Authors: Cody Palin and Tung Ho

Class: CS457 - Introduction to Artificial Intelligence

Overview:

This project develops an algorithm to bag groceries that have constraints on what items can be bagged with what, how much you can put in a bag, and the number of bags available. The program is given the name of a text file that defines the grocery bagging problem as the last command line argument. The first line in the file determines the number of bags available. The next line determines the maximum bag size. The remaining lines define the items that must be bagged. Each line has a distinct name, a number associated with it which defines its size, possibly followed by a + or a – which represents compatibility (aka positive compatibility) or incompatibility (aka negative compatibility) and followed by a list of items. The + or – and a list of items define the constraints on what can be bagged with the item. A + indicates that it is OK to bag the item with anything in the list that follows (i.e. positive compatibility), but nothing else. A – indicates that it is NOT OK to bag the item with anything in the list (i.e. negative compatibility), but you can bag the item with anything else. If no sign is given and no list is provided for the item on a given line, then the item is assumed to be compatible with everything in the problem. However, the above item can still be rejected by some other item(s) in the problem. In either case, the program will determine at least one way to bag all the items and at the same time satisfy the given constraints. Two uninformed search strategies will be implemented: depth first and breadth first. If depth first search is selected, then the program will print only the very first solution found and quit. If breadth first search is selected, then the program will print out all solutions found and quit.

Compiling and Using:

make clean && make

./bagit filename [-depth|-breadth]

Design Option 1 – Depth First Search:

The depth first design is completely different than the breadth first design. The main structure includes Item objects, Bag objects and State Objects. Item objects have a weight, the item’s name, an ID unique to that item and a set of all the items that cannot be put into a bag with that item, the constraints. Bag Objects have their maximum weight, as well as a weight total of all the items that are currently in the bag. The bag also has a set of constraints, this set is the union of all the sets of constraints of the items that are currently in the bag. State objects have a list of all the bags as they were when the bag was added to the stack, they also have an index for the item that is currently being checked.

First the file is read and all the items are initialized with their ID and their name, then the list of items are read through and the constraints are added, positive constraints are flipped around so that only negative constraints exist. Also, constraints are added in reverse, for example: Item 1 cannot be in a bag with item 2. Not only will item 1 get the constraint added for item 2, at the same time item 2 will get the constraint that it cannot be put into a bag with item 1. Once all the constraints are added, the stack begins with the state where no items are added to any bags. Now the depth first search begins. In a loop the following happens:

● The search checks whether it is in a goal state, if it is, the program succeeds.

● The next state on the top of the stack is loaded.

● For all the bags, if the item can be added to a bag, add the state where the item is added to the bag to the stack.

Because we have added the union set of all the items’ constraints to the bag, and the reverse constraints have been added, checking if an item can be added to a bag is much simpler and more efficient. After we check that the bag has enough room for this item, we simply check that the ID of the item we are trying to add does not exist in the list of constraints for the bag. If either of those checks fail, the item cannot be added to the bag.

Design Option 2 – Breadth First Search:

A unique set (i.e. *unique\_sets*) is first declared in order to represent distinct groups of items that are compatible to each other within that group. A hash map (i.e. *hm*) is declared in order to connect an item to its assigned ID. Another hash map (i.e. *hm\_weight*) is declared in order to connect the same item to its weight. Another hash map (i.e. *hm\_reverse*) is declared in order to retrieve an item’s name from its ID. There is one hash map for positive compatibility (i.e. *hm\_plus*), and another hash map for negative compatibility (i.e. *hm\_minus*).

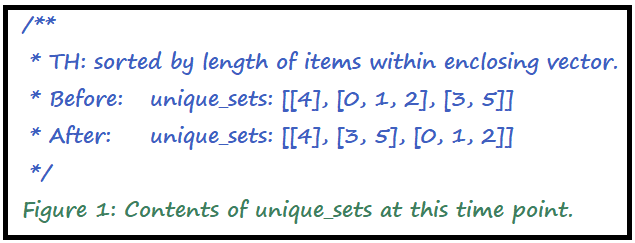
The program processes each line in the input file in order to capture the number of bags available (line 1), the maximum allowable weight per bag (line 2), and individual items along with their constraints on the remaining lines. If any item is NOT accompanied by either constraint, then that item is assumed to be compatible with everything in the problem. If an item is accompanied by a positive list, then a hash map (i.e. *pos\_rc\_pos\_list*) is generated in order to connect each ID to a list of IDs (for positive compatibility). Another hash map (i.e. *neg\_rc\_pos\_list*) is generated for negative compatibility. A vector (i.e. *pos\_log\_pos\_list\_vect*) is generated for positive logic from positive list, and another vector (i.e. *neg\_log\_pos\_list\_vect*) for negative logic from positive list. Similarly, if an item is accompanied by a negative list, then a hash map (i.e. *pos\_rc\_neg\_list*) is generated in order to connect each ID to a list of IDs (for positive compatibility). Another hash map (i.e. *neg\_rc\_neg\_list*) is generated for negative compatibility. A vector (i.e. *neg\_log\_neg\_list\_vect*) is generated for negative logic from negative list, and another vector (i.e. *pos\_log\_neg\_list\_vect*) for positive logic from negative list. The positive logic from negative list (i.e. *pos\_log\_neg\_list\_vect*) is then filtered out by negative logic from positive list (i.e. *neg\_log\_pos\_list\_vect*). An aggregate of negative logics (i.e. *neg\_log\_vect*) is created from negative logics of both lists. The positive logic from negative list (i.e. *pos\_log\_neg\_list\_vect*) is then filtered out by the aggregate of negative logics just created earlier. A vector for positive logic from negative list is also generated by stitching pieces together, and has its elements sorted and any duplicates removed. Both positive logic vectors then rearrange their elements by length in an increasing order and have all empty elements removed. Next, loners are recovered from the aggregate of negative logics (i.e. *neg\_log\_vect*) and added to a vector of *incompatibility* and to the unique set (i.e. distinct group). Next, loners are removed from positive logic from positive list (i.e. *pos\_log\_pos\_list\_vect*). Both positive logic vectors are now added to the unique set, which also sorts its elements and has any duplicates removed. New combinations for the unique set are generated by using the aggregate negative logic in order to filter out incompatible elements within each distinct group and to form new groups where the remaining elements are still compatible to each other within their groups. The newly formed sub-groups within the unique set are then sorted by length in an increasing order (Figure 1). At this point, it is critical to do some preliminary checking to make sure that no items are missing from the unique collection. A vector (i.e. *vect\_vect\_vect\_all*) is created in order to hold all possible combinations of items that are compatible to each other in each of the bags in the final solutions. A hash map (i.e. *hm\_vect\_vect\_vect\_all\_len*) is also created in order to connect each group ID to its length.

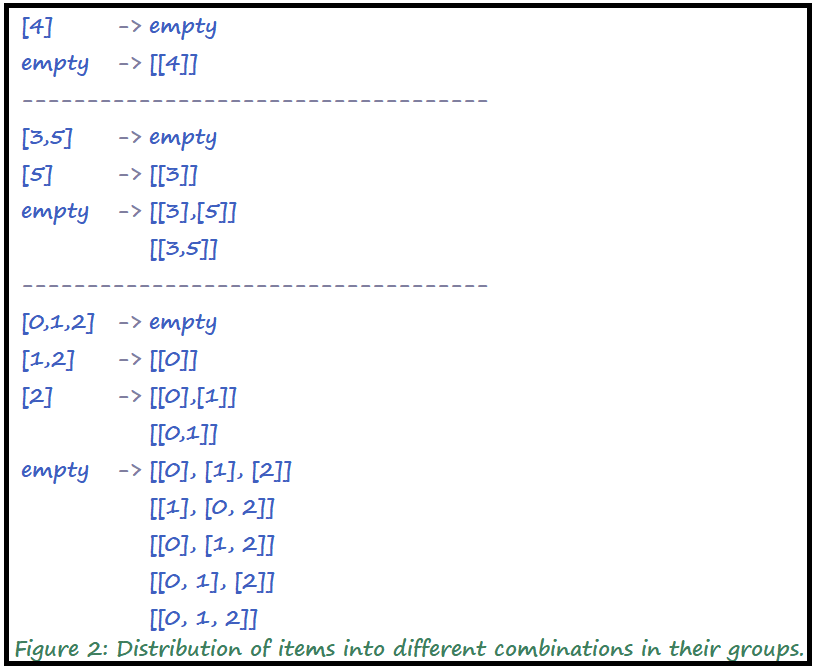
The breadth-first search starts out by pulling out all combinations within each group in the ‘*vect\_vect\_vect\_all*’ vector (Figure 2), starting at the lowest index. If the very first group is pulled out, then its elements will be put back to the ‘*vect\_vect\_vect\_all*’ vector at the highest index. The number of combinations that are put back to this vector are recorded as ‘*group\_length\_previous*’. These combinations will serve as a template so that more combinations can be formed when other groups get pulled out of the ‘*vect\_vect\_vect\_all*’ vector and combine themselves with the existing combinations. Overlaps will be resolved as new combinations are made along the way by eliminating redundancy in either existing groups or in newly arrived groups (Figure 3). At this point, breadth-first search candidates will be formed as a template before numerous permutations of bagging solutions can be generated, without any duplicates in symmetry. This is achieved by shuffling the items in each candidate several times in order to absorb elements together (Figure 4). For example, two items in two different bags can be combined into one bag if they are compatible to each other. The numerous permutations of bagging all the items are then collected in the ‘*vect\_vect\_vect\_all*’ vector, which also sorts its elements and has any duplicates removed. The next step is to iterate over potential bagging solutions and print out all valid solutions to the console in the correct format. At this point, the breadth-first search is complete.

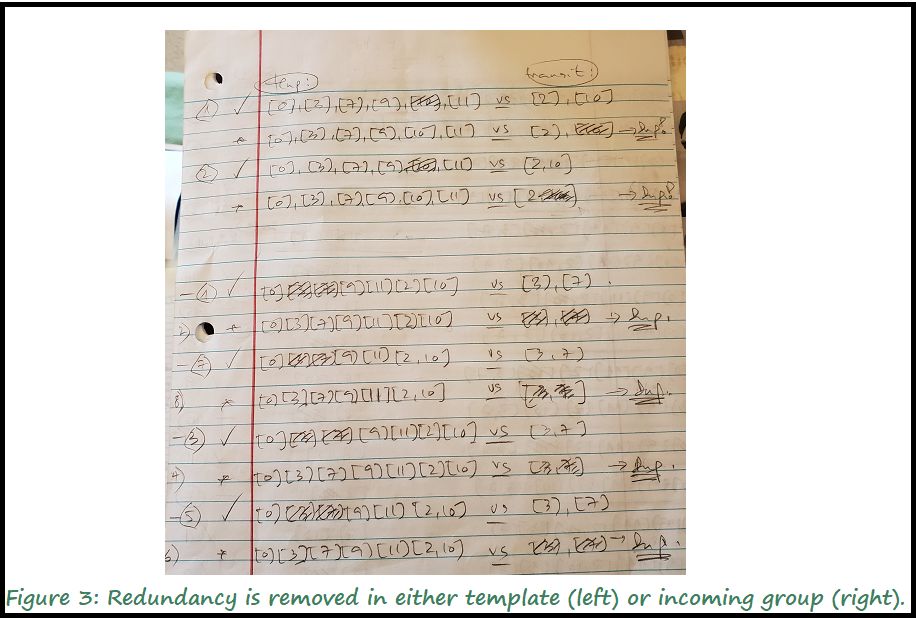
Testing and Results:

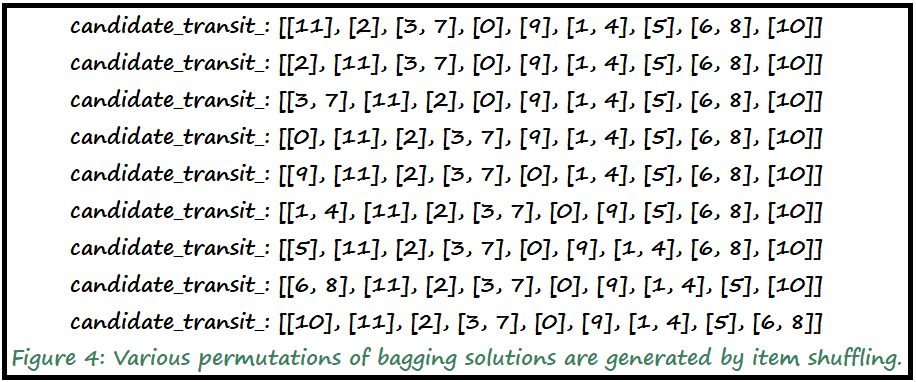
* The solution from depth-first search was matched against one of the solutions from the breadth-first search.
* Different constraint combinations were used: all positive constraints, all negative constraints, mixed constraints, items without any constraints, few bags than possible constraints, smaller bag size constraints.
* Depth-first solutions were found for problems with many items (up to 1000 items).

Figures:









References:

* We consulted the official JAVA API from Oracle for several data structures in this project.