Nicholas Gin
300107597

CSI 2120 Comprehensive Assignment – Part 3: Prolog Document

Contents

Basic Overview of the Implementation 2
Predicate Descriptions 3
References 9

Nicholas Gin
300107597

Basic Overview of the Implementation

This implementation of the Knapsack Problem essentially creates two Dynamic Programming tables, akin to the one included in the assignment outline, based on the information from the input file provided by the user. One of these tables contains the total maximum values of each knapsack that can be filled for a certain capacity and for a certain number of items. The other table contains the combination of items (as a list) that maximizes the total value of each knapsack for a certain capacity and for a certain number of items.

For example, “p1.txt” shall produce the following two tables.
Values Table:

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
No item	0	0	0	0	0	0	0	0
A	0	1	1	1	1	1	1	1
A, B	0	1	6	7	7	7	7	7
A, B, C	0	1	6	10	11	16	17	17
A, B, C, D	0	1	6	10	11	16	17	21

Names Table:

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
No item	[]	[]	[]	[]	[]	[]	[]	[]
A	[]	[‘A']	[‘A']	[‘A']	[‘A']	[‘A']	[‘A']	[‘A']
A, B	[]	[‘A’]	[‘B’]	[‘A’, ‘B’]	[‘A’, ‘B’]	[‘A’, ‘B’]	[‘A’, ‘B’]	[‘A’, ‘B’]
A, B, C	[]	[‘A']	[‘B’]	[‘C’]	[‘A’, ‘C’]	[‘B, ‘C’]	[‘A’, ‘B', ‘C’]	[‘A’, ‘B’, ‘C’]
A, B, C, D	[]	[‘A’]	[‘B’]	[‘C’]	[‘A’, ‘C’]	[‘B, ‘C’]	[‘A’, ‘B', ‘C’]	[‘B’,’D’]

The code will receive the first rows of each table as input and then simulate the iteration along this row to create the second row of the table. Through a recursive process, the code will receive the second row of the table as input and then simulate the iteration along this row to create the third row of the table and so on and so forth until all items have been considered and all rows in the table have been formed. The last values of each of the tables (the values in green) will be outputted to the user as the maximum total value of the items that can be stored in the knapsack and the combination of items that maximizes the total value of the knapsack respectively.

For a more detailed breakdown of how the code works, please refer to the comments of *solveKnapsack.pl*, as well as the descriptions of each predicate from *solveKnapsack.pl*, which begin on the following page.

Nicholas Gin
300107597

Predicate Descriptions

Further details describing how each predicate works can be found in the comments of *solveKnapsack.pl*.

solveKnapsack(Filename, Value, L_items_list)
This is the predicate that is called by the user; it calls all other predicates to solve the knapsack problem and to produce a solution file. It takes a Filename as input (the name of the file that will be read for the information about the items being considered in the knapsack problem) and outputs the total maximum value of the items that can be stored in the knapsack (Value) and the list of the names of the items that are stored in the knapsack which contains the combination of items that maximizes the total value of the knapsack (L_items_list).

readFile(Filename, N_numberOfItems, L_items_name, L_items_value, L_items_weight, N_capacity)
This is the predicate that reads the contents of the file with the inputted Filename. From the file, it extracts the number of items being considered (N_numberOfItems), the names of each item being considered (L_items_name), the values of each item being considered (L_items_value), the weights of each item being considered (L_items_weight), and the total capacity/weight that the knapsack can hold (N_capacity).

readLine(Input, NumberOfItems, [L_line|L_file])
This is the predicate we will use to read every line of the file that represents an item. Input indicates the Stream we are reading from. NumberOfItems indicates the number of lines this predicate will read from the file (the number of items we will extract information for). This predicate will output a list containing sublists that represent the information about each item in the file (L_file). Each sublist (L_line) will be added to L_file via recursive calls to this predicate.

readLine(_, _, []).
The base case of the readLine predicate, indicating when to stop the recursive process of filling the L_file list.

getInfoFromList([[Name, Value, Weight|_]|L_file], [C_Name|L_items_name], [N_Value|L_items_value], [N_Weight|L_items_weight])
This is the predicate that we will use to separate the information in each sublist of the list containing all of the information about the items (L_file) into separate lists for the names of each item (L_items_name), the values of each item (L_items_value) and the weights of each item (L_items_weight). Consider the first sublist in L_file (we take the head sublist of this list). Take the first element of the sublist (Name), convert it into a char (C_Name) and add it to L_items_name. Take the second element of the sublist (Value), convert it into a number (N_value) and add it to L_items_value. Take the third element of the sublist (Weight), convert it into a number (N_Weight) and add it to L_items_weight. Then repeat this process for every other sublist in L_file (by recursively calling this predicate using the tail we found earlier), until we reach the base case (there are no more sublists to consider).

getInfoFromList([], [], [], []).
The base case of the getInfoFromList predicate, indicating when to stop the recursive process of filling the lists.

writeFile(X, Filename)
This is the predicate we will use to write to our solution file. X refers to the information that we will write to the solution file and Filename refers to the name of the solution file.

knapsack(L_item_value, L_itemWeight, L_itemName, PreviousRow, PreviousNameRow, CurrentItemIndex, CurrentCapacity, MaxCapacity, Count, Z, A)
knapsack is used to recursively call solveRow to create each row in the Dynamic Programming tables. As parameters, knapsack takes the list of item values being considered (L_item_value), the list of item weights being considered (L_itemWeight), the list of item names being considered (L_itemName), the row of the dynamic programming table for item values that we will use to generate the next row in that table (PreviousRow), the row of the dynamic programming table for item names that we will use to generate the next row in that table (PreviousNameRow), the current item being considered represented by its ordering in the file (CurrentItemIndex), the current capacity of the knapsack being considered (CurrentCapacity), the maximum capacity that the knapsack will hold (MaxCapacity), and the number of times the knapsack predicate still needs to be recursively called/the number of rows in the tables that still need to be generated (Count). As outputs, we receive the last row of the table representing the item values (Z) and the last row of the table representing item names (A). So, if Count is greater than 0, generate the current rows of each of the tables (X and Y) based on the rows provided (PreviousRow and PreviousNameRow) by calling solveRow and then use those generated rows (X and Y) to generate the next rows of the table (Z and A) until we reach the base case; decrement Count by 1. In other words, call knapsack(L_item_value, L_itemWeight, L_itemName, X, Y, CurrentItemIndex + 1, CurrentCapacity, MaxCapacity, Count-1, Z, A).

knapsack(_L_item_value, _L_itemWeight, _L_itemName, PreviousRow, PreviousNameRow, _CurrentItemIndex, _CurrentCapacity, _MaxCapacity, Count, PreviousRow, PreviousNameRow)
The base case of the knapsack predicate, indicating when to stop the recursive calling of this predicate; the outputs are the inputs for the parameterized PreviousRow and the PreviousNameRow. This case occurs when Count is 0.

The solveRow predicates are the predicates that are used to generate a row of the tables.
As parameters, solveRow takes the list of item values being considered (L_item_value), the list of item weights being considered (L_itemWeight), the list of item names being considered (L_itemName), the row of the dynamic programming table for item values that we will use to generate the next row in that table (PreviousRow), the row of the dynamic programming table for item names that we will use to generate the next row in that table (PreviousNameRow), the

current item being considered represented by its ordering in the file (CurrentItemIndex), the current capacity of the knapsack being considered (CurrentCapacity), and the maximum capacity that the knapsack will hold (MaxCapacity). Each predicate will output the row of the value table that we are generating (NewRow) and the row of the item name table that we are generating (NewNameRow). There are six variants of the solveRow predicate, plus a base case.

solveRow(L_item_value, L_itemWeight, L_itemName, PreviousRow, PreviousNameRow, CurrentItemIndex, CurrentCapacity, MaxCapacity, [0|NewRow], [[]|NewNameRow])
This is Case 1. When the CurrentCapacity being considered is 0, add 0 to NewRow (as the total value when 0 items are placed in the knapsack is always 0). When the CurrentCapacity being considered is 0, add [] to NewNameRow (since 0 items are placed in the knapsack, there will be no item names). Then, recursively call solveRow by incrementing CurrentCapacity by 1, so we can consider the next capacity that the knapsack can hold (the next column of this row of the table); in other words, call solveRow(L_item_value, L_itemWeight, L_itemName, PreviousRow, PreviousNameRow, CurrentItemIndex, CurrentCapacity + 1, MaxCapacity, NewRow, NewNameRow).

solveRow(L_item_value, L_itemWeight, L_itemName, PreviousRow, PreviousNameRow, CurrentItemIndex, CurrentCapacity, MaxCapacity, [CurrentItemValue|NewRow], [CurrentItemName|NewNameRow])
This is Case 2. The CurrentCapacity being considered is less than or equal to the MaxCapacity that the knapsack can hold, and the weight of the current item that is being considered is equal to the CurrentCapacity being considered. We check if the singular value of the current item being considered is greater than the value of the combination of items that maximized the total value of the knapsack for the same capacity, but for one less item considered (the same column in the previous row of the values table/the column in the previous row of the values table with the same CurrentCapacity). If this is the case, add the value of the current item being considered to NewRow and add a sublist containing the name of this item to NewNameRow. The current item alone maximizes the total value of the knapsack better than the previous combination of items for this capacity. Then, recursively call solveRow by incrementing CurrentCapacity by 1, so we can consider the next capacity that the knapsack can hold (the next column of this row of the table); in other words, call solveRow(L_item_value, L_itemWeight, L_itemName, PreviousRow, PreviousNameRow, CurrentItemIndex, CurrentCapacity + 1, MaxCapacity, NewRow, NewNameRow).

solveRow(L_item_value, L_itemWeight, L_itemName, PreviousRow, PreviousNameRow, CurrentItemIndex, CurrentCapacity, MaxCapacity, [PreviousValue|NewRow], [PreviousItemName|NewNameRow])
This is Case 3. The CurrentCapacity being considered is less than or equal to the MaxCapacity that the knapsack can hold, and the weight of the current item that is being considered is equal to the CurrentCapacity being considered. We check if the singular value of the current item being considered is less than the value of the combination of items that maximized the total value of the knapsack for the same capacity, but for one less item considered (the same column in the

previous row of the values table/the column in the previous row of the values table with the same CurrentCapacity). If this is the case, add the value from the previous row to NewRow and add a sublist containing the names of the items from the previous row to NewNameRow. The previous combination of items for this capacity maximizes the total value of the knapsack better than the current item alone. Then, recursively call solveRow by incrementing CurrentCapacity by 1, so we can consider the next capacity that the knapsack can hold (the next column of this row of the table); in other words, call solveRow(L_item_value, L_itemWeight, L_itemName, PreviousRow, PreviousNameRow, CurrentItemIndex, CurrentCapacity + 1, MaxCapacity, NewRow, NewNameRow).

solveRow(L_item_value, L_itemWeight, L_itemName, PreviousRow, PreviousNameRow, CurrentItemIndex, CurrentCapacity, MaxCapacity, [PreviousValue|NewRow], [PreviousItemName|NewNameRow])

This is Case 4. The CurrentCapacity being considered is less than or equal to the MaxCapacity that the knapsack can hold, and the weight of the current item that is being considered is greater than the CurrentCapacity being considered. This means we can not add this item to the knapsack for this current capacity. Consider the combination of items that maximized the total value of the knapsack for the same capacity, but for one less item considered (the same column in the previous row of the values table/the column in the previous row of the values table with the same CurrentCapacity). Add the value from the previous row to NewRow and add a sublist containing the names of the items from the previous row to NewNameRow. Then, recursively call solveRow by incrementing CurrentCapacity by 1, so we can consider the next capacity that the knapsack can hold (the next column of this row of the table); in other words, call solveRow(L_item_value, L_itemWeight, L_itemName, PreviousRow, PreviousNameRow, CurrentItemIndex, CurrentCapacity + 1, MaxCapacity, NewRow, NewNameRow).

solveRow(L_item_value, L_itemWeight, L_itemName, PreviousRow, PreviousNameRow, CurrentItemIndex, CurrentCapacity, MaxCapacity, [NewValue|NewRow], [NewItemName|NewNameRow])

This is Case 5. The CurrentCapacity being considered is less than or equal to the MaxCapacity that the knapsack can hold, and the weight of the current item that is being considered is less than the CurrentCapacity being considered. We subtract the weight of the current item being considered from CurrentCapacity of the knapsack; call this difference RemainingCapacity. Consider the combination of items that maximized the total value of the knapsack for the RemainingCapacity, but for one less item considered. Sum this value with the value of the current item. If their summed value is greater than the value of the combination of items that maximized the total value of the knapsack for the same capacity, but for one less item considered (the same column in the previous row of the values table/the column in the previous row of the values table with the same CurrentCapacity), then add their sum to NewRow. Add the name of the current item to the list of item names for that combination of items from the previous row; add this sublist to NewNameRow. Then, recursively call solveRow by incrementing CurrentCapacity by 1, so we can consider the next capacity that the knapsack can hold (the next column of this row of the table); in other words, call solveRow(L_item_value, L_itemWeight,

L_itemName, PreviousRow, PreviousNameRow, CurrentItemIndex, CurrentCapacity + 1, MaxCapacity, NewRow, NewNameRow).

solveRow(L_item_value, L_itemWeight, L_itemName, PreviousRow, PreviousNameRow, CurrentItemIndex, CurrentCapacity, MaxCapacity, [PreviousValue|NewRow], [PreviousItemName|NewNameRow])
This is Case 6. The CurrentCapacity being considered is less than or equal to the MaxCapacity that the knapsack can hold, and the weight of the current item that is being considered is less than the CurrentCapacity being considered. We subtract the weight of the current item being considered from CurrentCapacity of the knapsack; call this difference RemainingCapacity. Consider the combination of items that maximized the total value of the knapsack for the RemainingCapacity, but for one less item considered. Sum this value with the value of the current item. If their summed value is less than the value of the combination of items that maximized the total value of the knapsack for the same capacity, but for one less item considered (the same column in the previous row of the values table/the column in the previous row of the values table with the same CurrentCapacity), add the value from the previous row to NewRow and add a sublist containing the names of the items from the previous row to NewNameRow. The previous combination of items for this capacity maximizes the total value of the knapsack better than the new combination we just found. Then, recursively call solveRow by incrementing CurrentCapacity by 1, so we can consider the next capacity that the knapsack can hold (the next column of this row of the table); in other words, call solveRow(L_item_value, L_itemWeight, L_itemName, PreviousRow, PreviousNameRow, CurrentItemIndex, CurrentCapacity + 1, MaxCapacity, NewRow, NewNameRow).

solveRow(_, _, _, _, _, _, CurrentCapacity, MaxCapacity, [], [])
The base case of the solveRow predicate. We have finished filling out a row of the tables, when the CurrentCapacity being considered is greater than the MaxCapacity that the knapsack can hold. Stop the recursive calling of solveRow.

getAtIndex([H], 0, H) :- !.
The first base case of getAtIndex; we have reached the index of the list that we want to retrieve an element from, so output the head of the list (H) as the element we want to retrieve.

getAtIndex([H|_], 0, H) :- !.
The second base case of getAtIndex; the index of the list that we want to retrieve an element from is 0. This means we want to retrieve the head of the list, so output the had of the list (H) as the element we want to retrieve.

getAtIndex([_|T], Index, Value)
A predicate that retrieves the element of a list (Value) at a given index (Index). Index represents the index of the list that we want to retrieve an element from, as well as the number of indexes (NextIndex) we have to iterate to the right until we reach the index of the element we want to

retrieve. We know we have reached the element we want to retrieve when the value of NextIndex is 0 (we have reached the first base case).

fillZero(Count, [0|LstOfZeroes])
A predicate that fills a list, LstOfZeros, with Count zeros. If Count is greater than 0, then add a 0 to the LstOfZeros. Then recursively call fillZero, but decrement Count by 1. We will eventually call this predicate Count times, reach the base case, and fill the LstOfZeros with Count zeros.

fillZero(Count, [])
The base case of fillZero; when Count is equal to 0, stop the recursive process of adding a 0 to LstOfZeros.

fillEmptyLists(Length, LstOfEmpties)
A predicate that fills a list (LstOfEmpties) with Count empty sublists. Use length to create a list, LstOfEmpties, of the inputted length, Length. Use maplist to fill each element in the LstOfEmpties with an empty list.

References

8_Prolog4_input_output. (n.d.). Brightspace. Retrieved April 6, 2021, from https://uottawa.brightspace.com/d2l/le/content/213483/viewContent/3528864/View.