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

1 The Brussels airport problem

1.1 Explain the given model

There are 5 data variables :
n, the number of plane.
idealTime, the prefered landing time for each plane.
penality, the penalty by unit time from the preffered 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.

```
...
...

minimize<cp>
    sum(i in P) delay[i] * delay[i] * penality[i]
subject to {
    // main constraint
  forall (i in P, j in P : j != i)
        cp.post(land[j] < land[i] || land[j] > land[i] + block[i]);
    // constraint to link variables
    forall(i in P)
        cp.post(land[i] == idealTime[i] + delay[i]);
} using {

   /* Strategie 1 */
   /*forall(i in P) by (idealTime[i])
      tryall<cp>(v in D : delay[i].memberOf(v) ) by (abs(v))
         label(delay[i],v);*/


   /* Strategie 2 */
   forall(i in P) by (-(block[i] * penality[i]))
      tryall<cp>(v in D : delay[i].memberOf(v) ) by (abs(v))
         label(delay[i],v);


   /* Strategie 3 */
  /* forall(i in P) by (-(block[i] * penality[i] / idealTime[i]))
      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

Instinctively, it is easy to consider the fact that this strategy is attracted by local optima. Indeed, this strategy does not present any mechanism of diversification hence "forces" the solver to choose the first solution to the CSP. With the following set :

	Preferred time	block	penalty
Plane 1	4	2	100
Plane 2	7	2	50
Plane 3	5	2	1

With a delay of 0, there are no solutions. With a delay of 1, we find 2 solutions but the optimal one is with the plane 1 with time slot 4-5, plane 3 with time slots 6-7 and plane 2 with time slots 8-9. The total penalty is 50. With a delay of 2, the optimal solution will be the same. The search would end here.
But this is not the optimal solution for the problem. With delay bigger allowed, we will find as optimal solution time slots 4-5 for plane 1, 7-8 for the second and 2-3 for the third, which equals to a total penalty of 3 which is really lower than the one found before.

2 The Knapsack Problem

2.1 A Branch & Bound approach

2.1.1 Model the knapsack problem as Constraint Optimization

```
...

// read the number of objects
int no = file.getInt();

// create a range for the objects
range P = 1..no;

// create data variables
int weight[P];          // weight of item
int usefulness[P];      // usefulness

// read data from file
forall(i in P) {
   file.getInt();
   weight[i] = file.getInt();
   usefulness[i] = file.getInt();
}

int C = file.getInt();

// model variables
var<CP>{int} bin[1..no](cp, 0..1);
var<CP>{int} load(cp, 0..C);



maximize<cp>
          sum(i in P) bin[i]  * usefulness[i]
subject to {
    cp.post(sum(i in 1..no) (bin[i] == 1) * weight[i] == load);
} using {


   /* First strategy */
   /*forall(i in P) by (weight[i])
      tryall<cp>(v in 0..1 : bin[i].memberOf(v)) by (-v)
         label(bin[i],v);*/

   /* Second strategy */
   forall(i in P) by (- usefulness[i])
      tryall<cp>(v in 0..1 : bin[i].memberOf(v)) by (-v)
         label(bin[i],v);


   /* Third strategy */
   /*forall(i in P) by (- usefulness[i] / weight[i])
      tryall<cp>(v in 0..1 : bin[i].memberOf(v)) by (-v)
         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.

```
...

// read the number of objects
int no = file.getInt();

// create a range for the objects
range P = 1..no;

// create data variables
int weight[P];          // weight of item
int usefulness[P];      // usefulness

// read data from file
forall(i in P) {
   file.getInt();
   weight[i] = file.getInt();
   usefulness[i] = file.getInt();
}

int C = file.getInt();
Integer ub(0);
ub := getUB(no, C, weight, usefulness);


// model variables
var<CP>{int} bin[1..no](cp, 0..1);
var<CP>{int} load(cp, 0..C);
var<CP>{int} totalUsefulness(cp, 0..ub);

whenever cp.getSearchController()@onCompletion(){
            ub := ub-1;
            cp.reStart();
```

```
33 }
34
35 whenever cp.getSearchController()@onFeasibleSolution(Solution s){
36             cp.exit();
37 }
38
39 solve<cp> {
40     cp.post(sum(i in 1..no) (bin[i] == 1) * weight[i] == load);
41     cp.post(sum(i in 1..no) (bin[i] == 1) * usefulness[i] ==
           totalUsefulness);
42
43 } using {
44       cp.post(totalUsefulness==ub);
45       forall(i in P) by (- usefulness[i])
46          tryall<cp>(v in 0..1 : bin[i].memberOf(v)) by (-v)
47             label(bin[i],v);
48 }
```

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. Suceed 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 totalUseffulness (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 an non optimal value.

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

See Figures 6, 7.

File	ub	totalUsefulness	time	# choices	# fail	# propag
A1	21	21	5	25	0	30
A2	21	21	5	25	0	30
A3	19	21	5	25	0	28
A4	21	21	5	25	0	30
A5	25	26	5	25	0	30
A6	32	34	6	25	0	30
A7	43	43	5	25	2	38
A8	40	44	7	25	3	41
A9	45	45	5	25	0	32
A10	47	47	8	25	0	32
A11	49	50	5	25	0	32
A12	65	65	5	25	0	34
A13	68	69	5	25	0	34
A14	73	73	5	25	0	36
A15	69	70	6	25	0	34
A16	77	78	6	25	1	39
A17	89	89	6	25	0	36
A18	86	86	5	25	0	36
A19	105	106	7	25	0	38
A20	109	110	5	25	0	38
A21	117	117	7	25	0	40
A22	104	104	5	25	0	38
A23	97	98	5	25	0	38
A24	130	131	5	25	0	42
A25	103	103	6	25	0	40

Figure 6 – Knapsack results with upper bound

2.2.7 Present and discuss the results in your report.

We can see that when the totalUsefulness is near the initial upper bound, the optimal solution is found quite directly. (A files) But when it is far from the upper bound, it can take a lot of time. More the solution is far, more time it will take.

File	ub	totalUsefulness	time	# choices	# fail	# propag
B1	544	921	1112	73715	4208	407136
B2	594	929	108	8546	1876	32075
B3	819	908	617	46118	1412	204306
B4	673	933	171	12373	976	56971
B5	670	862	124	9136	1013	41892
B6	695	809	396	28125	1029	146337
B7	742	810	170	13926	475	57613
B8	549	841	298	21285	965	102120
B9	663	806	161	12400	438	57102
B10	634	918	2244	144921	6126	832368
B11	642	855	1208	80406	2223	451408
B12	621	801	355	26862	1093	123243
B13	716	879	18	1089	540	4233
B14	644	822	184	14599	872	54322
B15	674	817	500	31753	2171	175186
B16	686	783	197	15538	1331	69005
B17	594	851	790	57157	2065	264039
B18	750	825	323	27840	262	109188
B19	650	822	354	28969	1396	115793
B20	730	901	60	3998	1222	20041
B21	596	806	170	12070	217	68842
B22	760	855	349	24653	671	129623
B23	674	773	265	23505	515	91134
B24	639	794	77	6389	515	26978
B25	734	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 ...

```
...

// read the number of objects
int no = file.getInt();

// create a range for the objects
range P = 1..no;

// create data variables
int weight[P];          // weight of item
int usefulness[P];      // usefulness

// read data from file
forall(i in P) {
   file.getInt();
   weight[i] = file.getInt();
   usefulness[i] = file.getInt();
}

int C = file.getInt();

Integer ub(0);
ub := getUB(no, C, weight, usefulness);

Integer lb(0);
lb := getLB(no, C, weight, usefulness);

// model variables
var<CP>{int} bin[1..no](cp, 0..1);
var<CP>{int} load(cp, 0..C);
var<CP>{int} totalUsefulness(cp, lb..ub);

whenever cp.getSearchController()@onCompletion(){
           ub := totalUsefulness;
           cp.reStart();
}

whenever cp.getSearchController()@onFeasibleSolution(Solution s){
           lb := totalUsefulness;
           if (lb == ub){
              cp.exit();
           }
           cp.reStart();
}

solve<cp> {
    cp.post(sum(i in 1..no) (bin[i] == 1) * weight[i] == load);
           cp.post(sum(i in 1..no) (bin[i] == 1) * usefulness[i] ==
               totalUsefulness);

```

```
} using {
      cp.post(totalUsefulness<=ub);
      cp.post(totalUsefulness>=ceil((lb+ub)/2.0));
      forall(i in P) by (- usefulness[i])
         tryall<cp>(v in 0..1 : bin[i].memberOf(v)) by (-v)
            label(bin[i],v);
}


...
```

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

In case of success, we have to update the lower bound to the solution we found during the new search which will be the new lower bound (since there were a solution with this bound) (i). If this new lower bound is equal to the upper bound, we have find 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 $[ceil(\frac{lb+ub}{2}), ub]$, thus we can change the upper bound to $ceil(\frac{lb+ub}{2}) - 1$ (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.

We can see that the time needed to find the optimal solution on the A files stays really low. Sometimes, we have a search that restarts, but it's rare. The time spend on B files is really lower than with the previous strategy. And number of search restart is also really lower (since for previous strategy, it was equal to the difference between the totalUsefulness and the initial ub). Using upper bound and lower bound reduces amazingly the time needed to find the optimal solution.

file	lb	ub	totalUsefulness	# search	time
A1	21	21	21	1	11
A2	21	21	21	1	10
A3	19	21	21	1	11
A4	21	21	21	1	10
A5	25	26	26	1	10
A6	32	34	34	1	10
A7	43	43	43	1	10
A8	40	44	44	2	10
A9	45	45	45	1	10
A10	47	47	47	1	9
A11	49	50	50	1	9
A12	65	65	65	1	10
A13	68	69	69	1	10
A14	73	73	73	1	10
A15	69	70	70	1	10
A16	77	78	78	1	10
A17	89	89	89	1	9
A18	86	86	86	1	10
A19	105	106	106	1	10
A20	109	110	110	1	10
A21	117	117	117	1	10
A22	104	104	104	1	10
A23	97	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	544	934	921	8	711
B2	594	934	929	6	150
B3	819	947	908	4	110
B4	673	943	933	7	150
B5	670	872	862	7	128
B6	695	835	809	6	146
B7	742	826	810	7	93
B8	549	859	841	8	173
B9	663	836	806	7	65
B10	634	951	918	9	934
B11	642	891	855	8	407
B12	621	817	801	7	203
B13	716	880	879	4	50
B14	644	836	822	6	118
B15	674	826	817	7	410
B16	686	796	783	7	171
B17	594	878	851	7	266
B18	750	858	825	7	113
B19	650	843	822	6	133
B20	730	904	901	6	146
B21	596	836	806	8	60
B22	760	883	855	5	94
B23	674	794	773	7	134
B24	639	802	794	6	89
B25	734	805	785	6	81

Figure 9 – Knapsack results with lower and upper bound

file	lb	ub	totalUsefulness	# search	time
C1	79	103	101	5	1477
C2	78	99	98	6	621
C3	77	94	94	3	130
C4	82	110	108	5	279
C5	86	108	106	4	316
C6	75	97	96	5	168
C7	71	113	112	6	1473
C8	109	128	126	4	316
C9	82	116	112	4	158
C10	90	105	102	5	688
C11	88	125	125	2	267
C12	81	115	112	4	602
C13	94	115	114	6	777
C14	81	115	113	5	862
C15	94	120	118	4	479
C16	79	116	114	5	704
C17	63	98	94	6	462
C18	88	111	109	5	395
C19	59	99	97	5	279
C20	74	117	115	5	733
C21	89	110	109	5	607
C22	70	108	108	4	284
C23	83	115	114	6	612
C24	100	109	107	4	120
C25	86	116	114	5	466

Figure 10 – Knapsack results with lower and upper bound