1 General instructions

It is your task to design and implement a local search method for finding solutions to a process re-assignment problem that minimize the total cost and satisfy all constraints, as described in Section 2. The program will be run for 5 minutes (CPU time) each for a set of problem instances of varying sizes. (**NB: do not limit the runtime of your program yourself**).

To simplify the execution, your program has to **append** a newly found solution to the output file whenever it finds a better solution. Each solution should be written as a single line of text, as described below, **terminated by a newline character** "`\n`". This is to ensure that it is possible to terminate the program from outside at any time, while the best solution found until then is stored at the end of the output file. During program evaluation we will consider the last (bottom) solution to be the final solution of the program.

Note the following requirements for a successful submission.

1. All solutions that are written as output have to be feasible!
2. Programs that only write the exact same initial assignment (part of input) to the output file are considered incomplete and hence do not count as a successfully completed programming assignment!
3. The program submitted must implement (at least) one of the high-level local search methods (e.g., simulated annealing, tabu search, etc.) as described in the lecture **or** in case some local search method not described in the lectures is implemented, provide reference to the algorithm used (i.e., it is possible to choose a more advanced local search method, but then you must include a pointer to the algorithm)!
4. Do not use any third-party code or non-standard libraries apart from the code provided in the assignment package.

The program is only allowed to use a single thread. The choice of programming language may be either Java or Python 2.

Test your solution with two sets of similar instances (available in the assignment package): `dms_assignment1_small/` and `dms_assignment1_large/`. For each set, test how good solutions can be found in 1 minute and in 3 minutes. The two sets also contain further information on the instances insider their `README.txt`.

In addition to the problem instance sets you can find a Java/Python program that reads and parses an input file and a (binary) solution checker in the assignment package. The checker is tested to run on the department's Linux machines and also on the Linux machines in Maarintalo. Further information on how to use the checker can be found in its Readme file. Note that we cannot release the source code of the checker.

1.1 Program syntax

The program should be run with the command (all in one line)

`java ProcessAssignment <instance_file> <initial_solution_file> <output_file>`

OR

`python ProcessAssignment.py <instance_file> <initial_solution_file> <output_file>`

where `<instance_file>` is the file name of the problem instance, `<initial_solution_file>` is the filename of the initial process-to-machine assignment and `<output_file>` is the name of the output file to which the best solution found is to be written. **Please make sure that your program can**

be called using exactly the syntax above. This is because we are using an automated system to check submissions and a different syntax would cause it to fail.

You can assume that the following versions of Java and Python are installed:

```
java -version
java version "1.7.0_65"
OpenJDK Runtime Environment (IcedTea 2.5.3) (7u71-2.5.3-2~deb7u1)
OpenJDK 64-Bit Server VM (build 24.65-b04, mixed mode)

python --version
Python 2.7.3
```

1.2 Deliverables

Your submission must contain two files, both are submitted through the Stratum system.

- *Commented* source code; and
- A short (max 3 A4 pages) report in pdf or ASCII text format that outlines your approach, lists your results for the two instance sets and briefly describes the computer (clock speed, memory) used. It is recommended to use the latex template provided in the assignment package.

The deadline for submissions is

March 1 (23:59).

Your code should be tidy and sufficiently commented to allow to easily understand what it does. Easy-to-read code and comments usually help troubleshooting problems in the submitted program.

1.3 Grading

The points in the Stratum system do not correspond to the final points you will receive. In the Stratum system the check is two-phase (automated / manual) for both the code and report. The points have the following meaning (see Checker output for more information):

0 points: No submission / submission not yet graded / failed submission

1 points: Submission has passed automated check (code is working) / submission has the right format (pdf/txt for report), manual check not yet done (e.g. style and commenting for code) / manual check failed

2 points: Submission has passed both manual and automated checks.

After both your report and code have passed both checks, your solution will be graded as follows. A correct solution on first try with a sufficient report earns 3 points. An additional 3 points are distributed according to a competition of the programs depending on the quality of the solutions obtained.

3 points: Adequate work: Solution is correct and report sufficient.

4 points: Same as above but quality of solutions are in the top 3/4 of all submissions.

5 points: Same as above but quality of solutions are in the top 1/2 of all submissions.

6 points: Same as above but quality of solutions are in the top 1/4 of all submissions.

Here, *quality of solutions* is the function

$$\sum_{I \in X} \frac{\text{MLCost}(A_I) - \text{TotalCost}(A_I, A_O)}{\text{MLCost}(A_I)},$$

where X is a set of instances different from (but similar to) the public ones and all other terms are described below. If the submitted solution does not work correctly, the source code is unreadable or not sufficiently commented, or the report is unclear or missing, you are asked to correct and resubmit your solution. Each re-submission is due within one week (in case a resubmission is needed, you will receive further instructions and a deadline for the resubmission) and decreases the maximum number of points achievable by 1.

The solutions submitted after the deadline (March 1, 23:59) are excluded from the competition, i.e., maximum number of points for a late submission is 3.

2 Problem description

We first discuss the problem informally and then give a formal definition. A number of *processes* are assigned to *machines* located in a datacenter. The assignment has to satisfy three types of constraints (restrictions).

Each process consumes a certain amount of *resources* that are available at the machines (e.g., CPU, RAM). For each resource and each machine there is an available *capacity* that the total amount of resource consumption of processes assigned to that machine must not exceed. For example, the total amount of memory consumed by the processes assigned to a specific machine must not exceed the machine's total amount of available memory. These are the *hard (machine) capacity constraints* (MCCon). **NB:** Capacities are assumed to be dimensionless (no units such as MHz, GB, MB, etc.).

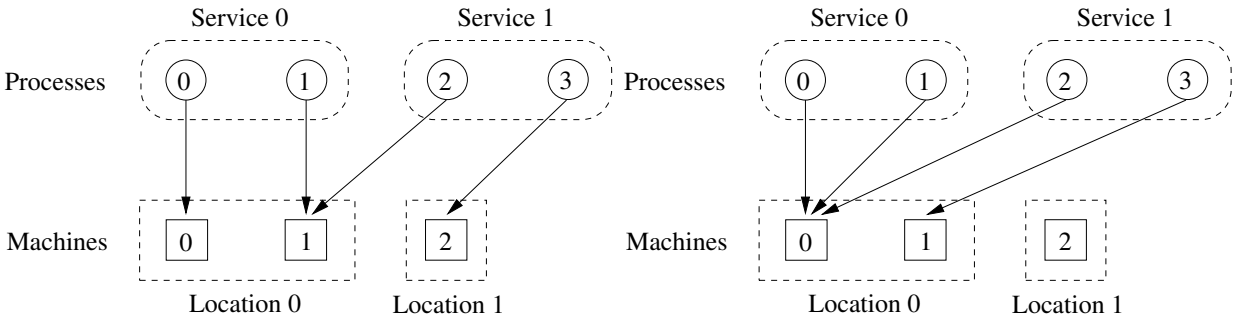
The set of processes is divided (partitioned) into *services* (e.g., email service, web service). Two processes that belong to the same service must not be assigned to the same machine. These are the *service conflict constraints*. (SCCon) **NB:** Services are disjoint.

The set of machines are partitioned into *locations*. Each service is required to operate at least within a certain number of different locations. The parameter that determines the number of different locations the service is required to operate in is called *minSpread*. These are the *service spread constraints*. (SSCon) **NB1:** Locations are disjoint. **NB2:** A service counts as operating in a certain location, if at least one of its processes is assigned to a machine in that location.

Figure 1 shows an example with four processes divided into two services and three machines divided into two locations. For the sake of the example, let us ignore capacity constraints. Here, Service 0 has a minimum spread of 1 and Service 1 has a minimum spread of 2. Hence, the left picture corresponds to a feasible assignment (disregarding machine capacity constraints), while the right picture violates both service conflict constraints for Service 0 and service spread constraints for Service 1.

The problem to be solved is the following: given an instance that consists of a number of machines, processes and resources, machine capacities, process resource and service spread requirements and a (feasible) initial assignment of processes to machines, find a new assignment that satisfies MCCon, SCCon and SSCon, while minimizing the following cost function that consists of two separate costs.

In order to not to overload the machines and provide a certain amount of spare capacity, if the consumption of a resource exceeds a given value that is deemed safe, a cost is incurred that is proportional to the excess resource consumption. As also hard resource capacities (see MCCon), *safe resource capacities* are resource and machine dependent. This cost is referred to as *machine load cost*. (MLCost) **NB:** One can assume that safe resource capacities are always lower than the corresponding hard resource capacities for all machines and all resources.



(a) Feasible assignment.

(b) Assignment violating SCCon and SSCon.

Figure 1: Example instance of four processes, two services, three machines and two locations. Note that Service 1 is required to be spread across at least two locations. Two assignments (solutions) are shown, one of which is feasible while the other one is not.

Some processes are difficult to move from one machine to another one. Hence, there is a cost that is incurred when a process is assigned to a machine that is different from the one that is given in the initial assignment provided as the input. This cost is referred to by *process move cost*. (PMCost) **NB1:** The cost of moving a specific process only depends on the process to be moved, independent of the target (or origin) machine it is assigned to. **NB2:** Note that although the cost of moving a specific process is independent of the machine to which the process was initially assigned to, the total PMCost incurred by an assignment not only depends on the assignment itself, but also on the initial assignment given as input to the program (see below).

3 Problem formulation

We begin by introducing notation and definitions and then formalize the various constraints and terms that are part of the cost function. Finally, we describe the input and output format.

3.1 Definitions

- Let $P = \{0, 1, 2, \ldots, N_P - 1\}$ be the set of processes, where we denote a single process by p. Similarly, let $M = \{0, 1, 2, \ldots, N_M - 1\}$ be the set of machines, where we denote a single machine by m. Let $R = \{0, 1, 2, \ldots, N_R - 1\}$ be the set of resources, where we denote a single resource by r.

- For each machine $m \in M$ and each resource $r \in R$, there is an integer capacity $C(m, r)$. Similarly, there is an integer safe capacity $C_S(m, r)$. Note that $C_S(m, r) \leq C(m, r)$.

- For each process $p \in P$ and each resource $r \in R$, there is an integer resource requirement $R(p, r)$.

- Let $S = \{0, 1, 2, \ldots, N_S - 1\}$ be the set of services. For each service $s \in S$, the set of processes that belong to that service is denoted by P_s. Recall that services are disjoint, so that $p \in P_s$ and $p \in P_{s'}$ implies $s = s'$.

- Let $L = \{0, 1, 2, \ldots, N_L - 1\}$ be the set of locations. For each location $l \in L$, the set of machines that belong to that location is denoted by M_l. Recall that locations are disjoint, so that $m \in M_l$ and $m \in M_{l'}$ implies $l = l'$.

- Let A_I be the initial assignment of processes to machines (given as part of the input). Note that A_I is an array that contains N_P integer values in the range of 0 to $N_M - 1$ with the interpretation

  $$A_I[p] = m \Leftrightarrow \text{process } p \text{ is assigned to machine } m.$$

  Similarly, let A_O be the re-assignment to be computed. The interpretation is the same as for A_I. **NB:** It can be assumed that the initial solution is feasible, i.e., it satisfies all constraints.

- For a machine $m \in M$ let $U(m, r)$ be the *usage* of resource r at machine m induced by an assignment A, which can be computed as

  $$U(m, r) = \sum_{p : A[p] = m} R(p, r),$$

  where the sum ranges over all processes that are assigned to machine m.

Example: Consider once more Figure 1. There are four processes split into two services to be assigned to three machines in two locations. So $N_P = 4$, $N_S = 2$, $N_M = 3$ and $N_L = 2$. Each of the services consist of two of the processes, so that $P_0 = \{0, 1\}$ and $P_1 = \{2, 3\}$. Similarly for the locations we have $M_0 = \{0, 1\}$ and $M_2 = \{2\}$. The assignment A on the left (picture a)) is specified as $A[0] = 0, A[1] = 1, A[2] = 1$ and $A[3] = 2$. Let's say there is only one resource ($N_R = 1$) and that processes that belong to Service 0 require 1 unit of this resource, while processes of Service 1 consume 2 units. This means that $R(0, 0) = R(1, 0) = 1$ and $R(2, 0) = R(3, 0) = 2$. Hence, for the assignment shown in Figure 1 a), we have that $U(0, 0) = 1$, $U(1, 0) = 3$ and $U(2, 0) = 2$.

3.2 Formulation of constraints

MCCon: For all machines $m \in M$ and all resources $r \in R$ it must hold that

$$U(m, r) = \sum_{p:A[p]=m} R(p, r) \leq C(m, r).$$

Note that the sum ranges over all processes that are assigned to machine m. The constraint thus specifies that the joint consumption of resource r at machine m must not exceed its capacity.

SCCon: For all services $s \in S$ and processes $p_1, p_2 \in P_s$ it must hold that

$$p_1 \neq p_2 \Rightarrow A[p_1] \neq A[p_2].$$

Note that the constraint specifies that two different processes of the same service must be assigned to different machines.

SSCon: For a service $s \in S$ let minSpread_s be the minimum number of distinct locations the service is required to operate in. Thus, for all services $s \in S$ it must hold that

$$\sum_{l \in L} \min\left(1, |\{p \in P_s \mid A[p] \in M_l\}|\right) \geq \text{minSpread}_s.$$

Note that the expression in the sum evaluates to the value 1 if there is at least one process of service s assigned to a machine in location l and 0 otherwise.

3.3 Formulation of costs

TotalCost: The total cost to be minimized by the re-assignment A_O, based on an initial assignment A_I, is determined as follows:

$$\text{TotalCost}(A_I, A_O) = \text{MLCost}(A_O) + \text{PMCost}(A_I, A_O).$$

MLCost: The machine load cost is determined based on the safe resource capacities of each machine for each resource. For a machine $m \in M$ and resource $r \in R$ recall that $U(m, r)$ is the usage of resource r at machine m, as defined above (dependent on assignment $A = A_O$).

$$\text{MLCost}(A_O) = \sum_{m \in M} \sum_{r \in R} \max\left(0, U(m, r) - C_S(m, r)\right).$$

Note that the expression in the inner sum evaluates to 0, whenever the joint usage of resource r at machine m is lower than its safe capacity. However, once the safe capacity has been reached, a cost that is equal to the excess usage is incurred.

line number	input line	entry name
L0	`1`	N_R
L1	`3`	N_M
L2	`0 2 1`	*location of machine 0*, $C(0,0)$ and $C_S(0,0)$
L3	`0 4 3`	*location of machine 1*, $C(1,0)$ and $C_S(1,0)$
L4	`1 2 1`	*location of machine 2*, $C(2,0)$ and $C_S(2,0)$
L5	`2`	N_S
L6	`1`	minSpread_0, i.e., minimum spread of service 0
L7	`2`	minSpread_1, i.e., minimum spread of service 1
L8	`4`	N_P
L9	`0 1 4`	*service of process 0*, $R(0,0)$ and pmc_0
L10	`0 1 2`	*service of process 1*, $R(1,0)$ and pmc_1
L11	`1 2 2`	*service of process 2*, $R(2,0)$ and pmc_2
L12	`1 2 4`	*service of process 3*, $R(3,0)$ and pmc_3

Table 1: Toy example of a problem-instance input file. The instance described is the same as shown in Figure 1. Note that there is a single whitespace between the different entries of the same line in the input file.

PMCost: The process move cost is a cost that is incurred based on the set of processes that were re-assigned to a different machine than in the initial assignment. Let pmc_p be the process move cost for process p (given as part of the input). The total process move cost is then determined as

$$\text{PMCost}(A_I, A_O) = \sum_{p \in Ps.t.\ A_O[p] \neq A_I[p]} \text{pmc}_p.$$

Note that the sum runs over all processes that are assigned to different machines in initial assignment A_I and output re-assignment A_O.

3.4 Input format

The input consists of two files: the *instance file* and the *initial solution*, which is a feasible but suboptimal assignment of processes to machines.

Note that in the assignment package you are provided Java/Python code for parsing the instance file. The input file format is described in Table 1 for the small example instance of Figure 1. As before, assume there is only one type of resource and that processes that belong to Service 0 consume 1 unit of this resource, while processes of Service 1 consume 2 units. Note that the assignment in Figure 1 a) satisfies capacity constraints (although it incurs a load cost at Machine 2), while the assignment of Figure 1 b) does not satisfy MCCon.

The second input file containing the initial machine assignment consists of a single line of space-delimited entries (no newline at the end). It contains all entries starting from $A_I[0]$ to $A_I[N_P - 1]$. For example, the the initial assignment corresponding to Figure 1 a) would be represented as

```
0 1 1 2
```

The general input format is described in Table 2. Note that all values are integers and below $2^{31} - 1$.

line number	entry name
L0	N_R
L1	N_M
L2-(2 + N_M − 1)	*location of machine* m, $C(m,0)$-$C(m,N_R-1)$, $C_S(m,0)$-$C_S(m,N_R-1)$
L(2 + N_M)	N_S
L(3 + N_M)-L(3 + N_M + N_S − 1)	minSpread_s, i.e., minimum spread of service s
L(3 + N_M + N_S)	N_P
L(4+N_M+N_S)-L(4+N_M+N_S+N_P−1)	*service of process* p, $R(p,0)$-$R(p,N_R-1)$ and pmc_p

Table 2: General description of the problem-instance input format. Note that there is a single white-space between the different entries of the same line in the input file.

3.5 Output format

The output format **for a single solution** is the same as the input format for the initial solution. More precisely, the output file has to consist of lines of text. Each line corresponds to a single solution and contains (delimited by a single space " ") all entries starting from $A_O[0]$ until $A_O[N_P-1]$ (naturally, in this order), which lists the machine that each process is assigned to in the best assignment found this far. For example, the final output assignment corresponding to Figure 1 a) would be represented as a single line of text:

```
0 1 1 2
```

Recall that your program is expected to **append** a newly found solution **as a new line** to the output file whenever it finds a better solution. Hence, we consider the last line of the output file to be the final solution used in the evaluation. Note that lines are expected to be **terminated by a newline character** "`\n`". We further recommend flushing the buffer of the output file after appending a new solution, since it may remain unwritten in the case that the program is terminated before it is flushed automatically.

4 Additional information

The problem that is subject of this programming assignment is a modified (and much simplified) version of the problem that is the subject of the ROADEF Challenge 2012. This challenge is an annual public competition organized by the French Operational Research and Decision Support Society (ROADEF). The main sponsor of the year 2012 is Google. For more information see

`http://challenge.roadef.org/2012/en/`.