UNIVERSITI TUNKU ABDUL RAHMAN

Lee Kong Chian Faculty of Engineering & Science

UECS2083/UECS2413

PROBLEM SOLVING WITH DATA STRUCTURES AND ALGORITHMS

May 2018

Group Assignment (20%)

Group members

Student Name	Student ID	Practical Group
1. Wong Jia Hau	1500181	P1
2. Lau Guo Ren	1400537	P1
3. Choo Jing Yuan	1500982	P1

Tutor: Mr.Wong Chim Chwee

Lecturer: Ms. Chean Swee Ling

Table of Content

Introduction

The bin-packing problem can be described as such according to Korf(2002):

> Given a sets of objects, and an infinite sets of bins of constant capacity,
> Find the minimum number of bins required to contain all the objects,
> Such that the sum of weightage assigned to each bin does not exceed the bin capacity.

There are actually many variants to the bin-packing problem, such as packing by weight, packing by cost, 2D packing and linear packing etc. The variant required by this assignment is also known as 1D packing as it only involves one dimension (which is the weight of parcels), thus this can be considered one of the most simplistic variants of bin-packing problem.

The bin-packing problem are also related to the memory management process of operating systems, where the system needs to allocate memory to newly started application into certain memory segment such that the fragmentation problem can be reduced. Fragmentation is a phenomenon where all segments are acquired, but not each of them are fully occupied, so although the sum of their remaining spaces can contain the next object, but none of the individual remaining spaces can fit in the next object. This situation is not desirable, though unavoidable. Thus, the allocation algorithm should aim to minimize the fragmentation problem.

Design (data structures and algorithms)

The data structure used to store list of bins is a custom data structure which is call PointingList. PointingList is used because it allows the bin-packing algorithms to be described more naturally as PointingList exposes methods such as goToFirst and goToNext. Of course, a simple array can be used in describing the algorithm as well, but doing so would require a lot of indices variables (such as i and j) and nested for loops, which could obscure the meaning of the code. PointingList is actually similar to Java's ArrayList, which is a wrapper around Java's array, where one can add an infinite amount of elements to it in theory (in practice the limitation is due to the physical memory limit). The only difference between ArrayList and PointingList is that PointingList has a custom iterator, which can be reset to the first position whenever needed.

The algorithm implemented in this assignments are First Fit, Best Fit, First Fit Decrease and Best Fit Decrease. All these algorithms are implemented as a Java class which implements the Solver interface. The Solver interface exposes the method solve which takes a stack of Weightables elements and the maximum capacity of each container. It does not take a stack of Parcels, because that would cause all bin-packing algorithm to be too specific and non-reusable. This is necessary because the bin-packing problem can be expanded to others similar problems such as pipe-dividing problem. Thus, the Solver's solve method take a stack of Weightables instead, so that it can be generalized, and increasing reusability.

The Weightable interface exposes a method getWeight. Note that the "weight" here is generic, because it has different meanings in different context. For example, in bin-packing problem, weight is the mass of parcels; in the pipe-dividing problem, weight is the length of pipe.

The First Fit (FF) algorithm is an algorithm can be described in two sentences:

From the array of opened containers, find the first container which can fit the next element, E.
If no container can fit E, open up a new container to take in E.

The Best Fit (BF) algorithm is a little more complicated, thus it requires three sentences to describe:

From the array of opened containers, find the container which best fit the next element, E.
Best fit means that the container will have the least capacity remaining after taking in E.
If no container can fit E, open a new container to take in E.

First Fit Decrease (FFD) algorithm is actually a variant of FF, but the elements are sorted descendingly according to their weight first before they are being solved by FF.

Similarly, Best Fit Decrease (BFD) also requires the elements to be sorted first before they are solved by BF.

Visualization of First Fit Algorithm

Let say we have a six parcels which have the following weights (note that the order cannot be changed):

Parcel	**Weight**
A	3
B	1
C	6
D	4
E	5
F	2

Assume that the maximum capacity of each bin is 7. The steps of First Fit Algorithm are shown below:

Step	Explanation	Bins	Parcels	Remaining capacity
1	Put A into Bin1	Bin1	A(3)	4
2	Put B into Bin1	Bin1	A(3), B(1)	3
3	Put C into Bin2	Bin1	A(3), B(1)	3
		Bin2	C(6)	1
4	Put D into Bin3	Bin1	A(3), B(1)	3
		Bin2	C(6)	1
		Bin3	D(4)	3
5	Put E into Bin4	Bin1	A(3), B(1)	3
		Bin2	C(6)	1
		Bin3	D(4)	3
		Bin4	E(5)	2
6	Put F into Bin1	Bin1	A(3), B(1), F(2)	1
		Bin2	C(6)	1
		Bin3	D(4)	3
		Bin4	E(5)	2

So, using First Fit algorithm, 4 bins are required.

Visualization of Best Fit Algorithm

Let say we have a six parcels which have the following weights (note that the order cannot be changed):

Parcel	**Weight**
A	3
B	1
C	6
D	4
E	5
F	2

Assume that the maximum capacity of each bin is 7. The steps of Best Fit Algorithm are shown below:

Step	Explanation	Bins	Parcels	Remaining capacity
1	Put A into Bin1	Bin1	A(3)	4
2	Put B into Bin1	Bin1	A(3), B(1)	3
3	Put C into Bin2	Bin1	A(3), B(1)	3
		Bin2	C(6)	1
4	Put D into Bin3	Bin1	A(3), B(1)	3
		Bin2	C(6)	1
		Bin3	D(4)	3
5	Put E into Bin4	Bin1	A(3), B(1)	3
		Bin2	C(6)	1
		Bin3	D(4)	3
		Bin4	E(5)	2
6	Put F into Bin1	Bin1	A(3), B(1)	3
		Bin2	C(6)	1
		Bin3	D(4)	3
		Bin4	E(5),F(2)	0

So, using Best Fit algorithm, 4 bins are required.

Visualization of First Fit Decrease (FFD) Algorithm

Since FFD will sort the parcels according to their weight descendingly, the initial order of the parcels are not important anymore, so the table below shall show parcels according to their descending order.

Parcel	**Weight**
C	6
E	5
D	4
A	3
F	2
B	1

Assume that the maximum capacity of each bin is 7. The steps of First Fit Decrease Algorithm are shown below:

Step	Explanation	Bins	Parcels	Remaining capacity
1	Put C into Bin1	Bin1	C(6)	1
2	Put E into Bin2	Bin1	C(6)	1
		Bin2	E(5)	2
3	Put D into Bin3	Bin1	C(6)	1
		Bin2	E(5)	2
		Bin3	D(4)	3
4	Put A into Bin3	Bin1	C(6)	1
		Bin2	E(5)	2
		Bin3	D(4), A(3)	0
5	Put F into Bin2	Bin1	C(6)	1
		Bin2	E(5), F(2)	0
		Bin3	D(4), A(3)	0
6	Put B into Bin1	Bin1	C(6), B(1)	0
		Bin2	E(5), F(2)	0
		Bin3	D(4), A(3)	0

From this table, we can see that FFD performs better than FF as FFD requires only 3 bins to pack all parcels, but FF requires 4 bins.

Visualization of Best Fit Decrease (BFD) Algorithm

Same as FFD, since BFD will sort the parcels according to their weight descendingly, the initial order of the parcels are not important anymore, so the table below shall show parcels according to their descending order.

Parcel	**Weight**
C	6
E	5
D	4
A	3
F	2
B	1

Assume that the maximum capacity of each bin is 7. The steps of Best Fit Decrease Algorithm are shown below:

Step	Explanation	Bins	Parcels	Remaining capacity
1	Put C into Bin1	Bin1	C(6)	1
2	Put E into Bin2	Bin1	C(6)	1
		Bin2	E(5)	2
3	Put D into Bin3	Bin1	C(6)	1
		Bin2	E(5)	2
		Bin3	D(4)	3
4	Put A into Bin3	Bin1	C(6)	1
		Bin2	E(5)	2
		Bin3	D(4), A(3)	0
5	Put F into Bin2	Bin1	C(6)	1
		Bin2	E(5), F(2)	0
		Bin3	D(4), A(3)	0
6	Put B into Bin1	Bin1	C(6), B(1)	0
		Bin2	E(5), F(2)	0
		Bin3	D(4), A(3)	0

From this table, we can see that BFD performs better than BF as BFD requires only 3 bins to pack all parcels, but FF requires 4 bins.

Summary

From the previous visualizations, it is notable that for the input case of: [A(1), B(2), C(3), D(4), E(5), F(6)], both Decrease algorithm performs better than their counterpart. The explanation for this results shall be discussed in Discussion.

Discussion (efficiency and complexities)

Before discussing the efficiency and complexities of each algorithm, we need to understand what is *Lower Bound*. Basically, *Lower Bound* is the minimum number of bins required given a certain sets of input and the maximum capacity of each bin. For example, suppose the input is a set of parcels:

Parcel	Weight
A	7
B	8
C	8

And the maximum capacity of each bin is 10, then the *Lower Bound* for this case can be calculated as such:

Lower Bound = Sum of Weights / Maximum capacity of each bin

= (7 + 8 + 8) / 10

= 30 / 10

= 3

Thus, the *Lower Bound(LB)* for this case is 3, means that the minimum number of bins required to pack all the given input is 3. The purpose of LB is to serve as a point-of-comparison for different bin-packing algorithms, since LB represents the best solution.

Time complexity

The time complexity of each algorithm will be discussed according to the Java code written by us. We will assume that the number of input (n) shall means the number of objects (parcels) to be packed.

Time complexity of First Fit algorithm

The time complexity of First Fit algorithm is at least *O(n)* since it contains a loop to loop through all the parcels. In the worst case scenario, where each parcels have the same weight as the maximum capacity of each bin, then for every parcels that passes the loop, there will be a same amount of bin opened, thus this algorithm would be similar to a Dependent Quadratic algorithm, where the upper bound of inner loop depends on the iterator value of the outer loop:

```
for (i = 0; i < parcels.length; i++)
      for(j = 0; j < bins.length; j++)
            // application code
```

Thus, the numbers of time where the `application code` would be executed would be:

$$0 + 1 + 2 + 3 + 4 + \ldots + n - 1 = \frac{n(n-1)}{2} = \frac{n^2-n}{2} = \frac{n^2}{2} - \frac{n}{2} \Rightarrow n^2$$

Thus, the First Fit Algorithm has a time complexity of $O(n^2)$ at worse case.

For best case scenario, where all the input parcels can fit into one bin, then the time complexity would be $O(n)$ since the inner loop can be neglected (because there will be only 1 bin).

So, for average case, the time complexity would be $x \text{ such that } O(n) < x < O(n^2)$.

Time complexity of Best Fit algorithm

Similar as First Fit(FF) algorithm, the time complexity of Best Fit(BF) algorithm is at least $O(n)$ since it has to loop through every input (which are parcels). However, the inner loop of BF will be different from that of FF, because in BF, all bins will be loop through no matter what, but in FF the inner loop will be terminated as soon a bin which can fit the next parcel is found. Thus, if we assume that on average case where an available bin can be found halfway through the loop, then FF will performs better than BF on average case scenario. For worst case scenario, the time complexity of BF would be the same as FF which is $O(n^2)$ for the same reason as FF. Likewise, for best case scenario the time complexity of BF would be same as that of FF, which is $O(n)$.

Time complexity of First Fit Decrease (FFD) and Best Fit Decrease (BFD) algorithm

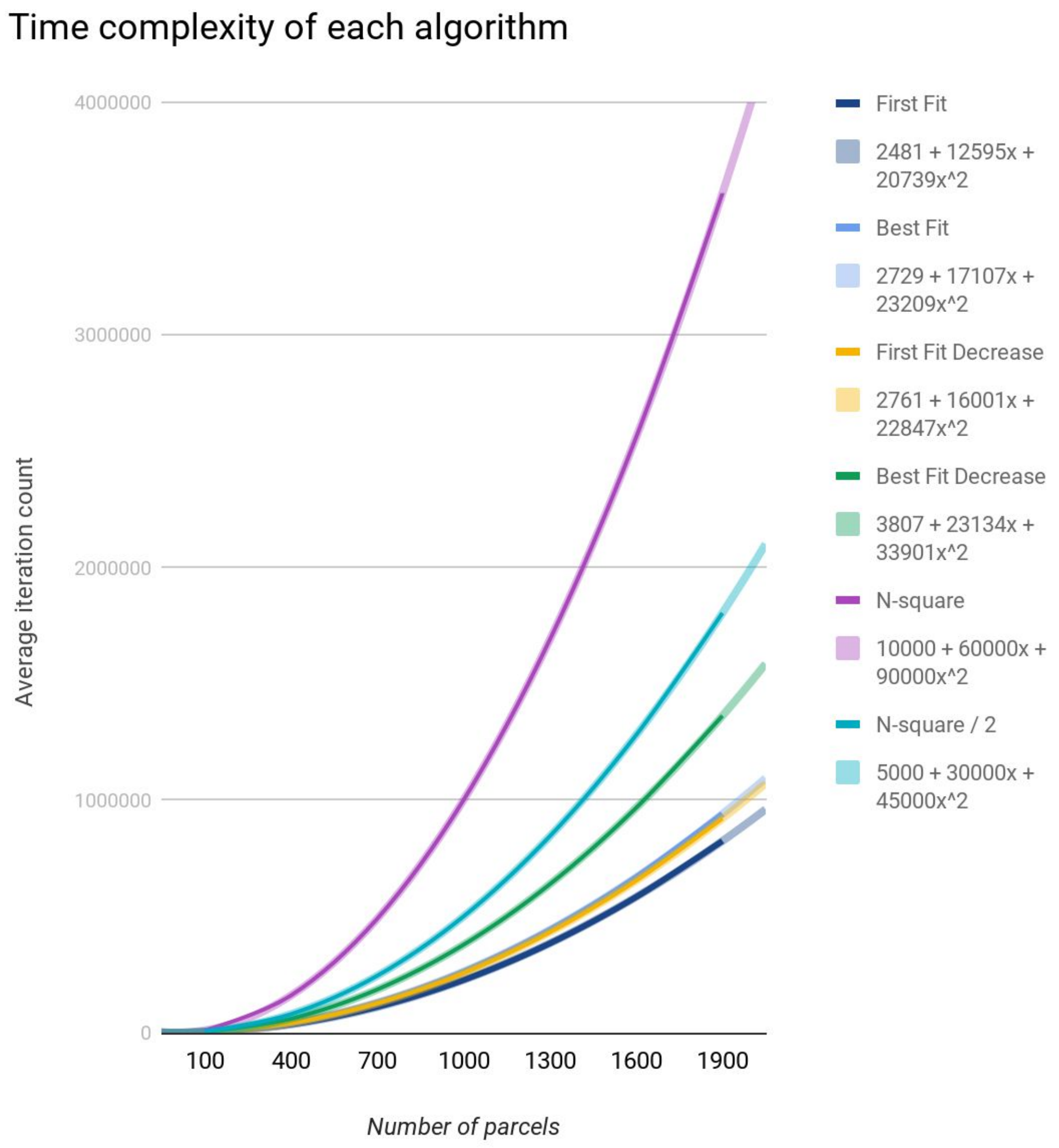
Both FFD and BFD are actually wrappers around their counterparts, because they require the input to be sorted first. In this assignment, we used stack sorting, which according to Amit(2015), have the time complexity of n^2 in worst case. Since both FF and BF have the time complexity of n^2 in worst case, the time complexity of FFD and BFD would be $n^2 + n^2 = 2n^2$ which can be reduced to n^2 as we can drop off the constant. Thus, for worst case, FFD and BFD would have the same time complexity as their counterparts.

Analysis

To find out whether the Big-O notation we formulated is correct, we created a Java method called `TimeComplexityBenchmark` to test the average number of iteration needed for each algorithm. The matrix below depicts the result. The first row is the input size (number of parcels), and the second to fifth row is the average number of iteration needed for each input size and algorithm.

Algorithm/ Input size	**100**	**400**	**700**	**1000**	**1300**	**1600**	**1900**
FF	2229	35948	111097	226944	384239	583731	824949
BF	2781	42764	130066	263120	442223	668403	940997
FFD	2653	41464	126592	256701	431842	653611	921604
BFD	3832	60654	185976	378320	638504	967145	1363055

Table: Iteration count of each algorithm with different input sizes



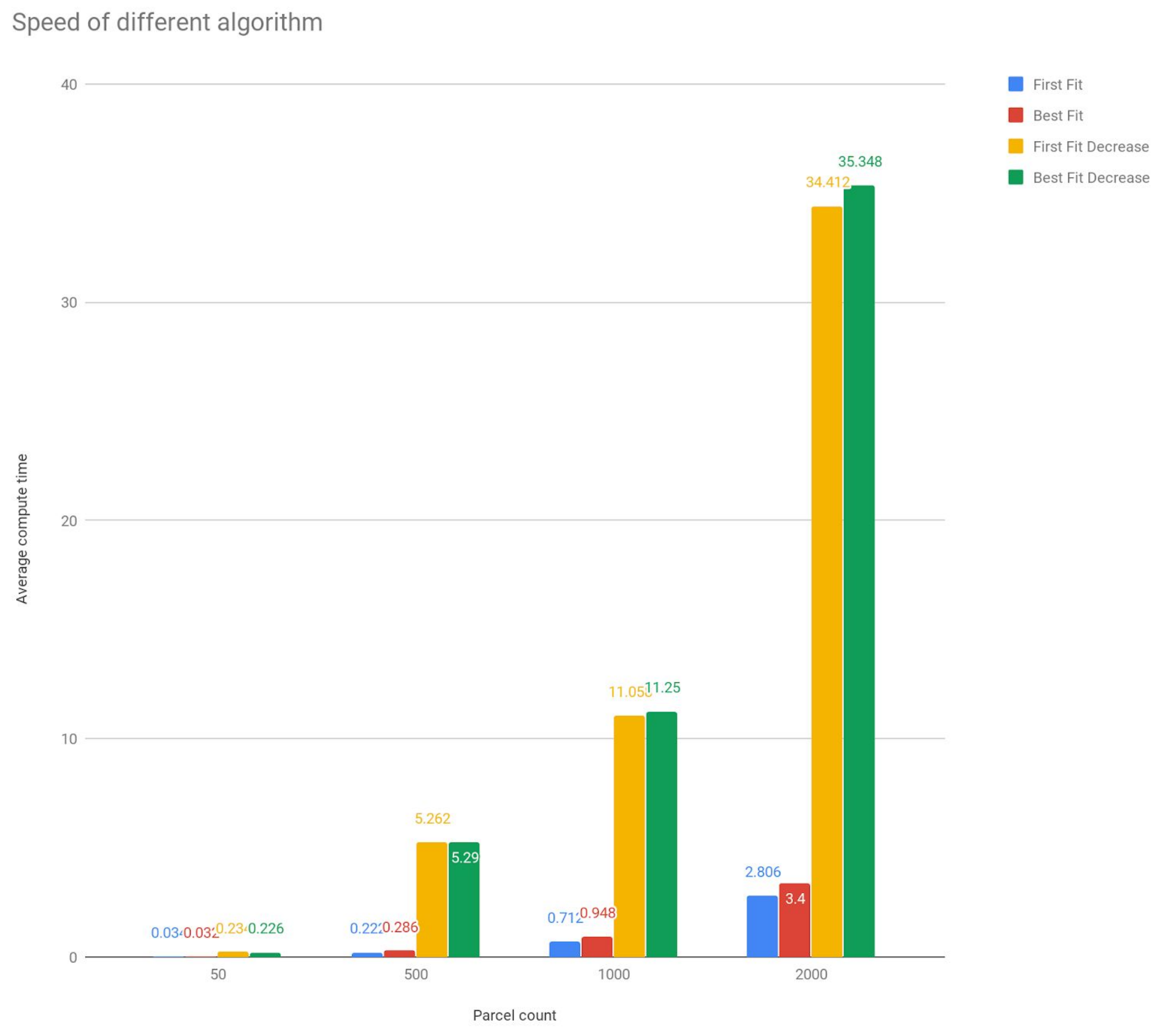
From the graph above, we can see that all the algorithm indeed have polynomial time complexity. All the shaded line are the line equation predicted by Google Sheet, and it is noticeable that all equations have the n^2 term. Thus, all of the algorithms does have time complexity of $O(n^2)$, and this proves that our aforementioned formulation is correct. Amongst all the 4 algorithms, Best Fit Decrease has the highest time complexity as it is the nearest to the $\frac{n^2}{2}$ line, and First Fit has the lowest time complexity as it has the shallowest curve.

Speed of each algorithm

To analyze the speed of each algorithm, we have created a benchmark function (named `SpeedBenchmark`) to calculate the time needed to computer the solution given a set of random parcels. This process is repeated 500 times so that we can obtain an average compute time.

Algorithm	**Random parcels count**	**Average compute time (ms)**
First Fit	50	0.034
	500	0.222
	1000	0.712
	2000	2.806
Best Fit	50	0.032
	500	0.286
	1000	0.948
	2000	3.4
First Fit Decrease	50	0.234
	500	5.262
	1000	11.058
	2000	34.412
Best Fit Decrease	50	0.226
	500	5.290
	1000	11.25

	2000	35.348



From the graph above, it is obvious that Best Fit Decrease(BFD) and First Fit Decrease(FFD) are much more slower than Best Fit(BF) and First Fit(FF), and Best Fit is slower than First Fit when the input size is large. The reason that causes FFD and BFD takes longer time to compute is because they need to sort the input parcels first. In short, FF is the fastest algorithm for the bin-packing problem.

Efficiency of each algorithm

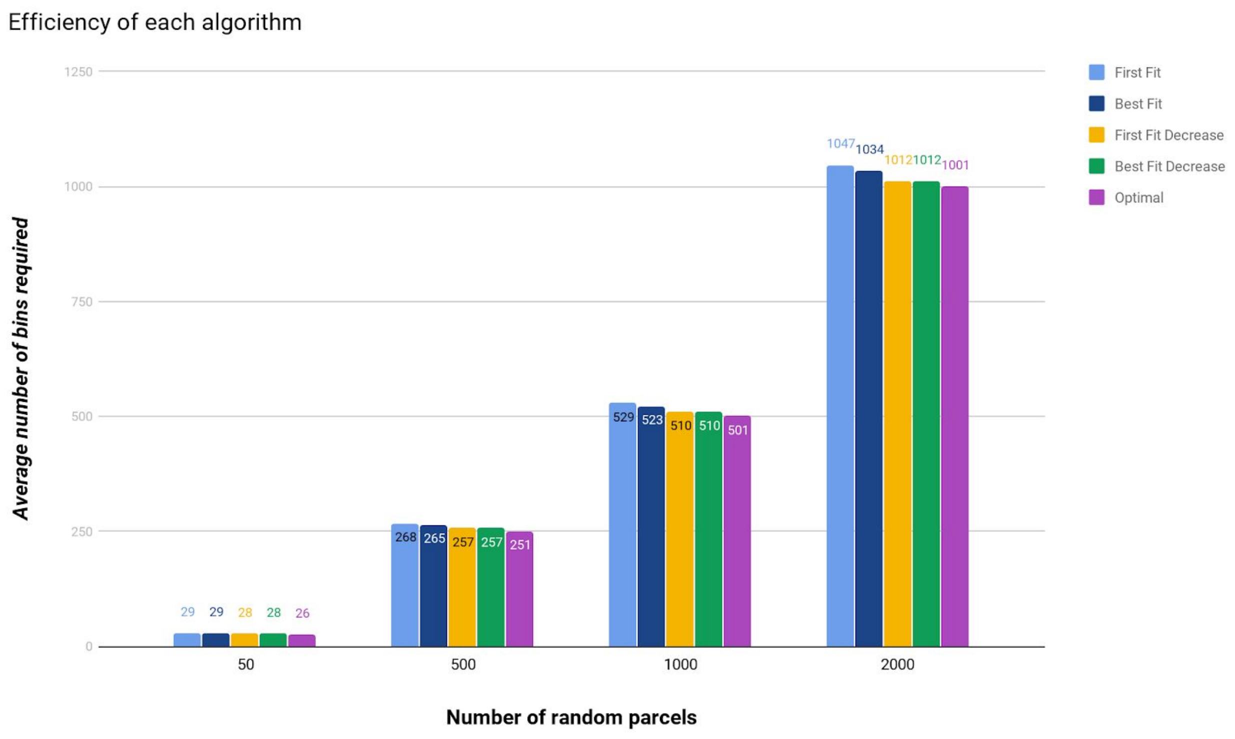
In the bin-packing problem, we will define *efficiency* as the ability to find the solution with the least number of bins required. To test the efficiency of each algorithms (FF, BF, FFD, BFD), we had created a benchmarking method (named `EfficiencyBenchmark`) that will generate a random maximum capacity, and an array of random parcels with random weight that does not exceed the maximum capacity, then it will run the inputs against each of the algorithm. This process is then repeated 500 times, so that we can obtain the average number of bins required for each algorithm. Moreover, to find out if the algorithm is really efficient, we created an optimal solver (which is impossible to implement) that will calculate the lower bound of a given input. Then, we can find out which algorithm produces results closest to the optimal result. The table on the next page depicts the results.

Legend: FF = First Fit, BF = Best Fit, FFD = First Fit Decrease, BFD = Best Fit Decrease, OP = Optimal

Random parcels count	**Algorithm**	**Average bins required**
50	FF	29
	BF	29
	FFD	28
	BFD	28
	OP	26
500	FF	268
	BF	265
	FFD	257
	BFD	257
	OP	251
1000	FF	529
	BF	523
	FFD	510
	BFD	510
	OP	501
2000	FF	1047
	BF	1034
	FFD	1012
	BFD	1012

	OP	1001

Table: Efficiency of each algorithm

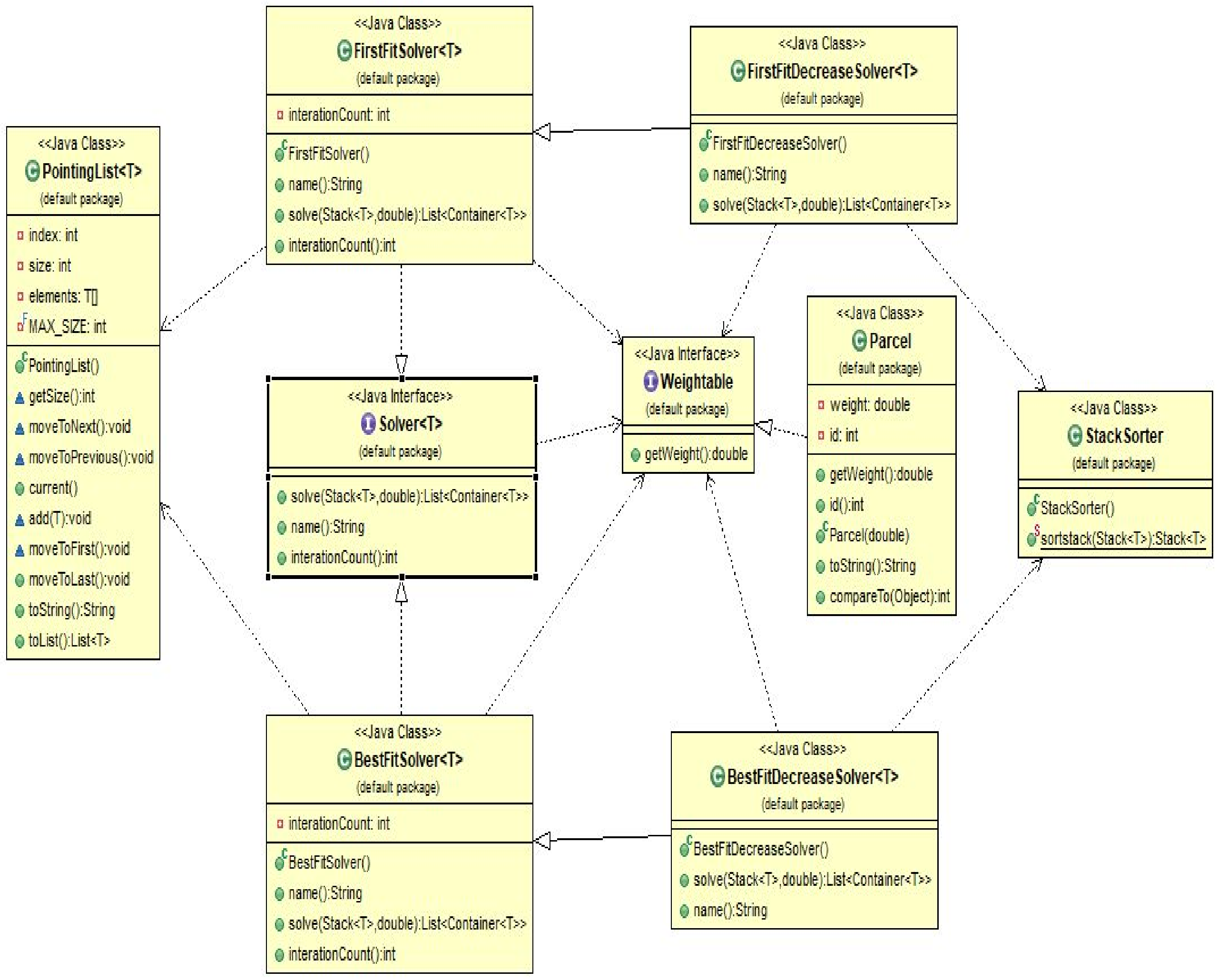


From the graph above, we can see that Best Fit Decrease and First Fit Decrease perform best regardless of the input size, and Best Fit is always better than First Fit. Thus, we can see that sorting the parcels descendingly first will improve the efficiency of the algorithm. Furthermore, it is also noticeable that when the number of inputs (number of random parcels) is getting larger, the solution computed by each algorithm sway further from the optimal algorithm. This indicates that the algorithms will better when the input size is smaller.

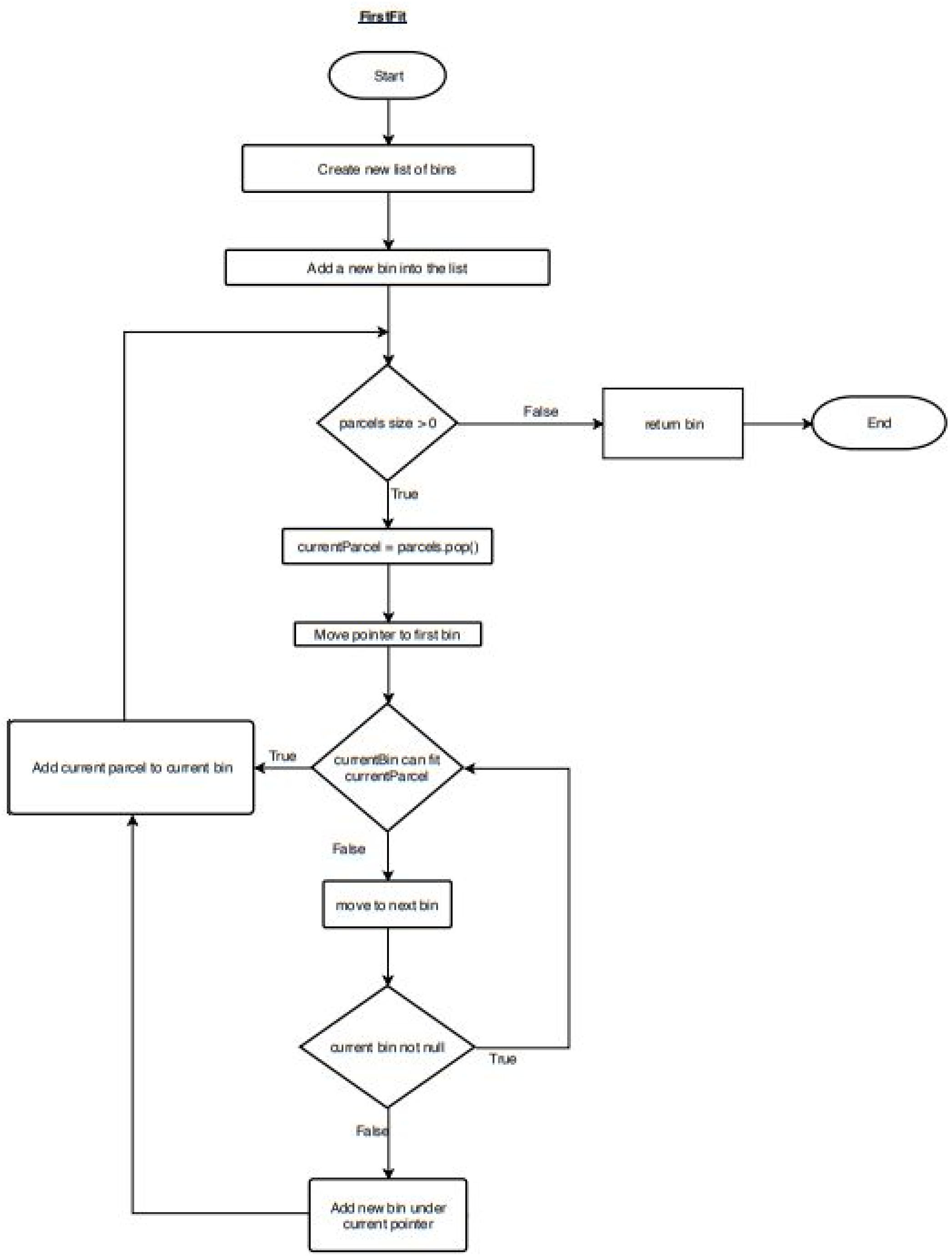
Summary

In summary FF and BF is much more faster than FFD and BFD, but FFD and BFD will give a better solution than FF and BF. This is a trade-off between speed and quality, just like food, fast food restaurant can serve you the food very quickly, but the food quality might not be very good; Michelin-rated restaurant take longer time to serves food, but the food quality is better than that of fast food. However, since the bin-packing problem urge for the solution with the best quality (minimum number of bins), we can concluded that FFD and BFD is indeed the best algorithm amongst the four algorithm discussed.

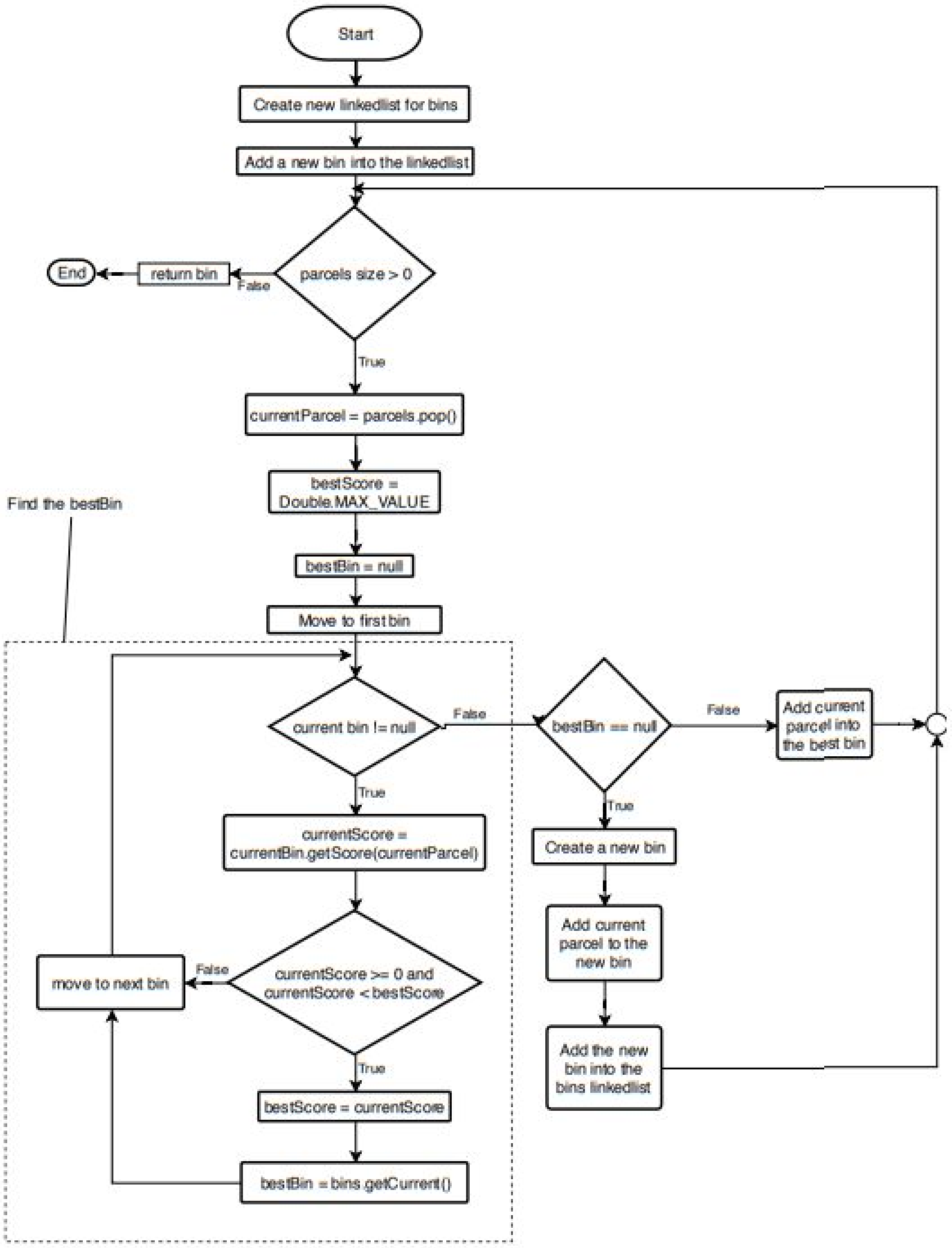
UML Diagram


<<Java Class>>
FirstFitSolver<T>
(default package)
interationCount: int
FirstFitSolver()
name():String
solve(Stack<T>,double):List<Container<T>>
interationCount():int
<<Java Class>>
FirstFitDecreaseSolver<T>
(default package)
FirstFitDecreaseSolver()
name():String
solve(Stack<T>,double):List<Container<T>>
<<Java Class>>
PointingList<T>
(default package)
index: int
size: int
elements: T[]
MAX_SIZE: int
PointingList()
getSize():int
moveToNext():void
moveToPrevious():void
current()
add(T):void
moveToFirst():void
moveToLast():void
toString():String
toList():List<T>
<<Java Interface>>
Solver<T>
(default package)
solve(Stack<T>,double):List<Container<T>>
name():String
interationCount():int
<<Java Interface>>
Weightable
(default package)
getWeight():double
<<Java Class>>
Parcel
(default package)
weight: double
id: int
getWeight():double
id():int
Parcel(double)
toString():String
compareTo(Object):int
<<Java Class>>
StackSorter
(default package)
StackSorter()
sortstack(Stack<T>):Stack<T>
<<Java Class>>
BestFitSolver<T>
(default package)
interationCount: int
BestFitSolver()
name():String
solve(Stack<T>,double):List<Container<T>>
interationCount():int
<<Java Class>>
BestFitDecreaseSolver<T>
(default package)
BestFitDecreaseSolver()
solve(Stack<T>,double):List<Container<T>>
name():String

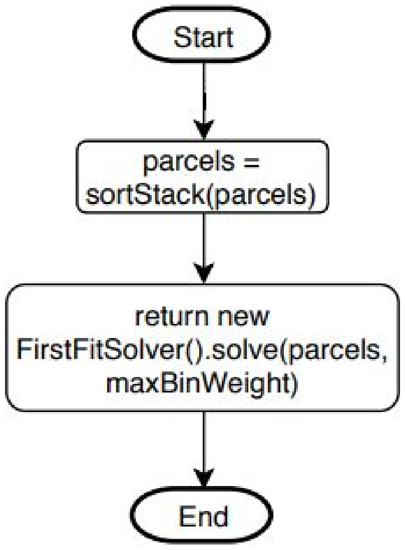
Flowchart


FirstFit
Start
Create new list of bins
Add a new bin into the list
parcels size > 0
False
return bin
End
True
currentParcel = parcels.pop()
Move pointer to first bin
currentBin can fit currentParcel
True
Add current parcel to current bin
False
move to next bin
current bin not null
True
False
Add new bin under current pointer

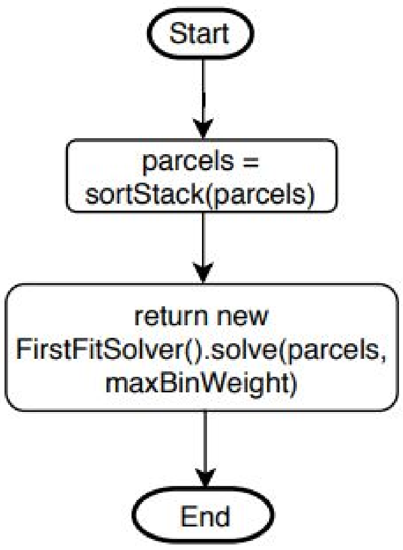
BestFit


Start
Create new linkedlist for bins
Add a new bin into the linkedlist
parcels size > 0
False
return bin
End
True
currentParcel = parcels.pop()
bestScore = Double.MAX_VALUE
bestBin = null
Move to first bin
Find the bestBin
current bin != null
False
True
currentScore = currentBin.getScore(currentParcel)
currentScore >= 0 and currentScore < bestScore
False
move to next bin
True
bestScore = currentScore
bestBin = bins.getCurrent()
bestBin == null
False
Add current parcel into the best bin
True
Create a new bin
Add current parcel to the new bin
Add the new bin into the bins linkedlist

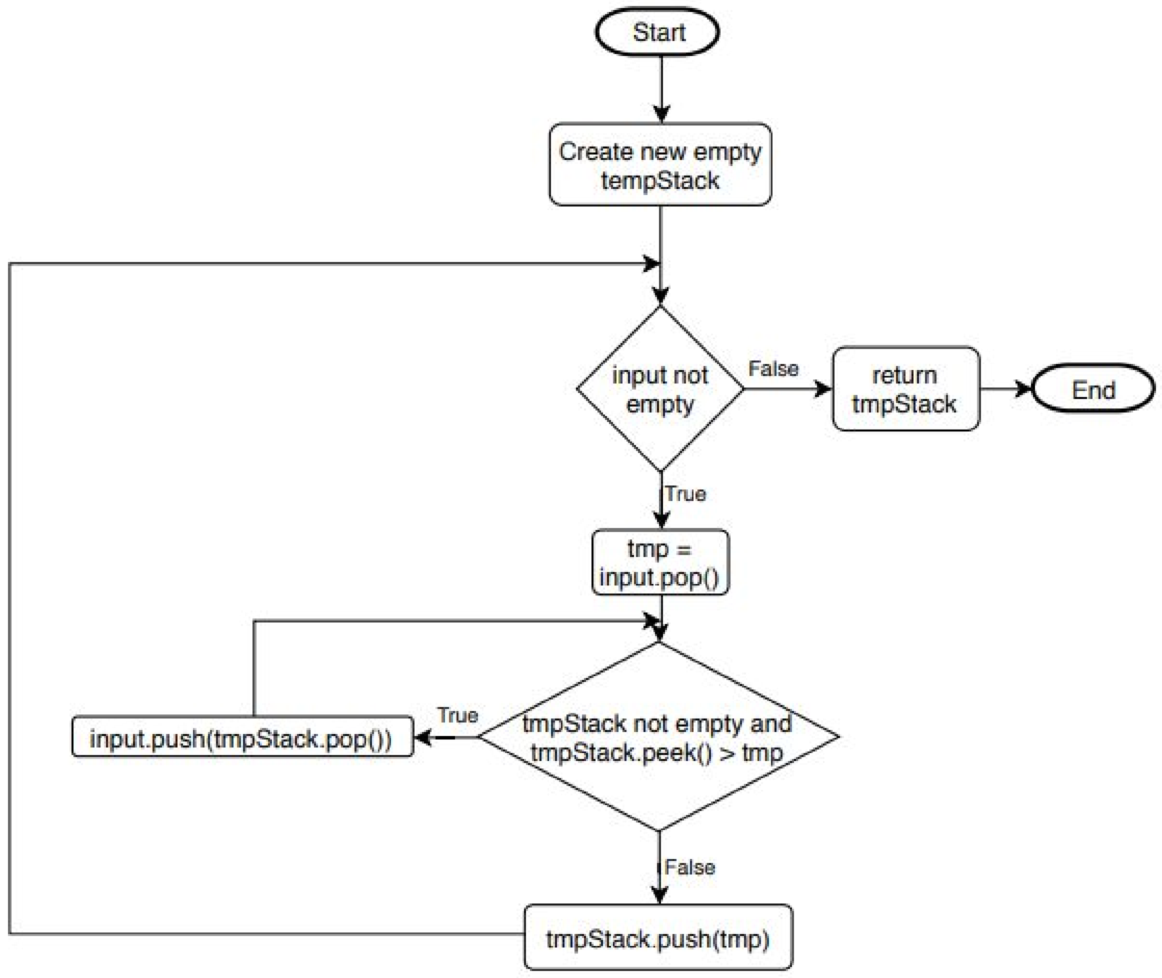
FirstFitDecrease


Start
parcels = sortStack(parcels)
return new FirstFitSolver().solve(parcels, maxBinWeight)
End

BestFitDecrease

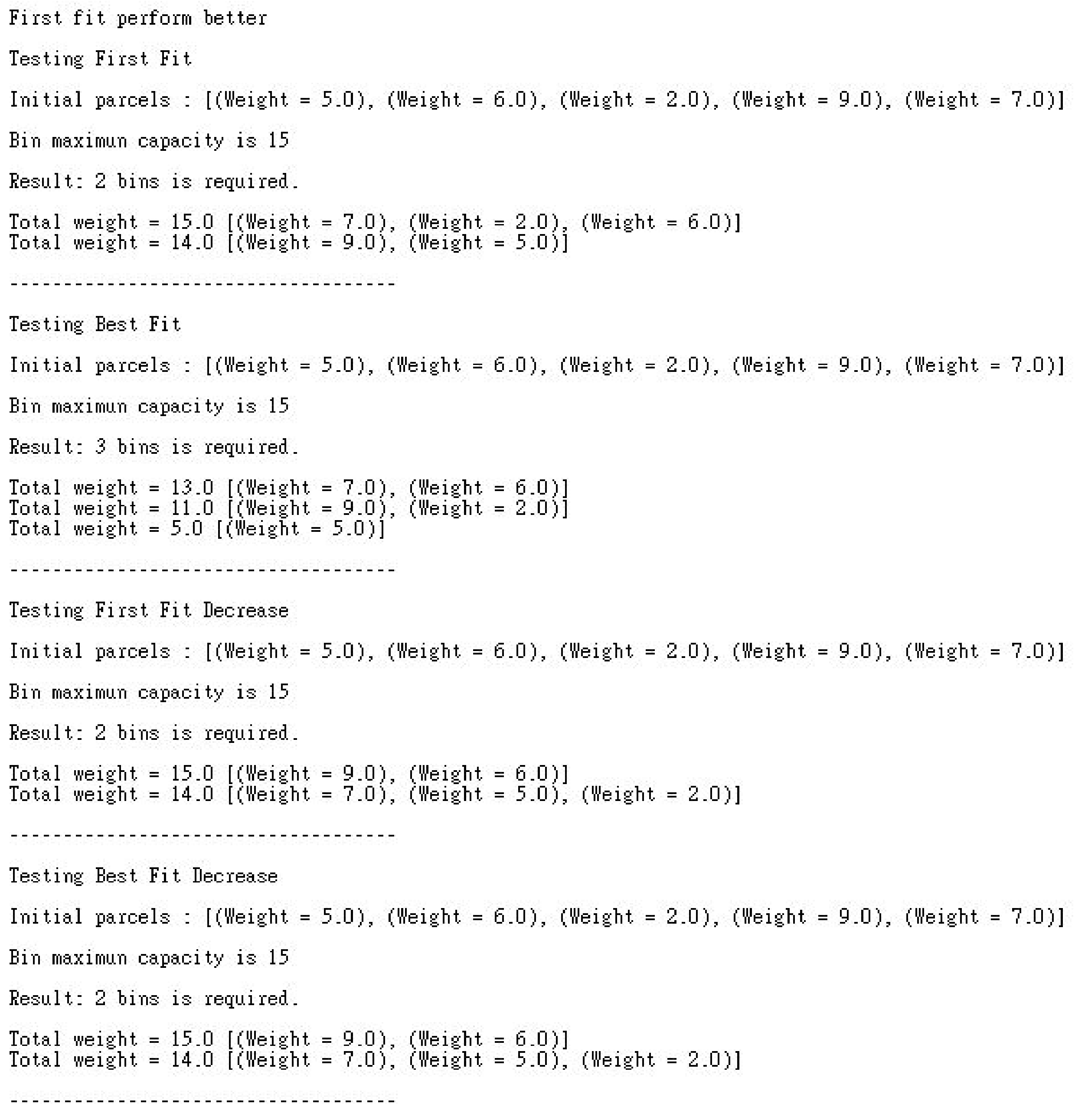

Start
parcels = sortStack(parcels)
return new FirstFitSolver().solve(parcels, maxBinWeight)
End

sortStack


Start
Create new empty tempStack
input not empty
False
return tmpStack
End
True
tmp = input.pop()
tmpStack not empty and tmpStack.peek() > tmp
True
input.push(tmpStack.pop())
False
tmpStack.push(tmp)

Sample input and output

Demo.java



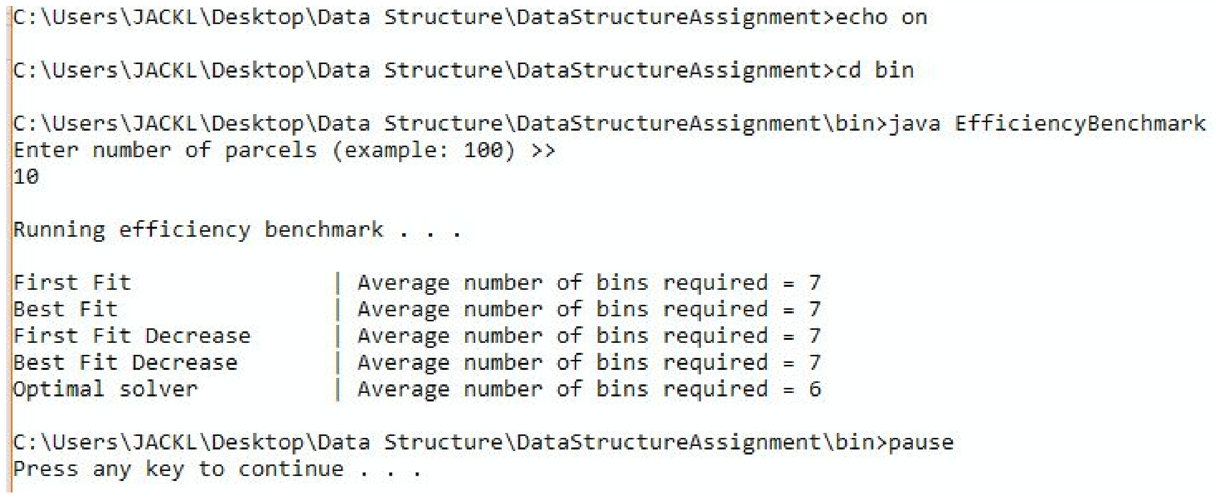
```
First fit perform better

Testing First Fit

Initial parcels : [(Weight = 5.0), (Weight = 6.0), (Weight = 2.0), (Weight = 9.0), (Weight = 7.0)]

Bin maximun capacity is 15

Result: 2 bins is required.

Total weight = 15.0 [(Weight = 7.0), (Weight = 2.0), (Weight = 6.0)]
Total weight = 14.0 [(Weight = 9.0), (Weight = 5.0)]

------------------------------------

Testing Best Fit

Initial parcels : [(Weight = 5.0), (Weight = 6.0), (Weight = 2.0), (Weight = 9.0), (Weight = 7.0)]

Bin maximun capacity is 15

Result: 3 bins is required.

Total weight = 13.0 [(Weight = 7.0), (Weight = 6.0)]
Total weight = 11.0 [(Weight = 9.0), (Weight = 2.0)]
Total weight = 5.0 [(Weight = 5.0)]

------------------------------------

Testing First Fit Decrease

Initial parcels : [(Weight = 5.0), (Weight = 6.0), (Weight = 2.0), (Weight = 9.0), (Weight = 7.0)]

Bin maximun capacity is 15

Result: 2 bins is required.

Total weight = 15.0 [(Weight = 9.0), (Weight = 6.0)]
Total weight = 14.0 [(Weight = 7.0), (Weight = 5.0), (Weight = 2.0)]

------------------------------------

Testing Best Fit Decrease

Initial parcels : [(Weight = 5.0), (Weight = 6.0), (Weight = 2.0), (Weight = 9.0), (Weight = 7.0)]

Bin maximun capacity is 15

Result: 2 bins is required.

Total weight = 15.0 [(Weight = 9.0), (Weight = 6.0)]
Total weight = 14.0 [(Weight = 7.0), (Weight = 5.0), (Weight = 2.0)]

------------------------------------
```

```
Best fit perform better

Testing First Fit

Initial parcels : [(Weight = 8.0), (Weight = 1.0), (Weight = 10.0), (Weight = 2.0), (Weight = 5.0)]

Bin maximun capacity is 15

Result: 3 bins is required.

Total weight = 8.0 [(Weight = 5.0), (Weight = 2.0), (Weight = 1.0)]
Total weight = 10.0 [(Weight = 10.0)]
Total weight = 8.0 [(Weight = 8.0)]

------------------------------------

Testing Best Fit

Initial parcels : [(Weight = 8.0), (Weight = 1.0), (Weight = 10.0), (Weight = 2.0), (Weight = 5.0)]

Bin maximun capacity is 15

Result: 2 bins is required.

Total weight = 15.0 [(Weight = 5.0), (Weight = 2.0), (Weight = 8.0)]
Total weight = 11.0 [(Weight = 10.0), (Weight = 1.0)]

------------------------------------

Testing First Fit Decrease

Initial parcels : [(Weight = 8.0), (Weight = 1.0), (Weight = 10.0), (Weight = 2.0), (Weight = 5.0)]

Bin maximun capacity is 15

Result: 2 bins is required.

Total weight = 15.0 [(Weight = 10.0), (Weight = 5.0)]
Total weight = 11.0 [(Weight = 8.0), (Weight = 2.0), (Weight = 1.0)]

------------------------------------

Testing Best Fit Decrease

Initial parcels : [(Weight = 8.0), (Weight = 1.0), (Weight = 10.0), (Weight = 2.0), (Weight = 5.0)]

Bin maximun capacity is 15

Result: 2 bins is required.

Total weight = 15.0 [(Weight = 10.0), (Weight = 5.0)]
Total weight = 11.0 [(Weight = 8.0), (Weight = 2.0), (Weight = 1.0)]

------------------------------------
```

```
Best fit and first fit will perform equally

Testing First Fit

Initial parcels : [(Weight = 8.0), (Weight = 1.0), (Weight = 7.0), (Weight = 2.0), (Weight = 7.0)]

Bin maximun capacity is 15

Result: 2 bins is required.

Total weight = 10.0 [(Weight = 7.0), (Weight = 2.0), (Weight = 1.0)]
Total weight = 15.0 [(Weight = 7.0), (Weight = 8.0)]

------------------------------------

Testing Best Fit

Initial parcels : [(Weight = 8.0), (Weight = 1.0), (Weight = 7.0), (Weight = 2.0), (Weight = 7.0)]

Bin maximun capacity is 15

Result: 2 bins is required.

Total weight = 10.0 [(Weight = 7.0), (Weight = 2.0), (Weight = 1.0)]
Total weight = 15.0 [(Weight = 7.0), (Weight = 8.0)]

------------------------------------

Testing First Fit Decrease

Initial parcels : [(Weight = 8.0), (Weight = 1.0), (Weight = 7.0), (Weight = 2.0), (Weight = 7.0)]

Bin maximun capacity is 15

Result: 2 bins is required.

Total weight = 15.0 [(Weight = 8.0), (Weight = 7.0)]
Total weight = 10.0 [(Weight = 7.0), (Weight = 2.0), (Weight = 1.0)]

------------------------------------

Testing Best Fit Decrease

Initial parcels : [(Weight = 8.0), (Weight = 1.0), (Weight = 7.0), (Weight = 2.0), (Weight = 7.0)]

Bin maximun capacity is 15

Result: 2 bins is required.

Total weight = 15.0 [(Weight = 8.0), (Weight = 7.0)]
Total weight = 10.0 [(Weight = 7.0), (Weight = 2.0), (Weight = 1.0)]

------------------------------------
```

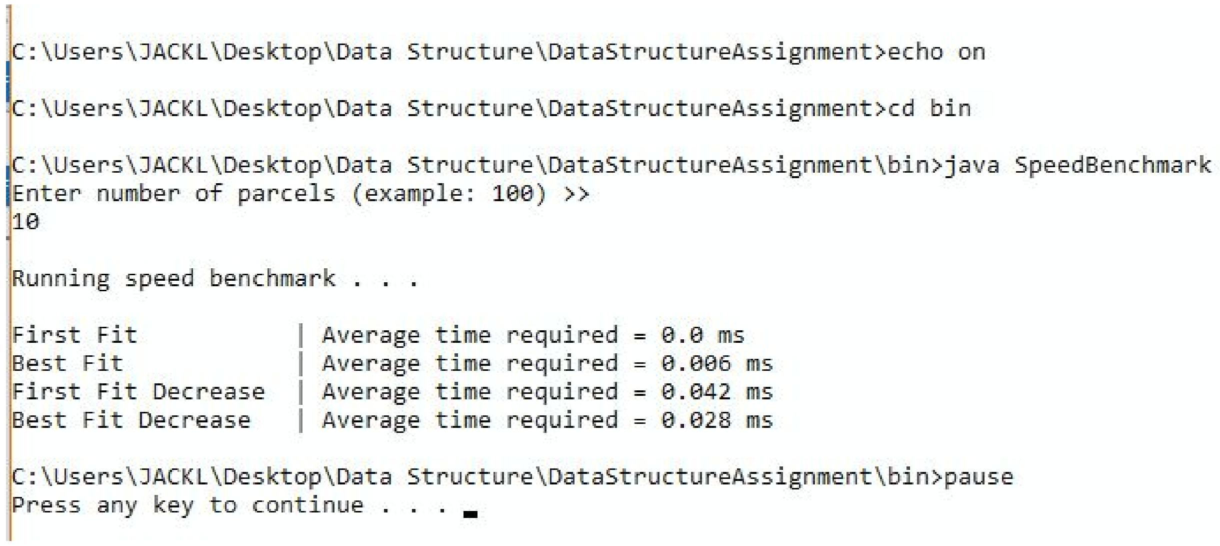
```
First fit decrease perform best

Testing First Fit

Initial parcels : [(Weight = 3.0), (Weight = 13.0), (Weight = 2.0), (Weight = 12.0)]

Bin maximun capacity is 15

Result: 3 bins is required.

Total weight = 14.0 [(Weight = 12.0), (Weight = 2.0)]
Total weight = 13.0 [(Weight = 13.0)]
Total weight = 3.0 [(Weight = 3.0)]

------------------------------------

Testing Best Fit

Initial parcels : [(Weight = 3.0), (Weight = 13.0), (Weight = 2.0), (Weight = 12.0)]

Bin maximun capacity is 15

Result: 3 bins is required.

Total weight = 14.0 [(Weight = 12.0), (Weight = 2.0)]
Total weight = 13.0 [(Weight = 13.0)]
Total weight = 3.0 [(Weight = 3.0)]

------------------------------------

Testing First Fit Decrease

Initial parcels : [(Weight = 3.0), (Weight = 13.0), (Weight = 2.0), (Weight = 12.0)]

Bin maximun capacity is 15

Result: 2 bins is required.

Total weight = 15.0 [(Weight = 13.0), (Weight = 2.0)]
Total weight = 15.0 [(Weight = 12.0), (Weight = 3.0)]

------------------------------------

Testing Best Fit Decrease

Initial parcels : [(Weight = 3.0), (Weight = 13.0), (Weight = 2.0), (Weight = 12.0)]

Bin maximun capacity is 15

Result: 2 bins is required.

Total weight = 15.0 [(Weight = 13.0), (Weight = 2.0)]
Total weight = 15.0 [(Weight = 12.0), (Weight = 3.0)]

------------------------------------
```

EfficiencyBenchmark.java



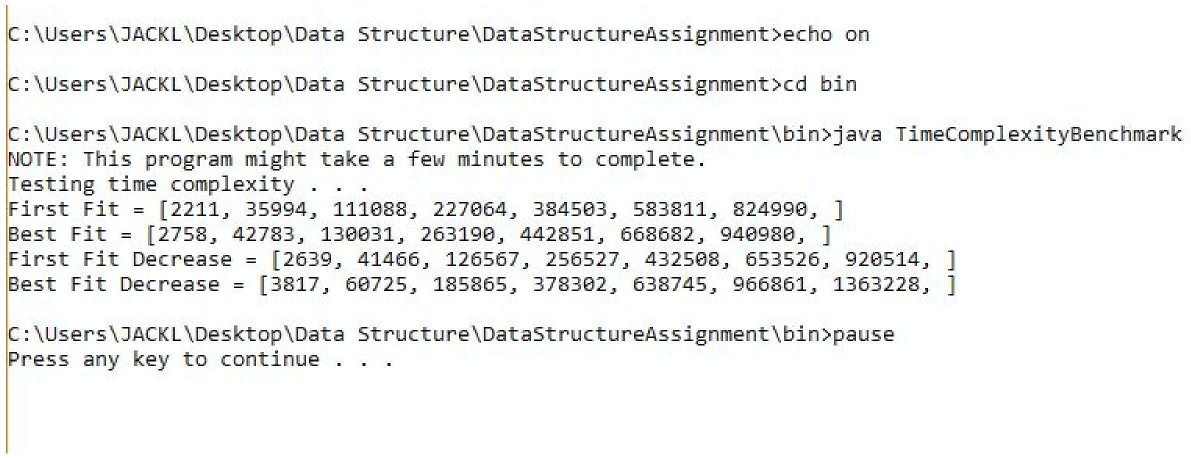
```
C:\Users\JACKL\Desktop\Data Structure\DataStructureAssignment>echo on

C:\Users\JACKL\Desktop\Data Structure\DataStructureAssignment>cd bin

C:\Users\JACKL\Desktop\Data Structure\DataStructureAssignment\bin>java EfficiencyBenchmark
Enter number of parcels (example: 100) >>
10

Running efficiency benchmark . . .

First Fit                | Average number of bins required = 7
Best Fit                 | Average number of bins required = 7
First Fit Decrease       | Average number of bins required = 7
Best Fit Decrease        | Average number of bins required = 7
Optimal solver           | Average number of bins required = 6

C:\Users\JACKL\Desktop\Data Structure\DataStructureAssignment\bin>pause
Press any key to continue . . .
```

SpeedBenchmark.java



```
C:\Users\JACKL\Desktop\Data Structure\DataStructureAssignment>echo on

C:\Users\JACKL\Desktop\Data Structure\DataStructureAssignment>cd bin

C:\Users\JACKL\Desktop\Data Structure\DataStructureAssignment\bin>java SpeedBenchmark
Enter number of parcels (example: 100) >>
10

Running speed benchmark . . .

First Fit            | Average time required = 0.0 ms
Best Fit             | Average time required = 0.006 ms
First Fit Decrease   | Average time required = 0.042 ms
Best Fit Decrease    | Average time required = 0.028 ms

C:\Users\JACKL\Desktop\Data Structure\DataStructureAssignment\bin>pause
Press any key to continue . . .
```

TimeComplexityBenchmark.java



```
C:\Users\JACKL\Desktop\Data Structure\DataStructureAssignment>echo on

C:\Users\JACKL\Desktop\Data Structure\DataStructureAssignment>cd bin

C:\Users\JACKL\Desktop\Data Structure\DataStructureAssignment\bin>java TimeComplexityBenchmark
NOTE: This program might take a few minutes to complete.
Testing time complexity . . .
First Fit = [2211, 35994, 111088, 227064, 384503, 583811, 824990, ]
Best Fit = [2758, 42783, 130031, 263190, 442851, 668682, 940980, ]
First Fit Decrease = [2639, 41466, 126567, 256527, 432508, 653526, 920514, ]
Best Fit Decrease = [3817, 60725, 185865, 378302, 638745, 966861, 1363228, ]

C:\Users\JACKL\Desktop\Data Structure\DataStructureAssignment\bin>pause
Press any key to continue . . .
```

References

Amit. 2015. *algorithm - time complexity for a sorting technique for stacks? - Stack Overflow*. [ONLINE] Available at: https://stackoverflow.com/questions/34384823/time-complexity-for-a-sorting-technique-for-stacks. [Accessed 26 August 2018].

Korf, R.E., 2002, July. A new algorithm for optimal bin packing. In AAAI/IAAI (pp. 731-736).