Computing celebrities: A lesson in problem solving

0 Celebrity cliques

In Anne Kaldewaij's textbook [2] the so-called *celebrity problem* is discussed and solved, as an illustration of the programming technique called *Searching by Elimination*. The problem of the *Celiberity Clique*, which I encountered in a recent publication by Richard Bird and Sharon Curtis [1], is a very nice generalization of this celebrity problem.

We consider a non-empty and finite group G of *persons*. A Celebrity Clique, or *C-Clique* for short, is a non-empty subset of G such that every member of G *knows* every member of the C-Clique, and every member of the C-Clique knows nobody outside the C-Clique. Depending on the properties of the "knows"-relation, group G may or may not contain a C-Clique, but if it exists the C-Clique is unique.

The (programming) problem is to construct a program to compute the C-Clique in a given group G, with a given "knows"-relation, and given that the C-Clique exists. Preferably, the program's time complexity is linear in the size of G, whereas the only primitive operation available is the evaluation of the boolean expression "x knows y", for any two group members x and y. Notice that the number of possible pairs of group members is quadratic in the size of the group; hence, any linear time algorithm can only evaluate this relation for a very restricted subset of all possible pairs.

* * *

We formalize the problem as follows. We consider a non-empty and finite set G, together with a given binary relation on G, here denoted by the infix symbol $\mapsto$ ("knows"). In what follows variables x, y, z range over this (non-empty and finite) set G.

The question whether or not a member of G knows itself is not a philosophical one, but is actually irrelevant for the problem. For purely pragmatic reasons – to save some case analysis –, however, we assume that every member knows itself, that is, $(\forall x :: x \mapsto x)$. In what follows, we will use this implicitly.

A *C-Clique* now is a non-empty subset C of G, having the following two properties, for all x, y (in G):

(0) $\quad y \in C \;\Rightarrow\; x \mapsto y$

(1) $\quad x \in C \wedge x \mapsto y \;\Rightarrow\; y \in C$

Depending on the binary relation $\mapsto$, set G may or may not contain a C-Clique, but it is not difficult to verify – see Section 4 – that G contains at most one C-Clique: if it exists a C-Clique is unique.

The problem now is: construct an efficient algorithm to compute the C-Clique in a given set G with a given binary relation $\mapsto$, with precondition that G contains a C-Clique. In what follows we use C to denote the C-Clique to be computed.

1 A "little theory"

Property (0) is easily exploited, as it can be rewritten as follows, for all x, y:

(2) $\quad \neg(x \mapsto y) \;\Rightarrow\; \neg(y \in C)$.

Hence, if $\neg(x \mapsto y)$ then y is not a member of C, so y can be safely discarded, without affecting C. Thus, with a constant amount of work the size of the set is reduced by one.

Exploiting property (1) is not as easy, though: from $x \mapsto y$ we cannot, for instance, – as in the "classical" celebrity problem – simply conclude $\neg(x \in C)$, which would yield another simple opportunity to reduce the size of the set. Therefore, we investigate (1) a little further, and try to derive some additional properties. Occasionally, this is the only thing one can do: to develop a little theory about the problem, so as to capture its essential mathematics. For our problem, "theory" is a big word here, because everything will remain relatively simple. The need of this "little theory" is, however, indicated: properties (0) and (1) do not seem to allow obvious generalizations leading to a solution of the problem.

By propositional calculus we rewrite (1) into, for all x, y:

(3) $\quad x \mapsto y \;\Rightarrow\; (x \in C \Rightarrow y \in C)$.

Now, we observe that property (2) has a symmetric counterpart, namely:

(4) $\quad \neg(y \mapsto x) \;\Rightarrow\; \neg(x \in C)$,

so we can confine our attention to the case $x \mapsto y \wedge y \mapsto x$, this being the only case still needing attention; we abbreviate this case to $x \leftrightarrow y$. Combining property (3) with its obvious symmetric counterpart, and by elementary propositional calculus, we obtain, for all x, y:

(5) $\quad x \leftrightarrow y \;\Rightarrow\; (x \in C \Leftrightarrow y \in C)$.

If $x \in C \Leftrightarrow y \in C$ we call x and y *equicelebrities*: either both are elements of the C-Clique or both are not elements of the C-Clique. In words, property (5) states that if x and y know each other then they are equicelebrities.

The essence of this observation is that "being equicelebrities" is an equivalence relation, whereas we have no such information about the relation "knowing each other". In particular, the relation "being equicelebrities" is transitive, and this is the key to the problem's solution. The equicelebrity relation has two equivalence classes: the C-Clique, C, and its complement, $G \setminus C$.

These considerations and property (5) inspire us to study so-called *equicelebrity sets*. A subset E of G is an equicelebrity set if (and only if):

(6) $\quad (\forall x, y : x, y \in E : x \in C \Leftrightarrow y \in C)$.

If E is an equicelebrity set, only two cases are possible: either $E \subseteq C$ or $E \subseteq G \setminus C$, which can also be rendered as:

(7) $\quad E \subseteq C \ \vee \ E \cap C = \phi$.

The relevance of this is that if $x \in E \wedge \neg(x \in C)$, for any x, then it is clear not only that x is not in C but also that *all* elements of E are not in C; this is the key to an efficient algorithm for the computation of C: equicelebrity sets provide a useful generalization of the notion of a C-Clique.

2 An efficient algorithm

A fairly general generalization technique to obtain algorithms for "processing" the elements of a given set is to bipartition the set into two disjoint parts: one subset represents the elements that "have already been processed" and the other subset represent the elements that "have not been considered yet".

In our case, the notion of an equicelebrity set suggests a tripartioning of group G: one subset represents the elements that definitely are not celebrities, one subset is an equicelebrity set, and the third subset represents the elements that "have not been considered yet". The subset containing definitely non-celebrities can remain anonymous; in what follows the equicelebrity set is named E and the subset of elements that "have not been considered yet" is named D. The elements of subset E may or may not be celebrities; all we know is that all elements of E will eventually share the same fate. The elements of D, too, may or may not be celebrities; after all, nothing is known about these elements. We do know, however, that the elements of the anonymous subset are not celebrities. So, C is a subset of $E \cup D$.

The algorithm is based on the following two properties of equicelebrity sets. The proofs of these properties are straightforward but do require property (5) and use of the fact that "being equicelibrities" is an equivalence relation.

Firstly, for every y, $y \in G$:

(8) $\{y\}$ is an equicelebrity set .

Secondly, for any equicelebrity set E and for every x, $x \in E$ and y, $y \in G$:

(9) if $x \leftrightarrow y$ then $E \cup \{y\}$ is an equicelebrity set .

Although we formulate the algorithm as a functional program, it can be equally well formulated as an iterative sequential program: the definition of the function turns out to be tail-recursive and the function's preconditions can be used as repetition invariants.

We introduce a function *celebs* with two parameters, E and D, subsets of G. The function's value is the C-Clique in G, named C. We recall that it also is a precondition that G contains a C-Clique; this is implied by the following condition (10). Conditions (11) and (12) reflect the tripartioning of G as suggested above; for technical reasons, to avoid some case analysis, E is non-empty. If G does not contain a C-Clique, that is, if C does not exist, the value of *celebs* is irrelevant and is left unspecified.

(10) C is the C-Clique in G

(11) $E \neq \phi \ \wedge \ E \cup D \subseteq G \ \wedge \ E \cap D = \phi$

(12) E is an equicelebrity set

(13) $C \subseteq E \cup D$

In conjunction, these conditions form the function's precondition. Thus, we obtain as specification for function *celebs*, for all sets C, D, E, G:

$$(10) \ \wedge \ (11) \ \wedge \ (12) \ \wedge \ (13) \ \Rightarrow \ celebs \cdot E \cdot D = C \ .$$

This specification can now be used to solve the problem, as follows. For any $y : y \in G$, the singleton $\{y\}$ satisfies condition (12); hence, $\{y\}$ and $G \backslash \{y\}$ are good values for parameters E and D, satisfying (10) through (13). As a result, we obtain:

(14) $C \ = \ celebs \cdot \{y\} \cdot (G \backslash \{y\})$ where $y : y \in G$ endwhere .

By means of the above properties of equicelebrity sets, a recursive definition for *celebs* is easily derived. Firstly, if $D = \phi$ then preconditions (11) and (13) boil down to:

$$E \neq \phi \ \wedge \ E \subseteq G \ \wedge \ C \subseteq E \ .$$

The conjunct $C \subseteq E$ is equivalent to $C \cap E = C$; because $C \neq \phi$ we conclude $C \cap E \neq \phi$, from which we derive, using property (7), that $E = C$.

Secondly, for the case $D \neq \phi$ and using that $E \neq \phi$, we assume x and y to be elements of E and D, respectively. By means of a three-way case analysis, we observe:

- if $x \leftrightarrow y$ then, by property (9) and precondition (12), the set $E \cup \{y\}$ is an equicelebrity set too; hence, $E \cup \{y\}$ and $D \setminus \{y\}$ (for E and D) satisfy the function's precondition as well.
- if $\neg(x \mapsto y)$ then, by property (2), value y can be discarded, that is, sets E and $D \setminus \{y\}$ satisfy the function's precondition.
- if $\neg(y \mapsto x)$ then, by property (4), we conclude $\neg(x \in C)$; hence, by property (7) we conclude that $E \cap C = \phi$. So, the whole of set E can now be discarded, and $\{y\}$ and $D \setminus \{y\}$ (for E and D) satisfy the function's precondition.

Thus, we obtain the following recursive definition for function *celebs*:

$$\begin{array}{l} \mathit{celebs} \cdot E \cdot D \ = \\ \quad \mathsf{if} \ \ D = \phi \ \rightarrow \ \ E \\ \quad [] \ \ D \neq \phi \ \rightarrow \ \ \mathsf{if} \quad x \leftrightarrow y \quad \rightarrow \ \ \mathit{celebs} \cdot (E \cup \{y\}) \cdot (D \setminus \{y\}) \\ \qquad\qquad\qquad\quad [] \ \neg(x \mapsto y) \ \rightarrow \ \ \mathit{celebs} \cdot E \cdot (D \setminus \{y\}) \\ \qquad\qquad\qquad\quad [] \ \neg(y \mapsto x) \ \rightarrow \ \ \mathit{celebs} \cdot \{y\} \cdot (D \setminus \{y\}) \\ \qquad\qquad\qquad\quad \mathsf{fi} \ \ \mathsf{where} \ \ x : x \in E \ \ \& \ \ y : y \in D \ \ \mathsf{endwhere} \\ \quad \mathsf{fi} \end{array}$$

This definition is obviously well-founded: in each recursive application of the function (finite) set D is replaced by a smaller one. Hence, the time complexity of this function is linear in the size of parameter D.

3 Implementation issues

The above abstract algorithm has been formulated in terms of sets, because the problem is about sets. Sets can be represented in several different ways, thus permitting several different implementations of the algorithm. In particular, the nondeterminism in the selections $y : y{\in}G$ –in (14)–, $x : x{\in}E$, and $y : y{\in}D$, can be refined to efficient instances, fitting the actual representation chosen.

Just for the fun of it, we present one of the many possibilities here. We represent set G as a non-empty interval $[0\,..\,N{+}1)$, with $0{\leq}N$, of the natural numbers. Parameter E being a non-empty and generally non-contiguous subset of G, it can be represented by (enumeration of its elements in) a non-empty finite list. In turn we represent the head and the tail of this list separately, as a single natural parameter m and a finite list s of naturals, with $E = [\![\, m{\triangleright}s \,]\!]$. (Here $[\![\cdot]\!]$ denotes the unavoidable "the set represented by" abstraction function.) If we now refine $y : y{\in}G$ and $y : y{\in}D$ to selections of the largest numbers in the sets, set D will (invariantly) remain an interval of the shape $[0\,..\,n)$; hence, D can be conveniently represented by a single natural parameter n, say. Finally, the function's value can be represented as a list as well.

Thus, function *celebs* is implemented by a new function *celebi*, say, with this specification, for all natural m, n and list s of naturals:

$$[\![\, celebi{\cdot}m{\cdot}s{\cdot}n \,]\!] \;=\; celebs{\cdot}[\![\, m{\triangleright}s \,]\!]{\cdot}[0\,..\,n) \quad .$$

If we now refine the selection $y : y{\in}G$, where $G{=}\,[0\,..\,N{+}1)$, to $y{=}N$, the original expression for the C-Clique:

$$celebs{\cdot}\{y\}{\cdot}(G{\setminus}\{y\}) \;\;\mathsf{where}\;\; y : y{\in}G \;\mathsf{endwhere} \quad ,$$

is implemented in terms of the new function *celebi* by:

$$celebi{\cdot}N{\cdot}[\,]{\cdot}N \quad .$$

It now is a straightforward exercise to derive the following recursive definition for *celebi*:

$$\begin{array}{lcl}
celebi{\cdot}m{\cdot}s{\cdot}0 & = & m \triangleright s \\
celebi{\cdot}m{\cdot}s{\cdot}(n{+}1) & = & \mathsf{if} \quad m \leftrightarrow n \;\rightarrow\; celebi{\cdot}n{\cdot}(m \triangleright s){\cdot}n \\
 & & [\!]\;\; \neg(m \mapsto n) \;\rightarrow\; celebi{\cdot}m{\cdot}s{\cdot}n \\
 & & [\!]\;\; \neg(n \mapsto m) \;\rightarrow\; celebi{\cdot}n{\cdot}[\,]{\cdot}n \\
 & & \mathsf{fi}
\end{array}$$

Because list parameter s represents a set, the order in which elements occur in s is irrelevant. In the above program, for instance, the recursive application $celebi \cdot n \cdot (m \triangleright s) \cdot n$ might have equally well been written as $celebi \cdot m \cdot (n \triangleright s) \cdot n$; the above version happens to be such that the elements of set E appear in $m \triangleright s$ in decreasing order, for whatever that is worth. As a result, the C-Clique is returned as a decreasing list.

4 Epilogue

The problem how to compute a Celebrity Clique is a nice generalization of the celebrity problem from [2], where it is used to illustrate Searching by Elimination. The solution presented here is essentially the same as the solution by Bird and Curtis [1]. This solution required some preliminary work but can also be viewed as a (somewhat more intricate) application of Searching by Elimination.

The problem of the celebrity cliques is harder than the original celebrity problem, because in the latter both $x \mapsto y$ and $\neg(x \mapsto y)$ allow elimination of one of the members of the set, thus reducing the set by one element. As we have seen, in the current problem the case $x \mapsto y$ is not that simple anymore.

Analyzing the situation I derived property (5) rather quickly, but at first I did not see how to exploit it. Only after a while, I became aware that "being equicelebrities" is transitive, that this is relevant, and that the notion of equicelebrity sets might be useful, as approximations of the C-Clique to be computed. Once this hurdle was taken, I could construct the solution without further difficulties. Neither the problem nor its solution have much to do with functional programming per se, though.

* * *

The discussion in [1] suffers from undue attention to functional programming techniques. As a result, the essential mathematics of the problem and its solution are obfuscated. The "straigthforward" exponential-time algorithm is not straightforward at all: to my taste, its correctness requires proof. Transformational programming in general, and fusion techniques in particular, have been shown to be very effective, but they are not always the best way to solve a problem. Occasionally, an implicit problem specification, that cannot be formulated in a functional programming language, is simpler, more elegant, and, therefore, more appropriate than a so-called "straightforward" functional program. (Sorting is the obvious example that comes to my mind.) This also reminds me of Edsger W. Dijkstra's old adagium: "Do not rush into coding".

Another aspect, another symptom of the same phenomenon actually, is the use of *lists* to formulate the problem, whereas the problem actually is about *sets*. In [1] this complicates the discussion, with issues like whether the empty list should or should not play a role, and whether or not the lists are allowed to contain duplicate elements. These issues are implementation details, relevant as such, but they are better ignored until the problem has been solved.

Finally, somewhere in the discussion in [1], the symbol $\bot$ and an "approximation ordering" $\sqsubseteq$ enter the scene, quite unnecessarily so.

* * *

Having read a draft of this paper, Jan Friso Groote made the following observation, for the sake of which we recall properties (0) and (1) that define subset C as the *C-Clique*; in addition C is non-empty. For all x, y:

(0) $\quad y \in C \;\Rightarrow\; x \mapsto y$

(1) $\quad x \in C \wedge x \mapsto y \;\Rightarrow\; y \in C$

Property (0) states that every element of C is known by all members of the group. Conversely, if some y, $y \in G$, is known by all members of G, that is, if $(\forall z :: z \mapsto y)$, then we derive, for any $x : x \in C$ – after all, C is non-empty –:

$$
\begin{array}{ll}
 & y \in C \\
\Leftarrow & \quad \{ \text{ property (1) } \} \\
 & x \in C \;\wedge\; x \mapsto y \\
\equiv & \quad \{ \; x \in C \; \} \\
 & x \mapsto y \\
\Leftarrow & \quad \{ \text{ instantiation } z := x \; \} \\
 & (\forall z :: z \mapsto y) \\
\equiv & \quad \{ \text{ assumption on } y \; \} \\
 & \mathsf{true} \quad .
\end{array}
$$

This proves that C is exactly the subset of those elements that are known by all elements of G:

$$y \in C \;\equiv\; (\forall z :: z \mapsto y) \quad , \text{ for all } y \quad .$$

This is a simple and explicit characterization of C that, therefore, also proves that C is unique. In addition, this characterization suggest a simple, albeit quadratic, algorithm for the computation of C.

References

[1] R. Bird, S. Curtis,
Finding celebrities: A lesson in functional programming.
Journal of Functional Programming **16**, pp. 13 - 20 , 2006.

[2] A. Kaldewaij, *Programming: The Derivation of Algorithms*.
Prentice-Hall, 1990.

Eindhoven, 23 August 2006

Rob R. Hoogerwoord
department of mathematics and computing science
Eindhoven University of Technology
postbus 513
5600 MB Eindhoven