First & last name	
NOMA UCLouvain	

LINFO2365: Constraint Programming (Prof. P. Schaus)

Final Exam - June 2026 - 2h (2h40 PEPS)

Exercise 1 Multiple Choice Questions (3 pts)

1. In MiniCP, the domain of an integer variable is implemented using a backtrackable sparse-set (represented by the `StateSparseSet` class). Let n be the capacity (initial domain size) and k the number of values currently remaining in the domain. For each operation, select its worst-case time complexity:

   - Checking if a value v is in the domain (`contains(v)`):
     ☒ $O(1)$ ☐ $O(k)$ ☐ $O(n)$ ☐ $O(\log k)$ ☐ $O(\log n)$ ☐ $O(k \log k)$ ☐ $O(n \log n)$
   - Removing a single value v from the domain (`remove(v)`):
     ☒ $O(1)$ ☐ $O(k)$ ☐ $O(n)$ ☐ $O(\log k)$ ☐ $O(\log n)$ ☐ $O(k \log k)$ ☐ $O(n \log n)$
   - Assigning the variable to a single value (`removeAllBut(v)`):
     ☒ $O(1)$ ☐ $O(k)$ ☐ $O(n)$ ☐ $O(\log k)$ ☐ $O(\log n)$ ☐ $O(k \log k)$ ☐ $O(n \log n)$
   - Iterating over all active values currently in the domain:
     ☐ $O(1)$ ☒ $O(k)$ ☐ $O(n)$ ☐ $O(\log k)$ ☐ $O(\log n)$ ☐ $O(k \log k)$ ☐ $O(n \log n)$
   - Updating the minimum or maximum value of the domain (`removeAbove/Below(v)`):
     ☐ $O(1)$ ☒ $O(k)$ ☐ $O(n)$ ☐ $O(\log k)$ ☐ $O(\log n)$ ☐ $O(k \log k)$ ☐ $O(n \log n)$

2. Consider the following Java code snippet executed using the MiniCP trailing mechanism:

```java
StateManager sm = new Trailer();
StateInt a = sm.makeStateInt(10);
sm.saveState();
a.setValue(15);
a.setValue(20);
sm.saveState();
a.setValue(25);
sm.saveState();
```

   Between the creation of `a` and the end of this snippet, how many total `StateEntry` objects have been pushed onto the trail (either in active or prior backups)?
   ☐ 4 entries ☐ 3 entries ☒ 2 entries ☐ 1 entry

3. Same code snippet but using copying mechanism instead at the first line.

```java
StateManager sm = new Copier(); // replacement of the first line
```

   How many total `StateEntry` objects have been pushed onto the state manager's backups?
   ☐ 4 entries ☒ 3 entries ☐ 2 entries ☐ 1 entry

Exercise 2 The Sum Constraint and Consistency (5 pts)

Consider the constraint $\sum_{i=1}^{n} x_i = 0$.

1. Which level of consistency is typically enforced for the `Sum` constraint in CP solvers (like MiniCP)? Justify why this level is chosen.

Bound consistency (specifically Bound(Z) consistency). Enforcing domain consistency for the sum constraint is NP-hard (since it generalizes the subset-sum / knapsack problem). In contrast, bound consistency can be checked and enforced in $O(n)$ time, offering a highly efficient trade-off between pruning power and search speed.

2. Provide the formal definition of the consistency level you identified in the previous question, for a general constraint $C(x_1, \dots, x_n)$ with domains $D(x_1), \dots, D(x_n)$.

A constraint $C(x_1, \dots, x_n)$ is bound-consistent if and only if for all $1 \leq i \leq n$ and all $v \in \{\min(D(x_i)), \max(D(x_i))\}$, there exists a tuple $(v_1, \dots, v_n)$ such that:

- $v_i = v$,
- $v_j \in [\min(D(x_j)), \max(D(x_j))] \cap \mathbb{Z}$ for all $j \neq i$,
- $C(v_1, \dots, v_n)$ is satisfied.

3. Give the filtering rule for the domain of x_i enforcing the consistency level given at the previous answer.

Using bounds notation $\underline{x}_i = \min(D(x_i))$ and $\overline{x}_i = \max(D(x_i))$, the domain bounds of x_i are updated as:

$$\underline{x}_i \leftarrow \max\left(\underline{x}_i,\ -\sum_{j \neq i} \overline{x}_j\right) \quad \text{and} \quad \overline{x}_i \leftarrow \min\left(\overline{x}_i,\ -\sum_{j \neq i} \underline{x}_j\right)$$

4. What is the time complexity to filter all variables $x_1, \dots, x_n$ from scratch and achieve the consistency level? Provide the best algorithm pseudo-code to achieve this complexity.

The time complexity is $O(n)$, as we can precompute S_{min} and S_{max} in $O(n)$ time and then update the bounds of each variable in $O(1)$ time.

Algorithm:

$S_{min} \leftarrow \sum_{j=1}^{n} \min(D(x_j))$
$S_{max} \leftarrow \sum_{j=1}^{n} \max(D(x_j))$
if $S_{min} > 0$ **or** $S_{max} < 0$ **then fail**
for $i \leftarrow 1$ **to** n **do**
  $D(x_i) \leftarrow D(x_i) \cap [\max(D(x_i)) - S_{max},\ \min(D(x_i)) - S_{min}]$
  if $D(x_i) = \emptyset$ **then fail**

Exercise 3 Scheduling: Cumulative Constraint (4 pts)

Consider a `Cumulative` constraint with capacity $C = 4$. We have three activities:

Task	Duration (p_i)	Demand (d_i)	est_i	lct_i
A	4	2	0	6
B	3	1	1	5
C	2	2	1	8

1. What is the *mandatory part* of an activity? Provide its formal definition as an interval based on est_i, lct_i and p_i.

The mandatory part of an activity is the time interval during which the activity must be executing regardless of when it starts within its time window $[est_i, lct_i]$. Formally, if $lct_i - p_i < est_i + p_i$, the mandatory part is the interval:

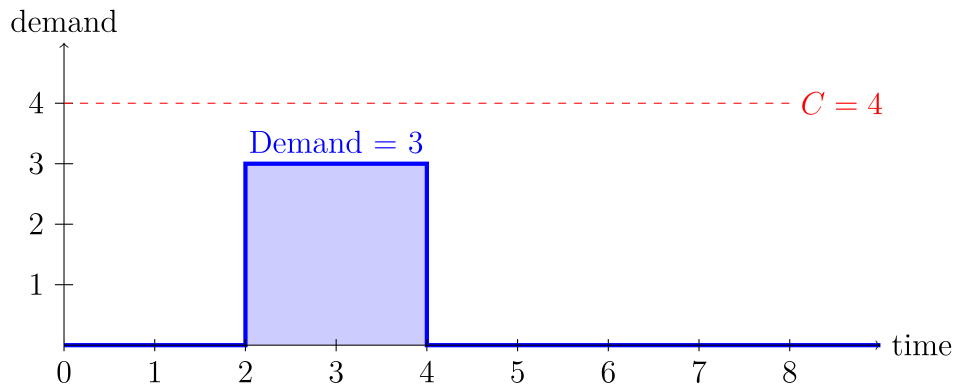
$$MP(i) = [lct_i - p_i,\ est_i + p_i]$$

Otherwise, if $lct_i - p_i \geq est_i + p_i$, the mandatory part is empty ($MP(i) = \emptyset$).

2. For each task (A, B, C), calculate its mandatory part (if any).

- **A**: $est_A = 0, lct_A = 6, p_A = 4$. $lct_A - p_A = 2 < 4 = est_A + p_A \implies MP(A) = [2, 4]$.
- **B**: $est_B = 1, lct_B = 5, p_B = 3$. $lct_B - p_B = 2 < 4 = est_B + p_B \implies MP(B) = [2, 4]$.
- **C**: $est_C = 1, lct_C = 8, p_C = 2$. $lct_C - p_C = 6 \geq 3 = est_C + p_C \implies MP(C) = \emptyset$.

3. Draw the *optimistic consumption profile* (time-table profile) based on the mandatory parts identified.



4. Can any time-window be tightened or can an inconsistency be detected based on time-table filtering? Justify your answer.

Yes, the time-window of Task C can be tightened. No inconsistency is detected. During the interval $[2, 4]$, the consumed capacity by the mandatory parts of A and B is $d_A + d_B = 2 + 1 = 3$. The remaining capacity is $4 - 3 = 1$.
Since Task C requires a demand of $d_C = 2 > 1$, it cannot overlap with any part of $[2, 4]$. Thus, we must have $s_C \geq 4$, allowing us to tighten the earliest start time of Task C: $est_C \leftarrow 4$.

Exercise 4 Modeling: Traveling Salesman Problem (4 pts)

You are asked to model the Traveling Salesman Problem (TSP) with n cities and a distance matrix $dist[i][j]$ using Constraint Programming.

1. Define the decision variables you would use. Explain what they represent and what their initial domains are.

We define the following decision variables:

- **Successor variables**: $next[i]$ for each city $i \in \{0, \dots, n-1\}$. The value of $next[i] = j$ means that city j is visited immediately after city i in the tour.
  Initial domains $D(next[i]) = \{0, \dots, n-1\} \setminus \{i\}$.
- **Cost variables**: $cost[i]$ for each city $i \in \{0, \dots, n-1\}$, representing the cost of the transition from city i to its successor.
  Initial domains $D(cost[i]) = [\min_{j \neq i} dist[i][j], \max_{j \neq i} dist[i][j]]$..
- **Objective variable**: $totalDist$, representing the total distance of the tour.
  Initial domain $D(totalDist) = [0, \sum_{i,j} dist[i][j]]$.

2. What are the constraints required to ensure a valid tour? Explain the purpose of each.

- **Circuit Constraint**: $\texttt{Circuit}(next)$.
  Enforces that the successor variables form a single Hamiltonian cycle visiting all n cities exactly once, preventing sub-tours.
- **Element Constraints** (for edge costs): $cost[i] = \texttt{Element}(dist[i], next[i])$ for each city i.
  Links the successor variable $next[i]$ to the distance/cost of the edge $dist[i][next[i]]$.
- **Sum Constraint** (for total distance): $totalDist = \sum_{i=0}^{n-1} cost[i]$.
  Computes the sum of all edge costs to obtain the objective value $totalDist$ to be minimized.

3. What branching variable and value selection heuristics would you choose? Explain the principle you follow for each choice and why it is appropriate for the TSP.

- **Variable Selection (Min-Regret / First-Fail Principle)**:
  Select the unassigned variable $next[i]$ with the largest *regret* (the difference between the closest and second-closest available successor cities).
  Reasoning: Following the first-fail principle, we want to branch first on the variables where making a sub-optimal choice is most costly.
- **Value Selection (Nearest Neighbor / Succeed-First Principle)**:
  For the selected variable $next[i]$, try the available successor city j that minimizes $dist[i][j]$ first.
  Reasoning: Following the succeed-first principle, edges with smaller distances are much more likely to be part of the optimal tour.

Exercise 5 Large Neighborhood Search (4 pts)

1. Explain briefly the principle of Large Neighborhood Search (LNS) and its three main phases.

Large Neighborhood Search (LNS) is an iterative optimization metaheuristic that combines local search and constraint programming. The three main phases are:

- **Relaxation / Destruction**: A subset of the decision variables (called a fragment) is unfixed (relaxed), while the remaining variables are frozen to their values in the current best solution.
- **Reconstruction / CP Search**: A Constraint Programming solver is used to explore the restricted search space (the neighborhood) to find a better assignment for the unfixed variables, typically under a small failure limit to prevent search explosion.
- **Acceptance / Restart**: If a new best solution is found, it is recorded. A new restart is initiated by selecting a different fragment to unfix.

2. Consider the following sketch of the Quadratic Assignment Problem solved via LNS in MiniCP. Complete the missing implementation to perform LNS search. The search must execute `nRestarts` restarts, be constrained by the objective `obj`, each restart is limited by `failureLimit` failures, and must fix a random 75% of the variables to their current best values.

```java
// Current best solution representation
int[] xBest = new int[n];
dfs.onSolution(() -> {
    for (int i = 0; i < n; i++) xBest[i] = x[i].min();
});

int nRestarts = 1000;
int failureLimit = 100;
Random rand = new Random(0);

// TODO: Complete the LNS search block here
```

```java
for (int i = 0; i < nRestarts; i++) {
dfs.optimizeSubjectTo(obj,
      stats -> stats.numberOfFailures() >= failureLimit,
      () -> {
  for (int j = 0; j < n; j++) {
    if (rand.nextInt(100) < 75) {
      cp.post(equal(x[j], xBest[j]));
    }
  }
});
}
```