Ivan Mykolenko (UG)

Newcastle University

08.12.2017

Algorithm design and analysis

Content

[**NFRCP: based on next fit on-line strategy for solving Bin Packing Problem.** 2](#_Toc500532037)

[**FFRCP: based on first fit on-line strategy for solving Bin Packing Problem.** 3](#_Toc500532038)

[**Implementation of the Program** 4](#_Toc500532039)

[**First-Fit and Implementation Logics** 7](#_Toc500532040)

[**Next-Fit and Implementation Logics** 7](#_Toc500532041)

[**Results** 8](#_Toc500532042)

[**Conclusion** 9](#_Toc500532043)

# **NFRCP: based on next fit on-line strategy for solving Bin Packing Problem.**

|  |  |
| --- | --- |
| Pseudo Code | |
| Algorithm | Next-Fit |
| Inputs | A: list of orders;  B: list of coils; |
| Variables | coilsUsed: Integer;  coilIndex: Integer;  orderIndex: Integer; |
| **Begin**  coilsUsed := 0  coilIndex := 0  orderIndex := 0  **while** (orderIndex < size(A)) **do**  **while** (orderIndex < size(A) **and** coilLen(B[coilIndex]) >= A[orderIndex]) **do**  B[coilIndex] cutOff A[orderIndex]  orderIndex := orderIndex + 1  **od**  **if** (coilLen(B[coilIndex]) < 5) **then**  remove (B[coilIndex])  **else**  coilIndex := coilIndex + 1  **fi**  coilsUsed := coilsUsed + 1  **od**  **return** coilsUsed  **End** | |

|  |
| --- |
| Java Implementation |
| **public** **int** **nextFit**(List<Integer> orders, List<Rope> coils) {  **int** **coilsUsed** = 0;  **int** **coilIndex** = 0;  **int** **orderIndex** = 0;  **while** (orderIndex < orders.size()) {  **while** (orderIndex < orders.size() && coils.get(coilIndex).getLength() >= orders.get(orderIndex)) {  coils.get(coilIndex).cut(orders.get(orderIndex));  orderIndex++;  }  **if** (coils.get(coilIndex).getLength() < 5) {  coils.remove(coilIndex);  } **else** {  coilIndex++;  }  coilsUsed++;  }  **return** coilsUsed;  } |

# **FFRCP: based on first fit on-line strategy for solving Bin Packing Problem.**

|  |  |
| --- | --- |
| Pseudo Code | |
| Algorithm | First-Fit |
| Inputs | A: list of orders;  B: list of coils; |
| Variables | coilIndex: Integer;  coilsUsed: set of coils used; |
| **Begin**  coilIndex := 0  coilsUsed  **for** i := 0 to size(A) – 1 **do**  **while** (coilLen(B[coilIndex]) < A[i]) **do**  coilIndex := coilIndex + 1  **od**  B[coilIndex] cutOff A[i]  B[coilIndex] addTo coilsUsed  **if** (coilLen(B[coilIndex]) < 5) then  remove (B[coilIndex])  **fi**  coilIndex := 0  **od**  **return** size(coilsUsed)  **End** | |

|  |
| --- |
| Java Implementation |
| **public** **int** **firstFit**(List<Integer> orders, List<Rope> coils) {  **int** **coilIndex** = 0;  Set<Rope> **coilsUsed** = **new** HashSet<Rope>();  **for** (**int** **i** = 0; i < orders.size(); i++) {  **while** (coils.get(coilIndex).getLength() < orders.get(i)) {  coilIndex++;  }  coils.get(coilIndex).cut(orders.get(i));  coilsUsed.add(coils.get(coilIndex));  **if** (coils.get(coilIndex).getLength() < 5) {  coils.remove(coilIndex);  }  coilIndex = 0;  }  **return** coilsUsed.size();  } |

# **Implementation of the Program**

Algorithms class

The class contains the two methods: First-Fit and Next-Fit implemented based on pseudo-codes described above. Both methods take 2 parameters: an array list with coils and array list with orders. These lists were created and filled with random data in Generator class and passed to the algorithms through the Test class. First-Fit and Next-Fit both return the number of coils they have used fulfilling the orders.

Generator class

The purpose of this class is to generate data and to make a deep copy of the lists generated, which allows to run two algorithms simultaneously on the same data sample. The class contains 4 methods:

* generateMultipleOrders – creates a list of orders and fills it with orders within specific range.
* orderRopeFromManufacturer – creates a list of coils and fills it with coils within specific range.
* getDeepCopyOfOrders – creates a new list of orders and copies all the orders from the original list into the new one.
* getDeepCopyOfCoils – creates a new list of coils and copies all the coils from the original list into the new one using Rope constructor with Rope object parameter.

Rope class

The Rope class is used for generating rope objects of specific length. It contains:

* Rope constructor with an int parameter. Used for specifying a rope’s length.
* Rope constructor with a Rope object parameter. This constructor assigns the length of the input parameter’s rope to the newly made rope. Used for generating a deep copy of Rope objects.
* getLength method – returns length of a rope.
* setLength method – sets length for a rope.
* cut method – takes int as a parameter and deducts it from the length of the coil.

Test class

The test class has been design to test the performance of two algorithms using identical input data in every run. The test can be launched using parameters such as number of tests, number of repetitions for each test, number of orders to test, increment for orders in each test. This class produces a table with information on how Next-Fit and First-Fit have performed on each number of orders. It also analyses the data and concludes upon the fastest and the most efficient algorithm. Algorithms are also compared against each other and the program outputs the comparison information in percentages.

* **main method** – launches the program. Contains method calls to the corectnessTesting and performanceTesting.
* **createOrdersForCorrectnessTest** – returns a list of orders, which is used to makes sure both algorithms operate as expected. These orders are hardcoded, thus I could calculate the sort of output I should expect from both of the algorithms.
* **createCoilsForCorrectnessTest** – returns a list of coils, which is used to makes sure both algorithms operate as expected.
* **corectnessTesting** – using coils and orders created by createCoilsForCorrectnessTest and createOrdersForCorrectnessTest methods, the corectnessTesting allows the user to visually inspect whether the algorithms behave as they should and return the correct number of coils used.

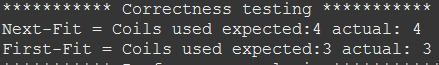


Figure : Correctness Testing

* **performanceTesting** – launches test of the two algorithms for the specified number of repetitions and orders. The performanceTesting method may run the algorithms multiple times on the same number of orders in order to provide the user with the best knowledge as to how efficient the algorithms are (how many coils they use).

The method captures start and end time of execution for the algorithms during each run. The times and coils used by each algorithms are stored in global variables to be retrieved and further analysed in the **analyzeResults** method**.**

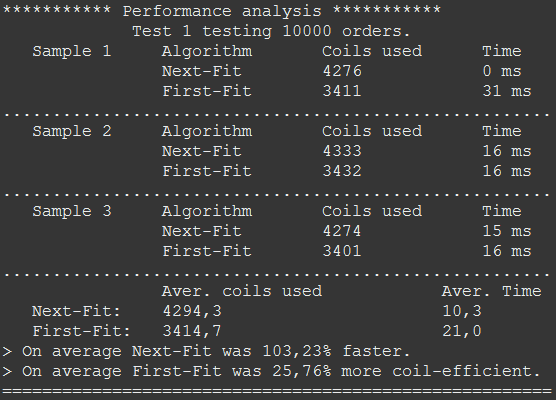
* **analyzeResults** – method displays the average performance (time and coils used)for every test of the two algorithms. Then results the two methods have produced are compared. The method calculates how quick and how efficient did one algorithm run in comparison to the other one and displays this as percentage.

Figure : Performance Testing and Analysis

The samples which, I’ve created in the createOrdersForCorrectnessTest and the createCoilsForCorrectnessTest methods allowed me to inspect whether the algorithms behave in the way they are supposed to. In order to see what is happening at each stage of the algorithms I have use “println” statements. Below are traces produced by the algorithms:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| I  n  p  u  t  D  a  t  a | Orders | 100 | 26 | 50 | 100 | | 50 |  |  |
| Coils | 130 | 100 | 120 | 130 | | 110 | 130 | 130 |
| Next-Fit Output | | | | | | First-Fit Output | | | |
| Coil 0 was: 130  Taking away: 100  Coil 0 became: 30  Coil 0 was: 30  Taking away: 26  Coil 0 became: 4  Coil 0 length is less than 5. Scrap it! | | | | | | Coil 0 was: 130  Taking away: 100  Coil 0 became: 30  Coil 0 was: 30  Taking away: 26  Coil 0 became: 4  Coil 0 length is less than 5. Scrap it! | | | |
| ///////////////////Comments////////////////////  Until the current coils has enough length to fulfil current order, the algorithm will keep re-using the coil.  The algorithm has used up all the available length in the first coil. Length became less than 5, so the coil is scraped. Coil list elements are shifted one place forward, so the coil 1 takes place of the coil 0.  coil counter = 1; | | | | | | ////////////////////Comments/////////////////////  Every time algorithm searches for the first coils in the list which has enough length to fulfil current order.  The algorithm has used up all the available length in the first coil. Length became less than 5, so the coil is scraped. Coil list elements are shifted one place forward, so the coil 1 takes place of the coil 0.  coil counter = 1; | | | |
| Coil 0 was: 100  Taking away: 50  Coil 0 became: 50 | | | | | | Coil 0 was: 100  Taking away: 50  Coil 0 became: 50 | | | |
| ///////////////////Comments////////////////////  Cut off the next order from the next coil.  coil counter = 2; | | | | | | ////////////////////Comments/////////////////////  Cut off the next order from the next coil.  coil counter = 2; | | | |
| Coil 1 was: 120  Taking away: 100  Coil 1 became: 20 | | | | | | Coil 1 was: 120  Taking away: 100  Coil 1 became: 20 | | | |
| ///////////////////Comments////////////////////  Cut off the next order from the next coil.  coil counter = 3; | | | | | | ////////////////////Comments/////////////////////  Cut off the next order from the next coil.  coil counter = 3; | | | |
| Coil 2 was: 130  Taking away: 50  Coil 2 became: 80 | | | | | | Coil 0 was: 50  Taking away: 50  Coil 0 became: 0  Coil 0 length is less than 5. Scrap it! | | | |
| ///////////////////Comments////////////////////  Although coil 0 was long enough to be used, the algorithm proceeded to the next coil, operating as expected.  coil counter = 4; | | | | | | ////////////////////Comments/////////////////////  Algorithm has come back to the coils 0 to fulfil this order, operating as expected.  coil counter = 3; | | | |
| Next-Fit = Coils used expected:4 actual: 4 | | | | | | First-Fit = Coils used expected:3 actual: 3 | | | |

# **First-Fit and Implementation Logics**

1. Starting form coil at position 0, the algorithm searches for the first coil that can fulfil current order.
2. When an appropriate coil has been found, the length of the order is deducted from the coil length.
3. The coil is marked as used.
4. If it turns out that current coil is less than 5 in length, then it gets chucked away, so the algorithm never looks at it again.
5. Proceed to the next order.
6. The above steps are repeated until the last order has been fulfilled.
7. Return the number of used coils.

# **Next-Fit and Implementation Logics**

1. Until current coil has enough length to fulfil next order – the algorithm keeps cutting the coil.
2. When the coil becomes too short to fulfil and order either:

* Chuck the coils away if its length is less than 5.
* Or just proceed to the next coil.

1. Increment used coils counter by one.
2. The above steps are repeated until the last order has been fulfilled.
3. Return used coils count.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Algorithm | 10000 Orders | | 20000 Orders | | 30000 Orders | | 40000 Orders | | 50000 Orders | |
| Average Time | Average Coils Used | Average Time | Average Coils Used | Average Time | Average Coils Used | Average Time | Average Coils Used | Average Time | Average Coils Used |
| Next-Fit | 1.1 ms | 4286.3 | 2.0 ms | 8555.3 | 4.3 ms | 12857.6 | 7.1 ms | 17146.3 | 10.6 ms | 21406.6 |
| First-Fit | 6.7 ms | 3411.0 | 19.6 ms | 6799.4 | 38.0 ms | 10206.9 | 66.0 ms | 13619.6 | 101.9 ms | 16978.0 |

# **Results**

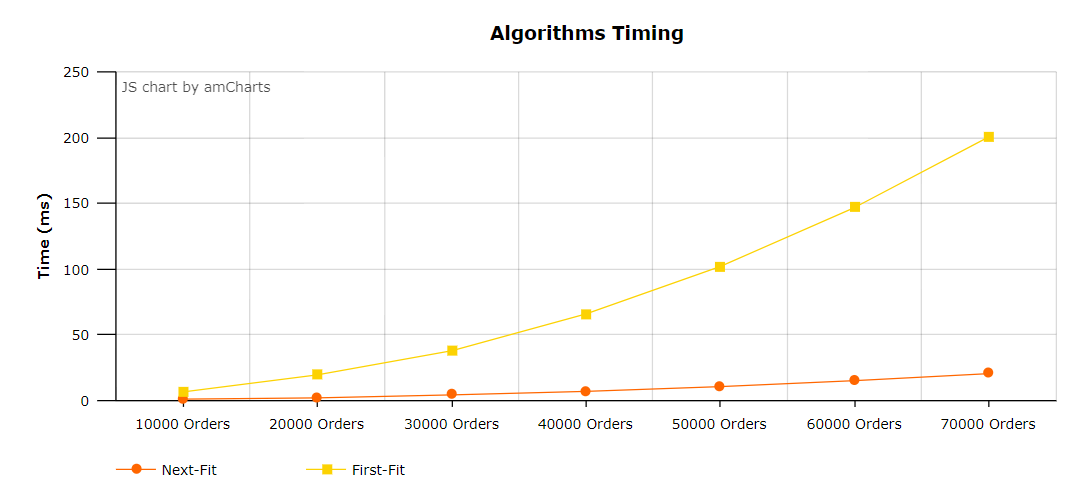
**

Figure : Visual representation of performance of First-Fit and Next-Fit algorithms

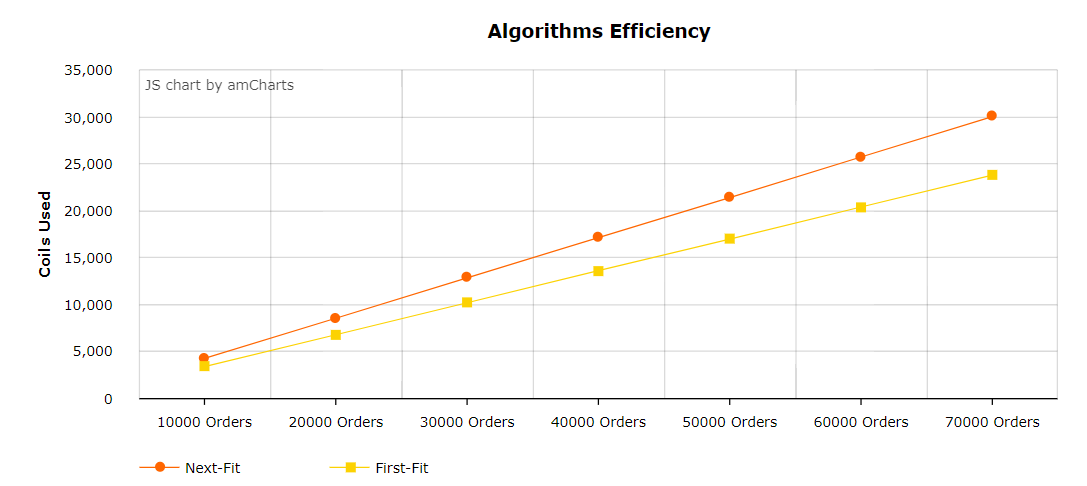
 The results for every number of orders shown above are produced based on 25 sample runs. For every sample new random orders and coils have been generated. Figure 3 shows that First-Fit has almost exponential increase in time as the number of orders grows, whereas Next-Fit maintains relatively little increase. For any orders size Next-Fit executes in less time compared to First-Fit. For instance: for 100 orders Next-Fit finishes at 1ms, First-Fit – 2ms. For 1000000 orders, on average Next-Fit finishes at 5539ms, which is 1207,96% faster than First-Fit with 72448ms.

Figure : Visual representation of performance of First-Fit and Next-Fit algorithms

On the other hand, as evident from figure 4, Next-Fit performs much poorer in terms of efficiency. The results have shown that on averaged First-Fit saves 26% more coils and thus fulfils orders in more efficient manner than Next-Fit does. For 500 orders First-Fit has consumed only 168 coils in-comparison to 206 for the Next-Fit. For 1000000 orders First-Fit: 339264 coils, Next-Fit: 428254.

# **Conclusion**

Having done multiple tests using different sizes of data samples on the two algorithms, it became evident that the design of the two algorithms described in this document work as expected and produces correct output. Analysis of the performance has shown that on average First-Fit algorithm is up to 26% more efficient in comparison to the Next-Fit, although Next-Fit takes less time to run.

During the implementation stage it has been discovered that removing coils from the coil list significantly slows algorithms down, since every element in list then has to be shifted one position forward. This could be an area for further improvement of the algorithms.