TP2: Implementation of a Timetable filtering algorithm for a Cumulative constraint
INF6101 — Constraint Programming

Thibault Noilly*, Mahshid Mohammadalitajrishi†, Jinling Xing‡

April 25, 2018

1 Introduction

TP2 required us to implement a timetable filtering algorithm for the cumulative constraint of MiniCP [MSH17b]. This work is described in [MSH17a] and is divided into 3 different parts. The first one is to build an optimistic profile from the mandatory parts of the different tasks considered in the constraint. The second part is to check that the profile is not exceeding the capacity given with the cumulative constraint. The third part is a filter of the earliest start of the activities. Once the implementation is done, we tested it with the given example "CumulSched.java" B to get the results.

2 Code and Explanation

In this part, we consider the implementation of the Cumulative and Time-Table Filtering algorithm with three TODO tasks. The whole implementation can be found in A.

2.1 Part 1: building the optimistic profile

```
public Profile buildProfile() throws InconsistencyException {
    ArrayList<Rectangle> mandatoryParts = new ArrayList<Rectangle>();
    for (int i = 0; i < start.length; i++) {
        // TODO 1: add mandatory part of activity i if any
        int startMax = start[i].getMax();
        int endMin = start[i].getMin() + duration[i];

        if(startMax < endMin) {
            mandatoryParts.add(
             new Profile.Rectangle(startMax, endMin, demand[i]));
        }
    }
    return new Profile(mandatoryParts.toArray(
     new Profile.Rectangle[mandatoryParts.size()]));
}
```

To build the profile, we need to detect the tasks that possess a mandatory part. To do so, we retrieve the latest possible starting time of a task and its earliest completion time. Indeed, a task will present a mandatory part if its earliest completion time is after its latest possible start. The contribution of this task to the profile will then be represented by the demand of the task.

*thibault.noilly@polymtl.ca
†Mahshid.Mohammadalitajrishi@polymtl.ca
‡jinling.xing@polymtl.ca

2.2 Part 2: checking the mandatory capacity

```
Profile profile = buildProfile();
// TODO 2: check that the profile is not exceeding the capa
// otherwise throw an INCONSISTENCY

for (int i = 0; i < profile.size(); i++) {
    // TODO: check
    int mandatoryCapacity = profile.get(i).height;
    if(mandatoryCapacity > capa)
    throw InconsistencyException.INCONSISTENCY;
}
```

After building the profile, we check if the mandatory capacity exceeds the capacity given when defining the constraint. This part is straightforward as we check the height of every rectangle that forms the profile. If the capacity is exceeded, we throw an inconsistency exception as the constraint cannot be solved.

2.3 Part 3: filter the earliest start time

```
for (int i = 0; i < start.length; i++) {
    if (!start[i].isBound()) {
        // j is the index of the profile rectangle overlapping t
        int j = profile.rectangleIndex(start[i].getMin());
        // TODO 3: push i to the right
        // hint:
        // You need to check that at every-point on the interval
        // [start[i].getMin() ... start[i].getMin()+duration[i]-1]
        //there is enough space.
        // You may have to look-ahead on the next profile rectangle(s)
        // Be careful that the activity you are currently pushing
        //may have contributed to the profile.

        // startMax and endMin are used to check if t is in the
        //mandatory part of the task
        int startMax = start[i].getMax();
        int endMin = start[i].getMin() + duration[i];

        for(int t = start[i].getMin(); t <= startMax; t++) {
            boolean conflict = false;
            int d = 0;

            // We check the current profile rectangle and the next
            // rectangles to see if the capacity is enough or if there
            // is a conflict
            while(!conflict && d <= duration[i]) {

                // We check if t is in the mandatory part
                if(startMax <= t+d && t+d <= endMin) {
                    // t is in the mandatory part => the demand of the task is
                    // already taken into account in the profile height
                } else {
                    // t isn't in the mandatory part, we can check normally
                    int mandatoryCapacity =
                     profile.get(profile.rectangleIndex(t+d)).height;
                    if(mandatoryCapacity + demand[i] > capa)
                    conflict = true;
```

```
                }
                d++;
            }

            // No conflict so t is the earliest start time
            // We can end the search and set t as min start
            if(!conflict) {
                // Set t as minimum start for task i
                start[i].removeBelow(t);
                // End the search
                break;
            }
        }
    }
}
```

To filter the starting time for each activity, we check each starting value possible for conflicts that would mean the value can be discarded and removed from the domain of the variable. To do so, we get the minimum value of start and we test it before incrementing it if we find a conflict.

A conflict is found if the capacity is exceeded for any of the time period representing the duration of the task. To check the capacity, we add the demand of the task currently filtered to the mandatory capacity of the time period.

We also took care of the task contributing to the profile. Indeed, if the task has a mandatory part and if the time currently tested is in the mandatory time period of the task, we don't have to check for the capacity as the demand of the task is already taken into account in the profile.

As we are seeking the earliest start possible, we can stop the search with the first time period which presents no conflict. We then remove any lesser value from the domain.

3 Results and Analysis

Below is the example of "CumulSched.java" given on Moodle and replicated in B:

```
// instance data; for each task: earliest start time, latest completion
// time, duration (on resource A, resource B -- both required),
// demand (on resource A, resource B -- both required)
int[] est = new int[]{0,0,6,5};
int[] lct = new int[]{7,5,9,9};
int[] durationA = new int[]{2,4,1,3};
int[] durationB = new int[]{1,2,2,2};
int[] demandA = new int[]{1,3,1,2};
int[] demandB = new int[]{4,1,3,2};
int capaA = 3;
int capaB = 4;
```

The result of this example with our implementation is below:

```
solution:[{4}, {0}, {7}, {5}]

#choice: 0
#fail: 0
#sols : 1
```

Moreover, we added 2 sets of easy schedules to test the constraint. The detailed code is below:

```
// This sample should throw be inconsistent (mandatory capacity > capacity)
int[] est = new int[]{2,0,2};
int[] lct = new int[] {6,5,7};
```

```
int[] durationA = new int[] {3,4,4};
int[] durationB = new int[] {0,0,0};
int[] demandA = new int[] {1,1,1};
int capaA = 2;

// This sample should give a solution (A0, B1, C2)
int[] est = new int[]{0,1,2};
int[] lct = new int[] {2,3,4};
int[] durationA = new int[] {2,2,2};
int[] durationB = new int[] {0,0,0};
int[] demandA = new int[] {1,1,1};
int capaA = 2;
```

The first set is throwing an inconsistency exception, as it should. The second gives us the solution we are looking for.

4 Conclusion

The timetable filtering algorithm for the cumulative constraint is a two stage algorithm. With the method *buildProfile()*, we added mandatory parts of activities to profile and we then checked whether the profile is exceeding the capacity to ensure that the constraint can be satisfied. For the second part of the algorithm, we needed to filter the start time of the activities to reduce the domains of the variables and thus to find solutions more efficiently. When we executed the test example, we got a solution without failures, which means the performance of the cumulative filtering algorithm is good as it reduces the possible values of start to consider for each activity.

A Implementation Cumulative.java

```
/*
 * mini-cp is free software: you can redistribute it and/or modify
 * it under the terms of the GNU Lesser General Public License v3
 * as published by the Free Software Foundation.
 *
 * mini-cp is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY.
 * See the GNU Lesser General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public License
 * along with mini-cp. If not, see http://www.gnu.org/licenses/lgpl-3.0.en.html
 *
 * Copyright (c) 2017. by Laurent Michel, Pierre Schaus, Pascal Van Hentenryck
 */

package minicp.engine.constraints;

import static minicp.cp.Factory.*;

import minicp.engine.core.Constraint;
import minicp.engine.core.IntVar;
import minicp.engine.constraints.Profile.Rectangle;
import minicp.util.InconsistencyException;
import minicp.util.NotImplementedException;

import java.util.ArrayList;

public class Cumulative extends Constraint {

    private final IntVar[] start;
    private final int[] duration;
    private final IntVar[] end;
    private final int[] demand;
    private final int capa;
    private final boolean postMirror;

    public Cumulative(IntVar[] start, int[] duration, int[] demand, int capa)
    throws InconsistencyException {
        this(start, duration, demand, capa, true);
    }

    private Cumulative(IntVar[] start, int[] duration, int[] demand, int capa,
    boolean postMirror) throws InconsistencyException {
        super(start[0].getSolver());
        this.start = start;
        this.duration = duration;
        this.end = makeIntVarArray(cp,start.length, i ->
        plus(start[i],duration[i]));
        this.demand = demand;
        this.capa = capa;
        this.postMirror = postMirror;
    }


    @Override
    public void post() throws InconsistencyException {
        for (int i = 0; i < start.length; i++) {
            start[i].propagateOnBoundChange(this);
        }

        if (postMirror) {
```

```
        IntVar[] startMirror = makeIntVarArray(cp, start.length, i ->
        minus(end[i]));
        cp.post(new Cumulative(startMirror, duration, demand, capa,
        false), false);
    }

    propagate();
}

@Override
public void propagate() throws InconsistencyException {

    Profile profile = buildProfile();
    // TODO 2: check that the profile is not exceeding the capa otherwise
    throw an INCONSISTENCY

    for (int i = 0; i < profile.size(); i++) {
        // TODO: check
     int mandatoryCapacity = profile.get(i).height;
     if(mandatoryCapacity > capa)
        throw InconsistencyException.INCONSISTENCY;
    }

    for (int i = 0; i < start.length; i++) {
        if (!start[i].isBound()) {
            // j is the index of the profile rectangle overlapping t
            int j = profile.rectangleIndex(start[i].getMin());
            // TODO 3: push i to the right
            // hint:
            // You need to check that at every-point on the interval
            // [start[i].getMin() ... start[i].getMin()+duration[i]-1]
            //there is enough space.
            // You may have to look-ahead on the next profile rectangle(s)
            // Be careful that the activity you are currently pushing
            //may have contributed to the profile.

            // startMax and endMin are used to check if t is in the
            //mandatory part of the task
            int startMax = start[i].getMax();
           int endMin = start[i].getMin() + duration[i];

            for(int t = start[i].getMin(); t <= startMax; t++) {
               boolean conflict = false;
               int d = 0;

               // We check the current profile rectangle and the next
               // rectangles to see if the capacity is enough or if there
                // is a conflict
               while(!conflict && d <= duration[i]) {

                 // We check if t is in the mandatory part
                 if(startMax <= t+d && t+d <= endMin) {
                    // t is in the mandatory part => the demand of the task is
                    // already taken into account in the profile height
                 } else {
                    // t isn't in the mandatory part, we can check normally
                    // int mandatoryCapacity = profile.get(j+d).height; <= seems
                        //wrong
                    int mandatoryCapacity =
                        profile.get(profile.rectangleIndex(t+d)).height;
                    if(mandatoryCapacity + demand[i] > capa)
                       conflict = true;
                 }
```

```
                d++;
              }

              // No conflict so t is the earliest start time
              // We can end the search and set t as min start
              if(!conflict) {
                // Set t as minimum start for task i
                start[i].removeBelow(t);
                // End the search
                break;
              }
            }
          }
        }
        // throw new NotImplementedException("Cumulative");
        // System.out.println("Not implemented yet");
    }

    public Profile buildProfile() throws InconsistencyException {
        ArrayList<Rectangle> mandatoryParts = new ArrayList<Rectangle>();
        for (int i = 0; i < start.length; i++) {
            // TODO 1: add mandatory part of activity i if any
         int startMax = start[i].getMax();
         int endMin = start[i].getMin() + duration[i];

         if(startMax < endMin) {
            mandatoryParts.add(new Profile.Rectangle(startMax, endMin, demand[i]));
         }
        }
        return new Profile(mandatoryParts.toArray(new
        Profile.Rectangle[mandatoryParts.size()]));
    }

}
```

B Implementation CumulativeSched.java

```
/*
 * mini-cp is free software: you can redistribute it and/or modify
 * it under the terms of the GNU Lesser General Public License v3
 * as published by the Free Software Foundation.
 *
 * mini-cp is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY.
 * See the GNU Lesser General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public License
 * along with mini-cp. If not, see http://www.gnu.org/licenses/lgpl-3.0.en.html
 *
 * Copyright (c) 2017. by Laurent Michel, Pierre Schaus, Pascal Van Hentenryck
 */

package minicp.examples;

import minicp.engine.constraints.Cumulative;
import minicp.engine.core.IntVar;
import minicp.engine.core.Solver;
import minicp.search.DFSearch;
import minicp.search.SearchStatistics;
import minicp.util.InconsistencyException;

import java.util.Arrays;

import static minicp.cp.Factory.*;
import static minicp.cp.Heuristics.firstFail;
import static minicp.search.Selector.branch;
import static minicp.search.Selector.selectMin;

public class CumulSched {
    public static void main(String[] args) throws InconsistencyException {

      // instance data; for each task: earliest start time, latest completion time,
          duration (on resource A, resource B -- both required), demand (on resource A,
          resource B -- both required)
        int[] est = new int[]{0,0,6,5};
        int[] lct = new int[]{7,5,9,9};
        int[] durationA = new int[]{2,4,1,3};
        int[] durationB = new int[]{1,2,2,2};
        int[] demandA = new int[]{1,3,1,2};
        int[] demandB = new int[]{4,1,3,2};
      int capaA = 3;
      int capaB = 4;

      // This sample should throw be inconsistent (mandatory capacity > capacity)
//       int[] est = new int[]{2,0,2};
//       int[] lct = new int[] {6,5,7};
//       int[] durationA = new int[] {3,4,4};
//       int[] durationB = new int[] {0,0,0};
//       int[] demandA = new int[] {1,1,1};
//       int capaA = 2;

      // This sample should give a solution (A0, B1, C2)
//       int[] est = new int[]{0,1,2};
//       int[] lct = new int[] {2,3,4};
//       int[] durationA = new int[] {2,2,2};
//       int[] durationB = new int[] {0,0,0};
//       int[] demandA = new int[] {1,1,1};
//       int capaA = 2;
```

```
        int n = est.length;

        Solver cp = makeSolver();
        IntVar[] start = new IntVar[n];
        for (int i = 0; i < n; i++)
            start[i] = makeIntVar(cp, est[i], lct[i]-Math.max(durationA[i],durationB[i]));


        cp.post(new Cumulative(start,durationA,demandA,capaA));
        cp.post(new Cumulative(start,durationB,demandB,capaB));


        DFSearch dfs = makeDfs(cp, firstFail(start));


        dfs.onSolution(() ->
                System.out.println("solution:"+ Arrays.toString(start))
        );

        SearchStatistics stats = dfs.start();

        System.out.println(stats);


    }
}
```

References

[MSH17a] Laurent Michel, Pierre Schauss, and Pascal Van Hentenryck. Exercises : Cumulative constraint: Time-table filtering, 2017.

[MSH17b] Laurent Michel, Pierre Schauss, and Pascal Van Hentenryck. Mini-cp: A minimalist open-source solver to teach constraint programming, 2017.

The document: TP 2 - Implantation d'un algorithme de filtrage