1, 2].

Definition 1.1. The adjacency matrix of a *n-vertices* $G = (V(G), E(G))$ graph is shown as $A(G)$. This matrix is of $n \times n$ structure, and the vertices forms the rows and columns of the matrix.

The elements of a *A(G)* matrix is defined as below [4].

$$a_{ij} = \begin{cases} 1 \;, if\; v_i v_j \in E(G) \\ 0 \;, if\; v_i v_j \notin E(G) \end{cases}$$

A sample 6 vertice graph and its adjacency matrix representation can be seen on Figure 1.



$$A(G) = \begin{array}{c|cccccc} & a & b & c & d & e & f \\ \hline a & 0 & 1 & 0 & 0 & 0 & 1 \\ b & 1 & 0 & 1 & 0 & 0 & 0 \\ c & 0 & 1 & 0 & 1 & 0 & 0 \\ d & 0 & 0 & 1 & 0 & 1 & 0 \\ e & 0 & 0 & 0 & 1 & 0 & 1 \\ f & 1 & 0 & 0 & 0 & 1 & 0 \end{array}$$

Figure 1. A Sample Graph and its adjacency matrix

The basic graph definitions and how the problem can be modelled with graph was shown on the introduction section. The Section 2 will be about the full details of the problem and the Paull-Unger algorithm which is related to our work. The details of our new algorithm will be introduced on Section 3, and algorithm analysis and comparations will be on Section 4. The last section will be about conclusion and future works that may improve our algorithm.

2. PAULL-UNGER ALGORITHM

Paull-Unger algorithm finds all maximal independent sets and independence number of a graph.

Definition 2.1. [5] By using an alphabet of $\sum = \{\sigma_1, \sigma_2, ..., \sigma_n\}$, we can create words $x = \sigma_{i_1}\sigma_{i_2}...\sigma_{i_k}$ (ε is a word of zero length). $\sum^*$ denotes the all words that can be generated from the $\sum$ alphabet.

$\sum^* = \{x : x, \sum \textit{is a word in alphabet}\}$

For all these words, a union operator can be defined. Let *x* be as above and $y = \sigma_{j_1}\sigma_{j_2}...\sigma_{j_l}$. We put the symbols of the second word after we write the first word.

$$x.y = \sigma_{i_1}\sigma_{i_2}...\sigma_{i_k}\sigma_{j_1}\sigma_{j_2}...\sigma_{j_l}$$

as seen, for every *x* word, $x.\varepsilon = \varepsilon.x = x$, then $\sum^* = (\sum^*, \cdot, \varepsilon)$ is a monoid. $\sum^*$ can be denoted as a *free monoid* that is extracted from $\sum$ alphabet.

Example. If $\sum = \{0,1\}$, then $\sum^* = \{0, 1, 00, 01, 10, 11, 000, 001, ...\}$. Some union operations of $\sum^*$ can be shown as below.

$$10 \cdot 001 = 10001$$
$$10 \cdot \varepsilon = 10$$

2.1. Algorithm

With the vertice set of infinite digraph G=(Σ^*, E) that is combined with x $\in$ y $\Leftrightarrow$ y = xσ ($\sigma \in \Sigma$) equation, lets denote the set of all words that can be generated from the vertice alphabet of graph G, as Σ^*[5].

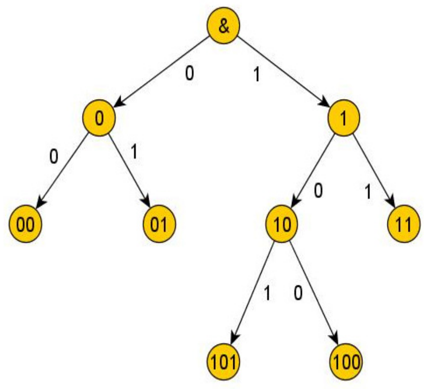


Figure 2. Infinite directed tree

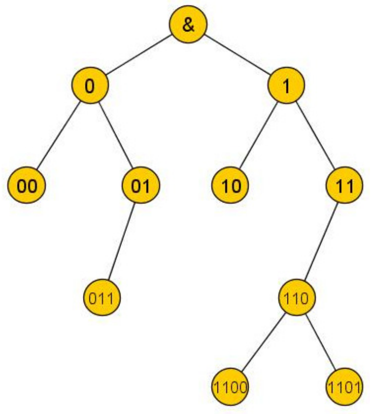


Figure 3. Binary Tree

An infinite directed tree graph for B* which is composed of $B = \{0,1\}$ binary alphabet, is shown on Figure 2. As shown on the figure, we label the edges that seem as directed branches, from left to right. This will be necessary if we need to look for the order which indicates B*. The first node is called root node. This is also the word null in the set of Σ^* .

V is a finite subset of B*, and T is in form $T = (V,E)$ and it is an infinite digraph binary tree,

(i) $x\sigma \in V \Rightarrow x \in V(\sigma \in B)$

(ii) $x \in y \Leftrightarrow \exists\sigma \in B, for\ y = x\sigma$

(i) guarantees that there is a path that connects every other node to the root node. *(ii)* is an edge connection that is mentioned before. The subset that shows the sets that belong to leaves can be defined as below.

$$L = L(T) = \{x \in V : for\ all\ \sigma \in B, x\sigma \notin V\} \subseteq V$$

Example. Figure 3 shows a binary tree. The equation (i) can be tested as below.

$1101 \in V \Rightarrow 110 \in V \Rightarrow 11 \in V \Rightarrow 1 \in V \Rightarrow \varepsilon \in V$

It is guaranteed from the equation (i) that, for the word x (x$\in\Sigma^*$) that each vertex represents, the all words that come before them will already exist on the tree. For example, in Figure 3**,** 1101 node is made of 110.1 and the node that comes before it is 110 , and it already exists in the tree. This special tree T has the set of vertices as below.

$$L = L(T) = \{00,10,010,1100,1101\}$$

$T = (V,E)$ can be used in algorithmic operations, for only finite number of vertices in V$\subseteq$B*. For this, the most suitable data type for T is data structure.

T: array B* of A

A is the algebra that is used for labelling the vertices. B^* is an infinite set, so labelling all vertices in the algorithm is impossible. Only the V$\subseteq$B* subset will be labelled. Because of the top to bottom ordering of the vertice labelling, we can say that the *V* subset provides the *(i)* situation. After the operation finishes, we have a labelled binary tree.

Labelled binary trees are used to find all maximal independent sets and find β independence number for a graph *G*. These calculations were first made by M.C.Paull and S.H.Unger. The solution can be shown by the algorithm below[5].

```
T: array $B^*$ of P (V)
β, i, j, n: positive integer
E: array $n^+ \times n^+$ of B
L, M: subset of $B^*$
v: array B of V
x: element of $B^*$
σ: element of $B^*$
begin
T ← ∅;
T[ε] ← V; L ← {ε};
    for j ←1 to n – 1 do
    for i ←j +1 to n do
    if E[i, j] = 1 then
    begin M ← {x ∈ L: $\{v_i, v_j\} \subseteq T[x]$};
          L ← L ~ M;
       v[0] ← $v_i$; v[1] ← $v_j$;
         for x ∈ M do
         for σ ∈ B do
       begin T[xσ] ← T[x] ~ {v[σ]};
              if ((T[xσ]$\not\subset$T[y]) for all y ∈ L then
                             L ← L ∪{xσ}
              end;
       end;
       β ← $\max_{x \in L}$ {|T[x]|}
  End.
```

This algorithm finds the β independence number for a graph *G*, and finds all sets of maximal independent sets. There will not be a bigger independent set.

Example. In a military warehouse, it is known that some materials react with each other when they are placed adjacently. In Figure 4, the materials are mapped to vertices, and reactions are mapped to edges.

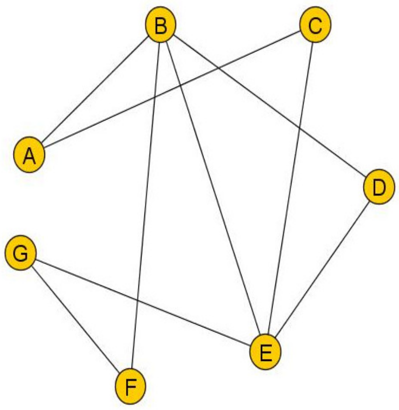


Figure 4. Sample Graph

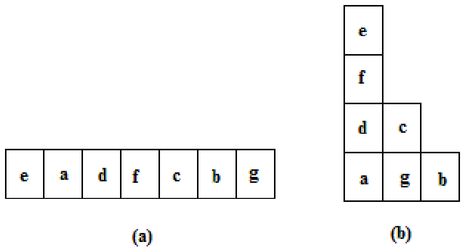


Figure 5. Two Sample Solutions

In the graph above, *a* and *b* may react with each other, but *a* and *d* don't react. This means that *a* and *d* can be put together. The problem is to find a compact solution that will place these materials together in a warehouse. Figure 5(a) and Figure 5(b) are some solutions for this problem. However, the solutions in Figure 5 are not feasible. The base and extended algorithms on the next section will find feasible solutions on the platform.

2. THE BASE ALGORITHM WITH NEW EXTENSIONS

In order to solve the problem, we need to put some rules that seperates feasible and infeasible solutions. According to this, for n number of materials and minimum t integer that provides $t^2 > n$, we can say that a solution which can fit into $t \times t$ square matrix is a feasible solution. Thus, we can make an algorithm for generating feasible solutions in a reasonable time.

There is no way to find an algoritm which will find the best solution logically, like many graph theoretic problems. Therefore, brute force[6] methods are more suitable to this type of problems. An heuristic approach to brute force that finds feasible solutions in a short time was introduced in study[7]. We have made extensions to this heuristic which improve the performance.

3.1. Classical Brute Force Approach

The most basic method to solve this problem will be to try every material to the square matrice randomly or in straight order, and backtrack when there is a conflict.

A basic order of trying to place materials to the corner by preserving the square shape can be as in Figure 6. According to this order, the materials are randomly placed as long as there are no conflicts. If the last material causes a conflict, then the material is changed with another available. If any of available material is suitable, then the previous material is also removed and another material is tried(backtracking). This method allows trying all *n!* combinations.

Searching the whole *n!* combinations becomes impossible for bigger *n* values. Because of this, another method that will help coming across a solution much faster, is needed.

1	2	5	10
4	3	6	11
9	8	7	12
16	15	14	13

Figure 6. A probable brute force order for placing materials to corner.

3.2. Base Heuristic Approach

According to the previous work[7], the base idea of this approach is to make mistakes as early as possible in trial-error stages. In order to do this, Paull-Unger algorithm is used to find maximal independent sets. Then, the elements in the sets are ordered according to the number of repetitions in these sets from the smallest to biggest. By doing this, we get a list of materials that are ordered from the most problematic to place, to the easiest one.

Then, starting with the most problematic material, the materials are placed to the left top corner in the order that is shown in Figure 7. The aim of this order is to detect a probable conflict as early as possible, and minimize the time spent on backtracking.

As an example, the situation in Figure 7 after placing the 9th material, can be observed. The following placements will have the risk of conflicting with the placement 5, 6, 7, 8 and 9. If backtracking is inevitable, then this should be done as early as possible, so the sides of the earliest placements should be covered first. In the figure, sides of 5, 6, 7, 8 and 9 are filled in the order.

1	**2**	**5**	**10**
3	**4**	**7**	**12**
6	**8**	**9**	**14**
11	**13**	**15**	**16**

Figure 7. New order for placing materials to corner

The recursive algorithm for this method is shown below.

```
Find(Loc L, int field[T][T], list PUList[N]){
  i:=0
  While i<N DO
     if (IsNodeUsed(PUList, i) = false) Then
        node := PUList[i];
        if(Control(L, node, field) = true ) Then
          field[L.X][L.Y] := node
          SetUsed(PUList, i)
          Find(NextLoc(L, field), field, PUList)
        end if
        SetFree(PUList, i)
     end if
     i=i+1;
  end while
} //End Function
```

The Paull Unger order that helps the speedup is taken as input. The *IsNodeUsed* function checks that if the material is used before. *NextLoc* function returns the next location to place, according to our order. *SetUsed* and *SetFree* functions marks that materials in the Paull Unger order, according to if they are used or not. *Control* function checks if the material that will be placed causes a conflict.

N → Number of materials.
T → Row number of the square matrix.
L → The location of the platform that will be filled.
PUList → The list of materials in the Paull Unger order.

It is observed that this algorithm finds solutions in much less steps with the help of Paull Unger material ordering and the specific placement order.

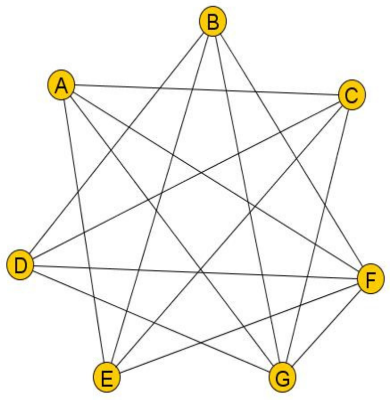
3.3. Extensions to Heuristic Approach

Three extensions are added into the base algorithm in this paper. These extensions are as below.

Extension 1- *Adding Blanks*

Heuristic approach finds solutions that fit into a minimum suitable square matrix. Total node number is generally smaller than the available places in the matrix, so there will usually be empty places. The heuristic always attempts to find fully filled solutions, and leaves empty spaces in only the lower right end of the matrix. This approach will not find solutions with blanks inside, as can be seen on the next figure.

This extension helps the algorithm to start finding the solutions first without any blanks, if there is any. But some graphs can have no such a solution; therefore it tries to find solutions with one blank in them, two blanks and so on.



A	B	C
D		F
E	G	

Figure 8. Finding solutions with blanks.

This improves the algorithm to find solutions for special graphs which has no blank-less solutions. An example of finding solutions with blanks is shown on Figure 8.
The main method of this extension is shown underneath.

```
for(blank = 0 ; blank <= N blank++){
    init_all();
```

```
        Find (tmp, Set, Arr);
        AddBlankNode();
    }
```

Extension 2- ***Placing in Any Rectangle***

According to this extension, algorithm can do placement in any rectangular area, while the base heuristic algorithm finds solutions for only square surface. This is achieved by limiting the dimensions and omitting unwanted solutions (placements). This extension provides flexibility for the heuristic. The method for the extension is shown below.

```
  Loc XYNext(int x, int y){
        Loc L;
        do{
            L = MyNext(x,y);
            x=L.X;
            y=L.Y;
        }while (L.X>=X||L.Y>=Y);
        return L;
    }
```

Extension 3- ***Reducing Steps***

The last and the most effective extension is about building a smarter heuristic which prunes unnecessary steps in backtracking.

While placing materials in the described heuristic, backtracking is made when there is no suitable material for placing to the current location. Some parts of those backtracking movements are redundant and those movements lead to unnecessary trials which will not affect the conflict on the current location. The idea is to backtrack directly to the source of the conflict, which is the neighbour location that causes conflict on the current place. Using these shortcuts to the conflict sources prunes the mostly redundant branches on the tree of possibilities. The pseudo code with the extension is as below.

```
Find(Loc L, int field[T][T], list PUList[N]){
  i:=0
  While i<N DO
     if (IsNodeUsed(PUList, i) = false) Then
        node := PUList[i];
        if(Control(L, node, field) = true ) Then
          field[L.X][L.Y] := node
          SetUsed(PUList, i)
          Find(NextLoc(L, field), field, PUList)
        if(btnode!=-1)            //if in backtrack mode
              if(btnode==node)  //btrack node reached
            btnode=-1;        //end backtrack
          else return;    // else backtrack until btnode
            end if
        end if
```

```
            SetFree(PUList, i)
        end if
        i=i+1;
    end while
    if (No node fits into Location L)
        btnode=nearest_conflicting_node;
    end if
}
```

The backtracking method using this shortcut may cause pruning some solutions, when a node placed in the backtracking path is suitable for the conflict. To avoid this, the placed nodes are checked for suitability while in backtrack mode. When a node in the path is found to be suitable for the conflict, backtracking ends, thus possibility of pruning solutions is prevented.

4. ANALYSIS OF ALGORITHM

The worst case of the run time can be stated as $N!$. The first material can be put on N different ways. The second is N-1, third is N-2 , and it goes on resulting a total combination of $N!$ steps of search in worst case. In realistic graphs, the state space will be lowered by the effect of pruning after each conflict, so the number of steps will be lower.

The base algorithm searches the whole state space and finds all the solutions that fit the rules. The first feasible solution is enough for us, so the base and the extended algorithm is optimised to find the first solution as fast as possible. With the help of Paull Unger material order and the order of filling locations, these algorithms searches the parts of the state space that seems to have more solutions. The elimination of problematic materials at the first steps will cut off most of the unnecessary combinations, prune the branches of the state space tree that are far to solution, and will lead to result. These optimizations will dramatically lower the steps needed to find a solution in compare to $N!$. The results of these optimizations can be seen on Figure 9. As seen on figure, the number of steps needed with using base Paull-Unger ordering is much lower than the reverse order and the random order.

#Solution	Worst Ordering	Random Ordering	Paull Unger Ordering (Base Algorithm)	The Extended Algorithm
1	1417	463	89	87
2	1481	511	294	269
3	2599	549	376	344
4	2655	2828	393	360
5	6516	4808	410	377
6	9289	5256	456	419
7	9419	5270	634	589
8	11951	5285	641	596
9	12457	5296	688	636
10	14055	9348	696	644
11	16403	9358	721	668
12	17086	9540	873	815

13	18467	9546	883	824

Figure 9. The number of steps for finding solutions with different orderings

Figure 10 shows the comparation of orderings in means of time in graphics. As seen on figure, the base algorithm has a linear graphic, while the other orderings tend to exponentially increase. We have observed that the time of a random ordering lies between the two orderings.

Our backtracking extension prunes unnecessary steps in backtracking in the case of a conflict. The effects of this extended algorithm can be seen on the rightmost column of Figure 9, and graphically on Figure 11. This improvement yields to an average of 8% decrease on the time needed to find a solution.

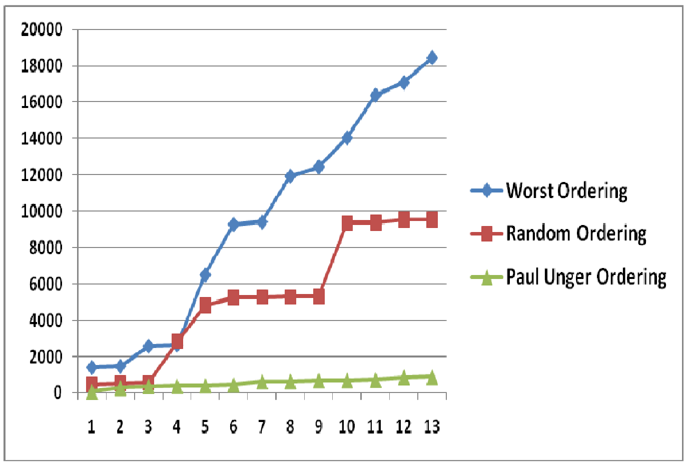


Figure 10. The time comparation of the new algorithm ordering and other orderings

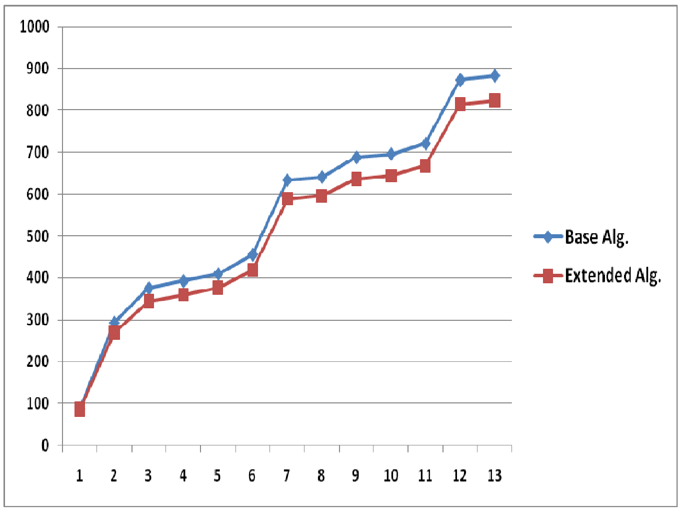


Figure 11. Comparison of steps for finding solutions

5. CONCLUSIONS AND FUTURE WORKS

According to previous work on this subject [7], a practical algorithm which approaches the settling problem with graph modelling and solves in much shorter time, has been made. The key ideas behind the speedup were that the materials are placed in an order generated by Paull Unger ordering, and locations are filled in a special order. The effect of Paull Unger ordering can be seen on Figure 10.

Three extensions have been made on this paper. There may be no gapless solution exist for some special graphs. Using the base algorithm will not return results for those graphs. For this case, the first extension is made to try several blanks in locations when there is no gapless result. The second extension has been made to add ability of situating materials in any type of arbitrary rectangles. Finally, the third and the biggest extension has been made to reduce backtracking steps by pruning unnecessary recursions during backtracks. This leads to using 8% less time to find a feasible solution, which is a notable performance improvement considering the problem is an NP-Complete replacement problem.

As a future work, we plan to design an algorithm for non-planar 3D environments, with dynamic programming methods.

6. REFERENCES

[1] Christofides,N.,Graph Theory an Algorithmic Approach, *Academic Pres*, London, (1986).

[2] West D.B., *Introduction to Graph Theory*, Prentice Hall, NJ, (2001).

[3] Blidia M., Chellali M., Favaron O., Meddah N., Maximal k- independent sets in graphs. *Discuss. Math. Graph Theory* 28 (2008) ,no.1,151-163.

[4] Chartrand G., Lesniak L., *Graphs & Digraphs*, Greg Hubit Bookworks, (1986).

[5] Prather Ronald E., Discrete Mathematical Structures for Computer Science, Houghton Mifflin Company, (1976).

[6] Coreman T., Leiserson C., Rivest R., Stein C., *Introduction to Algorithms*, 3rd Edition, The MIT Press, (2009).

[7] Dundar P., Tokatli G., Challenger M., Turaci T., *A Heuristic Algorithm for Placing Chemically Reacting Materials on a Platform*, National Conference on Informatics, Mugla University, Turkey, 10-12 Feb 2010.