

# Finding celebrities at a party

Uwe Hoffmann

May 2008

## Abstract

In this note we solve the celebrities problem, which is similar to [1]. The author doesn't have access to [1], so this is a guess of what that problem states.

**Problem:** Given a list of people at a party and for each person the list of people they know at the party, we want to find the celebrities at the party. A celebrity is a person that everybody at the party knows but that only knows other celebrities. At least one celebrity is present at the party.

## 1 Analysis

Let  $\mathbb{P}$  be the set of people at the party and  $\kappa : \mathbb{P} \rightarrow 2^{\mathbb{P}}$  the function that describes who knows whom at the party. The subset  $\mathbb{C} \subseteq \mathbb{P}$  of celebrities at the party is defined recursively as

**Definition 1.1.**

$$\mathbb{C} = \{c \in \mathbb{P} : (\forall p \in \mathbb{P} : c \in \kappa.p) \wedge (\kappa.c \subseteq \mathbb{C})\}$$

**Lemma 1.1.**

$$\forall p \in \mathbb{P} : \mathbb{C} \subseteq \kappa.p$$

*Proof.* Follows directly from the definition of celebrity. □

We introduce the notion of peer. A peer at the party is someone you know that knows you too. Function  $\pi : \mathbb{P} \rightarrow 2^{\mathbb{P}}$  returns all the peers of a person and function  $\eta : \mathbb{P} \rightarrow 2^{\mathbb{P}}$  all the non-peers:

**Definition 1.2.**

$$\forall p \in \mathbb{P} : \pi.p = \{q \in \kappa.p : p \in \kappa.q\}, \eta.p = \kappa.p \setminus \pi.p$$

**Lemma 1.2.**

$$\forall p \in \mathbb{P} : \eta.p = \emptyset \Leftrightarrow \pi.p \cup \{p\} = \mathbb{C}$$

*Proof.* First we prove that  $p \in \mathbb{C} \Leftrightarrow \pi.p \cup \{p\} = \mathbb{C}$ . Assume  $q \in \pi.p$  and  $q$  not a celebrity. This is a contradiction because  $p$  knows  $q$  and  $p$  is a celebrity hence it can only know celebrities. The reverse is obvious.

To prove the lemma, assume  $\eta.p \neq \emptyset$  and  $q \in \eta.p$ . If  $p$  were a celebrity then everybody would know him, but  $q$  doesn't know him, which is a contradiction. So  $p \in \mathbb{C} \Rightarrow \eta.p = \emptyset$ . The reverse follows like this: assume  $p$  is not a celebrity so it must know a celebrity (there is one present at the party) and that celebrity is not his peer otherwise he would be a celebrity. So that celebrity is in  $\eta.p$  which contradicts  $\eta.p = \emptyset$ .  $\square$

Armed with these two lemmas we can outline an algorithm to compute the set  $\mathbb{C}$  of celebrities. Lemma 1.1 tells us that we can start with any person  $p$  because the set of celebrities is a subset of  $\kappa.p$ . Lemma 1.2 tells us to look at the set of people that  $p$  knows that are not his peers:  $\eta.p$ . If  $\eta.p$  is empty then  $p$  and his peers are the celebrities. If it is not empty then  $p$  cannot be a celebrity but the set of celebrities is contained in  $\eta.p$ , which let's us recursively continue with a smaller problem (the set of people without  $p$  and his peers).

## 2 Implementation in Scala

This section shows a small Scala [2, 3] program that implements the outlined algorithm. The *Relationships* trait provides the  $\kappa$  function that returns who knows whom at the party, somebody's *rolodex* is the list of people he knows at the party. The *findCelebsThru* function implements the algorithm. It first partitions the rolodex into a tuple of two lists: peers and non-peers. It then decides according to the number of non-peers if it's done or if it has to call itself recursively. This function is tail-recursive.

The rest of the code just sets up a small example with a party of five people and two celebrities. All the *println* calls in the *main* function should return the same result, namely Alice and Peter, the celebrities in this example.

Listing 1: Scala code

```

package com.codemantic.scala.minis

import collection._

object celebs {

  trait Relationships [T] {
    def knows(p:T, q:T): Boolean
    def rolodex(p:T): List [T]
  }

  def findCelebsThru [T]( person:T,
                        rels: Relationships [T]): List [T] = {
    val parts =
      rels.rolodex(person).partition(rels.knows(_, person))
    parts._2 match {
    case List () => person :: parts._1
    case p :: ps => findCelebsThru [T](p, rels)
    }
  }

  def findCelebs [T]( people: List [T],
                    rels: Relationships [T]): List [T] = {
    people match {
    case List () => List ()
    case p :: ps => findCelebsThru [T](p, rels)
    }
  }

  // ----- example -----

  class SimpleRelationships (data: Map [String , List [String]])
  extends Relationships [String] {
    private val rels = data
    override def rolodex (p: String): List [String] = {
      rels (p)
    }
    override def knows (p: String , q: String): Boolean = {

```

```

    rels(p).exists(_ == q)
  }
}

def main(args: Array[String]): Unit = {
  val party = List("Adam", "Dan", "Alice", "Peter", "Eva")
  val partyRels = new SimpleRelationships(
    collection.immutable.Map(
      "Adam" -> List("Dan", "Alice", "Peter", "Eva"),
      "Dan" -> List("Adam", "Alice", "Peter"),
      "Eva" -> List("Alice", "Peter"),
      "Alice" -> List("Peter"),
      "Peter" -> List("Alice")
    )
  )
  println(findCelebsThru("Adam", partyRels))
  println(findCelebsThru("Dan", partyRels))
  println(findCelebsThru("Eva", partyRels))
  println(findCelebsThru("Alice", partyRels))
  println(findCelebsThru("Peter", partyRels))
  println(findCelebs(party, partyRels))
}
}

```

The program does terminate because at each recursive call the set of people shrinks by at least one person. The program only works correctly if at least one celebrity is present. If no celebrities are present the program will stop at a person that doesn't have any non-peers instead of returning an empty list. Consider Figure 1 and imagine the algorithm starts at one of the green nodes. It will end in one of the blue nodes which are not celebrities because not every node in the figure knows them.

The worst case runtime is  $O(n^2)$ , where  $n$  is the number of people at the party. We see this with the following setup as suggested by Figure 2:  $p_0, p_1, \dots, p_{n-1}$  are the people at the party and  $\forall i < j : (p_j \in \kappa.p_i) \wedge (p_i \notin \kappa.p_j)$ . Assuming *findCelebsThru* gets called with  $p_0$  and *Relationships* are returned such that the recursive calls will be in order  $p_i$ ,  $1 \leq i < n$  then the partitioning into peers and non-peers incurs the  $O(n^2)$  cost.

This can be turned into a linear program if *Relationships* returns peers first and then non-peers in its rolodex lists and the partitioning code stops

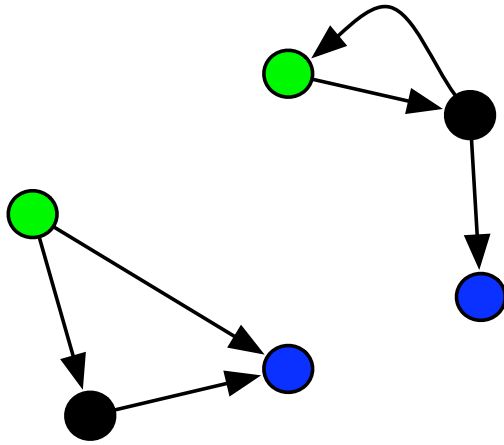


Figure 1: No celebrities at the party.

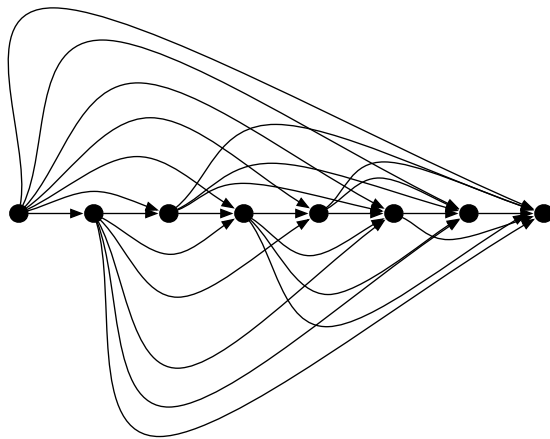


Figure 2:  $O(n^2)$  runtime.

at the first non-peer. Since peers get discarded they don't get visited again and since the first non-peer moves the algorithm forward it becomes linear.

## References

- [1] Richard Bird and Sharon Curtis. Functional pearls: Finding celebrities: A lesson in functional programming. *J. Funct. Program.*, 16(1):13–20, 2006.
- [2] Martin Odersky. Scala by example. 2008.
- [3] Martin Odersky and Others. The scala language specification. 2008.