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

23 mars 2014

Table des matières

1 The Brussels airport problem

1.1 Explain the given model

TODO

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

TODO

1.3 Which criteria are meaningful for comparing different search strategies ?

TODO

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

TODO

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

TODO

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 :	H1	H2	H3	H4	H5	H6	H7	H8	H9	H10
s1	76	37	21	50	73	95	219	318	1098	2189
s2	15	10	9	14	20	27	63	124	213	349
s3	29	21	22	28	34	63	145	166	899	1476
ff	17	15	10	18	26	45	98	113	672	1034

Figure 1 – Search time 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 ... TODO

2.1.5 Present and discuss the results in your report

TODO

#choices	H1	H2	H3	H4	H5	H6	H7	H8	H9	H10
s1	2151	1068	609	1488	2058	2668	6177	10206	36020	73265
s2	489	343	329	569	697	891	2309	3986	8218	12734
s3	1219	678	736	858	1016	2103	4877	5850	29973	49940
s4	538	468	339	617	860	1590	3756	4431	26737	41081

Figure 2 – Number of choices depending on instances and heuristics

#fail	H1	H2	H3	H4	H5	H6	H7	H8	H9	H10
s1	4266	2098	1140	2940	3975	5298	12193	20306	71907	146472
s2	796	507	473	790	1186	1451	3792	7176	13027	21112
s3	1731	1120	1045	1454	1852	3667	8608	10247	55578	90988
s4	722	574	437	847	1149	1966	4781	5723	33206	54804

Figure 3 – Number of failures depending on instances and heuristics

#propag :	H1	H2	H3	H4	H5	H6	H7	H8	H9	H10
s1	25586	12644	7115	16302	25829	36401	81296	129069	427978	819394
s2	3634	2197	1840	2451	4878	6845	15387	34646	50685	91129
s3	6250	5299	3798	6189	8609	17699	41002	48293	280535	391867
s4	6306	5371	3867	6210	8680	17772	41030	48393	280603	391948

Figure 4 – Number of propagation depending on instances and heuristics

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();
}
```

```
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
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.

File	ub	totalUsefulness	time	# choices	# fail	# propag
A1	21	21	9	25	0	27
A2	21	21	9	25	0	27
A3	21	21	9	25	0	27
A4	21	21	9	25	0	27
A5	26	26	9	25	0	28
A6	34	34	9	25	0	27
A7	43	43	10	25	0	28
A8	44	44	10	25	0	28
A9	45	45	9	25	0	28
A10	47	47	8	25	0	28
A11	50	50	9	25	0	28
A12	65	65	9	25	0	29
A13	69	69	9	25	0	29
A14	73	73	9	25	0	30
A15	70	70	9	25	0	29
A16	78	78	9	25	0	29
A17	89	89	9	25	0	30
A18	86	86	9	25	0	30
A19	106	106	9	25	0	31
A20	110	110	10	25	0	31
A21	117	117	9	25	0	32
A22	104	104	8	25	0	33
A23	98	98	9	25	0	31
A24	131	131	9	25	0	33
A25	103	103	9	25	0	32

2.2.7 Present and discuss the results in your report.

TODO

2.3 Optimization via divide and conquer

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

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

```
...

// 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);

} 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

TODO

2.3.4 Present and discuss the results in your report.

TODO