



CATHOLIC UNIVERSITY OF LOUVAIN

PROJECT 3 : SEARCH

---

# LINGI2365 - Constraint Programming

---

*Auteurs :*

Vanwelde Romain (3143-10-00)

Crochelet Martin (2236-10-00)

Groupe 7

*Superviseurs :*

Pr. Yves Deville

François Aubry

24 mars 2014

# Table des matières

<b>1</b>	<b>The Brussels airport problem</b>	<b>1</b>
1.1	Explain the given model . . . . .	1
1.2	Design 2 different variable and/or value ordering heuristics for this problem.	1
1.3	Which criteria are meaningful for comparing different search strategies? . .	2
1.4	Based on your criteria, compare your heuristics with the labelFF heuristic by testing them on the instance on iCampus. . . . .	2
1.5	Consider the following strategy. . . . Give an example with three planes where this strategy is wrong . . . . .	2
<b>2</b>	<b>The Knapsack Problem</b>	<b>2</b>
2.1	A Branch & Bound approach . . . . .	2
2.1.1	Model the knapsack problem as Constraint Optimization . . . . .	2
2.1.2	Describe your model in the report. . . . .	3
2.1.3	Design 3 different heuristics for variable selection. . . . .	4
2.1.4	Test your heuristics and the labelFF heuristic on the knapsack-A instances. . . . .	4
2.1.5	Present and discuss the results in your report . . . . .	5
2.2	Optimization over iterations . . . . .	6
2.2.1	Model the knapsack problem as a Constraint Satisfaction Problem.	6
2.2.2	In order to implement the optimization over iterations . . . . .	7
2.2.3	Which of these points (i., ii., iii.) do you need to execute on which events? . . . . .	7
2.2.4	How do you modify the value of ub to be sure to find the optimal solution? . . . . .	7
2.2.5	Can you explain why we initialize ub with an upper bound instead of any other value? . . . . .	7
2.2.6	Experiment this program on the instances knapsack-A, -B. . . . .	7
2.2.7	Present and discuss the results in your report. . . . .	7
2.3	Optimization via divide and conquer . . . . .	9
2.3.1	In order to implement the optimization via divide and conquer you will have to . . . . .	9
2.3.2	Which of these points (i., ii., iii., iv.) do you need to execute on which events? . . . . .	11
2.3.3	Experiment this version on the instances knapsack-A,-B,-C . . . . .	11
2.3.4	Present and discuss the results in your report. . . . .	11

# 1 The Brussels airport problem

## 1.1 Explain the given model

There are 5 data variables :

**n**, the number of plane.

**idealTime**, the preferred landing time for each plane.

**penalty**, the penalty by unit time from the preferred landing time.

**block**, the time the plane blocks the lane.

**maxDelay**, maximum difference of time between landing and preferred landing time.

Then, we have two decision variables :

**delay**, the delay of each plane.

**land**, the landing time of each plane.

Lastly, we use the constraint optimisation problem structure to solve our problem with an objective function which is the sum of all penalties of each planes, the constraints, and the heuristics. It's obvious that we want to minimize the accumulated penalties.

## 1.2 Design 2 different variable and/or value ordering heuristics for this problem.

```

1  ...
2  ...
3
4  minimize<cp>
5      sum(i in P) delay[i] * delay[i] * penalty[i]
6  subject to {
7      // main constraint
8      forall (i in P, j in P : j != i)
9          cp.post(land[j] < land[i] || land[j] > land[i] + block[i]);
10     // constraint to link variables
11     forall(i in P)
12         cp.post(land[i] == idealTime[i] + delay[i]);
13 } using {
14
15     /* Strategie 1 */
16     /*forall(i in P) by (idealTime[i])
17         tryall<cp>(v in D : delay[i].memberOf(v) ) by (abs(v))
18         label(delay[i],v);*/
19
20
21     /* Strategie 2 */
22     forall(i in P) by (-(block[i] * penalty[i]))
23         tryall<cp>(v in D : delay[i].memberOf(v) ) by (abs(v))
24         label(delay[i],v);
25
26
27     /* Strategie 3 */
28     /* forall(i in P) by (-(block[i] * penalty[i] / idealTime[i]))
29         tryall<cp>(v in D : delay[i].memberOf(v) ) by (abs(v))

```

```

30         label(delay[i],v);*/
31
32
33     //labelFF(delay);
34 }
35 ...

```

airport.co

### 1.3 Which criteria are meaningful for comparing different search strategies ?

We want our heuristic to minimize the number of next choices (we want good pruning), but what we basically want is a resolution as fast as possible.

### 1.4 Based on your criteria, compare your heuristics with the labelFF heuristic by testing them on the instance on iCampus.

	Strategy 1	Strategy 1	Strategy 1	labelFF
Time	75	34	37	OUT OF TIME

**Figure 1** – Time taken for different strategies

Based on the execution time, we will choose the 2nd strategy which is the best. LabelFF ran out of time because it implement the First Fail strategy, without any "ordering" on the values in the domain. Since the amount of variables and values inside their domains is quite enormous, it will take a lot of time to complete.

### 1.5 Consider the following strategy. ... Give an example with three planes where this strategy is wrong

## 2 The Knapsack Problem

### 2.1 A Branch & Bound approach

#### 2.1.1 Model the knapsack problem as Constraint Optimization

```

1 ...
2
3 // read the number of objects
4 int no = file.getInt();
5
6 // create a range for the objects
7 range P = 1..no;
8
9 // create data variables

```

```

10 int weight[P];           // weight of item
11 int usefulness[P];       // usefulness
12
13 // read data from file
14 forall(i in P) {
15     file.getInt();
16     weight[i] = file.getInt();
17     usefulness[i] = file.getInt();
18 }
19
20 int C = file.getInt();
21
22 // model variables
23 var<CP>{int} bin[1..no](cp, 0..1);
24 var<CP>{int} load(cp, 0..C);
25
26
27
28 maximize<cp>
29     sum(i in P) bin[i] * usefulness[i]
30 subject to {
31     cp.post(sum(i in 1..no) (bin[i] == 1) * weight[i] == load);
32 } using {
33
34
35     /* First strategy */
36     /*forall(i in P) by (weight[i])
37         tryall<cp>(v in 0..1 : bin[i].memberOf(v)) by (-v)
38         label(bin[i],v);*/
39
40     /* Second strategy */
41     forall(i in P) by (- usefulness[i])
42         tryall<cp>(v in 0..1 : bin[i].memberOf(v)) by (-v)
43         label(bin[i],v);
44
45
46     /* Third strategy */
47     /*forall(i in P) by (- usefulness[i] / weight[i])
48         tryall<cp>(v in 0..1 : bin[i].memberOf(v)) by (-v)
49         label(bin[i],v);*/
50
51     /* First Fail strategy */
52     //labelFF(bin);
53
54 }
55
56 ...

```

knapsack.co

### 2.1.2 Describe your model in the report.

We use 4 data variables :

- **no**, the number of objects
- **weight**, an array with the weight of each object

- **usefulness**, an array with the usefulness of each object
- **C**, the capacity of the knapsack

and 2 model variables :

- **bin**, with a domain wich is  $[0,1]$ . When an object has the value 0, it is not in the knapsack, and when it has value 1, it is in the knapsack.
- **load** is the weight of all the objects wich are curently in the knapsack.

Then we use the structure to solve constraint optimisation problems :

```

maximize<cp>
    objective_function
subject to {
    constraints
} using {
    search_heuristic
}

```

The objective function (that we maximize) is the total of the actual usefulness of the knapsack.

The constraints prevent to fill the knapsack with more weight than allowed.

Search\_heuristic are discussed in the following section.

### 2.1.3 Design 3 different heuristics for variable selection.

**The first heuristic** assigns variables which have the smallest weight first. Indeed, we could probably put more object in the knapsack, and fill it better with this strategy.

**The second heuristic** assigns variables which have the biggest utility first. Since the utility of those objects is bigger, they will probably be in the final solution.

**The third heuristic** tries to combine the 2 presented here above. It assigns objects with the higher value for ratio usefulness/weight. Object with higher ratio will also tend to be in the final solution.

### 2.1.4 Test your heuristics and the labelFF heuristic on the knapsack-A instances.

time	A1	A2	A3	A4	A5	A6	A7	A8	A9	A10
Heuristic 1	76	37	21	50	73	95	219	318	1098	2189
Heuristic 2	15	10	9	14	20	27	63	124	213	349
Heuristic 3	29	21	22	28	34	63	145	166	899	1476
labelFF	17	15	10	18	26	45	98	113	672	1034

**Figure 2** – Search time depending on instances and heuristics

#choices	A1	A2	A3	A4	A5	A6	A7	A8	A9	A10
Heuristic 1	2151	1068	609	1488	2058	2668	6177	10206	36020	73265
Heuristic 2	489	343	329	569	697	891	2309	3986	8218	12734
Heuristic 3	1219	678	736	858	1016	2103	4877	5850	29973	49940
labelFF	538	468	339	617	860	1590	3756	4431	26737	41081

**Figure 3** – Number of choices depending on instances and heuristics

#fail	A1	A2	A3	A4	A5	A6	A7	A8	A9	A10
Heuristic 1	4266	2098	1140	2940	3975	5298	12193	20306	71907	146472
Heuristic 2	796	507	473	790	1186	1451	3792	7176	13027	21112
Heuristic 3	1731	1120	1045	1454	1852	3667	8608	10247	55578	90988
labelFF	722	574	437	847	1149	1966	4781	5723	33206	54804

**Figure 4** – Number of failures depending on instances and heuristics

#propag :	A1	A2	A3	A4	A5	A6	A7	A8	A9	A10
Heuristic 1	25586	12644	7115	16302	25829	36401	81296	129069	427978	819394
Heuristic 2	3634	2197	1840	2451	4878	6845	15387	34646	50685	91129
Heuristic 3	6250	5299	3798	6189	8609	17699	41002	48293	280535	391867
labelFF	6306	5371	3867	6210	8680	17772	41030	48393	280603	391948

**Figure 5** – Number of propagation depending on instances and heuristics

Results here above only show the results for the 10 first files. The other results are available in the directory `Tests_Q_2_3_1`.

### 2.1.5 Present and discuss the results in your report

We can observe on results here above that the second heuristic is always better while looking at time values (behave on file A7, but difference is minor), which is the most important criterium.

Looking at the table with number of choices, we can observe, once again, that the second heuristic is better since it has less different choices (It shows a better pruning).

Looking at table with number of failures, we can observe that the second heuristic is better since it has a low number of failures. It converges more quickly on good values.

Finally, we can observe that with heuristic 2, we have less propagation than with other heuristics.

In fact, those four tables are linked. Since this heuristic has better pruning (low # of choices), it will converges faster (low # of fails), which implies less propagation and faster times.

Considering all those results, the search strategy we will use for the remainder of this assignment is the second one.

## 2.2 Optimization over iterations

### 2.2.1 Model the knapsack problem as a Constraint Satisfaction Problem.

```

1  ...
2
3  // read the number of objects
4  int no = file.getInt();
5
6  // create a range for the objects
7  range P = 1..no;
8
9  // create data variables
10 int weight[P];           // weight of item
11 int usefulness[P];       // usefulness
12
13 // read data from file
14 forall(i in P) {
15     file.getInt();
16     weight[i] = file.getInt();
17     usefulness[i] = file.getInt();
18 }
19
20 int C = file.getInt();
21 Integer ub(0);
22 ub := getUB(no, C, weight, usefulness);
23 //cout << "UB: " << ub << endl;
24
25 // model variables
26 var<CP>{int} bin[1..no](cp, 0..1);
27 var<CP>{int} load(cp, 0..C);
28 var<CP>{int} totalUsefulness(cp, 0..ub);
29
30 whenever cp.getSearchController().@onCompletion() {
31     //cout << "FAIL" << endl;
32     ub := ub-1;
33     cp.reStart();
34 }
35
36 whenever cp.getSearchController().@onFeasibleSolution(Solution s){
37     cp.exit();
38 }
39
40 solve<cp> {
41     cp.post(sum(i in 1..no) (bin[i] == 1) * weight[i] == load);
42     cp.post(sum(i in 1..no) (bin[i] == 1) * usefulness[i] ==
43         totalUsefulness);
44 } using {
45     cp.post(totalUsefulness==ub);
46     forall(i in P) by (- usefulness[i])
47         tryall<cp>(v in 0..1 : bin[i].memberOf(v)) by (-v)

```



```

48 |         label ( bin [ i ] , v ) ;
49 |     }

```

knapsackUB.co

### 2.2.2 In order to implement the optimization over iterations ...

We added Integer ub which will be the upper bound, and is assigned thanks to the `getUB` function. We added a decision variable `totalUsefulness` which will be forced to take the value of the upper bound. Succeed event that will finish the search, and failure events that will reduce the upper bound from 1 and restart the search.

### 2.2.3 Which of these points (i., ii., iii.) do you need to execute on which events ?

In case of success, we succeed in finding the best value for the `totalUsefulness` (since it can't be bigger than the upper bound). So we end the search (iii). Otherwise, in case of failure, we modify ub value (we decrease by 1), and we restart the search (i and ii). We will continue like that until the upper bound will be the optimal solution, and then will succeed and end the search.

### 2.2.4 How do you modify the value of ub to be sure to find the optimal solution ?

Since we only use integer, we can decrease ub by 1 at each failure.

### 2.2.5 Can you explain why we initialize ub with an upper bound instead of any other value ?

If ub is not an upper bound, it means that the search could end with a non optimal value.

### 2.2.6 Experiment this program on the instances knapsack-A, -B.

See Figures 6, 7.

### 2.2.7 Present and discuss the results in your report.

The first thing we observe is that the upper bound determined at the beginning is always the `totalUsefulness`. It means that we don't change the upper bound value during the execution of the program. The execution time is really better than before, and same for the number of choices and number of propagations. Let's observe that the number of failures is always at 0, which means that we immediately get the optimal solution.

File	ub	totalUsefulness	time	# choices	# fail	# propag
A1	21	21	5	25	0	30
A2	21	21	5	25	0	30
A3	21	21	5	25	0	28
A4	21	21	5	25	0	30
A5	26	26	5	25	0	30
A6	34	34	6	25	0	30
A7	43	43	5	25	2	38
A8	44	44	7	25	3	41
A9	45	45	5	25	0	32
A10	47	47	8	25	0	32
A11	50	50	5	25	0	32
A12	65	65	5	25	0	34
A13	69	69	5	25	0	34
A14	73	73	5	25	0	36
A15	70	70	6	25	0	34
A16	78	78	6	25	1	39
A17	89	89	6	25	0	36
A18	86	86	5	25	0	36
A19	106	106	7	25	0	38
A20	110	110	5	25	0	38
A21	117	117	7	25	0	40
A22	104	104	5	25	0	38
A23	98	98	5	25	0	38
A24	131	131	5	25	0	42
A25	103	103	6	25	0	40

**Figure 6** – Knapsack results with upper bound

File	ub	totalUsefulness	time	# choices	# fail	# propag
B1	921	921	1112	73715	4208	407136
B2	929	929	108	8546	1876	32075
B3	908	908	617	46118	1412	204306
B4	933	933	171	12373	976	56971
B5	862	862	124	9136	1013	41892
B6	809	809	396	28125	1029	146337
B7	810	810	170	13926	475	57613
B8	841	841	298	21285	965	102120
B9	806	806	161	12400	438	57102
B10	918	918	2244	144921	6126	832368
B11	855	855	1208	80406	2223	451408
B12	801	801	355	26862	1093	123243
B13	879	879	18	1089	540	4233
B14	822	822	184	14599	872	54322
B15	817	817	500	31753	2171	175186
B16	783	783	197	15538	1331	69005
B17	851	851	790	57157	2065	264039
B18	825	825	323	27840	262	109188
B19	822	822	354	28969	1396	115793
B20	901	901	60	3998	1222	20041
B21	806	806	170	12070	217	68842
B22	855	855	349	24653	671	129623
B23	773	773	265	23505	515	91134
B24	794	794	77	6389	515	26978
B25	785	785	176	14402	240	59226

Figure 7 – Knapsack results with upper bound

## 2.3 Optimization via divide and conquer

**2.3.1** In order to implement the optimization via divide and conquer you will have to ...

```

1 ...
2
3 // read the number of objects
4 int no = file.getInt();
5
6 // create a range for the objects
7 range P = 1..no;
8
9 // create data variables
10 int weight[P];           // weight of item
11 int usefulness[P];       // usefulness
12
13 // read data from file
14 forall(i in P) {

```

```

15     file.getInt();
16     weight[i] = file.getInt();
17     usefulness[i] = file.getInt();
18 }
19
20 int C = file.getInt();
21
22 Integer ub(0);
23 ub := getUB(no, C, weight, usefulness);
24 //cout << "UB: " << ub << endl;
25 Integer lb(0);
26 lb := getLB(no, C, weight, usefulness);
27 //cout << "LB: " << lb << endl;
28
29 // model variables
30 var<CP>{int} bin[1..no](cp, 0..1);
31 var<CP>{int} load(cp, 0..C);
32 var<CP>{int} totalUsefulness(cp, lb..ub);
33
34 whenever cp.getSearchController().@onCompletion(){
35     //cout << "FAIL" << endl;
36     ub := (int) ceil((lb+ub)/2.0) - 1;
37     //cout << lb << "-" << ub << endl;
38     cp.reStart();
39 }
40
41 whenever cp.getSearchController().@onFeasibleSolution(Solution s){
42     //cout << "SUCEED" << endl;
43     lb := (int) ceil((lb+ub)/2.0);
44     //cout << lb << "-" << ub << endl;
45     if (lb == ub){
46         cp.exit();
47     }
48     cp.reStart();
49 }
50
51 solve<cp> {
52     cp.post(sum(i in 1..no) (bin[i] == 1) * weight[i] == load);
53     cp.post(sum(i in 1..no) (bin[i] == 1) * usefulness[i] ==
54         totalUsefulness);
55 } using {
56     cp.post(totalUsefulness<=ub);
57     cp.post(totalUsefulness>=ceil((lb+ub)/2.0));
58     forall(i in P) by (- usefulness[i])
59         tryall<cp>(v in 0..1 : bin[i].memberOf(v)) by (-v)
60         label(bin[i], v);
61 }
62
63 ...
64

```

knapsackUB\_LB.co

### 2.3.2 Which of these points (i., ii., iii., iv.) do you need to execute on which events ?

Note : Since the search runs on domain  $[\text{ceil}(\frac{lb+ub}{2}), ub]$ , we are sure that if there is a solution, this is not the lb. So, if  $lb=34$  and  $ub=35$ , ceil function will make the search on the domain  $[35,35]$  and not  $[34,35]$  if we had used the floor function, which would have made the rest of the program more complicated.

In case of success, we have to update the lower bound to the lower value of the domain used for the search which is the new lower bound (since there was a solution with this bound) (i). If this new lower bound is equal to the upper bound, we have found the optimal value, and we can end the search (iv), otherwise, we have to continue the search (iii).

In case of failure, we have no solutions in the interval  $[\text{ceil}(\frac{lb+ub}{2}), ub]$ , thus we can change the upper bound to  $\text{ceil}(\frac{lb+ub}{2}) - 1$  (ii), and restart the search (iii).

### 2.3.3 Experiment this version on the instances knapsack-A,-B,-C

See Figures 8, 9, 10.

### 2.3.4 Present and discuss the results in your report.

The time needed to find the optimal solution is really low. We can also observe that the solution is found with few search restart (until 9), and same with search restart, it stays really fast. Using upper bound and lower bound reduces amazingly the time needed to find the optimal solution.

file	lb	ub	totalUsefulness	# search	time
A1	20	21	21	1	10
A2	21	21	21	1	9
A3	21	21	21	1	10
A4	21	21	21	1	10
A5	25	26	26	1	10
A6	33	34	34	1	9
A7	43	43	43	1	10
A8	44	44	44	1	10
A9	45	45	45	1	11
A10	47	47	47	1	9
A11	50	50	50	1	10
A12	65	65	65	1	11
A13	69	69	69	1	11
A14	73	73	73	1	10
A15	69	70	70	1	11
A16	76	78	78	2	10
A17	89	89	89	1	11
A18	86	86	86	1	10
A19	106	106	106	1	10
A20	110	110	110	1	10
A21	117	117	117	1	9
A22	104	104	104	1	10
A23	98	98	98	1	10
A24	130	131	131	1	10
A25	103	103	103	1	10

**Figure 8** – Knapsack results with lower and upper bound

file	lb	ub	totalUsefulness	# search	time
B1	622	934	934	9	15
B2	642	934	934	9	15
B3	774	947	947	8	15
B4	696	943	943	8	15
B5	602	872	872	9	15
B6	651	835	835	8	14
B7	589	826	826	8	15
B8	636	859	859	8	15
B9	582	836	836	8	15
B10	811	951	951	8	15
B11	775	891	891	7	14
B12	676	817	817	8	15
B13	671	880	880	8	16
B14	592	836	836	8	15
B15	706	826	826	7	14
B16	738	796	796	6	14
B17	618	878	878	9	15
B18	628	858	858	8	15
B19	642	843	843	8	15
B20	734	904	904	8	14
B21	640	836	836	8	14
B22	741	883	883	8	14
B23	670	794	794	7	15
B24	524	802	802	9	15
B25	665	805	805	8	14

**Figure 9** – Knapsack results with lower and upper bound

file	lb	ub	totalUsefulness	# search	time
C1	62	103	103	6	16
C2	72	99	99	5	15
C3	77	94	94	5	15
C4	86	110	110	5	15
C5	87	108	108	5	15
C6	86	97	97	4	14
C7	87	113	113	5	16
C8	87	128	128	6	16
C9	92	116	116	5	15
C10	85	105	105	5	15
C11	93	125	125	6	15
C12	67	115	115	6	16
C13	91	115	115	5	14
C14	76	115	115	6	16
C15	99	120	120	5	15
C16	91	116	116	5	15
C17	67	98	98	5	16
C18	83	111	111	5	15
C19	72	99	99	5	15
C20	75	117	117	6	16
C21	73	110	110	6	15
C22	84	108	108	5	15
C23	73	115	115	6	17
C24	90	109	109	5	15
C25	82	116	116	6	16

**Figure 10** – Knapsack results with lower and upper bound