We created a new method in Optimisation.java to implement our local-search optimisation algorithms.

```
private static int[] generateNeighbour(int[] packingPlan, i
nt possibleOperation) {}
```

In the method, we actually implemented 3 different local-search algorithms.

This method takes in 2 arguments, packingPlan (the current packing plan) and possibleOperation (an integer indicating which specific local-search algorithm to use)

In each iteration, before the application of any of the following 3 algorithms, there is a process of possibly assigning a position in packingPlan array to 1. This is important because all the following algorithms would only change the sequence of elements of the packingList, but not change the number of 0s and 1s. A process to increase the 1s in packingList is therefore necessary

Local-search Algorithm 1: Exchange

```
int loop=0;
do {
    Random rand_index = new Random();
    index_1 = rand_index.nextInt(neighbour.length);
    index_2 = rand_index.nextInt(neighbour.length);
    loop++;
} while (index_1 == index_2 || (neighbour[index_1]==neighbo
ur[index_2] && loop<=10));

 // Swap two elements
 temp = neighbour[index_1];
 neighbour[index_1] = neighbour[index_2];
 neighbour[index_2] = temp;
```

This algorithm randomly choose 2 positions (index_1 and index_2) in the packingPlan array, and exchange their values with each other.

If index_1 == index_2, which they are in fact the same position, go back to regenerate 2 numbers.

If the packingPlan values at index_1 and index_2 are the same (which is highly possible because the packingPlan has only 0 and 1 values), go back to regenerate 2 numbers.

However, considering the high possibility, especially in the beginning, that two values in packingPlan array are the same (both are 0), it is necessary to to limit the search of different values to 10 loops every time to avoid wasting too much time.

Local-search Algorithm 2: Jump

```
do {
    Random rand_index = new Random();
    index_1 = rand_index.nextInt(neighbour.length);
    index_2 = rand_index.nextInt(neighbour.length);
} while (index_1 == index_2);

// Jump element at index_1 and shift other elements
if (index_1 < index_2) {
    temp = neighbour[index_1];
    System.arraycopy(neighbour, index_1 + 1, neighbour, ind
ex_1, index_2 - index_1);
    neighbour[index_2] = temp;
} else {
    temp = neighbour[index_1];
    System.arraycopy(neighbour, index_2, neighbour, index_2
 + 1, index_1 - index_2);
    neighbour[index_2] = temp;
}
```

This randomly generates 2 integers representing 2 different positions in the packingPlan array, move everything between index_1 and index_2 away from index_1 for 1 position, and move the value previously at index_1 to the new empty position near index_2.

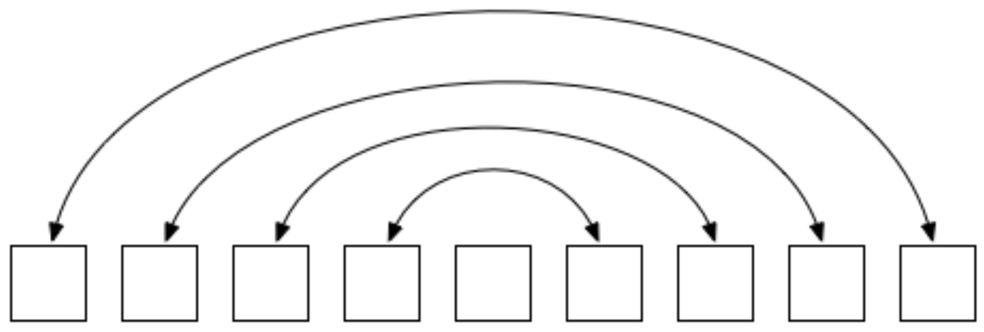
Local-search Algorithm 3: Inversion

```
do {
    Random rand_index = new Random();
    index_1 = rand_index.nextInt(neighbour.length);
    index_2 = rand_index.nextInt(neighbour.length);
} while (index_1 == index_2);

int lower_index = Math.min(index_1, index_2);
int bigger_index = Math.max(index_1, index_2);

//Do invert the sub elements among these two indices
while (lower_index <= bigger_index) {
    temp = neighbour[lower_index];
    neighbour[lower_index] = neighbour[bigger_index];
    neighbour[bigger_index] = temp;
    lower_index++;
    bigger_index--;
}
break;
```

Again, randomly select two different positions in packingPlan, treat everything between index_1 and index_2 as a new array and flip it: exchange the first and last element, then exchange the 2nd and 2nd last element, and so on.



Inversion

Some Reflections:

In our current implementation, the use of these 3 local-search algorithms is 1:1:1, because the possibleOperation argument is randomly generated between 1, 2 and 3. We have tried different mixes, e.g. give the 1st algorithm better possibility of being used. These attempts generated slightly different results, better in some parts but worse in others. From current results, we consider the impact of the portion this mix is minor, therefore we keep on using these 3 algorithms in 1:1:1.