Brooklyn Grant

Megan Aker

CS 457, Intro to AI

Tim Anderson

October 7th, 2024

Grocery Bagging Project Write Up

Project Overview:

The objective of the project is to design and implement a program that solves a constrained grocery bagging problem. Given a set of groceries, each with a size and specific packing constraints, the program determines how to place items into a limited number of bags. The program ensures that all constraints, such as which items can or cannot be bagged together, are met, while adhering to the maximum capacity of each bag. This write up aims to explain the thought behind our code and test suites used to solve this problem. We both worked equally on this project, both coding and contributing to different parts of the writeup.

The Design:

Within the grocery bagging project, there are a few key files that each play a role in the project functionality. BagIt is the shell script that passes the command line parameters it receives. A Makefile is also included to help build the program and remove .class files. We created our own random generator, GroceryProbGen, that followed the constraints specified and created test files with it. We specify grocery item names as well

as their size, bag size, and number of bags available (following the example file on the Canvas project description). The main driver behind our design is BaggingProblem.java and it handles the bulk of the logic. The program defines two main entities: Item and Bag. Each item has a name, size, and a set of constraints that determine which other items it can or cannot be packed with. Bags have a maximum size, and items must fit within these size limits while respecting the packing constraints.

The constructor of the BaggingProblem class reads input from a file, initializing items and bags. It also sets up a matrix, canPackWith, to track which items can be packed together without violating constraints. These constraints are processed based on the input, and the canPackWith matrix is updated to reflect item compatibility.

The program's packing logic is implemented in the Bag class, where the pack() method attempts to add an item to a bag. Before packing, it checks if the item fits in the bag and if it is compatible with the other items already packed in that bag. If the item fits and does not violate any constraints, it is successfully packed.

The core of the solution is a recursive backtracking algorithm implemented in the search() method. It uses a priority queue to process the items, prioritizing larger items with more constraints. The algorithm tries to pack each item into a bag, and if it cannot, it backtracks and tries a different configuration, exploring all possible packing arrangements.

Finally, if the search is successful, meaning all items have been packed into bags without violating any constraints, the program calls printPacking() to display the arrangement of items in the bags. If it is not, it will output "failure". This code effectively

solves the problem of packing items into bags while satisfying the constraints, using a priority queue and a recursive search algorithm to find a valid packing solution.

Testing:

Test files were created randomly by calling on GroceryProbGen.java to test the efficiency of our bagging solution. The test files differ in bag size, number of bags available, item constraints, and overall complexity. They are named bag0.txt, bag1.txt, bag2.txt etc. In theory, if our solution works properly, the output will print out the word "success", followed by, for each bag, the items that have been calculated that should go into that bag. If no solution is found, it will output "failure". In addition, a test script was created to efficiently test all files at once and pass them through bagit.sh as well as cspsolve.

Test:	Content:	Solution?	PASS/FAIL
bag0	3 7 bread 3 + rolls rolls 2 + bread squash 3 - meat meat 5 lima_beans 1 - meat	yes	PASS
bag1	5 10 meat 5 tomatoes 3 + meat rolls 5 - meat bread 5 + meat	yes	PASS
bag2	5 9 yogurt 4 apples 2 + yogurt tomatoes 3 - apples	yes	PASS
bag3	5 7 tomatoes 3 bread 3 + tomatoes meat 1 - tomatoes	yes	PASS

bag4	4 7 apples 2 lima_beans 1 + apples cheese 1 + apples milk 2 - lima_beans	yes	PASS
bag5	3 10 lima_beans 1 milk 2 - lima_beans squash 3 - milk meat 3 + squash rolls 2 + lima_beans	yes	PASS
bag6	2 8 apples 2 meat 3 - apples lima_beans 2 + meat	yes	PASS
bag7	3 5 tomatoes 2 apples 1 + tomatoes cheese 1 - tomatoes squash 3 - cheese meat 3 + squash milk 3 - meat	no	PASS
bag8	3 5 apples 3 milk 2 + apples lima_beans 1 - apples yogurt 4 + lima_beans	yes	PASS
bag9	2 6 rolls 4 apples 4 + rolls meat 1 + rolls squash 2 - apples	yes	PASS
bag10	2 7 apples 2 tomatoes 5 + apples lima_beans 5 - tomatoes squash 3 - lima_beans	no	PASS
bag11	4 10 apples 5 bread 5 - apples tomatoes 4 - bread meat 5 + bread milk 5 - bread	yes	PASS
bag12	5 9 meat 5 yogurt 1 - meat squash 4 - yogurt	yes	PASS

bag13	2 5 meat 3 apples 5 - meat bread 2 + meat tomatoes 2 - apples squash 5 - bread	no	PASS
bag14	3 9 squash 5 yogurt 1 + squash apples 5 - squash rolls 3 - squash	yes	PASS
bag15	2 10 bread 3 + rolls rolls 2 + bread squash 3 - meat meat 5 lima_beans 1 - meat yogurt 6 - cheese cheese 4 apples 10 - milk milk 1 tomatoes 9 + milk	no	PASS

An example output for test 14 and test 15:

```
---------------------------------------
Test File 14:
success
yogurt  squash
rolls   apples
checkbag output: Good solution!
Test File 14 PASS
---------------------------------------
Test File 15:
failure
Bagging failed for bag15.txt. See if this is supposed to fail with cspsolve.
cspsolve output: failure
Test File 15 PASS
---------------------------------------
```

Results:

When the test script ran, all of our created tests passed. The ones that were successful had a good solution according to the CheckBag.jar file, and the ones that had failed were supposed to fail due to checking with the cspsolve file. Essentially, after

conducting further investigation into this file, cspsolve calculates the correct solution to the test. For instance, we purposely made bag15 difficult to solve as it has 10 total items with a limited amount of bags available and a higher item size. Thus, when the test ran, no solution was found, as to be expected.

Reflection:

We were having trouble with the algorithm, because it seemed to be not including some items, but still being successful. In our original code, the reason some items weren't being packed was because the search wasn't properly backtracking and revisiting items when they couldn't fit in a particular bag. Once an item failed to be packed in any bag, the search gave up on it and moved on, leaving out that item so it would eventually not be included into the final result. This meant it was successful without all of the items. We decided to fix this, we would re-add the item back to the list of items to be packed. This way, the search can keep trying to add it to the bags rather than forgetting certain items. Now, it explores all possible ways to pack the groceries while respecting the constraints, increasing the chance of finding a valid solution where everything fits. We did find this bug before Professor Andersen revealed this bug in class, which would have been better to know before we looked for this bug for an hour!

It also was taking a lot of time to test, it was very tedious to slowly run all the commands needed to: make clean, make, bagit, and then CheckBag or cspsolve to see if our result was correct. So after we were sure a lot of our code was compilable, we created our test.sh script that would run through and make sure all of our generated text files were passing our tests. We found most of our errors using this test. We found that

making a script would make it easier for testing moving forward, and ends up being more efficient in the end!

Conclusion:

In summary, this project demonstrates the practical application of constraint based algorithms to solve the grocery bagging problem. By implementing techniques such as node and arc consistency checking, along with intelligent variable and value ordering, the program efficiently explores possible packing configurations to find a valid solution, if one exists. Through thorough testing with varying input scenarios, the project highlights where it performs well and where it faces challenges when attempting to find solutions with difficult constraints. This project fulfills the primary goal of producing a correct packing solution and explores the behavior of constraint based searches.