Report for Project 5

Author: Yihong Zhou (yzhou8) & Mengwen Li (mli2)

Instructions on compiling and running your program

The program is written in Java. IntelliJ was used to compile the project and generating the “project5.jar” file.

To run “project5.jar” file with input file and output file, use following command:

java -jar project5.jar <input filename> > <output filename>

The default method is backtracking + MRV + degree heuristic + LCV + Forward Checking.

To switch between methods, go to “BackTrack.java” class and uncomment the desired methods.

For the bonus point, we tried to implement AC3 preprocessing. The AC3 algorithm is in “BackTrack.java” file and it’s partially working.

detailed description of your approach. That is, what search method and what heuristics were used, how was AC employed, etc. Please try to describe your approach in an algorithmic manner (i.e., include pseudo-code of the overall approach).

In our implementation, backtracking search method was used. Also, minimum remaining values (MRV), degree heuristic to break ties and least constraining value (LCV) heuristics were implemented.

Forward checking was implemented to ensure arc consistency as the algorithm goes.

The screenshot of our implementation is shown in the following:

```
66      public Solution backtrack(Solution s, Items items, Bags bags, Matrices matrices){
67          // If solution contains all items, finish.
68          if(s.containsAllItem(items)){
69              // Check for global constraints.
70              if(this.finishConsistant(bags)){
71                  return s;
72              }
73              return new Solution();
74          }
75          // Get an unused item according to solution.
76          Item varItem = items.getUnusedItem(s);
77          // Get the domain for this item.
78          ArrayList<Integer> values = varItem.updateAndGetDomain(matrices);
79          if(items.getSize() == 1){
80              values = bags.getBagsIds();
81          }
82          for(int i = 0; i < values.size(); i ++){
83              Bags tmpBags = new Bags(bags);
84              // Update bag with item.
85              tmpBags.updateBag(values.get(i), varItem);
86              // Check for constraints.
87              if(consistant(tmpBags)){
88                  Pair pair = new Pair(varItem, bags.getBagById(values.get(i)));
89                  // Check for binary constraints.
90                  if(!this.binaryConsistant(pair, s)){
91                      continue;
92                  }
93                  // Add to solution.
94                  s.addPair(pair);
95                  Solution tmpSolution = backtrack(s, items, tmpBags, matrices);
96                  // If no solution found, remove this pair and try next value.
97                  if(tmpSolution.isEmpty()){
98                      s.removePair(pair);
99                  }else{
100                     // If found a solution, return.
101                     return tmpSolution;
102                 }
103             }
104         }
105         // If all values are tried and no solution found, return empty.
106         return new Solution();
107     }
```

This is the implementation for only backtracking algorithm.

First, in line 68, the program checks whether the solution contains all the items. If it does, it checks for minimum fitting limit constraint and minimum weight limit constraint to make sure that the solution found is valid. If it is valid, the solution is returned, if not, an empty solution is returned.

Then, in line 76, the program gets an unused item, then, it gets the domain for this item from constraint matrices. Then, in line 82, the program tries all the values for this variable until it found a solution.

In line 85, the bag corresponding to the selected value is updated, the item-bag pair is pushed into solution and backtrack is called again to get value for the next variable.

On success, a solution is returned, on failure, an empty solution is returned.

To implement MRV and degree heuristic to break ties, “items.getUnusedItem()” is changed to “items.getUnusedItemMRV”.

The implementation for “getUnusedItemMRV” is as following:

```
65      public Item getUnusedItemMRV(Solution sol, Matrices matrices){
66          ArrayList<Item> usedItems = sol.getUsedItems();
67          ArrayList<Item> tmpList = new ArrayList<~>();
68          for(int i = 0; i < items.size(); i ++){
69              tmpList.add(new Item(items.get(i)));
70          }
71          tmpList.removeAll(usedItems);
72          for(int i = 0; i < tmpList.size(); i ++){
73              tmpList.get(i).updateAndGetDomain(matrices);
74          }
75          Collections.sort(tmpList);
76          int minRemVal = tmpList.get(0).getDomain().size();
77          ArrayList<Item> equalList = new ArrayList<~>();
78          for(int i = 0; i < tmpList.size(); i ++){
79              if(tmpList.get(i).getDomain().size() == minRemVal){
80                  equalList.add(tmpList.get(i));
81              }
82          }
83          if(equalList.size() == 1){
84              return equalList.get(0);
85          }else{  // Apply degree heuristic for items with same domain count.
86              int maxCons = Integer.MIN_VALUE;
87              int rtnInd = 0;
88              for(int i = 0; i < equalList.size(); i ++){
89                  int consCont = equalList.get(i).getConsCnt();
90                  if(consCont > maxCons){
91                      maxCons = consCont;
92                      rtnInd = i;
93                  }
94              }
95              if(rtnInd == 0){
96                  int maxWeight = Integer.MIN_VALUE;
97                  for(int i = 0; i < equalList.size(); i ++){
98                      if(equalList.get(i).getWeight() > maxWeight){
99                          maxWeight = equalList.get(i).getWeight();
100                         rtnInd = i;
101                     }
102                 }
103             }
104             return equalList.get(rtnInd);
105         }
106     }
```

This method first gets the domains for all the unused items, then, it sorts the unused items according to the number of elements in their domains. If there's only one variable has the smallest value, this variable is returned. If multiple variables all have the same smallest value, the degree heuristic is used. The program tries to find the variable with the most constraints. If multiple variables all have the max number of constraints, the method selects the variable with the maximum weight.

To implement LCV, "ArrayList<Integer> values = varItem.updateAndGetDomain(matrices);" is replaced by "ArrayList<Integer> values = varItem.sortDomain(matrices);".
The implementation for "sortDomain()" method is as following:

```
68      // Sort the domain according to Least Constraining Value.
69      public ArrayList<Integer> sortDomain(Matrices matrices){
70          this.setDomain(matrices);
71          ArrayList<Integer> rtnList = new ArrayList<~>();
72          // Get the total available count list.
73          ArrayList<Integer> tacList = new ArrayList<~>();
74          for(int i = 0; i < domain.size(); i ++){
75              int tac = matrices.getTotalAvalCnt(this, domain.get(i));
76              tacList.add(tac);
77          }
78          // System.out.println(tacList);
79          ArrayList<Wrapper> wList = new ArrayList<~>();
80          for(int i = 0; i < domain.size(); i ++){
81              Wrapper w = new Wrapper(domain.get(i), tacList.get(i));
82              wList.add(w);
83          }
84          Collections.sort(wList);
85          for(int i = 0; i < wList.size(); i ++){
86              rtnList.add(wList.get(i).getA());
87          }
88          Collections.reverse(rtnList);
89          return rtnList;
90      }
```

The "sortDomain()" method first gets total available count for all the item's neighbours if this value is taken for each value in the domain. Then, it sorts the domain according to the total available count descendingly. Then, the list is returned. After doing this, the first value in the list will have be the least constraining one.

To implement forward checking, the following lines are added:

```
204             // Do forward checking by updating the matrices according to current taken item, its bag and constraints.
205             Matrices tmpMatrices = new Matrices(matrices);
206             tmpMatrices.updateMatrices(pair);
207             this.updateMatricesCons(tmpMatrices, pair, items, tmpBags);
```

First, the matrices that manages the domain of variables are updated according to the chosen pair. Then, the constraints are used to maintain arc consistency between the selected variable and all its neighbours.

The “updateMatricesCons()” method is implemented so that it checks all the not null constraints and update matrices according to constraint.

describe which tests you ran to try out your program. How did your program perform?

We run our program using just backtracking, backtracking + MRV + degree heuristic + LCV and backtracking + MRV + degree heuristic + LCV + forward checking on all the given test inputs for five times each and record the average runtime in ms.
The result we got is shown in the following table:

Run Time(ms)	Backtracking	BT+MRV+LCV+degree heuristic	BT+heuristic+FC
Input1	0.6	1	0.8
Input2	1	2	2
Input3	1	1.8	2.2
Input4	2	3	3.4
Input5	4.4	11.4	16.2
Input6	1.4	2.6	3.6
Input7	1.2	2.8	2
Input8	2	6.4	9.2
Input9	1	1.8	1.6
Input 10	1	2.2	2.4
Input 11	1.8	2.8	3
Input 12	1.4	2.6	2.8
Input13	1.2	2.6	2.6
Input 14	1.4	2.4	2.6
Input 15	1.4	2.2	2
Input 16	1.4	2.4	2.2
Input17	1.6	2.4	3
Input 18	1.6	3.4	3.4

Input 19	1.4	2.2	2.4
Input 20	1.8	2.4	2.6
Input 21	1.2	2.2	2.6
Input 22	1.8	5.4	5.4
Input 23	1.8	5.2	6.6
Input 24	3.2	9.8	12.2
Input 25	10.8	85	21.8
Input 26	>5 minutes	2	50.2
Input 30	3081.6	11.4	16.6
Input 31	3855.4	329	142.8
Input 32	1232.4	121.4	254
Input 33	1237.4	890.4	918.2

From the table, we can tell that with heuristics and forward checking, the algorithm runs much faster on complicated test cases. On some simple test cases, forward checking is a little slower than just using heuristics, this may because that in our implementation, forward checking needs to constantly update the matrices to maintain arc consistency. This may cause it to be a little slow on simple cases. On some complex cases at the end, the combination of everything is much faster than only using backtracking, but still a bit slower than not the combination of everything expect forward checking. We speculated that the forward checking will change matrices every time that an item was put into a bag. Checking and Change will takes a lot of time.

describe the strengths and the weaknesses of your program.

The strength of our program is that we use constraint matrices to keep track of the domain of variables. As a result, updating domain, getting domain and maintaining arc consistency becomes easy to code.
The weaknesses is that since we have to constantly accessing the constraint matrices, the run time may be longer. Especially when running forward checking algorithm, each time putting an item into a bag will result to go through other matrices to make change, which cause a long time run.