OptimalJobScheduling.java

```
1   package com.example;
2
3   public class OptimalJobScheduling {
4
5       private final int numberProcesses;
6       private final int numberMachines;
7       private final int[][] Run;
8       private final int[][] Transfer;
9       private final int[][] Cost;
10
11      /**
12       * Constructor of the class.
13       * @param numberProcesses ,refers to the number of precedent processes(N)
14       * @param numberMachines ,refers to the number of different machines in our disposal(M)
15       * @param Run , N*M matrix refers to the cost of running each process to each machine
16       * @param Transfer ,M*M symmetric matrix refers to the transportation delay for each pair of
17       *     machines
18       */
19      public OptimalJobScheduling(int numberProcesses, int numberMachines, int[][] Run, int[][] Transfer) {
20          this.numberProcesses = numberProcesses;
21          this.numberMachines = numberMachines;
22          this.Run = Run;
23          this.Transfer = Transfer;
24          this.Cost = new int[numberProcesses][numberMachines];
25      }
26
27      /**
28       * Function which computes the cost of process scheduling to a number of VMs.
29       */
30      public void execute() {
31  1       this.calculateCost();
32  1       this.showResults();
33      }
34
35      /**
36       * Function which computes the cost of running each Process to each and every Machine
37       */
38      private void calculateCost() {
39
40  3       for (int i = 0; i < numberProcesses; i++) { // for each Process
41
42  3           for (int j = 0; j < numberMachines; j++) { // for each Machine
43
44                  Cost[i][j] = runningCost(i, j);
45              }
46          }
47      }
48
49      /**
50       * Function which returns the minimum cost of running a certain Process to a certain Machine.In
51       * order for the Machine to execute the Process ,he requires the output of the previously
52       * executed Process, which may have been executed to the same Machine or some other.If the
53       * previous Process has been executed to another Machine,we have to transfer her result, which
54       * means extra cost for transferring the data from one Machine to another(if the previous
55       * Process has been executed to the same Machine, there is no transport cost).
56       *
57       * @param process ,refers to the Process
58       * @param machine ,refers to the Machine
59       * @return the minimum cost of executing the process to the certain machine.
60       */
61      private int runningCost(int process, int machine) {
62
63  1       if (process == 0) // refers to the first process,which does not require for a previous one
64                            // to have been executed
65  1           return Run[process][machine];
66          else {
67
68              int[] runningCosts = new int[numberMachines]; // stores the costs of executing our Process depending on
69                                                            // the Machine the previous one was executed
70
71  3           for (int k = 0; k < numberMachines; k++) // computes the cost of executing the previous
72                                                       // process to each and every Machine
73  3               runningCosts[k] = Cost[process - 1][k] + Transfer[k][machine] + Run[process][machine]; // transferring the result to our Machine and executing
74                                                                                                        // the Process to our Machine
75
76  1           return findMin(runningCosts); // returns the minimum running cost
77          }
78      }
79
80      /**
81       * Function used in order to return the minimum Cost.
82       * @param cost ,an Array of size M which refers to the costs of executing a Process to each
83       *     Machine
84       * @return the minimum cost
85       */
86      private int findMin(int[] cost) {
87
88          int min = 0;
89
90  3       for (int i = 1; i < cost.length; i++) {
91
92  2           if (cost[i] < cost[min]) min = i;
93          }
94  1       return cost[min];
95      }
96
97      /**
98       * Method used in order to present the overall costs.
99       */
100     private void showResults() {
101
102 2       for (int i = 0; i < numberProcesses; i++) {
103
104 3           for (int j = 0; j < numberMachines; j++) {
105 1               System.out.print(Cost[i][j]);
106 1               System.out.print(" ");
107             }
108
109 1           System.out.println();
110         }
111 1       System.out.println();
112     }
113
114     /**
```

```
115       * Getter for the running Cost of i process on j machine.
116       */
117      public int getCost(int process, int machine) {
118 1        return Cost[process][machine];
119      }
120  }
```

Mutations

31	1. removed call to com/example/OptimalJobScheduling::calculateCost → KILLED
32	1. removed call to com/example/OptimalJobScheduling::showResults → SURVIVED
40	1. changed conditional boundary → KILLED 2. Changed increment from 1 to -1 → KILLED 3. negated conditional → KILLED
42	1. changed conditional boundary → KILLED 2. Changed increment from 1 to -1 → KILLED 3. negated conditional → KILLED
63	1. negated conditional → KILLED
65	1. replaced int return with 0 for com/example/OptimalJobScheduling::runningCost → KILLED
71	1. changed conditional boundary → KILLED 2. Changed increment from 1 to -1 → KILLED 3. negated conditional → KILLED
73	1. Replaced integer subtraction with addition → KILLED 2. Replaced integer addition with subtraction → KILLED 3. Replaced integer addition with subtraction → KILLED
76	1. replaced int return with 0 for com/example/OptimalJobScheduling::runningCost → KILLED
90	1. changed conditional boundary → KILLED 2. Changed increment from 1 to -1 → KILLED 3. negated conditional → KILLED
92	1. changed conditional boundary → SURVIVED 2. negated conditional → KILLED
94	1. replaced int return with 0 for com/example/OptimalJobScheduling::findMin → KILLED
102	1. changed conditional boundary → KILLED 2. Changed increment from 1 to -1 → KILLED 3. negated conditional → SURVIVED
104	1. changed conditional boundary → KILLED 2. Changed increment from 1 to -1 → KILLED 3. negated conditional → SURVIVED
105	1. removed call to java/io/PrintStream::print → SURVIVED
106	1. removed call to java/io/PrintStream::print → SURVIVED
109	1. removed call to java/io/PrintStream::println → SURVIVED
111	1. removed call to java/io/PrintStream::println → SURVIVED
118	1. replaced int return with 0 for com/example/OptimalJobScheduling::getCost → KILLED

Active mutators

- BOOLEAN_FALSE_RETURN
- BOOLEAN_TRUE_RETURN
- CONDITIONALS_BOUNDARY_MUTATOR
- EMPTY_RETURN_VALUES
- INCREMENTS_MUTATOR
- INVERT_NEGS_MUTATOR
- MATH_MUTATOR
- NEGATE_CONDITIONALS_MUTATOR
- NULL_RETURN_VALUES
- PRIMITIVE_RETURN_VALS_MUTATOR
- VOID_METHOD_CALL_MUTATOR

Tests examined

- com.example.OptimalJobSchedulingTest.testOptimalJobScheduling1(com.example.OptimalJobSchedulingTest) (1 ms)
- com.example.OptimalJobSchedulingTest.testOptimalJobScheduling3(com.example.OptimalJobSchedulingTest) (1 ms)
- com.example.OptimalJobSchedulingTest.testOptimalJobScheduling2(com.example.OptimalJobSchedulingTest) (0 ms)

Report generated by PIT 1.5.0