CMSC210-04

The instructor's comments have been added in red.

Final Report

December 10, 2002

Friendly Conflict

In high school, I got along with different groups of people and had many friends. However, there was always a problem when attempting to organize a get-together. Due to the range of my friends, some of them did not like or get along with others. How could I avoid any arguments and how many parties would I have to give to spend time with all my friends? These were the questioned I always asked myself.

In order to represent everyone, each person's name in put in a set, *Friends*. Then there can be a relation, known as *conflict*, that will yield ordered pairs of people that do not get along with each other. So logically the compliment will give the people that do get along.

Friends: a set of all my friends
$conflict \subseteq Friends \times Friends$
$noConflict = conflict'$

It is important to use a relation rather than a function, because one person can dislike more than one of my other friends. However, if there is a Boolean type set, *dislikes*, then a function can be used. This is similar to the virtual pet example.

$conflictFunc : Friends \times Friends \rightarrow dislikes$

It would look similar to this:

conflictFunc = {((person1, person2), true), ((person1, person3), false), …}

Another problem was always what to do after avoiding conflicts. My friends may get along now, but their idea of what to do as a group (or subgroups) may differ. A function to *conflictFunc* can be used to find out what two people like to do with each other the most.

favorite: $Friends \times Friends \rightarrow Events$

i.e. *favorite* = {((person1, person2), ping pong), …)}

Below are all of the elements I have listed above. And they can be placed in a structure named **Party**.

Party = (*Friends*, *conflict*, *noConflict*, *conflictFunc*, *favorite*, *Events*)

- *Friends*: a set of all my friends
- $conflict \subseteq Friends \times Friends$
- $noConflict = conflict'$
- $conflictFunc : Friends \times Friends \rightarrow dislikes$
- $favorite: Friends \times Friends \rightarrow Events$
- *Events*: a set of events, e.g., ping pong

Here is a concrete example of **Friendly Conflict** with schematic representations as well.

Party = (*Friends*, *conflict*, *noConflict*, *conflictFunc*, *favorite*, *Events*)

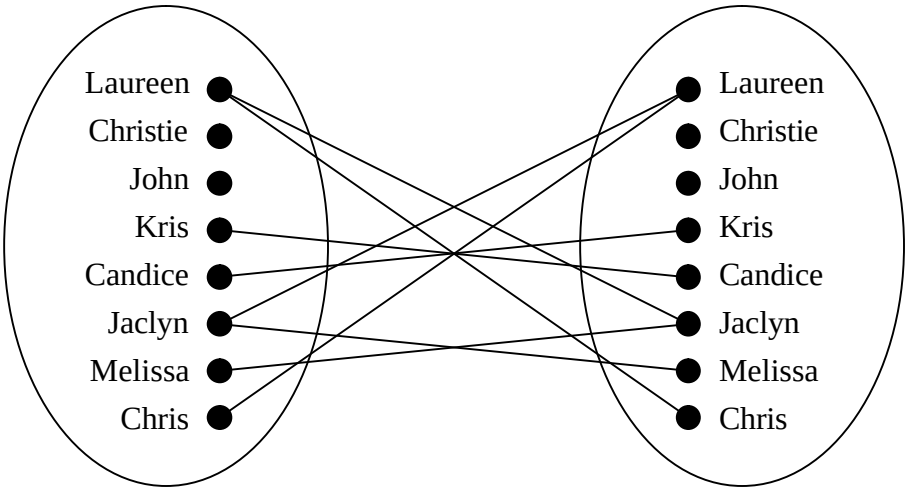
- *Friends*: a set of all my friends
- $conflict \subseteq Friends \times Friends$
- $noConflict = conflict'$
- $conflictFunc : Friends \times Friends \rightarrow dislikes$
- $favorite: Friends \times Friends \rightarrow Event$
- *Events*: a set of events, e.g., ping pong

I exclude myself from the *Friends* set only because if anyone did conflict with me, then he or she would not be my friend. Therefore, it can be assumed that every member of *Friends* gets along with me.

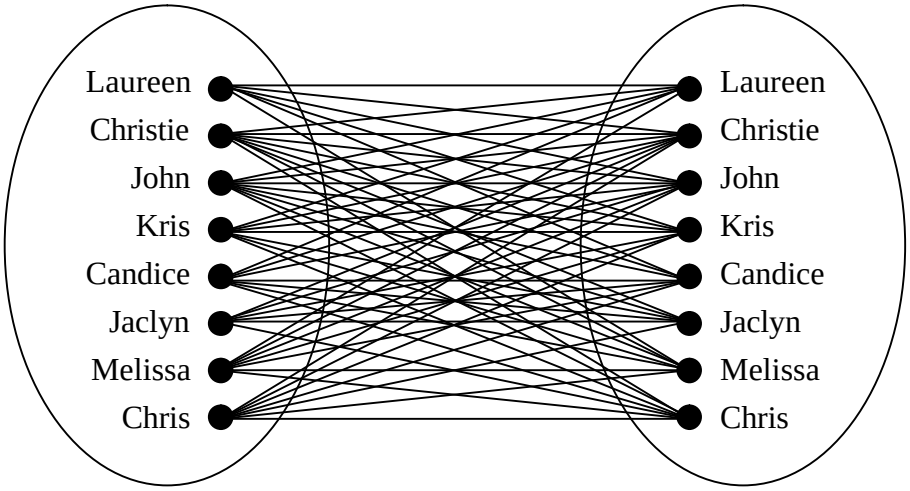
Friends = {Laureen, Christie, John, Kris, Candice, Jaclyn, Melissa, Chris}

conflict = {(Laureen, Jaclyn), (Laureen, Chris), (Kris, Candice), (Candice, Kris), (Jaclyn, Laureen), (Jaclyn, Melissa), (Melissa, Jaclyn), (Chris, Laureen)}

It turns out that for every relation $(x, y) \in Friends$, $(y, x) \in Friends$ holds, which makes the relation symmetrical. This situation can be represented in the following way.



Then the *noConflict* relation can be represented (*conflict'*). It is noticeable in this drawing that the relation is also reflexive. It makes sense to say that a person gets along with him or her self.



noConflict may have sounded like a useful relation, but only until the schematic is drawn. This does not mean that it is less useful; it just depends on the situation. If there were more conflicts than people who get along, then this relation would be easier to use.

If the drawings get too complicated, the *conflictFunc* function can be used to see precisely who conflicts with others because it includes everyone. The problem with this method is that as the number of people increase so does the elements of the function (n^2).

conflictFunc =
{ ((Laureen, Laureen), false), ((Laureen, Christie), false), ((Laureen, John), false), ((Laureen, Kris), false), ((Laureen, Candice), false), ((Laureen, Jaclyn), true), ((Laureen, Melissa), false), ((Laureen, Chris), true),
((Christie, Laureen), false), ((Christie, Christie), false), ((Christie, John), false), ((Christie, Kris), false), ((Christie, Candice), false), ((Christie, Jaclyn), false), ((Christie, Melissa), false), ((Christie, Chris), false),
((John, Laureen), false), ((John, Christie), false), ((John, John), false), ((John, Kris), false), ((John, Candice), false), ((John, Jaclyn), false), ((John, Melissa), false), ((John, Chris), false),
((Kris, Laureen), false), ((Kris, Christie), false), ((Kris, John), false), ((Kris, Kris), false), ((Kris, Candice), true), ((Kris, Jaclyn), false), ((Kris, Melissa), false), ((Kris, Chris), false),
((Candice, Laureen), false), ((Candice, Christie), false), ((Candice, John), false), ((Candice, Kris), true), ((Candice, Candice), false), ((Candice, Jaclyn), false), ((Candice, Melissa), false), ((Candice, Chris), false),
((Jaclyn, Laureen), true), ((Jaclyn, Christie), false), ((Jaclyn, John), false), ((Jaclyn, Kris), false), ((Jaclyn, Candice), false), ((Jaclyn, Jaclyn), false), ((Jaclyn, Melissa), true), ((Jaclyn, Chris), false),
((Melissa, Laureen), false), ((Melissa, Christie), false), ((Melissa, John), false), ((Melissa, Kris), false), ((Melissa, Candice), false), ((Melissa, Jaclyn), true), ((Melissa, Melissa), false), ((Melissa, Chris), false),

((Chris, Laureen), true), ((Chris, Christie), false), ((Chris, John), false), ((Chris, Kris), false), ((Chris, Candice), false), ((Chris, Jaclyn), false), ((Chris, Melissa), false), ((Chris, Chris), false) }

The *favorite* function is similar to the *conflictFunc* because it will have the same number of elements (This will be shortened). *favorite* will include ∅ in the case that the pair is reflexive or the two do not get along.

favorite =

{ ((Laureen, Laureen), ∅), ((Laureen, Christie), talk), ((Laureen, John), listen to music), ((Laureen, Kris), play pool), ((Laureen, Candice), ping-pong), ((Laureen, Jaclyn), ∅), …
((Chris, Jaclyn), play video games), ((Chris, Melissa), talk), ((Chris, Chris), ∅) }

Now to solve the problem:

John and Christie are special elements because they get along with everyone. The best way to figure out who to have over my house is to first look at who has the most conflicts (Laureen or Jaclyn). Laureen can be grouped with Melissa and Candice. Jaclyn can be placed with Chris and Kris. To even out the groups (or subsets of *Friends*), John can be in the first group and Christie in the second. This means I will have to have two different groups over my house at different times:

Group1: {Laureen, Melissa, Candice, John}
Group2: {Jaclyn, Chris, Kris, Christie}