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))
```

```
        label(delay[i],v);*/


    //labelFF(delay);
}
...
```

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

```
...

// 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);*/

   /* First Fail strategy */
   //labelFF(bin);

}

...
```

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);
//cout << "UB: " << ub << endl;

// 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(){
              //cout << "FAIL" << endl;
              ub := ub-1;
              cp.reStart();
}

whenever cp.getSearchController()@onFeasibleSolution(Solution s){
              cp.exit();
}

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);
        forall(i in P) by (- usefulness[i])
           tryall<cp>(v in 0..1 : bin[i].memberOf(v)) by (-v)
```

```
        label(bin[i],v);
}
```

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 2.2.6, 6.

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
A-1.txt	21	21	5	25	0	30
A-2.txt	21	21	5	25	0	30
A-3.txt	21	21	5	25	0	28
A-4.txt	21	21	5	25	0	30
A-5.txt	26	26	5	25	0	30
A-6.txt	34	34	6	25	0	30
A-7.txt	43	43	5	25	2	38
A-8.txt	44	44	7	25	3	41
A-9.txt	45	45	5	25	0	32
A-10.txt	47	47	8	25	0	32
A-11.txt	50	50	5	25	0	32
A-12.txt	65	65	5	25	0	34
A-13.txt	69	69	5	25	0	34
A-14.txt	73	73	5	25	0	36
A-15.txt	70	70	6	25	0	34
A-16.txt	78	78	6	25	1	39
A-17.txt	89	89	6	25	0	36
A-18.txt	86	86	5	25	0	36
A-19.txt	106	106	7	25	0	38
A-20.txt	110	110	5	25	0	38
A-21.txt	117	117	7	25	0	40
A-22.txt	104	104	5	25	0	38
A-23.txt	98	98	5	25	0	38
A-24.txt	131	131	5	25	0	42
A-25.txt	103	103	6	25	0	40

Figure 6 – Knapsack results with upper bound

File	ub	totalUsefulness	time	# choices	# fail	# propag
B1	934	934	9	50	0	53
B2	934	934	10	50	0	52
B3	947	947	10	50	0	52
B4	943	943	9	50	0	55
B5	872	872	10	50	0	53
B6	835	835	10	50	0	52
B7	826	826	10	50	0	54
B8	859	859	10	50	0	53
B9	836	836	10	50	0	54
B10	951	951	10	50	0	54
B11	891	891	9	50	0	54
B12	817	817	9	50	0	54
B13	880	880	10	50	0	55
B14	836	836	10	50	0	54
B15	826	826	9	50	0	52
B16	796	796	11	50	0	54
B17	878	878	10	50	0	54
B18	858	858	10	50	0	55
B19	843	843	9	50	0	54
B20	904	904	9	50	0	53
B21	836	836	9	50	0	52
B22	883	883	9	50	0	54
B23	794	794	9	50	0	53
B24	802	802	10	50	0	55
B25	805	805	9	50	0	56

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);
//cout << "UB: " << ub << endl;
Integer lb(0);
lb := getLB(no, C, weight, usefulness);
//cout << "LB: " << lb << endl;

// 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(){
           //cout << "FAIL" << endl;
           ub := (int) ceil((lb+ub)/2.0) - 1;
           //cout << lb << "-" << ub << endl;
           cp.reStart();
}

whenever cp.getSearchController()@onFeasibleSolution(Solution s){
           //cout << "SUCEED" << endl;
           lb := (int) ceil((lb+ub)/2.0);
           //cout << lb << "-" << ub << endl;
           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 lower value of the domain used for the search which is 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 7, 8, 9.

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 (untill 9), and same with search restart, it stay 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