21035_Qn1_Tutorial2

November 19, 2023

1 QN1-S.MEENAKSHI - CB.EN.U4CSE21035

1.0.1 Suppose you are organizing a party for a large group of your friends. Your friends are pretty opinionated, though, and you don't want to invite two friends if they don't like each other. So you have asked each of your friends to give you an "enemies" list, which identifies all the other people among your friends that they dislike and for whom they know the feeling is mutual.

1.0.2 Your goal is to invite the largest set of friends possible such that no pair of invited friends dislike each other. How would you solve this problem. Model this problem appropriately and explain the modeln the model.

1.1 1. Give a greedy algorithm for the solving the problem. Prove that it is optimal or give a counter example

1.2 Soln.

1.2.1 Example Illustrating the Limitation of Relative's Greedy Algorithm:

Consider a scenario where we need to invite friends to a party, and the friends are:

- Ram
- Raju
- Balu
- Krishna
- Radha
- Sita
- Balram

Each friend has an enemy list as follows:

- **Ram:** Raju, Balu, Krishna
- **Raju:** Ram, Radha, Sita, Balram
- **Balu:** Ram, Radha, Sita, Balram
- **Krishna:** Ram, Radha, Sita, Balram
- **Radha:** Raju, Balu, Krishna, Sita, Balram
- **Sita:** Raju, Balu, Krishna, Radha, Balram
- **Balram:** Raju, Balu, Krishna, Sita, Radha

According to the relative's greedy algorithm, we would invite the person with the fewest enemies first, which in this case is Ram. However, this choice leads to a suboptimal solution.

If we invite Ram, then Raju, Balu, and Krishna cannot be invited due to their enmity with Ram. The next eligible candidates are Radha, Sita, and Balram. However, choosing any one of them results in a situation where only two members, Ram and the chosen friend, can be invited because theThe optimal solution, in this case, is to invite Raju, Balu, and Krishna, as they form a group of three friends who do not have enmity with each other.eiends without mutual enmities. h each other.h each other.d.

```
[26]: def invite_friends(friends, enemies):
    invited = []

    while friends:
        # Count the number of enemies each friend has
        enemy_counts = {friend: sum(1 for enemy in enemies[friend] if enemy not␣
 ↪in invited) for friend in friends}

        # Find the friend with the fewest enemies
        chosen_friend = min(enemy_counts, key=enemy_counts.get)

        # Check if the chosen friend is not an enemy of someone already invited
        if all(enemy not in invited for enemy in enemies[chosen_friend]):
            invited.append(chosen_friend)
            friends.remove(chosen_friend)
        else:
            # If the chosen friend has enemies among the already invited,␣
 ↪remove them from the friends list
            friends.remove(chosen_friend)

    return invited

# Updated names
friends = ["Ram", "Raju", "Balu", "Krishna", "Radha", "Sita", "Mohan"]
enemies = {
    "Ram": ["Raju", "Balu", "Krishna"],
    "Raju": ["Ram", "Radha", "Sita", "Mohan"],
    "Balu": ["Ram", "Radha", "Sita", "Mohan"],
    "Krishna": ["Ram", "Radha", "Sita", "Mohan"],
    "Radha": ["Raju", "Balu", "Krishna", "Sita", "Mohan"],
    "Sita": ["Raju", "Balu", "Krishna", "Radha", "Mohan"],
    "Mohan": ["Raju", "Balu", "Krishna", "Sita", "Radha"]
}

invited_friends = invite_friends(friends.copy(), enemies.copy())
print("Optimal invitation list:", invited_friends)
```

```
Optimal invitation list: ['Ram', 'Radha']
```

1.2.2 Conclusion:

This example demonstrates that the relative's greedy algorithm **may not always yield the optimal solution**. In this specific case, inviting Ram first does not maximize the number of friends at the party. Instead, a better choice is to invite Raju, Balu, and Krishna, forming a group of three friends without mutual enmities.

1.3 2. Can this be solved using DP or Backtracking. Explain your answer and give a suitable algorithm.

1.4 SOLN..

This problem is essentially finding the maximum independent set in an undirected graph. In graph theory, an independent set is a set of vertices in a graph, no two of which are adjacent (i.e., there is no edge connecting them). The maximum independent set is the largest possible independent set in the graph.

Model: - **Vertices:** Each friend is represented as a vertex in the graph. - **Edges:** If two friends dislike each other, there is an edge between the corresponding vertices. The goal is to find the maximum independent set, which represents the largest group of friends who do not dislike each other. - **Objective:** Maximize the size of the independent set.

We are finding the largest set of vertices (friends) such that no two vertices are connected by an edge (no two friends dislike each other).

Backtracking is a more suitable approach for this problem. The basic idea is to explore the solution space by trying different combinations of friends and backtracking when a solution is not feasible.

1.4.1 Algorithm

1. **Represent Relationships as Undirected Graph:**
   - Construct an undirected graph where each friend is a node, and there is an edge between nodes if the corresponding friends dislike each other.
2. **Initialize Variables:**
   - Create an empty set (`independentSets`) to represent the currently invited friends.
   - Create an empty set (`maximalIndependentSets`) to store the largest sets encountered.
3. **Backtracking Algorithm:**
   - Begin with the first friend:
     – Add them to the invited set (`tempSolutionSet`).
     – Recursively explore the solution space.
     – At each step:
       * Consider the next friend.
       * If adding them to the set doesn't create a conflict (check with `isSafeForIndependentSet` function), proceed recursively.
       * Backtrack if necessary by removing the friend from the set.
4. **Base Case:**
   - When there are no more friends to consider:
     – Check if the size of the current set is larger than the largest set encountered so far.
     – Update the largest set (`maximalIndependentSets`) if needed.
5. **Print Results:**

- Print all independent sets using the `printAllIndependentSets` function.
- Print the maximal independent sets using the `printMaximalIndependentSets` function.

2 Different Test cases Explored

2.0.1 Basic Case:

Input: A small number of friends with no conflicts.

Expected Output: All friends could be invited together.

2.0.2 Conflict Resolution:

Input: Friends with conflicting relationships.

Expected Output: The algorithm identified and excluded conflicting friends from the same set.

2.0.3 Multiple Independent Sets:

Input: Friends that can form multiple independent sets.

Expected Output: The algorithm found and displayed all possible independent sets.

2.0.4 Large Friendship Network:

Input: A large number of friends with various relationships.

Expected Output: The program efficiently handled a larger dataset.

2.0.5 No Friendships:

Input: Friends with no conflicts.

Expected Output: All friends could be invited together; the program handled this scenario gracefully.

2.0.6 All Friends Dislike Each Other:

Input: All friends have conflicts with every other friend.

Expected Output: No friends could be invited together.

2.0.7 Empty Input:

Input: No friends provided.

Expected Output: The program handled empty input gracefully.

[ ]:

```
```