Department of Mathematics and Computer Science
University of Southern Denmark, Odense

September 29, 2016
Marco Chiarandini

DM841 - Heuristics and Constraint Programming for Discrete Optimization

Obligatory Assignment 1, Autumn 2015[pdf format]

Deadline: 8th October 2015 at noon.

Solution
Included.

1. This is the first of two preparatory, obligatory assignments on constraint programming with pass/fail evaluation.
2. The aim of this first assignment is to get you have hands on experience with modeling in CP and the implementation in Gecode. Both skills will be tested in the final assignment that will be graded.
3. You are encouraged to search feedback and inspiration among your peers and ask questions in class related to the assignment. Working in pairs is allowed, but the final submission must be individual. You are therefore recommended to develop source code on your own and write the report individually.
4. Note: Changes to this document are to be expected. They will be listed in the last section: "Additions"
5. You have to submit a PDF document containing max two pages of description of the model and the source code of your implementation of the model in Gecode. The implementation must be compiling and working on any IMADA machine. More details on the requirements for the submission will come soon.
6. The subject of the assignment is problem 73 from the CSPLIB:

   `http://www.csplib.org/Problems/prob073/`

   The instances that you have to solve are those listed in the section "results". You can download these precise instances from

   `http://www.imada.sdu.dk/~marco/DM841/Files/P1-ob1/inst.tgz`

7. You are given a starting package to parse the input files.

Update
Submit at the following link:

`http://valkyrien.imada.sdu.dk/DOApp/`

Solution

Problem definition and notation Let $T = \{1..n\}$ be the set of tasks (or test), $M = \{1..m\}$ the set of machines and $R = \{1..r\}$ the set of resources. For each task $j \in T$ we are given the following parameters:

- a duration d_j
- a set of allowed machines $M(j) \subseteq M$
- a set of used resources $R(j) \subseteq R$
- an usage u_{rj} for each resource $r \in R$ (u_{rj} is 1 for the resources in $R(j)$ and 0 otherwise)

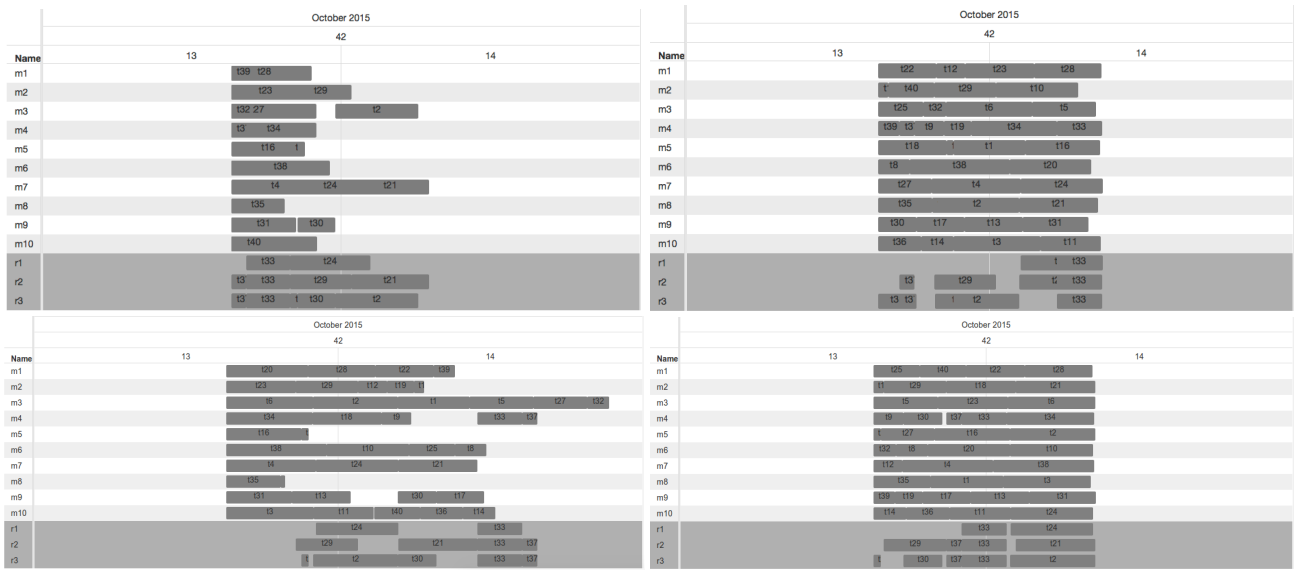

Figure 1:

Each resource has a capacity limit C_r, $r \in R$.
The makespan is the completion time of the last task finished.
Our goal is to find an assignment of tasks to machines and a starting time for each task such that:

- tasks do not overlap on the machines;
- at any time the use of each resource does not exceed the capacity limit;
- the makespan is minimized

An example It is good to have a small numerical example to represent the problem, test ideas and test implementations. We use here the smallest instance available, which is anyway already too big for the purpose of a small numerical example but we try anyway (one could remove a number of tasks and make thus the instance smaller). Below there are three Gantt chart to represent the problem. The first Gantt chart represents an schedule that violates the constraint that each machine can process only one job at a time. There are indeed overlaps among the tasks on the machines. The second Gantt chart shows a schedule that violates the constraints on the resources. There is indeed more than one task consuming a unary measure of resource running at the same time thus violating the capacity limit of one. Finally, the last two Gantt charts shows feasible schedules that can be bad (left) or good (right). We know that the optimal solution on this instance has makespan 1725.

CP model We assume that time is discretized and goes from 0 to Q, where Q is an upper bound to the makespan. A simple upper bound is the sum of all processing times of the tasks.
We then define the main variables:

- $X = \{x_1, x_2, ..., x_n\}$ with $D(x_j) = M$, for $j = 1..n$ to decide for each test on which machine it will be executed
- $S = \{s_1, s_2, ..., s_n\}$ with $D(s_j) = [0..Q]$ for $j = 1..n$ to decide at what time the execution of each task is started.

We will also need some auxiliary variables:

- $E = \{e_1, e_2, ..., e_n\}$ with $D(e_j) = [0..Q]$ to represent the end times of each task
- a with $D(a) = [0..Q]$ to carry the makespan.

Further, we define the constraints:

$$x_j \in M(j) \qquad \forall j \in T \tag{1}$$
$$s_j + d_j \leq e_j \qquad \forall j \in T \tag{2}$$
$$(s_i + d_i \leq d_j) \vee (s_j + d_j \leq s_i) \iff x_i = x_j \qquad \forall i, j \in T, i \neq j \tag{3}$$
$$\sum_{j \in \{j | s_j \leq t \leq s_j + d_j\}} u_{rj} \leq C_r \qquad \forall t \in [0..Q] \; \forall r \in R \tag{4}$$
$$e_j \leq a \qquad \forall j \in T \tag{5}$$
$$\min a \tag{6}$$

Constraints (1) restrict the domain to the allowed machines. Constraints (2) impose for each task j that its end time is smaller than equal the sum of its starting time and its duration. The minimization of the makespan will take care of making the smaller or equal relation an equality relation. Constraint (3) are the disjunctive constraints on each machine: for any pair of jobs i, j that are assigned to the same machine, either job i is processed before job j or job j is processed before job i. Constraint (4) enforce that for each resource and for each instant t, that is overlapped by the processing of at least one task j, the sum of the usage parameter of all the tasks j that are being processed at time t is less or equal than the capacity limit C_r. Finally, constraints (5) set the value of the makespan to the time when the last job is finished.

Constraints (1) can be implemented in Gecode with the `dom` constraints or by defining the precise domain when initializing the variables. Constraints (3) are disjunctive constraints that in gecode are implemented by the `unary` constraints. There will be one `unary` constraint for each machine but it has to act only on the jobs assigned to that machine. We will see two ways to achieve this, one with `cumulatives` and one one with reifying Boolean variables.
Constraints (4) are cumulative constraints that in gecode can be implemented with the `cumulative` constraints. Note that, due to the unary values of capacities and usage in our specific case, the constraints (4) become also disjunctive type of constraints and can therefore be implemented in gecode with `unary` constraints.

Gecode Implementation Perhaps the easiest way to formulate the problem in Gecode is via `cumulatives` to take care of the assignment of the tasks to machines and `cumulative` or `unary` to avoid violations on the resources.
We define the main variables and some constants that will be needed by our model:

```
class TestScheduling: public Script {
protected:
  Input *p_in;
  IntVarArray mach;
  IntVarArray start;
  IntVar makespan;
  int N;
  int M;
  int R;
public:
        ...
}
```

Then the constructor where the model is defined is:

```
  TestScheduling(const InstanceOptions& opt, Input *ind) : Script(opt), p_in(ind),
       mach(*this, ind->getNumTasks(), 0, ind->getNumMachines() - 1),
       start(*this, ind->getNumTasks(), 0, ind->getMaxTime()),
       makespan(*this, 0, ind->getMaxTime()),
       N(ind->getNumTasks()), M(p_in->getNumMachines()), R(p_in->getNumResources()) {

    // We limit the domains of the variables mach to only the allowed machines
    for (int j = 0; j < N; j++) {
      if (p_in->getTask(j).machines.size() > 0) {
        IntArgs tmp;
        for (const string & i : p_in->getTask(j).machines) {
          int k = 0;
          while (i != p_in->getMachine(k).name)
            k++;
```

```
15            tmp << k;
16          }
17          dom(*this, mach[j], IntSet(tmp));
18        }
19      }
20
21      // Machines
22      IntArgs duration(N);
23      IntArgs height(N);
24      for (int j = N; j--;) {
25        duration[j] = p_in->getTask(j).duration;
26        height[j] = 1;
27      }
28      IntVarArgs e(N);
29      for (int i = N; i--;)
30        e[i] = expr(*this, start[i] + duration[i]);
31      cumulatives(*this, mach, start, duration, e, IntArgs::create(N, 1, 0),
32        IntArgs::create(M, 1, 0), true);
33
34      // Resources
35      for (int i = R; i--;) {
36        IntArgs d_aux;
37        IntVarArgs s_aux;
38        for (int j = N; j--;) {
39          if (vector_contains(p_in->getTask(j).resources, p_in->getResource(i).name)) {
40            s_aux << start[j];
41            d_aux << p_in->getTask(j).duration;
42          }
43        }
44        cumulative(*this,1,s_aux,d_aux,IntArgs::create(s_aux.size(), 1, 0));
45        // "unary" would also work in this special case of all unary capacities
46        // unary(*this, s_aux, d_aux);
47      }
48      // Makespan
49      max(*this, e, makespan);
50
51      // Branchingc
52      Rnd r(opt.seed());
53      branch(*this, mach, INT_VAR_RND(r), INT_VAL_RND(r));
54      branch(*this, start, INT_VAR_SIZE_MIN(), INT_VAL_SPLIT_MIN());
55      branch(*this, makespan, INT_VAL_MAX());
56    }
```

Since we are solving an optimization problem we need define the constraints on the makespan that must be added each time a new feasible solution is found.

```
1  virtual void constrain(const Space& _b) {
2      const TestScheduling & b = static_cast<const TestScheduling &>(_b);
3      if (b.makespan.assigned())
4        rel(*this, makespan, IRT_LE, b.makespan.val());
5    }
```

This model works badly and achieves on the easiest instance, where the optimal value is 1725, a value of 3023 in 20 seconds of running time. The reason is that the "cumulatives" propagator is more general than what is actually needed. Indeed, there is no cumulative usage in these problem since both usages and capacities are unitary. Hence, we try to model everything with the unary constraint. For the resources we substitute line 44 with the commented line 46. Handling the machine constraints is slightly more complicated because only few tasks will be executed on every machine, not all. However, we can handle the situation with reification introducing an array of m Boolean variables for each task. We then channel the binary variables with the machine variables.
In practice, we work with a matrix of size $n \times m$ Boolean variables. Having declared the variables `BoolVarArray b;` we initialize them in the constructor:

```
1  b(*this, ind->getNumTasks() * ind->getNumMachines(), 0, 1)
```

and use them later in the model as:

```
Matrix < BoolVarArgs > bmat(b, M, N); // watch out! first number of columns and then number of rows
for (int j = N; j--;)
  channel(*this, bmat.row(j), mach[j]);
for (int i = M; i--;)
  unary(*this, start, duration, bmat.col(i));
```

For the new model better results are achieved with the following branching rules:

```
Rnd r(opt.seed());
branch(*this, mach, INT_VAR_RND(r), INT_VAL_RND(r));
branch(*this, start, INT_VAR_SIZE_MAX(), INT_VAL_SPLIT_MIN());
branch(*this, makespan, INT_VAL_MAX());
```

The model with the unary constraints reaches a solution of 1741 after 12 seconds running on one single thread. The solution is depicted in the fourth Gantt chart (bottom right corner) of Figure 1.
Before showing the results I report also the main function to show the use of `Statistics` and `TimeStop` which are useful to show the statistics of the propagation and search process and to stop the search at a time limit. An alternative would be to execute the search with the minimodel short cut `Script::run<*m, BAB, Options >(opt);`

```
int main(int argc, char* argv[]) {
  InstanceOptions opt("TestScheduling");
  opt.instance("../data/t40m10r3-2.pl");

  opt.time(20 * 1000);

  opt.parse(argc, argv);
  Input *p = new Input(opt.instance());

  Search::Options so;
  Search::TimeStop* ts;
  ts = new Search::TimeStop(opt.time());
  so.stop = ts;

  Search::Statistics stat;

  // Timer
  Support::Timer t;
  t.start();

  TestScheduling* m = new TestScheduling(opt, p);

  SpaceStatus status = m->status(stat);
  print_stats(stat); // a function define to output the statistics

  BAB<TestScheduling> e(m, so);
  delete m;
  while (TestScheduling* s = e.next()) {
    ofstream myfile;
    myfile.open("solution.txt", ios::out);
    s->printSol(myfile);
    myfile.close();
    stat = e.statistics();
    print_stats(stat);
    cout << "\ttime: " << t.stop() / 1000 << "s" << endl;
    delete s;
  }
  if (e.stopped()) {
    cout << "WARNING: solver stopped, solution is not optimal!\n";
    if (ts->stop(e.statistics(), so)) {
      cout << "\t Solver stopped becuase of TIME LIMIT!\n";
    }
  }
  print_stats(stat);
```

instance	prop. at root	prop at end	depth	nodes	Makespan
t40m10r3-2	127	6293696	592071	385	1741
t50m10r3-9	157	75125	4901	491	7279
t500m50r5-5	1550	1550	0	0	[801..206476]
t500m100r10-1	1606	1606	0	0	[799..202156]
t500m100r10-2	1609	1609	0	0	[796..197238]
t500m100r10-3	1609	1609	0	0	[801..197161]
t500m100r10-4	1607	1607	0	0	[798..199680]
t500m100r10-5	1608	1608	0	0	[800..194259]
t500m100r10-6	1607	1607	0	0	[801..206713]
t500m100r10-7	1609	1609	0	0	[800..200054]
t500m100r10-8	1611	1611	0	0	[800..205732]
t500m100r10-9	1609	1609	0	0	[800..199521]
t500m100r10-10	1610	1610	0	0	[795..202601]

Table 1:

```
45    cout << "\ttime: " << t.stop() / 1000 << "s" << endl;
46    return 0;
47  }
```

The Gantt chart in Figure 1 are produced with angular-gantt on a json file which is produced in a similar fashion as the `printSol(file)` function above.
The full script is available at

`http://www.imada.sdu.dk/~marco/DM841/Files/scheduling.cpp`

Results I run the script with a time limit of 20 seconds on all instances. The results are reported in Table 1. We see that a feasible solution is found in only two instances.