CS4341

Connor Porell (cgporell)

Andrew Roskuski (ajroskuski)

Project 4 Report

**Running the Program**

This program was implemented in Java 8. This program has 3 classes with main functions; each of these is a different solver. To compile the program, simply . The program comes with a .jar file for each solver, so you can skip this step if you already have the jars.

To run the program, open a command line, and type the following:

java -jar RoskuskiPorellSolver.jar name\_of\_input\_file MRV\_flag LCV\_flag forward\_flag

To activate the MRV, LCV, and Forward flags, simply type "1" in their spots. For example, to run the MRV program, type:

java -jar RoskuskiPorellSolver.jar input.txt 1

This will run the MRV search as part of solving the CSP in input.txt

When the program runs, it will display to the console the result of the CSP once it is solved.

**Description of Approach**

A preliminary note: Forward Checking was always "on" when we collected data with the heuristics separated. When we are stacking the various searches, Forward Checking is handled independently. It is built-in to each of the search functions.

We implemented Backtracking Search, Minimum Values Remaining Search, Least Constraining Value Search, and Forward Checking Search. We implemented Backtracking Search at the start, but we realized that everything followed a very similar structure early on, so we just made a general solver that looks to see what type of search we are trying to do; this is defined by an enum chosen by the main class (which is, in turn, chosen from whatever class is the Main class). This is our implementation of Backtracking Search:

boolean backtrackingSolve(HashMap<String, Item> items)

if(items.size == 0) then check constraints on Bags

copy all items

for all items **i**

for all bags **b**

check consistency of assignment of **i** to **b**

if assignment of **i** to **b** is consistent then put **i** in list of consistent values for **b**

remove **i** from assignment to **b**

for all items **i**

if **i** has no consistent assignments then return **false**

store all items in ArrayList<Item> **itemArray**

for all items **i** in **itemArray**

store all bags in ArrayList<Bag> **bagArray**

for all bags **b** in **bagArray**

check consistency of assignment of **i** to **b**

check constraints on assignment of **i** to **b**

if assignment of **i** to **b** is unconsistent then unassign **i** from **b**

else put all items in HashMap<String, Item> **temp**

remove **i** from **temp**

result = backtrackingSolve(**temp**)

if result is true then *return* **true**

else unassign **i** from **b**

*return* **false**

As stated above, our implementation of MRV search was fairly similar to the implementation of Backtracking Search. The main difference was a sorting of the itemArray variable before the function iterated through it. It is called when the MRV solver class is set to be the Main class. It can be run using the MRVSolver jar. Below is our pseudocode for MRV search:

boolean MRVSolve(HashMap<String, Item> items)

if(items.size == 0) then check constraints on Bags

copy all items

for all items **i**

for all bags **b**

check consistency of assignment of **i** to **b**

if assignment of **i** to **b** is consistent then put **i** in list of consistent values for **b**

remove **i** from assignment to **b**

for all items **i**

if **i** has no consistent assignments then return **false**

store all items in ArrayList<Item> **itemArray**

sort **itemArray** from lowest to highest

for all items **i** in **itemArray**

store all bags in ArrayList<Bag> **bagArray**

for all bags **b** in **bagArray**

check consistency of assignment of **i** to **b**

check constraints on assignment of **i** to **b**

if assignment of **i** to **b** is unconsistent then unassign **i** from **b**

else put all items in HashMap<String, Item> **temp**

remove **i** from **temp**

result = backtrackingSolve(**temp**)

if result is true then *return* **true**

else unassign **i** from **b**

*return* **false**

Similarly, our implementation of LCV Search followed a very similar pattern to the Backtracking Search. The main difference was a series of loops that the program would run after creating **itemArray** and **bagArray**. These loops would run checks to make see which assignments cause the fewest conflicts with future assignments. Our LCV implementation is run by selecting the LeastConstrainingSolver as the Main class at compilation, or by running the LCVSolver jar. Below is our implementation of LCV Search:

boolean LCVSolve(HashMap<String, Item> items)

if(items.size == 0) then check constraints on Bags

copy all items

for all items **i**

for all bags **b**

check consistency of assignment of **i** to **b**

if assignment of **i** to **b** is consistent then put **i** in list of consistent values for **b**

remove **i** from assignment to **b**

for all items **i**

if **i** has no consistent assignments then return **false**

store all items in ArrayList<Item> **itemArray**

for all items **i** in **itemArray**

store all bags in ArrayList<Bag> **bagArray**

for all bags **b** in **bagArray**

assign **i** to **b**

for all items **i2**

if **i2** does not equal **i**

for all bags **b2**

check consistency of the assignment of **i2** to **b2**

check consistency of assignment and constraints

if assignment is consistent then increment **possibleAssignments**

unassign **i2** from **b2**

unassign **i** from **b**

sort **bagArray** in order from highest to lowest

for all bags **b** in **bagArray**

check consistency of assignment of **i** to **b**

check constraints on assignment of **i** to **b**

if assignment of **i** to **b** is unconsistent then unassign **i** from **b**

else put all items in HashMap<String, Item> **temp**

remove **i** from **temp**

result = backtrackingSolve(**temp**)

if result is true then *return* **true**

else unassign **i** from **b**

*return* **false**

Lastly, we created an implementation that runs Backtracking

**Testing the Program**

To test the efficiency of the program, we ran a few time tests. Without a doubt, MRV is the most efficient search by far, as it completed each test file nearly instantly. The regular Backtracking Search clearly outperforms LCV search. While neither perform nearly as quickly as MRV, the disparity between times for Backtracking and LCV is large enough to declare Backtracking the faster of the two.

**Strengths and Weaknesses of the Program**

The program is strong in that it can find solutions to CSPs of this nature within a minute in most cases. When running each solver on each of the test cases, the problem was solved within a few seconds for smaller cases. The solve time never exceeded a minute. In the case of MRV, many CSPs were solved nearly instantaneously.

The program's main weakness is that it uses a lot of memory. This is partly due to the depth-first search nature of the problem, but also is due in part to Java being memory-inefficient.

**Comparison of Average Times**

|  |  |  |  |
| --- | --- | --- | --- |
|  | Backtracking | Backtracking+heuristics | Backtracking+heuristics+Forward Checking |
| Test File 1 | 0.081s | 0.072s | 0.072s |
| Test File 2 | 0.073s | 0.072s | 0.077s |
| Test File 3 | 0.067s | 0.066s | 0.073s |
| Test File 4 | 0.079s | 0.084s | 0.075s |
| Test File 5 | 12.76s | 0.089s | 0.091s |
| Test File 6 | 24.83s | 0.105s | 0.096s |