Mobility and Smart Cities

Guillaume MARTIN - Master Internet Of Things - 2023/2024

Tutored Work Report

This report presents all the contributions to the TD exercises and explain the design, tests and evaluation of the solutions.

During the course, we shared code and ideas between students, but the final solutions are personal. I mostly have exchanged with the following students on the Make Change exercise:

- Killian Maxel
- Antonin Winterstein

Therefore, you may found some similarities between our solutions.

The files are organized as follow, each folder contains the work to do for a TD. Another folder, called `Algo`, is storing the explanation on the different algorithms presented in class. You will find a jupyter notebook and its conversion to MarkDone, pdf and python code. I recommend to read the notebook or the PDF.

TD1 & TD2 - Make Change

The Make Change exercise has its own report inside the jupyter notebook.

Examples used to compare the algorithms

To compare each algorithm, I have used different set of coins and different amount to change :

Data 1 - 12.35:

- Coins : [5,2,1,0.5,0.2,0.1] (ordered or not)
- amount : 12.35 (or 12.36 to have an example where the problem is not solvable)

This example is shown in the slides. Playing with the order of the coins we can see how it affects some algorithms. For example, the greedy algorithm can give a valid solution, but it won't be the optimal one. For the solution to be optimal (meaning that we want to give as little change as possible), we need to order the coins from the biggest to the smallest, we do not take into account the number of coin available and the set of coin must allow to reach the target amount.

Data 2 - 8 :

- Coins : [5,2,1.5] (ordered or not)
- amount : 8

This data is used to see how the algorithm could behave. In this case, the greedy algorithm will give an incomplete solution even if the problem is solvable. It get stuck to a local optimum.

Therefore, I have used those data to compare the different algorithms. Unfortunately, the iterative and recursive algorithms that generates all the solution are sometimes to slow to be used with the first example.

Evaluation of the algorithms

To evaluate the algorithm I have used the following metrics :

- Number of coins used
- Validity of the solution
- Time to compute the solution

Design

Greedy

The greedy approach consists in selecting the largest available coin at each step to minimize the number of coins used. However, it may not always find the optimal solution in terms of the minimum number of coins, especially if the coin are not sorted in descending order of value.

It's important to note that this algorithm does not guarantee the globally optimal solution for all cases, but it provides a quick and simple way to make change.

Input Parameters

- `amount`: The target amount for which you want to make change.
- `coins`: A list of coin denominations (unsorted) that can be used to make change.

Algorithm Steps

1. Initialize Variables:
    - `i`: Initialize a variable `i` to 0. This variable represents the index of the current coin denomination being considered.
    - `change`: Initialize an empty list called `change` to store the coins used to make change.
2. Main Loop:
    - The algorithm enters a while loop that continues as long as the `amount` is greater than 0 and there are more coin denominations in the `coins` list to consider (`i` is within the bounds of the list).
3. Coin Counting:
    - For each coin denomination in the `coins` list, the algorithm calculates how many times that coin can be used to make change for the remaining `amount`. This is done by dividing the `amount` by the coin's value using integer division (`//`).

4. Update Change:

    - For each calculated count of a coin, the algorithm appends that coin's denomination to the `change` list. This represents adding that coin to the change being given.

5. Update Remaining Amount:

    - After adding the appropriate number of coins to the `change` list, the algorithm calculates the remaining amount by taking the modulo (`%`) of the `amount` with the coin's value. This ensures that the remaining amount is updated correctly.

6. Increment Coin Index:

    - The algorithm increments the index `i` to move on to the next coin denomination in the `coins` list.

7. Return Change:

    - Once the loop finishes, the function returns the `change` list, which contains the minimum number of coins needed to make change for the given `amount` using the unsorted coin denominations provided.

Iterative

This solution iteratively generates all the possible combinations of coins that sum up to the target amount.

Input Parameters

- `coins`: A list of coin denominations (e.g., [1, 2, 5]) that can be used to make change.
- `amount`: The target amount for which you want to calculate all combinations of coins.

Algorithm Steps

1. Convert Amount to Cents

    - The algorithm starts by converting the target `amount` to cents by multiplying it by 100 and then converting it to an integer. This ensures that all calculations are performed in cents to avoid precision issues with floating-point numbers.

2. Create a List of Coin Values in Cents

    - The coin denominations in the `coins` list are also converted to cents and stored in a new list called `coin_values_cents`. This list contains the values of the coins in cents, making it easier to work with integers.

3. Initialize Data Structures

   - The algorithm uses a list called `combinations` to store all valid combinations of coins that sum up to the target amount.
   - It also initializes a stack-based list called `list`. Each element in this list represents the current state of exploration, consisting of three components:
     - `current_amount`: The current amount in cents being considered.
     - `current_combination`: The current combination of coins being used.
     - `current_coin_i`: The index of the current coin being considered.

4. Main Loop

   - The algorithm enters a while loop that continues as long as the `list` is not empty, meaning there are states to explore.

5. Explore States

   - Inside the loop, the algorithm pops the last state from the `list`. This state represents a particular combination of coins being explored.

6. Check for Valid Combination

   - The algorithm checks if the `current_amount` is equal to the `amount_cents`. If they are equal, it means that the current combination of coins sums up to the target amount. In this case, the combination is added to the `combinations` list.

7. Explore Possibilities

   - If the `current_amount` is less than the `amount_cents` and there are more coins to explore (`current_coin_i` is less than the length of `coin_values_cents`), the algorithm proceeds to explore different possibilities.
   - It calculates the maximum count of the current coin that can be used without exceeding the target amount.
   - It then iterates through different counts of the current coin (from 0 to the maximum count) and calculates the new amount and new combination for each possibility.
   - These new states are pushed onto the `list` for further exploration.

8. Repeat and Return

   - The algorithm continues this process until all possible combinations have been explored, and the `combinations` list contains all valid combinations.
   - Finally, the function returns the `combinations` list, which contains all combinations of coins that add up to the target amount.

Recursive

Input Parameters

- `amount`: The target amount to be reached in euros and cents (e.g., 12.50 euros).

- `available_coins`: A list of available coin denominations, represented as floats (e.g., [0.01, 0.05, 0.10]).
- `max_coins`: A list representing the maximum number of each coin type allowed in the combinations (corresponding to `available_coins`).

Algorithm Steps

1. Convert Amount to Cents:
    - The `amount` is converted to cents (an integer) by multiplying it by 100. This ensures that all calculations are performed in cents to avoid precision issues with floating-point numbers.
2. Initialize Data Structures:
    - Initialize an empty list called `all_combinations` to store all generated combinations.
3. Create a List of Coins (in Cents):
    - A list called `coins_list` is created by iterating through the `available_coins` and repeating each coin type based on the maximum allowed count. The coin values are converted to cents.
4. Generate Combinations:
    - A loop iterates from 1 to the `amount_cents` (inclusive) to generate combinations of different lengths.
    - For each length `r`, the `generate_combinations` function is called to generate combinations of that length. If valid combinations are found, they are converted back to euros and cents and returned.
5. Return Combinations:
    - The function returns the valid combinations of coins that sum up to the target amount in euros and cents.

Helper Function: `generate_combinations`

The `generate_combinations` function is a recursive helper function that generates all combinations of a given length `r` from a list of coin values to reach a target amount. It takes the following parameters:

- `input_list`: The list of available coin values.
- `r`: The desired length of combinations.
- `target_amount`: The target amount to be reached.
- `current_combination`: A list representing the current combination being constructed.

TD4 Complexity evaluation of Make Change

To evaluate the time complexity of the algorithm, I've used different set of coins and different amount to change.

Changing the amount of change to give does not really impact the time complexity. However adding small coins to the list of coin available will increase the time complexity of some algorithms as it will have more possibilities to check.

The results are available in the TD4 folder, with graph images and json data.

TD3B - Matrix

In this tutored work we had to implement some tests and operation on matrices. The code is available in the TD3B folder.

Validation

To validate the function I have used the different matrices given in the slide to compare the results.

Design

`reflexive(R)`

- **Description:** The `reflexive` function checks if a given matrix `R` is reflexive. A matrix is reflexive if all of its diagonal elements are equal to 1.
- **Input:**
    - `R`: A square matrix (2D list) to be checked for reflexivity.
- **Output:**
    - Returns `True` if the matrix is reflexive; otherwise, returns `False`.

`symmetrical(R)`

- **Description:** The `symmetrical` function checks if a given matrix `R` is symmetric. A matrix is symmetric if it is equal to its transpose.
- **Input:**
    - `R`: A square matrix (2D list) to be checked for symmetry.
- **Output:**
    - Returns `True` if the matrix is symmetric; otherwise, returns `False`.

`anti_symmetrical(R)`

- **Description:** The `anti_symmetrical` function checks if a given matrix `R` is anti-symmetric. A matrix is anti-symmetric if for all pairs of distinct elements `R[i][j]` and `R[j][i]`, one of them is zero.
- **Input:**
    - `R`: A square matrix (2D list) to be checked for anti-symmetry.
- **Output:**
    - Returns `True` if the matrix is anti-symmetric; otherwise, returns `False`.

`asymmetrical(R)`

- **Description:** The `asymmetrical` function checks if a given matrix `R` is asymmetrical. A matrix is asymmetrical if it is neither symmetric nor anti-symmetric, which means there exist elements `R[i][j]` and `R[j][i]` that are both greater than zero.
- **Input:**
  - `R`: A square matrix (2D list) to be checked for asymmetry.
- **Output:**
  - Returns `True` if the matrix is asymmetrical; otherwise, returns `False`.

`irreflexive(R)`

- **Description:** The `irreflexive` function checks if a given matrix `R` is irreflexive. A matrix is irreflexive if all of its diagonal elements are equal to zero.
- **Input:**
  - `R`: A square matrix (2D list) to be checked for irreflexivity.
- **Output:**
  - Returns `True` if the matrix is irreflexive; otherwise, returns `False`.

`transitive(matrix)`

- **Description:** The `transitive` function checks if a given matrix `matrix` represents a transitive relation. A matrix represents a transitive relation if for all `i`, `j`, and `k`, if `matrix[i][j] > 0` and `matrix[j][k] > 0`, then `matrix[i][k] > 0`.
- **Input:**
  - `matrix`: A square matrix (2D list) to be checked for transitivity.
- **Output:**
  - Returns `True` if the matrix represents a transitive relation; otherwise, returns `False`.

`intransitive(matrix)`

- **Description:** The `intransitive` function checks if a given matrix `matrix` represents an intransitive relation. A matrix represents an intransitive relation if for all `i`, `j`, and `k`, if `matrix[i][j] > 0` and `matrix[j][k] > 0`, then `matrix[i][k]` is not equal to 1.
- **Input:**
  - `matrix`: A square matrix (2D list) to be checked for intransitivity.
- **Output:**
  - Returns `True` if the matrix represents an intransitive relation; otherwise, returns `False`.

`non_transitive(matrix)`

- **Description:** The `non_transitive` function checks if a given matrix `matrix` represents a non-transitive relation. A matrix represents a non-transitive relation if there exist `i`, `j`, and `k` such that `matrix[i][j] > 0`, `matrix[j][k] > 0`, and `matrix[i][k]` is equal to 0.
- **Input:**

    - `matrix`: A square matrix (2D list) to be checked for non-transitivity.
- **Output:**
    - Returns `True` if the matrix represents a non-transitive relation; otherwise, returns `False`.

`equivalence_relation(matrix)`

This function is used to get the equivalence relation of a matrix.

Input Parameters

- `R`: A square relation matrix (2D list) representing a relation.

Algorithm Steps

1. Initialize Variables:
    - `n`: The size of the matrix (number of rows or columns).
    - `classes`: An empty list to store the equivalence classes found.
    - `scores`: An empty list to store the scores for each row in the matrix, indicating the number of related elements in each row.
2. Calculate Row Scores:
    - Iterate through each row of the matrix (indexed by `x`), and for each row, count the number of related elements (elements with values greater than 0) in that row. Store these counts in the `scores` list.
3. Initialize Variables for Iteration:
    - `lastScore`: Initialize a variable to store the score of the previous row (initialized to 0).
4. Find Equivalence Classes:
    - Perform the following steps for each row in the matrix:
        - Find the row with the highest score (`max_score`) and its corresponding index (`max_index`) in the `scores` list.
        - If `max_score` is equal to `lastScore`, it means that the current row belongs to the same equivalence class as the previous row. In this case, append the row index (`max_index`) to the last equivalence class in the `classes` list.
        - If `max_score` is different from `lastScore`, it means that a new equivalence class is starting. Create a new equivalence class by appending a list containing the row index (`max_index`) to the `classes` list.
        - Update `lastScore` to `max_score` to keep track of the previous row's score.
        - Set the score of the row at index `max_index` in the `scores` list to 0 to prevent it from being considered in future iterations.

5. Return Equivalence Classes and Scores:

    - The function returns two lists:
        - `classes`: A list of equivalence classes, where each class is represented as a list of row indices.
        - `scores`: A list of scores, where each score corresponds to the number of related elements in the corresponding row of the matrix.

Transitive Closure

The `transitive_closure` realizes the transitive closure of a given matrix. The transitive closure of a relation represents all transitive relationships derived from the original relation.

Input Parameters

- `matrix`: A square relation matrix (2D list) representing a relation.

Algorithm Steps

1. Initialize a Modification Flag:

    - Set `modification` to `True` to indicate that modifications to the matrix are possible.

2. Transitive Closure Computation Loop:

    - Enter a `while` loop that continues as long as `modification` is `True`, indicating that modifications to the matrix are still happening.

3. Nested Loop Iteration:

    - Iterate through each pair of rows `i` and `j` and each column `k` in the matrix:
        - Check if the values at `matrix[i][j]`, `matrix[j][k]`, and `matrix[i][k]` meet the following conditions:
            - `matrix[i][j]` is greater than 0 (there is a relation from `i` to `j`).
            - `matrix[j][k]` is greater than 0 (there is a relation from `j` to `k`).
            - `matrix[i][k]` is 0 (there is no direct relation from `i` to `k`).
        - If these conditions are met, it implies that there is an indirect transitive relationship from `i` to `k` through `j`. In this case:
            - Update `matrix[i][k]` to the sum of `matrix[i][j]` and `matrix[j][k]`, representing the transitive relationship.
            - Set `modification` to `True` to indicate that a modification was made to the matrix.

4. Return Transitive Closure:

    - Once no more modifications are possible, indicating that the transitive closure has been fully computed, the function returns the modified `matrix`, which now represents the transitive closure of the original relation.

Removing Transitive Relationships

The `remove_transitive_closure` function removes transitive relationships from a given relation matrix. It iteratively checks for transitive relationships and eliminates them, returning a matrix with only direct relationships.

Input Parameters

- `matrix`: A square relation matrix (2D list) representing a relation with transitive relationships.

Algorithm Steps

1. Initialize Variables:
    - `n`: The size of the matrix (number of rows or columns).
    - `modification`: Set `modification` to `True` to indicate that modifications to the matrix are possible.
2. Transitive Relationship Removal Loop:
    - Enter a `while` loop that continues as long as `modification` is `True`, indicating that transitive relationships are still being removed.
3. Reset Modification Flag:
    - Reset the `modification` flag to `False` at the beginning of each iteration of the outer loop.
4. Nested Loop Iteration:
    - Iterate through each pair of rows `i` and `j` in the matrix:
        - Check if there is a direct relationship from `i` to `j` (i.e., `matrix[i][j]` is greater than 0).
        - If such a relationship exists, proceed to check for transitive relationships by iterating through columns `k` starting from `j` to the end of the matrix:
            - For each `k`, check if there are relationships from `i` to `k` and from `j` to `k` (i.e., `matrix[i][k]` and `matrix[j][k]` are both greater than 0).
            - If both relationships exist, it indicates a transitive relationship from `i` to `k` through `j`. In this case:
                - Set `matrix[i][k]` to 0 to remove the transitive relationship.
                - Set `modification` to `True` to indicate that a modification was made to the matrix.
                - Print a message indicating that the transitive relationship from `i` to `k` was removed.

        - Restart the while loop to recheck for any additional transitive relationships.

5. Return Modified Matrix:
    - Once no more modifications are possible, indicating that all transitive relationships have been removed, the function returns the modified `matrix` with only direct relationships remaining.

TD4 - Slide 22 - Computing Node Ordering Based on Distance from a Specific Node

In this exercise, we are ask to compute the list of nodes ordered in the ascending order of distances to a particular node.

We used the data provided on the slide 22.

Design

The `compute` function calculates a list of nodes ordered by their distances from a specified reference node within a subset of nodes and a set of edges.

Input Parameters

- `node`: The reference node for distance calculations.
- `subset`: A list of nodes representing the subset of nodes of interest.
- `edges`: A list of edges, where each edge is represented as a tuple of two nodes.

Algorithm Steps

1. Extract Linked Edges:
    - Create an empty list `linked_edges` to store edges linked to the specified node or between nodes within the subset.
    - Iterate through the `edges` list and add edges to `linked_edges` if they are linked to the specified node or if both endpoints are in the `subset`.
2. Generate Matrix Representation:
    - Create a matrix representation of the graph using the `graph_to_matrix` function, considering the `subset` of nodes and the `linked_edges`.
3. Compute Transitive Closure:
    - Calculate the transitive closure of the matrix using the `transitive_closure` function.
4. Symmetrize the Matrix:
    - Make the matrix symmetrical to ensure that it represents relationships in both directions.

5. Calculate Distances:
    - Compute the distances between the reference node and all other nodes in the `subset`.
6. Sort Distances:
    - Sort the distances in ascending order.
7. Return Sorted Node List:
    - Return a list of nodes ordered by their distances from the specified reference node.