# Comparing Heuristic Search Methods for Selecting Sequence of Refactoring Techniques Usage for Code Changing

Ratapong Wongpiang, Pornsiri Muenchaisri

*Abstract*— **Refactoring is the process of changing the internal structures, that preserves external behaviors of software. To improve software maintainability, we can apply several refactoring techniques to source code; applying different sequence of refactoring techniques to different parts of the source code results in different code changes and different level of software maintainability. This research uses Heuristic Search methods to find a sequence of refactoring techniques usage for code changing from a search space. Each Heuristic Search method has different characteristics and algorithm to reach an optimal result in solving the problem. Heuristic Search methods including Greedy Algorithm, Breadth First Search, Hill Climbing and A* (A Star) are used to search for sequence of refactoring techniques usage from search space and compare the effort and result of the search methods. The purpose of the research is to find the most appropriate Heuristic Search method for searching sequence of refactoring techniques usage with maximum software maintainability and least searching time. The researcher evaluates each Heuristic Search method with source code containing Long Method, Large Class and Feature Envy bad smell. The result shows that Greedy Algorithm shows the best results with maximum software maintainability and the least searching time.**

*Index Terms*—**Refactoring, Refactoring Sequencing, Heuristic Search, Software Maintainability**

## I. Introduction

BAD SMELL [1] is characterized by bad design or bad coding of software developers. Source code with bad smell makes software complex, low software quality and reduces software maintainability. To resolve this problem, refactoring technique has been introduced. Refactoring [1] is the process of changing the internal structure of software that preserves its external behavior. Martin Fowler [1] has identified characteristics of bad smell. Each refactoring can change source code that impacts internal attributes such as size, complexity, coupling and cohesion differently. In some software, many refactoring techniques are used to apply to several parts of source code. Different of refactoring techniques usages becomes a choice for developer to change the source code. Selecting the appropriate sequence of refactoring techniques to obtain the changed source code with optimal software maintainability value is investigated. In general, if we search for optimal result (maximum or minimum value depend on domain problem) without using search algorithm, we have to create all possible paths and then select the best path that resolves a problem with optimal result. To resolve the problem without using search algorithm, we cannot find an appropriate result because the problem has many paths or large space. It is a waste of time to find a result. So search method helps us to resolve a problem obtaining optimal result and least searching time.

The purpose of the research is to find the most appropriate Heuristic Search method for selecting sequence of refactoring techniques usage for code changing with maximum software maintainability and least searching time. In this paper, we use Heuristic Search methods to search for sequence of refactoring techniques usage for code changing. Heuristic Search methods that we focus are Greedy Algorithm, Breadth First Search, Hill Climbing and A* (A Star). The research evaluates each Heuristic Search method with the source code containing Long Method, Large Class and Feature Envy bad smells. The result shows that Greedy Algorithm shows the best results with maximum software maintainability and least searching time.

This paper is organized as follows. Section 2 introduces the related work. Section 3 describes algorithm of each Heuristic Search method. Section 4 describes the search methods steps for searching sequence of refactoring techniques usage. Section 5 presents an experiment for the results of each Heuristic Search method. Finally, section 6 is discusses conclusions and future work.

## II. Related Work

T. Mens, G.Taentzer and O.Runge [2] present refactoring techniques as graph transformations, the techniques of critical pair analysis and sequential dependency analysis to detect the implicit dependencies among refactoring techniques. Their approach can suggest developer to select appropriate refactoring techniques and refactoring order to be applied.

Eduardo Pivetam, Joao Araujo, Marcelo Pimenta, Pedro Guerrerirro and R. Tom Price [3] present an approach to reduce the search space for refactoring opportunities, by

Manuscript received January 30, 2014.

Ratapong Wongpiang is with Center of Excellence in Software Engineering Department of Computer Engineering, Chulalongkorn, Bangkok, Thailand; e-mail: Ratapong.W@Student.chula.ac.th).

Pornsiri Muenchaisri is with Center of Excellence in Software Engineering Department of Computer Engineering, Chulalongkorn, Bangkok, Thailand; e-mail: Pornsiri.Mu@chula.ac.th).

providing mechanisms to create and simplify a Deterministic Finite Automata representing the applicable refactoring sequences in existing software. They exemplify the approach with five refactoring patterns; Equivalent, Commutative, Inverse, Forbidden and Parallel, to further reduce the scope of refactoring.

Sanjay K.D. and Ajay R. [4] propose a model and analyze the effects of relationship between the CK metrics [10] and the software maintainability to assess software system quality. They show that keeping low values of CK metrics results in invariably software's maintainability improving, for a qualitative utility.

R. Wongpiang and P. Muenchaisri [5] propose an approach to select sequence of refactoring techniques usage for code changing, using Greedy Algorithm. They use Greedy Algorithm to obtain a sequence of refactoring techniques usage from search space, with optimal software maintainability and less searching time. They show that the changed source code using sequencing refactoring techniques usage improves software maintainability better than that without the techniques.

## III. Algorithms of Heuristic Search Method

### A. Greedy Algorithm

Greedy Algorithm [11] is an algorithm that follows the problem solving heuristic of making the locally optimal choice at each stage with the hope of finding a global optimum. This search method doesn't consider the previous stages, so that reduces the search space and gets the result in short time. In general, Greedy Algorithm have five components:

1. A candidate set, from which a solution is created
2. A select function, which chooses the best candidate can be used to contribute to a solution
3. A feasibility function, that is used to determine if a candidate can be used to contribute to a solution
4. An objective function, which assigns a value to a solution, or a partial solution
5. A solution function, which will indicate when we have discovered a complete solution

The General From of Greedy Algorithm can be shown on Fig 1.

```
function select (C : candidate_set) return candidate;
function solution (S : candidate_set) return boolean;
function feasible (S: candidate_set) return boolean;
function greedy (C: candidate_set) return candidate_set is
   x : candidate; S : candidate_set;
begin
      S := {};
      while (not solution(S)) and C /= {} loop
         x := select (C);  C:= C – {x};
         if feasible (S union{x})  then
            S := S union{x};
      if solution(S) then return S;
      else return es;
```

Fig 1 General Form of Greedy Algorithm

### B. Breadth First Search

Breadth First Search [12] is a graph search algorithm which is limited to essentially two operations: (a) visit and inspect a node of a graph and (b) gain access to visit the neighbor nodes of currently visited node. That means this searching method considers a previous stage for choosing the next nodes from a currently selected node. The searching process can choose to go to the previous nodes, from a currently selected node, to get the better result in the end of the process; If the next node give a worse result than a previous one. This algorithm use a queue data structure to store intermediate results as it traverse the graph, as follows:

1. Enqueue the root node
2. Dequeue a node and examine it

2.1 If the element sought is found in this node, quit the search and return a result.

2.2 Otherwise enqueue any successors (the direct child nodes) that have not yet been discovered.

3. If the queue is empty, every node on the graph has been examined – quit the search and return "not found".
4. If the queue is not empty, repeat from step 2.

The psedocode of Breadth First Search for traversing from node A to G can be shown in Fig 2.

```
Procedure BFS(G,v) is
   create a queue Q
   create a set V
   enqueue v onto Q
   add v to V
   while Q is not empty loop
      t ←Q.dequeue()
      if t is what we are looking for then return t
      for all edges e in G.adjacenEdges(t) loop
         u ← G.adjacentVertex(t,e)
         if u is not in V then add u to V enqueue y onto Q
   return none
```

Fig 2 Psedocode of Breadth First Search

### C. Hill Climbing

Hill Climbing [13] is a mathematical optimization technique which belongs to the family of local search. It is an iterative algorithm that starts with an arbitrary solution to a problem, then attempts to find a better solution by incrementally changing a single element of the solution. If the change produces a better solution, an incremental change is made to the new solution, repeating until no further improvements can be found. This search algorithm always head towards a state which is better than the current one and terminates when are better than the current state itself. The psedocode of Hill Climbing can be shown in Fig 3.

```
Proceduce Hill Climbing
   currentNode = startNode;
```

Fig 3 Psedocode of Hill Climbing (1)

ISBN: 978-988-19252-5-1
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

```
loop do
L = NEIGHBORS(currentNode)
   nextEval = -INF
   nextNode = NULL
   for all x in L
      if(EVAL(x) > nextEval)
         nextNode = x
         nextEval = EVAL(x)
   if nextEval <= EVAL(currentNode)
      Return currentNode
   currentNode = nextNode
```

Fig 3 Psedocode of Hill Climbing (2)

### D. A* (A Star)

A* [14] uses a Best First to search and find the least cost path from a given initial node to a goal node, one or more possible goals. As A* traverses the graph, it follows a path of the lowest expected total cost or distance and keeps a sort priority queue of alternate path segments along the way. A* considers two functions for selecting node to traverse: the past path cost function (g(x)) which is the known distance from the start node to the current node x and the future path cost function (h(x)). The equation of A* to get the goal state can be defined as:

$$f(x) = g(x) + h(x)$$

- f(x) refers to a goal stage
- g(x) refers to the past path cost function, which is the known distance from the staring node to the current node x
- h(x) refers to a future path-cost function, which is admissible heuristic of the distance from x to the goal

The psedocode of A* can be shown in Fig 4.

```
Procedure A*(star, goal)
   closedset := the empty set
   openset := {start}
   came_from := the empty map
   g_score[start] := 0
   f_score[start] := g_score[start] +
heuristic_cost_estimate(start, goal)

while openset is not empty
   current := the node in openset having the lowest
f_scroe[] value
   if current = goal
      return reconstruct_path(came_from, goal)
   remove current from openset
   add current to closedset
   for each neighbor in neighbor_nodes(current)
      if neighbor in closedset
         continue
      tentative_g_score := g_score[current] +
dis_between(current,neighbor)
   if neighbor not in openset or tentative_g_score <
g_score[neighbor]
```

Fig 4 Psedocode of A* (1)

```
      came_from[neighbor] := current
      g_score[neighbor] := tentative_g_score
      f_score[neighbor] := g_score[neighbor] +
heuristic_cost_estimate(neighbor, goal)
      if neighbor not in open
return failure

function reconstruct_path(came_from, current_node)
   if current_node in came_from
      p := reconstruct_path(came_from,
came_from[current_node])
      return (p + current_node)
   else
      return current_node
```

Fig 4 Psedocode of A* (2)

## IV. Comparing Four Search Methods in Selecting Sequence of Refactoring Techniques Usage for Code Changing

To compare the search methods in selecting sequence of refactoring techniques usage for code changing, we consider for two criterias: maintainability of changed source code after sequencing of refactoring techniques usage and searching time (number of retrieved nodes) to get a result. We want to find which Heuristic Search method is appropriate for selecting refactoring techniques usage of code changing with optimal software maintainability and the least searching time. The steps of searching for sequence of refactoring usage paths of the search methods are defined as follow.

### A. Greedy Algorithm

1. Apply each refactoring technique to the positions to be changed in source code.
2. Calculate software maintainability of changed source codes that have already been applied by refactoring techniques.
3. Select a path or changed source code which has maximum software maintainability to be applied refactoring techniques in the remained positions.
4. Check that there are positions to be applied refactoring techniques or not.

4.1 If there are positions then repeats from step 1 to step 4 until there are no positions to be changed or no refactoring techniques to apply.

4.2 If there are no position then stop searching process.

5. Obtain the sequence of refactoring techniques usage which makes the changed source code with optimal software maintainability.

The number of nodes which has to be retrieved for the result using Greedy Algorithm can be found by formula as follow:

Total Number of Retrieved Nodes $_{GA}$ = Sum of considered refactoring techniques each round ( $\sum_{r=1}^{z} \mathrm{Re}_r$ )

ISBN: 978-988-19252-5-1
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

$\mathrm{Re}_r$ = Sum of considered refactoring techniques each position to be applied ( $\sum_{i=1}^{z} P_i$ )

- *Re* refers to number of refactoring techniques to consider each round.
- *P* refers to number of refactoring techniques to consider each position.
- *r* refers to round of searching.
- *z* refers to number of positions to be applied refactoring techniques.
- *i* refers to position number to be applied refactoring technique.

From the formula, the number of positions to be applied refactoring technique decreases by one position at the end of each round.

### B. Breadth First Search

1. Apply each refactoring technique to the positions to be changed in source code.
2. Calculate software maintainability of changed source codes that have already been applied by refactoring techniques.
3. Select a path or changed source code which has maximum software maintainability to be applied refactoring techniques in the remained positions by considering previous nodes (except root node or the first round of searching process).
4. Check that there are positions to be applied refactoring techniques or not.

4.1 If there are positions then repeats from step 1 to step 4 until there are no position to be changed or no refactoring techniques to apply.

4.2 If there are no position then stop searching process.

5. Obtain the sequence of refactoring techniques usage which makes the changed source code with optimal software maintainability.

The number of nodes which has to be retrieved for the result using Breath First Search can be found by formula as follow:

Total Number of Retrieved Nodes $_{BFS}$ = Sum of considered refactoring techniques each position to be applied in the 1st round ( $\sum_{i=1}^{z} P_i$ ) + Sum of considered refactoring techniques from 2nd round on ( $\sum_{r=2}^{z} \mathrm{Re}_r$ )

$\mathrm{Re}_r$ = Sum of considering refactoring techniques each position to be applied ( $\sum_{i=1}^{z} P_i$ ) + (Sum of considering refactoring techniques of previous round - 1)

From the formula, the number of positions to be applied refactoring technique decreases by one position at the end of each round.

### C. Hill Climbing

1. Apply each refactoring technique to the positions to be changed in source code.
2. Calculate software maintainability of changed source codes that have already been applied by refactoring techniques.
3. Select a path or changed source code which has software maintainability better than current path to be applied refactoring techniques in the remained path. Select path which has maximum software maintainability in case there are many better paths.
4. Check that there are better positions to be applied refactoring techniques or not.

4.1 If there are position then repeats from step 1 to step 4 until there are no positions to be changed or no better positions than current path.

4.2 If there are no position or no better positions than current path then stop searching process.

5. Obtain the sequence of refactoring techniques usage which makes the changed source code with optimal software maintainability.

From the step of searching process, the searching process can be stop if the software maintainability of current path is better than other next paths. As a result, some positions may not be changed by refactoring techniques after finishing process. The number of nodes which has to be retrieved for the result using Hill Climbing can be found by formula as follow:

Total Number of Retrieved Nodes $_{HC}$ = Sum of considering refactoring techniques each round ( $\sum_{r=1}^{z} \mathrm{Re}_r$ )

$\mathrm{Re}_r$ = Sum of considered refactoring techniques each position to be applied ( $\sum_{i=1}^{z} P_i$ )

From the formula, the number of positions to be applied refactoring technique decreases by one position at the end of each round.

### D. A* (A Star)

The equation of A* to get the sequence of refactoring techniques usage can be defined as in

$$f(x) = g(x) + h(x)$$

- f(x) refers to goal stage which all positions are applied refactoring techniques.
- g(x) refers to software maintainability of current node.
- h(x) refers to average of software maintainability of other refactoring techniques to go to the goal stage from current node.

The step of searching refactoring techniques usage path using A* can be defined as follow.

1. Apply each refactoring technique to the positions to be changed in source code.
2. To find g(x), calculate software maintainability of changed source codes that have already been applied by each refactoring from step 1.

ISBN: 978-988-19252-5-1
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

3. To find h(x), apply refactoring techniques at all other positions from changed source code in step 1. And then calculates average of software maintainability of all possible paths from changed source code.

4. Consider sum of g(x) and h(x) of each path from step 1 by select a path which has maximum value.

5. Check that there are positions to be applied refactoring techniques or not.

5.1 If there are positions then repeats from step 1 to step 5 until there are no position to be changed or no refactoring techniques to apply.

5.2 If there are no position then stop searching process.

6. Obtain the sequence of refactoring techniques usage which makes the changed source code with optimal software maintainability.

The number of nodes which has to be retrieved for the result using A* can be found by formula as follow:

Total Number of Retrieved Nodes $_{A*}$ = Sum of considered refactoring techniques each round ( $\sum_{r=1}^{z} \mathrm{Re}_r$ )

$\mathrm{Re}_r$ = Sum of all possible nodes of refactoring techniques of $P_i$ with other positions.

From the formula, the number of positions to be applied refactoring technique decreases by one position at the end of each round.

## V. EXPERIMENT

In the experiment, we apply each Heuristic Search method to search for sequence of refactoring techniques usage to change Statement method of Customer class on Movie Rental System [1]. The Statement method contains Long Method, Large Class and Feature Envy bad smells. We apply refactoring techniques at three positions to remove bad smells. Each position can be applied by two refactoring techniques: Extract Method and Move Method (shown in Table I). To calculate software maintainability, we focus on three Object Oriented Metrics: Weight Method per Class (WMC), Lack of Cohesion in Method (LCOM) and Coupling between Object Classes (CBO) [7]. For Coupling between Object Classes metric, we consider two couplings: Efferent Coupling (EC) and Afferent Coupling (AC). The relationship between the metrics and software maintainability are inverse [6].

```
01 Class Customer {
02   public String statement(){
03     double totalAmount = 0;
04     int frequentRenterPotints = 0;
05     Enumeration  rentals = _rentals.elements();
06 String result = "Rental Record for " + getName(); + "\n"
07   while(rentals.hasMoreElements()){
08     double thisAmount = 0;
09     Rental each = (Rental) rentals.nextElement();
```

Fig 5 Customer Class (1)

```
10     if(each.getMovie().getPriceCode() ==
Movie.REGULAR)){
11       thisAmount += 2;
12       if(each.getDaysRented() > 2)
13         thisAmount += (each.getDaysRented() - 2) * 1.5;
14     }
15    else if(each.getMovie().getPriceCode() ==
Movie.NEW_RELEASE){
16          thisAmount += each.getDaysRented() * 3;
17     }
18     else if(each.getMovie().getPriceCode() ==
Movie.CHILDRENS){
19          thisAmount += 1.5;
20          if(each.getDaysRented() > 3)
21         thisAmount += (each.getDaysRented() - 3) * 1.5;
22     }
23     if((each.getMovie().getPriceCode() ==
Movie.NEW_RELEASE) && each.getDaysRented() > 1)
24       frequentRenterPoints++;
25       result += "\t" + each.getMovie().getTitle() + "\t" +
String.valueOf(thisAmount) + "\n";
26       totalAmount += thisAmount;
27      result += "Amount owed is " + totalAmount
28      result += "You earned " + frequentRenterPoints + "
frequent renter points";
29    return result;
```

Fig 5 Customer Class (2)

TABLE I
LIST OF REFACTORING TECHNIQUES TO APPLY WITH CUSTOMER CLASS

| Position | Applied refactoring techniques |
|---|---|
| P1 (line 8-23) : Calculating movie charge part | R1 : Extract this part to new method of Customer class. |
| | R2 : Move this part to a new method of Rental Class. |
| P2 (line 24-25) : Calculating frequency rental point part | R3 : Extract this part to new method of Customer class. |
| | R4 : Move this part to a new method of Rental Class. |
| P3 (line 28) : Calculating total movie charge part | R5 : Extract this part to new method of Customer class. |
| | R6 : Move this part to a new method of Rental Class. |

TABLE II
THE RESULT OBJECT ORIENTED METRICS OF CUSTOMER CLASS AFTER APPLYING REFACTORING TECHNIQUES USAGE USING HEURISTIC SEARCH METHODS

| Heuristic Search | Object Oriented Metrics | | | | Sequence |
|---|---|---|---|---|---|
| | WMC | LCOM | AC | EC | |
| GA | 9 | 0.6 | 0 | 1 | R2,R4,R6 |
| BFS | 9 | 0.6 | 0 | 1 | R2,R4,R6 |
| HC | 7 | 0.625 | 0 | 1 | R2,R4 |
| A* | 9 | 0.6 | 0 | 1 | R2,R4,R6 |

From Table II, the sequence of refactoring techniques usage from Greedy Algorithm, Breadth First Search and A* to change Statement method are S[R2,R4,R6] with WMC = 9, LCOM = 0.6, AC = 0 and EC = 1. But the sequence of refactoring techniques usage from Hill Climbing is S[R2,R4] which finishes searching process after changing only two positions because the Object Oriented Metrics values of changed source code on position 2 and 4 are better than changed source code after all positions. That causes

ISBN: 978-988-19252-5-1
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

TABLE III
THE NUMBER OF RETRIEVED NODES TO SEARCH FOR SEQUENCE OF REFACTORING TECHNIQUES USAGE USING HEURISTIC SEARCH METHODS TO CHANGE CUSTOMER CLASS

| Round | GA | BFS | HC | A* |
|---|---|---|---|---|
| 1st | 6 | 6 | 6 | 48 |
| 2nd | 4 | 9 | 4 | 16 |
| 3rd | 2 | 5 | - | 2 |

the changed source code of Statement method using Hill Climbing still has bad smell on position 3. So the quality of changed source code using Greedy Algorithm, Breadth First Search and A* is better than using Hill Climbing. In searching time, we consider the number of retrieved nodes to search for the sequence of refactoring techniques of each Heuristic Search method. The results can be shown in Table III. In the first round of Table III, the number of retrieved nodes of Breadth First Search is as same as the number of retrieved nodes of Greedy Algorithm and Hill Climbing because the first round starts from root node and there are no previous nodes for Breadth First Search to consider. The number of retrieved nodes of A* are as same as the number of all possible nodes for applying refactoring techniques to change the Statement method. Because A* has to consider about the future path cost, function (h(x)) of six refactoring techniques and each refactoring technique has eight possible paths to change the Statement method. So there are forty-eight paths to calculate for h(x) of the first round. In the second round, the number of considering positions decreases by one position, that results the number of retrieved nodes of Greedy Algorithm and Hill Climbing decrease by two nodes. So there are four remaining nodes for Greedy Algorithm and Hill Climbing to consider in the second round. In Breadth First Search, there are nine nodes to be retrieved (six from remaining nodes and three from previous nodes). A* has to consider sixteen node for h(x) calculating from two remaining positions. In the third round, there is only one remaining position or two nodes to retrieve for Greedy Algorithm. For the last position to consider for A*, current remaining node has already considered all positions completely that causes h(x) equals 0 or it no need to calculate for h(x). So the number of retrieved nodes to consider is two nodes. In Breadth First Search, there are five nodes to retrieve for the last position (two nodes from remaining nodes and three nodes from previous nodes). But there are no nodes to consider in Hill Climbing for the last position because the searching process has already stopped in the end of previous round.

In searching time, we conclude that Greedy Algorithm to search for sequence of refactoring techniques usage to change the Statement method uses less time than Breadth First Search and A*. Although Hill Climbing use less time than Greedy Algorithm, it doesn't remove bad smell completely.

## VI. Conclusion

Our research finds the most appropriate Heuristic Search method for searching sequence of refactoring techniques usage with maximum software maintainability and the least searching time. We evaluate four Heuristic Search methods; Greedy Algorithm, Breadth First Search, Hill Climbing and A* to change the Statement method of Customer class containing Long Method, Large Class and Feature Envy bad smell. In the experiment, we compare sequences of refactoring techniques usage (changed source code) and searching time (number of retrieved nodes) to get a result of each Heuristic Search methods. The result of the experiment shows that the quality changed source code using Greedy Algorithm, Breadth First Search and A* is better than using Hill Climbing because the changed source code using Hill Climbing still contains bad smell. In searching time, the number of retrieved nodes to search for sequence of refactoring techniques usage using Greedy Algorithm is less than using Breadth First Search and A*.

In our future work, we will classify characteristic of source code that can be improved by the sequence of refactoring techniques usage. So, it will help developers consider applying refactoring techniques to improve their software maintainability.

## References

[1] M. Fowler, K. Beck, J. Brant, W. Opdyke and D. Roberts, Refactoring: Improving the Design of Existing Code. Addision Wesley Professional, 1990, pp. 13-72.

[2] T. Mens, G. Taentzer, O. Runge, "Analysing Refactoring Dependencies Using Graph Transformation," *Software and Systems Modeling*, vol.6, no. 3, 2007, pp. 269-285.

[3] Eduardo Piveta, Joao Araujo, Marcelo Pimenta, Ana Moreira, Pedro Gurreriro, R. Tom Price, "Searching for Opportunities of Refactoring Sequences : Reducing the Search Space", *The Annual IEEE International Computer Software and Applications Conference*, 2008.

[4] Sanjay K. D., Ajay R., "Assessment of Maintainability Metrics for Object-Oriented Software System", *ACM SIGSOFT Software Engineering Notes*, vol. 36, pp. 1-5, September,2011.

[5] R. Wongpiang, P. Muenchaisri, "Selecting Sequence of Refactoring Techniques Usage for Code Changing Using Greedy Algorithm," *The 2013 IEEE 4th International Conference on Electronics Information and Emergency Communication (ICEIEC 2013),* China, 2013.

[6] Bansiya J. and C. G. Davis, "A Hierarchical Model for Object-Oriented Design Quality Assessment," *IEEE Transactions on Software Engineering*, vol. 28, 2002, pp. 4-16.

[7] S. R. Chidamber and C.F. Kemerer, "A Metrics Suit for Object-Oriented Design", IEEE Transactions on Software Engineering, Vol 20, No. 6, pp.476-493.,Jun.

[8] KR Chowhary, "Artifical Intelligence (Heuristic Search)," presented at Department of Computer Science and Engineering MBM Engineering College, Jodhpur.

[9] P. Meananeatra, S. Rongviriyapanish and T. Apiwattanapong, "Identifying Refactoring Through Formal Model Based on Data Flow Graph," *The 5th Malaysian Conference in Software Engineering (MySEC)*, Malaysia, 2011.

[10] Hui Liu, Limei Yang, Zhendong Niu, Zhiyi Ma, Weizhong Shao, "Facilitating Software Refactoring with Appropriate Resolution Order of Bad Smells," *ESEC-FSE*, Amsterdam, The Netherlands , August, 2009.

[11] Paul E. Dunne. Greedy Algorithm. Available: http://cgi.csc.liv.ac.uk/~ped/teachadmin/algor/greedy.html

[12] Knuth, Donald E, The Art Of Computer Programming Vol 1. 3rd ed, Boston, 1997, pp. 590-597.

[13] Russell, Stuar J., Norvig, Peter, Artificial Intelligence: A Modern Approach 2nd edition, New Jersey, 2003, pp. 111-114.

[14] Delling D., Sanders P., Schultes D., Wagner D, Engineering route planning algorithms. Algorithmics of large and complex networks, Springer, 2009, pp. 117-139.