Notes on algorithms: 3

Version 1 at 25 September 2016

0 Contents

1 Introduction

This Note is less about an algorithm, more a proof that an algorithm is possible. The Note is about a class of structures that will be called Well-Founded Multigraphs here, for want of a better name. They are a generalisation of the more familiar Lists, Trees, and Well-Founded Relations.

The note starts with an informal description of these structures and of some important properties they have (section 2). It then shows how these structures relate to some simpler and better known structures (section 3). It finishes with a detailed mathematical model of the structures followed by proofs of their important properties (section 4).

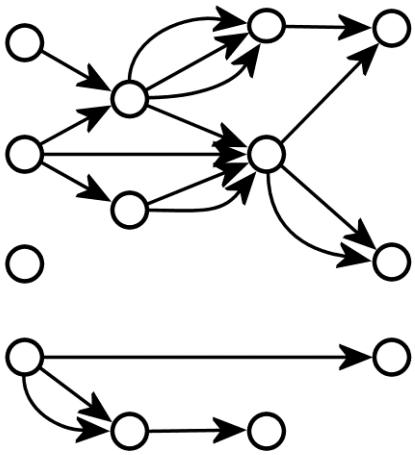
Notes on algorithms: 3

2 Outline

A graph consists of points with links that connect some of these points. The number of points and links can be finite or infinite. In this note the links are oriented. A link goes from a particular point to a particular point so a link can be drawn in pictures as an arrow. Each arrow goes from the point at its tail to the point at its head. The prefix “multi” in **multigraph** indicates that there can be several arrows with the same tail point and head point.

This note is about multigraphs. Figure 2.1 shows a picture of one.

Figure 2.1 A multigraph, one that happens to be well-founded



In a practical application of multigraphs the points and arrows in the system under study often have properties. The multigraph itself ignores these properties and just shows the structure. It may be that some of the properties need to be proved, defined, or calculated. Can the multigraph structure assist in proving, defining, and calculating? It can if it is a particular kind of multigraph. In Figure 2.1 start at any point; follow an arrow backwards from head to tail; repeat, stopping only if there is no arrow to follow. Notice that you always come to a stop after a finite number of steps. You never find yourself back at the same point going round and round for ever. This kind of multigraph can have an infinite number of points. If so, there will still be a finite number of points before coming to a stop. You never find yourself stepping through an infinite number of points and so stepping for ever.

This kind of multigraph where stepping backwards always comes to a stop is called **Well-Founded**. This note is specifically about Well-Founded Multigraphs. The multigraph shown in Figure 2.1 is one of them.

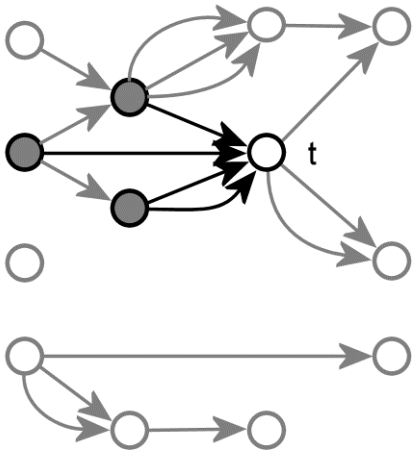
In any Well-Founded Multigraph there will be at least one point with no incoming arrows; that is, it is not at the head of any arrow. These can be called **base points** or **root points**. They are shown on the left in Figure 2.1.

Notes on algorithms: 3

Starting from the base points and following arrows forwards from point to point it is possible to reach every point in a Well-Founded Multigraph after a finite number of steps, even if there are an infinite number of points in the multigraph. This, in effect, is how proofs, definitions, and calculations can propagate from the base points to all the other points in the multigraph.

An important construct when proving, defining, and calculating are the **predecessors** of each point. In Figure 2.2 the highlighted points are the predecessors of the point *t*. An arrow goes from each predecessor to the point *t*. In some contexts it can be preferable to call them **immediate predecessors** to distinguish them from points more than one arrow away, but this is not necessary in this note. The base points have no predecessors, of course; their predecessor sets are empty.

Figure 2.2 A point, *t*, and its predecessors



Well-Founded multigraphs can be distinguished from other multigraphs by the fact that following arrows backwards always comes to a stop. There is another, entirely equivalent, way of distinguishing them that is more convenient for a mathematical treatment. If *every* non-empty subset of the multigraph's points contains at least one point with no predecessors in the subset then the multigraph is Well-Founded; otherwise it is not. Such a point might have no predecessors at all, or have predecessors but none of them in the subset. If the multigraph has any points at all then the subset consisting of all its points will contain at least one point with no predecessors, a base point, as said earlier. (Its predecessors cannot be outside the subset that contains all the points, obviously).

Notes on algorithms: 3

2.1 Induction

How can the structure of a Well-Founded multigraph be used to prove that something is true at all its points? All that needs to be done is to prove that :

1 It is true at each base point;
2 For any other point, *t*, if it is true for all of *t*'s predecessors then it must be true for *t*.

Once this has been done then it is certain that it is true at all points in the multigraph. (This is the **Induction Principle**, proved in section 4).

Here is an example :

To prove
A. Every point in any Well-Founded multigraph can be reached from a base point by following a finite number of arrows.

Proof

For convenience define the term "path" : a path is a sequence of
point, arrow, point, arrow, point, …
for as long as needed, starting and ending at a point. It records the result of following arrows forward from point to point. Its length is the number of arrows in the sequence.

Consider a Well-Founded Multigraph, *G*.

If *G* has no points then there is nothing to prove; otherwise :

1 For a base point, *b*, in *G* :
The (only) path from a base point to *b* is the sequence *b*, so the number of arrows and hence length is zero, which is finite.

2 For any other point, *t*, in *G* :
If every path from the base points to each predecessor of *t* is finite, then every path to *t* will have just one more arrow so is also finite.

Conclude that for each point of *G* there is a finite path to the point from some base point.

As *G* is any Well-Founded Multigraph conclude that **A** is true.

QED

In fact we have proved a stronger theorem : that *every* path to each point is finite, not just some path.

Notes on algorithms: 3

2.2 Recursion

How can the structure of a Well-Founded multigraph help to define a function that assigns a value to each of its points? In some cases the structure is not relevant. For instance, if each point has a length and a width associated with it then the definition of a function to associate an area with each point requires no help from the structure.

On the other hand, if the value at a point is partly determined by the function's values at the point's predecessors then the structure matters a lot. For instance, if a predecessor of a predecessor of the point *t* is *t* itself then the value at *t* must be found by solving an equation. The solution might not exist or there might be many possible values.

However, when the structure of the points is a Well-Founded multigraph then there is a straightforward way to define such functions. State a rule (define an auxiliary function) which for any point *t* gives a value determined by the value of the function at *t*'s predecessors and by values already associated with *t*, *t*'s predecessors, and each arrow from *t*'s predecessors to *t*, and by no other points or arrows. Once this is done then it is certain that there is a function that gives each point a value, that the value at each point obeys the rule, and that there is only one possible such value at each point. (This is the **Recursion Theorem**, proved in section 4).

The rule does not have to use all the values it is allowed to use. For instance, the example of areas given above can be defined using this technique; then the rule uses values already associated with *t* and no others. Values do not have to be numbers. They can be strings, sets of points, paths, vehicles, people, etc. In general, they can be any mathematical objects and they can be any objects that can be modelled by mathematical objects.

Here is an example :

Requirement
Define the function *MaxPathLen* that for each point returns the length of the longest path from a base point to the point.

We proved in section 2.2 that each path is finite, but there is a complication. If there are an infinite number of points then it is possible to have an infinite number of (finite) path lengths. This introduces complications in the definition of "longest" which are best avoided in a simple example. *MaxPathLen* will be defined only for a Well-Founded multigraph with a finite number of points. Most physical systems have a finite structure so this restriction is not oppressive.

Definition
Given some Well-Founded Multigraph, *G*, where *G* has a finite number of points, then the function *MaxPathLen*, defined on all points of *G*, is such that at any point *t* in *G* :

1 **if** *t* is a base point, then
MaxPathLen(*t*) = 0

2 **otherwise**
MaxPathLen(*t*) = 1 + **maximum**[*MaxPathLen*(*p*) for all predecessors, *p*, of *t*]

Notes on algorithms: 3

We can be certain that *MaxPathLen* exists, is defined for all points of *G*, and that no other function obeys the definition. As *G* could be any finite Well-Founded Multigraph it would be more accurate to indicate this by marking the function name, e.g *GMaxPathLen* or $MaxPathLen_G$.

Notice that base points and other points are treated separately. This will often be the case. Here it avoids saying that the **maximum** of no numbers at all is declared to be **-1**.

Notes on algorithms: 3

2.3 Calculation

How can the structure of a Well-Founded multigraph help to calculate the values at its points? This depends on how the values are defined. If they are defined recursively in the way described in section 2.2 then the structure indicates two ways to go about doing the calculations.

One way can be thought of as the bottom up way. Calculate the values at all the base points, then follow arrows and calculate the values for points whose predecessor values have already been calculated. Continue until all points have been calculated. Store the results for future use. The value at each point is calculated by following the rule used in defining the function being implemented.

This way calculates the values at all points, which might be far more than will be needed so it might be seen as a waste of time. It might also take too much time finding points that can now be calculated. (Doing a topological sort on the points first would help here).

The other way can be thought of as the top down way. When asked to calculate the value of the point *t*, go to *t* and follow the rule used in defining the function being implemented. If the rule requires the values at the predecessors of *t* then for each predecessor, *p*, calculate the value at *p* in the same way. Eventually point(s) will be reached that have no predecessors (guaranteed, as usual) so following predecessors downward is always a procedure that terminates.

This way's purpose is to calculate the value at one point. It is unlikely to be efficient for calculating the values at all points. It will calculate the right value even if the structure or values associated with it have changed since the previous calculation. The main danger in computer programming is not noticing that a base point has been reached and then doing something inappropriate.

The second way can be adapted to calculate the values at all points. Calculate the value at an arbitrary point, but store the values calculated while doing so and use the stored values when calculating the values at other points. Repeat until all points have been calculated.

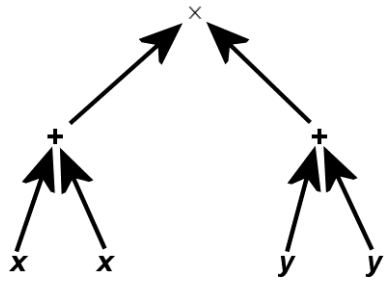
Notes on algorithms: 3

2.4 Is this useful ?

Are these structures of any practical significance?

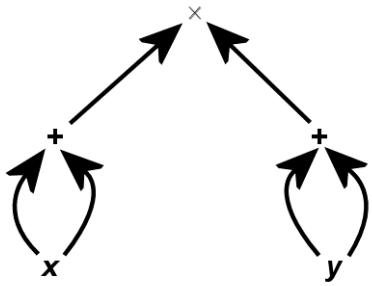
Yes! Consider the expression **(*x* + *x*) × (*y* + *y*)**. Its parse tree is shown in figure 2.4.1.

Figure 2.4.1 Parse tree for (x + x) × (y + y)



This parse tree is rather misleading. It suggests that when calculating the value of the expression it would be possible to start with different values in the two occurrences of *x*. The same is true of *y*. A more honest picture of the parse 'tree' is shown in Figure 2.4.2.

Figure 2.4.2 More honest parse 'tree' for (x + x) × (y + y)



Now there is no doubt that *x* has only one value, and so does *y*. This parse 'tree' is a multigraph (as you might have guessed it would be).

When values are given to *x* and *y* can we be sure that the value of the expression will be calculated correctly? The Recursion Theorem tells us that a value for all the points, especially the top point, does exist. It also tells us that there is only one answer. Therefore the calculation will certainly give the correct answer as it is the only answer.

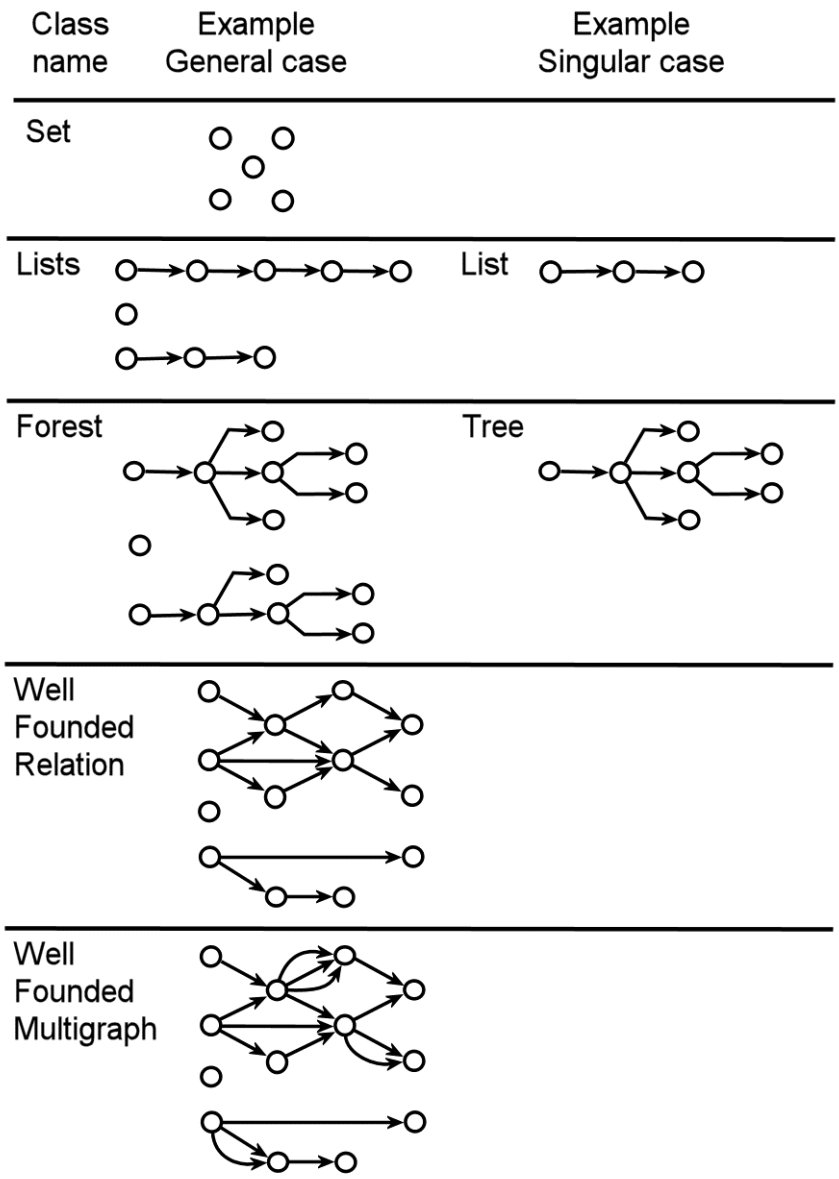
A full treatment of expressions would need to prove some things not mentioned so far : for instance that every Well Formed Expression can be modelled as a Well-Founded Multigraph. This multigraph might have its arrows pointing in the wrong direction. Provided the number of points is finite then reversing all the arrows leads to another Well-Founded Multigraph (proved in section 4). The treatment would also need to introduce something that distinguishes one arrow from the other in sub-expressions such as *a* - *b* so that *b* will be subtracted from *a*, not *a* from *b*.

Notes on algorithms: 3

3 A taxonomy

Well-Founded Multigraphs are a generalisation of some familiar structures. Figure 3.1 shows examples of these structures.

Figure 3.1 A taxonomy



The different kinds of structure are distinguished by restrictions on the arrows. All the structures are Well-Founded : following arrows backwards from any point always comes to a stop at a point with no incoming arrows (a root point).

For **Set** no arrows are allowed.

For **Lists** (plural) at most one arrow is allowed to arrive at each point and at most one arrow is allowed to leave it. For the more familiar **List** (singular) there is only one point with no arrows arriving at it (the root point).

Notes on algorithms: 3

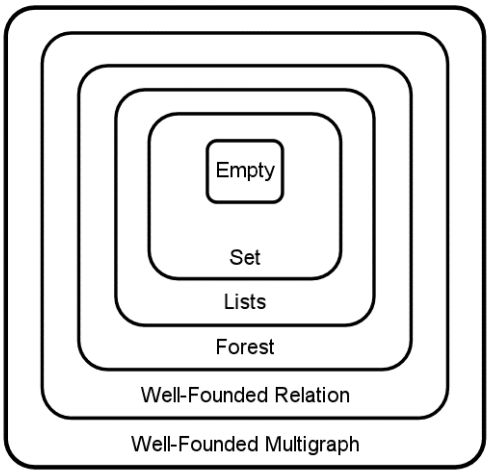
For **Forest** at most one arrow is allowed to arrive at each point, but there is no restriction on the number of arrows that can leave it. For the more familiar **Tree** there is only one point with no arrow arriving at it (the root point).

For **Well-Founded Relation** any two points are allowed to have at most one arrow between them.

For **Well-Founded Multigraph** there are no restrictions on the arrows apart from the need to be Well-Founded.

Each kind of structure is a special case of the kinds shown lower in the picture. To emphasise this, Figure 3.2 shows them as subclasses of the Well-Founded Multigraph class.

Figure 3.2 Classes and their subclasses



One subclass not described earlier is the **Empty** class. Its only member has no points and hence no arrows. It is a subclass of all the other classes but is sometimes excluded in discussions of the other kinds of structure.

Every structure belonging to these subclasses is also a Well-Founded Multigraph so the Induction Principle and Recursion Theorem apply to all of them. Obviously, in the case of Set neither theorem is of any practical use.

What is not shown in Figure 3.2 is that the class of all Well-Founded Multigraphs is a subclass of the class of all multigraphs.

Notes on algorithms: 3

4 Mathematics

4.1 Notation

First, Fig 4.1.1 lists some symbols that might not be familiar.

Figure 4.1.1 Some notation

iff	If and only if, alias $\Leftrightarrow$
$_{d}$	By definition
$=_{d}$	Is defined to be equal to, as in $x =_{d} a \cap b$
$\subseteq_{d}$	Is defined to be a subset of, as in $A \subseteq_{d} B$
:	Is defined to be a member of, as in $x : Y$
$\exists 1$	There is exactly one, as in $\exists 1\ x \bullet x = \varnothing$
$\langle$	Domain restriction, as in $X \langle F$ (It is defined at the end of this section)

Next, in the following subsections each function is represented by nothing more than a set of couples (alias 'ordered pairs'). A defined Domain and Codomain are not included as they are usually of little interest here and not always obvious in the case of functions defined by recursion.

Finally, we come to the notation for describing the internal structure of complex objects. The usual way to describe a multigraph would be to write :

A multigraph G is an ordered 4-tuple $G =_{d} (V, A, s, t)$ *where V is … .*

An example can be seen at [1]. This has a problem when another structure of the same kind is needed. Do you say

$H =_{d} (W, B, u, v)$ or $G' =_{d} (V', A', s', t')$?

In either case, do you (re)define *W*, *B*, etc, *V′*, *A′*, etc, or do you require the reader to guess correctly?

There is another problem. Some component parts may have their own internal structure. For instance, the function *s* would have a Domain of Definition, alias Definition Domain, usually given as Def(*s*). There are now two different ways of referring to the internal structure of *G*!

To avoid these problems this Note uses Scheurer's Feature Notation [2] to define structures and to prove results. Feature Notation is a systematic way to name the components of structures built from sets. The definition of *G* above would start by declaring that *G* is any member of a class of structures of a particular kind

G : Multigraph

It then defines and describes the component parts of *G* that distinguish one multigraph from another (indicated by the asterisks)

*	*GV* : Set	Points
*	*GA* : Set	Arrows
*	*Gs* : $GA \rightarrow GV$	*Gs*(*a*) is the point at the head of the arrow *a*
*	*Gt* : $GA \rightarrow GV$	*Gt*(*a*) is the point at the tail of the arrow *a*

Notes on algorithms: 3

Then the Definition Domain of the function *Gs* is given the name
GsDef Definition Domain of function *Gs*
and similarly for other sub-parts.

Note : A symbol $f : A \to B$ is to be interpreted in this Note as a constraint on the function *f* rather than making Domains and Codomains part of the structure of functions. The symbol declares that the function is defined for every member of *A* and no other, and that the function's graph is a set of couples that is a subset of $A \times B$.

What about *H* and *G′*? *H* can now be introduced by writing *H* : Multigraph. It has component parts *HV, HA, Hs, Ht, HsDef*, etc. *G′* can be introduced by *G′*: Multigraph with component parts *G′V, G′A, G′s, G′t, G′sDef*, etc. They are defined as above.

Figure 4.1.2 An example of Feature Notation

F : Function

*	*FGr* : Set	*F* 's graph, where $F(x) = y$ iff $\langle x, y \rangle \in FGr$
		F 's graph is a set of Couples
	FCond1 **.:** $\forall z : FGr \bullet \exists x, y : \text{Set} \bullet z = \langle x, y \rangle$	
		F is a function
	FCond2 **.:** $\forall x, y_1, y_2 : \text{Set} \bullet$ $[(\langle x, y_1 \rangle \in FGr) \wedge (\langle x, y_2 \rangle \in FGr)] \Rightarrow (y_1 = y_2)$	
	FDef $=_d \{ x \mid \langle x, y \rangle \in FGr \}$	*F* 's Definition Domain
	FRan $=_d \{ y \mid \langle x, y \rangle \in FGr \}$	*F* 's Range

---

Fig 4.1.2 is an example of Feature Notation. It gives the definition of functions that is used in this Note. *FCond1* and *FCond2* are conditions, alias constraints that must be true if *F* is to be a member of Function. The **.:** symbol is punctuation that separates the name from the formula. Some constraints are implicit. For instance, *FGr* is required to be a set.

If something behaves like a function but has a graph that is a proper class, that is, the graph is too big to be a set, then it is not a member of Function. For instance, the union operator $\cup$ behaves like a function but is defined for all sets so it cannot be a member of Function. Enderton [3] calls such a function a **function-class**.

Notes on algorithms: 3

To complete the description of possibly unfamiliar notation used in this Note here is the definition of the domain restriction operator. Its purpose is to restrict the definition domain of a function.

Figure 4.1.3 The Domain Restriction operator

$\langle$ Domain Restriction operator

Given any X : Set and any F : Function then

$X \langle F =_d F^*$ where F^* : Function and

$F^*Gr =_d \{ \langle x, y \rangle \mid \langle x, y \rangle \in FGr \ \wedge \ x \in X \}$

Note that X does not have to be a subset of $FDef$, though it often will be.

Notes on algorithms: 3

4.2 The definition of Well-Founded Multigraphs

The model of Well-Founded multigraphs used here is defined in two parts. First, figure 4.2.1 defines the class of all multigraphs, both Well-Founded and not Well-Founded.

Figure 4.2.1 Definition of the class Multigraph

G : Multigraph

* *GPnts* : Set — The points in the multigraph *G*

* *GArrs* : Set — The arrows in the multigraph *G*

* *GHead* : *GArrs* → *GPnts* — *GHead*(*a*) is the point at the head of arrow *a*

* *GTail* : *GArrs* → *GPnts* — *GTail*(*a*) is the point at the tail of arrow *a*

GArrivs : *GPnts* → Pow(*GArrs*) — Arrows arriving at *t*

$\forall\ t : GPnts \bullet GArrivs(t) =_d \{\ a : GArrs \mid GHead(a) = t\ \}$

GPreds : *GPnts* → Pow(*GPnts*) — Predecessors of *t*

$\forall\ t : GPnts \bullet$

$GPreds(t) =_d \{\ x : GPnts \mid \exists\ a : GArrs \bullet GTail(a) = x \wedge GHead(a) = t\ \}$

Recall that the features marked with an asterisk are those that distinguish one multigraph from another. For instance, two multigraphs with different points are obviously different multigraphs. The unmarked features are features derived from the marked features.

Second, figure 4.2.2 defines the subclass consisting of all Well-Founded multigraphs.

Figure 4.2.2 Definition of WFGraph, a subclass of Multigraph

G : WFGraph — alias Well-Founded Multigraph

GCond1 **.:** $G \in$ Multigraph — *G* is a member of Multigraph

G is Well-Founded

GCond2 **.:** $\forall\ S \subseteq_d GPnts \bullet S \neq \varnothing \Rightarrow (\ \exists\ x : S \bullet (S \cap GPreds(x)) = \varnothing\ )$

The only features are constraints that select some multigraphs and reject others, and indeed anything else.

Notes on algorithms: 3

4.3 Proof of the Induction Principle

The proof is based on Enderton's proof of the Induction Principle for Well-Founded relations [3].

Here is the detailed definition of the Induction Principle :

Figure 4.3.1 Transfinite Induction Principle for Well-Founded Multigraphs

---

Induction Principle

Given any G : WFGraph then

$$\forall A \subseteq_d GPnts \bullet [\ \forall t : GPnts \bullet GPreds(t) \subseteq A \Rightarrow t \in A\ ] \Rightarrow A = GPnts$$

To Prove :
The Transfinite Induction Principle given in Figure 4.3.1.

Proof :
Assume that G is a member of WFGraph.

if $GPnts = \varnothing$
then For any subset A of $GPnts$, $A = \varnothing = GPnts$

otherwise :
Assume that A is a subset of $GPnts$ with the special property that for every point t in $GPnts$
(1) $GPreds(t) \subseteq A \Rightarrow t \in A$

If A is a proper subset of $GPnts$ so that $(GPnts - A) =_d B$ is not empty
then by *GCond2* (that G is Well-Founded) there is a point $x : B$ such that
$(B \cap GPreds(x)) = \varnothing$

Thus either $GPreds(x) = \varnothing$ or every member of $GPreds(x)$ is a member of A.
In either case, $GPreds(x) \subseteq A$ so by (1) x is a member of A and hence not a member of B after all.

Consequently B has no members so $A = GPnts$.

Combine the cases $GPnts = \varnothing$ and $GPnts \neq \varnothing$, then generalise to give the final result

$$\forall G : \text{WFGraph} \bullet \forall A \subseteq_d GPnts \bullet [\ \forall t : GPnts \bullet GPreds(t) \subseteq A \Rightarrow t \in A\ ] \Rightarrow A = GPnts$$

QED

The Induction Principle can be used to prove that something is true of every point of G. Define A to be the subset of points for which the something is true. Now prove that the condition in the Induction Principle is true at all points. If this is done, then by the

Notes on algorithms: 3

Induction Principle *A* contains all the points of *G* so the something is true for all points of *G*.

A subset *A* need not be mentioned in a proof by Induction. Figure 4.3.2 shows the use of the Induction Principle in proofs.

Figure 4.3.2 Proof schema for Well-Founded Multigraphs

Proof schema for induction

Given any *G* : WFGraph and any Well Formed Formula *P*(*v*) then

$$[\ \forall\ t : GPnts \bullet\ [\ \forall\ x : GPreds(t) \bullet P(x)\ ]\ \Rightarrow\ P(t)\ ]$$
$$\Rightarrow$$
$$\forall\ t : GPnts \bullet P(t)$$

All that needs to be proved is the formula

$$[\ \forall\ x : GPreds(t) \bullet P(x)\ ]\ \Rightarrow\ P(t)$$

for every point *t* in *G* and the result follows.

Note : $P(t)$ must be proved unconditionally true when *t* has no predecessors, that is, when $GPreds(t) = \varnothing$.

Notes on algorithms: 3

4.4 Proof of the Recursion Theorem Schema

The proof is based on Enderton's proof of the Recursion Theorem Schema for Well-Founded relations [3].

Here is the detailed definition of the Recursion Theorem Schema. It is described as a schema as there is a separate theorem for each possible auxiliary function **H**.

Figure 4.4.1 Transfinite Recursion Theorem Schema for Well-Founded Multigraphs

---

Recursion Theorem Schema (for points)

Given any G : WFGraph and any ternary function-class **H** defined for all sets, then

$\exists 1\ F$: Function •
$FDef = GPnts$
$\wedge$
$\forall\ t : GPnts \bullet F(t) = \mathbf{H}(a, f, t)$
where
$a =_d GArrivs(t) \lhd GTail$ (Note domain restriction)
$f =_d GPreds(t) \lhd F$ (Note domain restriction)

The auxiliary function **H** is given the point t and two functions. These two functions provide information about the arrows and points other than t that are relevant to the definition of $F(t)$. From these two functions one can obtain :

aDef	The arrows arriving at t
aRan	The points these arrows come from, i.e t's predecessors
fDef	t's predecessors, again
f	The values of F at t's predecessors (and at no other points)

In addition one can use any values already associated with t, with the arrows arriving at t, and with t's predecessors (but not $F(t)$ of course).

Note that the auxiliary function **H** must be defined whatever the sets a, f, t might be. If this is difficult for arguments that will not arise in practice, such as when a is not a member of Function, then **H** can be defined to return some arbitrary constant such as 0 or ∅. Also, it must be obvious or else proved that **H** is a function. Whatever the sets a, f, t might be there must be only one result that satisfies the definition of **H**.

Recall that the class Function was defined in figure 4.1.2, and that for each function, F : Function, the only feature of F that distinguishes it from other functions is *FGr*.

To prove : The Transfinite Recursion Theorem Schema given in Figure 4.4.1.

Proof
Assume that G is a member of WFGraph.

if $GPnts = \varnothing$
then There is a function, $F^{\varnothing}$: Function, such that $F^{\varnothing}Gr = \varnothing$.
Now $F^{\varnothing}Def = \varnothing$ so

Notes on algorithms: 3

$F^{\varnothing}Def = GPnts \wedge \forall\ t : GPnts \bullet F^{\varnothing}(t) = \mathbf{H}(\ a, f, t\ )$ (vacuously)
Clearly $F^{\varnothing}$ exists and is unique.

otherwise :

0 Outline

Define partial solutions to the problem. Form the union of all partial solutions and call it *F*. Prove that *F* is a set, that it is a set of couples, that it is a function, that it is defined for all the points of *G* and nothing else, that it obeys the given definition, and that it is unique.

1 Define partial solutions to the problem; call them 'acceptable' functions

For the purposes of this proof, call a function, *V*, **acceptable** iff
(1.1) $V \in$ Function;
$VDef \subseteq GPnts$;
Whenever $x \in VDef$
then $GPreds(x) \subseteq VDef$
and $V(x) = \mathbf{H}(\ (\ GArrivs(x) \langle\ GTail\ ),\ (\ GPreds(x) \langle\ V\ ),\ x\ )$.

2 Show that any two acceptable functions agree at every point in common

For any two acceptable functions, V_1 and V_2, form the two sets
$A =_d \{\ x : (V_1Def \cap V_2Def) \mid V_1(x) = V_2(x)\ \}$ (Points where they agree)
$B =_d (V_1Def \cap V_2Def) - A$ (Points where they disagree)

If *B* is not empty then by *GCond2* (that *G* is Well-Founded) there is a point $x : B$ such that
$B \cap GPreds(x) = \varnothing$
V_1 and V_2 are acceptable so
$GPreds(x) \subseteq V_1Def$ and $GPreds(x) \subseteq V_2Def$
Therefore $GPreds(x) \subseteq A$

From the definition of *A*
$GPreds(x) \langle\ V_1 = GPreds(x) \langle\ V_2$
and so from the definition of acceptable functions at (1.1)
$V_1(x) = V_2(x)$.

The two functions agree at *x* so *x* is not a member of *B* after all; *B* has no members.
(2.1) Consequently V_1 and V_2 have the same value at all points in common.

3 Show that there is a set consisting of all acceptable functions

Define the formula
$\varphi(U, V) =_d [\ U \subseteq GPnts \wedge V$ is an acceptable function such that $VDef = U\ ]$

It follows from (2.1) that
$\varphi(U, V_1) \wedge \varphi(U, V_2) \Rightarrow V_1 = V_2$
so by a Replacement axiom there exists a set *J* such that

Notes on algorithms: 3

$V \in J \Leftrightarrow (\exists U \subseteq GPnts \bullet \varphi(U, V))$

(3.1) That is to say there is a set, *J*, that contains all the acceptable functions and nothing else.

Remark

φ defines a class, Φ, of couples that is functional in nature : for any set *u* there is at most one couple in Φ of the form $\langle u, v \rangle$ so Φ defines a function-class (a partial function as it happens). A Replacement axiom states that the image of any set *X* via Φ exists and is a set, even if Φ is a proper class. The image, Φ[[*X*]], of *X* via Φ is defined by

$v \in \Phi[[X]] \Leftrightarrow (\exists u : X \bullet \langle u, v \rangle \in \Phi)$

This definition allows for the possibility that Φ is a partial function, as it is here.

4 Show that the 'Union' of all acceptable functions is a function, call it *F*

The set *J* described at (3.1) consists of all the acceptable functions. Now construct the set *K* of the union of all their graphs

$K =_d \bigcup \{ VGr \mid V \in J \}$.

Each *VGr* is a set of couples so *K* is also a set of couples, and

(4.1) $\langle x, y \rangle \in K \Leftrightarrow V(x) = y$ for some acceptable function *V*

If $\langle x, y_1 \rangle \in K$ and $\langle x, y_2 \rangle \in K$ then $y_1 = y_2$ as by (2.1) any two acceptable functions agree wherever both are defined. Therefore *K* is a set that satisfies the Conditions for being the graph of a member of Function (see figure 4.1.2).

Define *F* to be this function, so

(4.2) *F* : Function such that $FGr =_d K$.

5 Show that *F* is acceptable

To prove that *F* is acceptable it is necessary to prove that all the clauses of the definition of acceptable functions at (1.1) are true for *F*.

From (4.2) *F* is a member of Function.

Consider any $x \in FDef$. By the construction of *F* given in (4) there exists some acceptable function, *V*, such that $x \in VDef$. By the definition of acceptable functions $VDef \subseteq GPnts$ so $x \in GPnts$. Generalising this, $FDef \subseteq GPnts$.

By the definition of acceptable functions and of *F*

$GPreds(x) \subseteq VDef \subseteq FDef$.

And to determine *F*(*x*) we have

	$V(x) = \mathbf{H}(\ (\ GArrivs(x) \langle\ GTail\ ),\ (\ GPreds(x) \langle\ V\ ),\ x\ )$	by acceptability of *V*
	$(\ GPreds(x) \langle\ V\ ) = (\ GPreds(x) \langle\ F\ )$	by (4.1) and (2.1)
	$V(x) = F(x)$	by (4.1)
so	$F(x) = \mathbf{H}(\ (\ GArrivs(x) \langle\ GTail\ ),\ (\ GPreds(x) \langle\ F\ ),\ x\ )$	

Putting all these together gives

$F \in$ Function;

$FDef \subseteq GPnts$;

Notes on algorithms: 3

Whenever $x \in FDef$
then $GPreds(x) \subseteq FDef$
and $F(x) = \mathbf{H}(\ (\ GArrivs(x) \langle\ GTail\ ),\ (\ GPreds(x) \langle\ F\ ),\ x\ )$

(5.1) From the definition at (1.1) conclude that F is an acceptable function.

6 Show that *FDef = GPnts*

In outline, if F is not defined at some points then form an extended function that includes one of these points. Show that this extended function is acceptable and so F was defined at that point after all.

Assume some points of *GPnts* are undefined by F so
$B =_d GPnts - FDef$
is not empty, then by *GCond2* (that G is Well-Founded) there is a point $x : B$ such that
$(\ B \cap GPreds(x)\ ) = \varnothing$
Either x has no predecessors or all its predecessors are members of *FDef*. Either way,
$GPreds(x) \subseteq FDef$
while $x \notin FDef$.

Let y be the unique value
$y =_d \mathbf{H}(\ (\ GArrivs(x) \langle\ GTail\ ),\ (\ GPreds(x) \langle\ F\ ),\ x\ ).$
Form the set
$FGr \cup \{\ \langle x, y\rangle\ \}.$
As this is the union of two sets of couples and $x \notin FDef$ it is a set that satisfies the Conditions for being the graph of a member of Function (see figure 4.1.2). Define
$\tilde{V}$: Function where $\tilde{V}Gr =_d FGr \cup \{\ \langle x, y\rangle\ \}.$

Now show that $\tilde{V}$ is an acceptable function as defined at (1.1).

By construction $\tilde{V} \in$ Function and $\tilde{V}Def \subseteq GPnts$.

Also by construction $FDef \subseteq \tilde{V}Def$.

Given any $t : \tilde{V}Def$ then

if $t \in FDef$
then By (5.1) F is acceptable, so by (1.1)
$GPreds(t) \subseteq FDef \subseteq \tilde{V}Def$
so
$F(t) = \mathbf{H}(\ (\ GArrivs(t) \langle\ GTail\ ),\ (\ GPreds(t) \langle\ F\ ),\ t\ )$
but
$(\ GPreds(t) \langle\ F\ ) = (\ GPreds(t) \langle\ \tilde{V}\ )$
so
$\tilde{V}(t) = F(t) = \mathbf{H}(\ (\ GArrivs(t) \langle\ GTail\ ),\ (\ GPreds(t) \langle\ \tilde{V}\ ),\ t\ )$

else $t \notin FDef$
$t = x$ by the definition of x
$GPreds(t) \subseteq FDef \subseteq \tilde{V}Def$ by the definition of x and of $\tilde{V}$
By construction

Notes on algorithms: 3

$\tilde{V}(t) = \tilde{V}(x) = y = \mathbf{H}(\ (\ GArrivs(t) \langle\ GTail\ ),\ (\ GPreds(t) \langle\ F\ ),\ t\ )$
$(\ GPreds(t) \langle\ F\ ) = (\ GPreds(t) \langle\ \tilde{V}\ )$
so
$\tilde{V}(t) = \mathbf{H}(\ (\ GArrivs(t) \langle\ GTail\ ),\ (\ GPreds(t) \langle\ \tilde{V}\ ),\ t\ )$

Putting these together gives
$\tilde{V} \in$ Function;
$\tilde{V}Def \subseteq GPnts$;
Whenever $t \in \tilde{V}Def$
then $GPreds(t) \subseteq \tilde{V}Def$
and $\tilde{V}(t) = \mathbf{H}(\ (\ GArrivs(t) \langle\ GTail\ ),\ (\ GPreds(t) \langle\ \tilde{V}\ ),\ t\ )$
so by (1.1) $\tilde{V}$ is acceptable.

By the construction of $\tilde{V}$, $x \in \tilde{V}Def$ for the acceptable function $\tilde{V}$ so, by the definition of F at (4.2), $x \in FDef$ and so is not a member of B after all. Conclude that $B = \varnothing$ and so
$FDef = GPnts$.

7 Show that *F* is unique

Apply Transfinite Induction.

Assume that F_1, F_2 : Function both obey the definition in figure 4.4.1.

Assume that $t \in GPnts$ and that $\forall\ x : GPreds(t) \bullet F_1(x) = F_2(x)$.

Now, by the definition,
$F_1(t) = \mathbf{H}(\ (\ GArrivs(t) \langle\ GTail\ ),\ (\ GPreds(t) \langle\ F_1\ ),\ t\ )$
$F_2\ (t) = \mathbf{H}(\ (\ GArrivs(t) \langle\ GTail\ ),\ (\ GPreds(t) \langle\ F_2\ ),\ t\ )$
But, by assumption
$(\ GPreds(t) \langle\ F_1\ )\ =\ (\ GPreds(t) \langle\ F_2\ )$ (Note : $\varnothing \langle\ F_1 = \varnothing \langle\ F_2$)
so
$F_1(t) = F_2\ (t)$

Generalising
$\forall\ t : GPnts \bullet\ (\ \forall\ x : GPreds(t) \bullet F_1\ (x) = F_2(x)\ )\ \Rightarrow\ F_1(t) = F_2(t)$
Therefore, by Induction,
$F_1 = F_2$
so F is unique.

8 Finish

Conclude that
F is a member of Function, is unique, and
$FDef = GPnts \wedge\ \forall\ t : GPnts \bullet F(t) = \mathbf{H}(\ a, f, t\ )$
where
$a =_d GArrivs(t) \langle\ GTail$
$f =_d GPreds(t) \langle\ F$

Notes on algorithms: 3

Combine the cases $GPnts = \varnothing$ and $GPnts \neq \varnothing$, then generalise to give the final result

Given any ternary function-class, **H**, defined for all sets, then

$\forall\ G$: WFGraph •

$\exists 1\ F$: Function • $FDef = GPnts \wedge \forall\ t : GPnts \bullet F(t) =$ **H**$(\ a, f, t\ )$

where

$a =_{\mathrm{d}} GArrivs(t) \rangle\ GTail$

$f =_{\mathrm{d}} GPreds(t) \rangle\ F$

QED

The proof applies whether there are a finite number of points and arrows or an infinite number, for any size of infinity. That is why the title in figure 4.4.1 uses the word ‘Transfinite’.

It is interesting to compare this proof with Enderton’s proof for Well-Founded Relations. The notation used is different, obviously, but allowing multiple links between two points *x* and *y* makes only two differences. First, the links have to be represented by separate arrows whereas for a relation they can be represented by the simple couple ⟨x, y⟩. Second, the auxiliary function **H** is a ternary function instead of a binary function. This is necessary because more information must be passed to **H**.

Notes on algorithms: 3

4.5 Proof of a Reversed Arrows theorem

Suppose the arrows in a multigraph are reversed. That is, for each arrow the head point becomes the tail point and the tail point becomes the head point. Clearly the result is also a multigraph.

If the original multigraph was Well-Founded then the result is not necessarily also Well-Founded, but there are cases where it will be. The one that is easiest to describe and is more likely to be useful in practice is the case where the multigraph has a finite number of points. This section proves that the result of reversing the arrows in a Well-Founded Multigraph with a finite number of points is also Well-Founded.

Start by defining the Reverse function. Notice that the definition of the result specifies just those features of a member of Multigraph that distinguish one member from another. Reverse is not a member of Function as it has a graph that is a proper class.

Figure 4.5.1 The Reverse function on members of Multigraph

Reverse

Given any G : Multigraph

Reverse(G) $=_d$ G^* where G^* : Multigraph and

$G^*Pnts =_d GPnts$
$G^*Arrs =_d GArrs$
$\forall\ a : G^*Arrs \bullet G^*Head(a) =_d GTail(a)$
$\forall\ a : G^*Arrs \bullet G^*Tail(a) =_d GHead(a)$

For convenience define a further feature of multigraphs; it continues the definition in Figure 4.2.1 :

Figure 4.5.2 Successors of point t

G : Multigraph Continued

$GSuccs : GPnts \to Pow(GPnts)$ Points at the heads of arrows leaving t
$\forall\ t : GPnts \bullet$
$GSuccs\ (t) =_d \{ x : GPnts \mid \exists\ a : GArrs \bullet GTail(a) = t \wedge GHead(a) = x\}$

Notes on algorithms: 3

Now state the theorem that will be proved :

Figure 4.5.3 Theorem for the finite case

Reversing theorem for members of WFGraph with a finite number of points

Given any *G* : WFGraph
if *GPnts* is finite
then Reverse(*G*) ∈ WFGraph

To Prove : The theorem given in Figure 4.5.3.

Proof

Assume that *G* is a member of WFGraph and that *GPnts* is Finite.

if $GPnts = \varnothing$
then Reverse(*G*) is a member of WFGraph as
WFGraph's Condition 1 is true by definition of *G* and
Condition 2 is true vacuously (see figure 4.2.2)

otherwise :
Assume that *S* is a non-empty subset of *GPnts*, so
$S \subseteq_d GPnts \wedge S \neq \varnothing$.

Form a sequence of points in *S* starting at any point in *S*. At each point, *t*, in the sequence the next point, *t′*, if any, is given as follows : If *t* has no successors in *S* then *t* is the last point in the sequence; Otherwise, *t′* is some successor of *t* that is in *S*. So

(1) **if** $(S \cap GSuccs(t)) = \varnothing$
then *t* is the last point in the sequence
else $t' \in (S \cap GSuccs(t))$

Now *t′* is a point that does not already occur in the sequence. For if not, there would be a subset of *S*, and hence a subset of *GPnts*, in which each point has a predecessor that is in the subset. This would make *GCond2* false which cannot be as *G* is Well-Founded.

GPnts is Finite so *S* is Finite. As each point of *S* occurs at most once in the sequence the sequence cannot have more points than there are in *S* so the length of the sequence is also Finite. Consequently there is a last point, *e*, in the sequence. So from (1)
$(S \cap GSuccs(e)) = \varnothing$
Hence
(2) $\exists x : S \bullet (S \cap GSuccs(x)) = \varnothing$

Therefore, generalising (2),
(3) $\forall S \subseteq_d GPnts \bullet S \neq \varnothing \Rightarrow (\exists x : S \bullet (S \cap GSuccs(x)) = \varnothing)$

Form *G** : Multigraph, the Reverse of *G* :
$G^* =_d$ Reverse(*G*)

Notes on algorithms: 3

Observe that *G*Pnts* = *GPnts* so any point *t* in *G*Pnts* is also in *GPnts*. Observe also that the successors of *t* in *GPnts* are the predecessors of *t* in *G*Pnts*. These are immediate from the definitions of Reverse, of *GSuccs*, and of *G*Preds*. So

(4) $GPnts = G^*Pnts$ and
$\forall t : G^*Pnts \bullet GSuccs(t) = G^*Preds(t)$

Use (4) to replace *GPnts* and *GSuccs* in (3) by *G*Pnts* and *G*Preds* respectively giving
$\forall S \subseteq_d G^*Pnts \bullet S \neq \varnothing \Rightarrow (\exists x : S \bullet (S \cap G^*Preds(x)) = \varnothing)$

But this is WFGraph's Condition 2 (see figure 4.2.2). $G^* \in$ Multigraph by the definition of Reverse, so satisfying WFGraph's Condition 1. Therefore $G^* \in$ WFGraph. That is,

(5) $\text{Reverse}(G) \in \text{WFGraph}$

Combine the cases $GPnts = \varnothing$ and $GPnts \neq \varnothing$, then generalise to give the final result
$\forall G : \text{WFGraph} \bullet GPnts \text{ is Finite} \Rightarrow \text{Reverse}(G) \in \text{WFGraph}$

QED

Notes on algorithms: 3

5 References

[1] Wikipedia entry for a directed multigraph [as at September 2016].
<https://en.wikipedia.org/wiki/Multigraph#Directed_multigraph_.28edges_with_own_identity.29>

[2] Scheurer, T [1994]. Foundations of Computing : System development with set theory and logic.
Addison-Wesley.

[3] Enderton, H B [1977]. Elements of set Theory.
Academic Press Inc.