The *Specification* Track

?
←
→
C
I
S

9 An Input/Output Specification

This hyperbook is mainly about systems. We call something a *system* when we are interested in its behavior—that is, what is happening while the system is operating or being executed. These systems are also called *concurrent systems*, because we usually consider them to be composed of multiple components that may be doing things at the same time. What this hyperbook calls a system has also been called a *reactive system* because such systems usually interact with their environment throughout the course of their execution.

?
←
→
C
I
S

Sometimes we are interested in conceptually simpler systems that interact with their environment only by taking an input when they are started and then producing an output and stopping. We can specify such a system by an *Input/Output Specification* (*I/O Spec* for short) that describes the relation between its input and its output.

An I/O Spec may consist of a function F whose domain is the set of possible inputs, where $F[x]$ is the output that should be produced by the input x in DOMAIN F. It may also be more convenient to write it as an operator Op rather than a function, where $Op(x)$ is the output that should be produced by the possible input x. If more than one possible output is permitted for a single input, then an I/O Spec consists of a relation. The relation may be described by a set R of ordered pairs, where $\langle x, y \rangle$ is an element of R iff x is a legal input that allows y as an output. It may be more convenient to describe the relation by a Boolean-valued operator Op, where $Op(x, y)$ is true iff y is an allowed output for input x.

Often, the I/O system we are specifying takes as input an operation and a collection of arguments. In that case, we may specify each possible operation separately.

We can write I/O specifications as well as system specifications in TLA$^+$. I/O specs are chararcterized by not using variables. The only parameters of an I/O spec are constants. I/O specs are conceptually simpler than system specs, requiring only ordinary math without any temporal logic. They provide a good way to familiarize yourself with ordinary math, without the complications of dealing with behaviors.

An I/O spec is ordinary math, which means that if you can explain it mathematically then you can easily write it in TLA$^+$. However, you might not be able to check the spec because TLC can't evaluate the math or can't evaluate it efficiently enough to check a large enough model. In this section, I illustrate how to write an I/O spec that TLC can handle. In effect, I will show you how to use TLA$^+$ as a programming language for mathematics.

9.1 The Example

The I/O spec described here is a real-life example. I am a member of a committee that chooses the recipient of an award. Before the committee meets, each member ranks the candidates, and these individual rankings are combined into a single ranking. This combined ranking is meant to serve only as an indication of the committee members' views; it does not determine the recipient. However, it inevitably influences the outcome.

?
←
→
C
I
S

A few hundred years of research into elections has shown that there is no perfect method of choosing a winner. A result known as Arrow's Theorem proves that no voting system can satisfy a small number of clearly desirable properties. Reasonable systems can produce significantly different outcomes. I decided that it would be better for the committee to use more than one system for ranking the candidates.

The obvious thing to do would have been to write a little program to compute each ranking. For some voting systems, computing a ranking can be tricky. When faced with the task of writing a tricky piece of code, it's a good idea to first write and debug a PlusCal algorithm that computes the desired result and then hand-translate the PlusCal code into the programming language. To get the PlusCal algorithm right, you can specify the ranking with a TLA$^+$ operator and have the algorithm compare the value it computes with the value of the operator. You can then check the algorithm with TLC on a large collection of small instances.

This is a procedure I use regularly when writing code that is at all tricky and that I want to be correct. But, getting this program correct was not important enough to warrant spending that much time. Even if I didn't do it right, it would still have taken me more time to write the program than I felt like spending. However, there are only about 10 members of the committee and about the same number of candidates. This meant that TLC could compute the rankings directly from their high-level TLA$^+$ specifications. This seemed like it would be easier and more fun than writing a program to do it, since I prefer writing TLA$^+$ specs to programming. So, that's what I did.

It turned out that the simplest, most obviously correct specifications could not be executed efficiently enough by TLC even on the committee's small elections. The thing to do in such a case is to write two versions of some definitions: one that is simpler and more obviously correct, and another that TLC can evaluate more efficiently. You can then use TLC to check the equivalence of the two definitions on small examples. I was confident enough in the correctness of the more efficient definitions not to bother doing this. However, I will do it here to show you how. It's what I would have done if getting the right result had been more important.

The process of writing two versions of an operator's definition—a simpler version and one that TLC can evaluate more efficiently—tends to be more useful for system specifications than for I/O specifications. You use the simpler definition to make sure that you are specifying the system you intend to, and then have TLC use the second definition for checking the specification. The voting example will show how this sort of TLA$^+$ "programming" is done.

I decided to specify two ranking systems, known as the Borda and Condorcet systems. The Borda system assigns a score to each candidate, and candidates are ranked by their score. I wanted the output of this system to be a list of candidates and their scores, sorted by score. This requires specifying sorting. The standard *TLC* module defines a specfication of sorting that is implemented efficiently by TLC. However, the operator it defines is not exactly the one we need, and sorting provides a nice example of an I/O specification. So, let's specify it from scratch.

?
←
→
C
I
S

9.2 Sorting

We want to specify what it means to sort a set of elements. For simplicity, let's assume that each element is a record$^\square$ with a *key* component, and the elements are to be sorted in increasing order of their *key* value.

Sorting a set means arranging its elements in a list. A sorting of a set S is therefore a list of elements of S that contains each element exactly once and is sorted according to the elements' *key* values. To describe this precisely, we must define:

- What a list of elements of S is.

- What it means for such a list to contain each element of S exactly once.

- What it means for the list to be sorted according to elements' *key* values.

The mathematical representation of a list is a finite sequence$^\square$, also known as a tuple. In TLA$^+$, the sequence $\langle e_1, \ldots, e_n \rangle$ is defined to be the function$^\square$ f with domain $1 \,..\, n$ such that $f[i]$ equals e_i, for each i in $1 \,..\, n$. This sequence represents a list of elements of S iff each e_i is an element of S.

TLA$^+$ defines $[1 \,..\, n \rightarrow S]$ to be the set of all functions f with domain $1 \,..\, n$ such that $f[i]$ is in S for all i in $1 \,..\, n$. Therefore, $[1 \,..\, n \rightarrow S]$ represents the set of all lists of length n of elements in S. Such a list contains each element of S exactly once iff n equals the number of elements of S and every element of S appears in the list. The standard *FiniteSets* module defines $\mathit{Cardinality}(S)$ to be the cardinality$^\square$ of a finite set S. A sequence *seq* is therefore a sorting of S iff it satisfies the following conditions:

1. *seq* is an element of $[1 \,..\, \mathit{Cardinality}(S) \rightarrow S]$

2. Each element of S equals $\mathit{seq}[i]$ for some i in $1 \,..\, \mathit{Cardinality}(S)$.

3. *seq* is sorted, which means: For each i and j in $1 \,..\, \mathit{Cardinality}(S)$, if $i < j$ then $\mathit{seq}[i] \leq \mathit{seq}[j]$.

?
←
→
C
I
S

It is obvious how to write condition 1 as a mathematical formula. Conditions 2 and 3 are also easily expressed as formulas:

S2. $\forall\, s \in S \,:\, \exists\, i \in 1 \,..\, \mathit{Cardinality}(S) \,:\, \mathit{seq}[i] = s$

S3. $\forall\, i, j \in 1 \,..\, \mathit{Cardinality}(S) \,:\, (i < j) \Rightarrow (\mathit{seq}[i].\mathit{key} \leq \mathit{seq}[j].\mathit{key})$

Since multiple elements of S can contain the same *key* value, there is no unique way to sort the elements of S. We specify sorting by defining the set $\mathit{Sortings}(S)$ of all sortings of the elements of S. This set is the set of all elements *seq* of $[1 \,..\, \mathit{Cardinality}(S) \rightarrow S]$ satisfying S2 and S3. We write the set of all elements x in a set T satisfying a condition $P(x)$ as $\{x \in T : P(x)\}$. Therefore, we can define $\mathit{Sorting}(S)$ to equal

$$\begin{array}{l}\{\mathit{seq} \in [1 \,..\, \mathit{Cardinality}(S) \rightarrow S] \,:\, \\ \quad \land\ \forall\, s \in S \,:\, \exists\, i \in 1 \,..\, \mathit{Cardinality}(S) \,:\, \mathit{seq}[i] = s \\ \quad \land\ \forall\, i, j \in 1 \,..\, \mathit{Cardinality}(S) \,:\, (i < j) \Rightarrow (\mathit{seq}[i].\mathit{key} \leq \mathit{seq}[j].\mathit{key})\,\}\end{array}$$

This is a perfectly fine definition of $\mathit{Sortings}(S)$, but we can express condition 2 in a different way that I find a little more elegant. The set of all elements in the sequence *seq* can be written as

$$\{\mathit{seq}[i] \,:\, i \in 1 \,..\, \mathit{Cardinality}(S)\}$$

In general, $\{f(x) \,:\, x \in T\}$ is the set of all elements of the form $f(x)$ for x an element of the set T. Condition 2 asserts that this set equals S, so it can be written as:

S2a. $S = \{\mathit{seq}[i] \,:\, i \in 1 \,..\, \mathit{Cardinality}(S)\}$

This condition is partially redundant. Condition 1 implies that $\mathit{seq}[i]$ is in S, for all i in $1 \,..\, \mathit{Cardinality}(S)$, which implies that $\{\mathit{seq}[i] \,:\, i \in 1 \,..\, \mathit{Cardinality}(S)\}$ is a subset of S. Therefore, the two sets are equal iff S is a subset of $\{\mathit{seq}[i] \,:\, i \in 1 \,..\, \mathit{Cardinality}(S)\}$. We can thus replace S2a by:

S2b. $S \subseteq \{\mathit{seq}[i] \,:\, i \in 1 \,..\, \mathit{Cardinality}(S)\}$

This leads to the definition:

$$\begin{array}{l}\mathit{Sortings}(S) \triangleq \\ \quad \{\mathit{seq} \in [1 \,..\, \mathit{Cardinality}(S) \rightarrow S] : \\ \qquad \land\ S \subseteq \{\mathit{seq}[i] : i \in 1 \,..\, \mathit{Cardinality}(S)\} \\ \qquad \land\ \forall\, i,\, j \in 1 \,..\, \mathit{Cardinality}(S) : (i < j) \Rightarrow (\mathit{seq}[i].\mathit{key} \leq \mathit{seq}[j].\mathit{key})\}\end{array}$$

The expression $1 \,..\, \mathit{Cardinality}(S)$ occurs three times in the definition. We can use a LET / IN□ construct to make the definition easier to read by locally defining an identifier to equal that expression. Let's use the identifier D (for *domain*) and write:

$$
\begin{array}{l}
\mathit{Sortings}(S) \triangleq \\
\quad \text{LET } D \triangleq 1 \,..\, \mathit{Cardinality}(S) \\
\quad \text{IN } \;\; \{\mathit{seq} \in [D \to S] : \\
\qquad\qquad \land\; S \subseteq \{\mathit{seq}[i] : i \in D\} \\
\qquad\qquad \land\; \forall\, i, j \in D : (i < j) \Rightarrow (\mathit{seq}[i].\mathit{key} \leq \mathit{seq}[j].\mathit{key})\}
\end{array}
$$

?
←
→
C
I
S

Let's start creating our specification and test this definition. In the Toolbox, open a new specification named *CandidateRanking*. We will need the standard modules *Integers*, *Sequences*, and *FiniteSets* (for the definition of *Cardinality*), so begin the module with:

EXTENDS *Integers*, *Sequences*, *FiniteSets*

```
EXTENDS Integers, Sequences, FiniteSets
```

Now insert the definition of *Sortings*(*S*). Let's check the definition by evaluating the operator *Sortings* with some arguments. First, create a new model. The model will open on the *Model Checking Results*. We enter the expression we want to evaluate in the *Expression* field of the Evaluate Constant Expression section.

The argument of *Sortings* should be a set of records□ with *key* fields. Let's start simply, with the set containing the a single element $[\mathit{key} \mapsto 42]$, which is a record with only a *key* field having value 42. Enter the expression

$$\mathit{Sortings}(\{[\mathit{key} \mapsto 42]\})$$

```
Sortings({[key |-> 42]})
```

and run TLC to evaluate it. TLC gives the value $\{\langle [\mathit{key} \mapsto 42] \rangle\}$, a set containing the single sequence having that record as its one element.

Question 9.1 What does *Sortings* applied to the empty set of records equal? Let TLC check your answer.

Let's test the definition on a more interesting set of records. To allow different records with the same *key* value, the records must have at least one additional field. We want to test our definitions on sets with different numbers of elements, so let's define *TestSet*(*n*) to be the set of all records having a *key* field in the set $1 \,..\, n$ and a *val* field in the set {"x", "y"}. (There are $2n$ such records.)

$$\mathit{TestSet}(n) \;\triangleq\; [\mathit{key} : 1 \,..\, n,\; \mathit{val} : \{\text{“x”}, \text{“y”}\}]$$

```
TestSet(n) == [key : 1..n, val : {"x", "y"}]
```

Since this definition is just for testing, we don't want to make it part of our specification. Instead, copy the definition into the Additional Definitions section of the model's Advanced Options page. Now have TLC evaluate *Sortings*(*TestSet*(3)). It reports a set of 8 sequences, including

$$
\begin{array}{l}
\langle\; [\mathit{key} \mapsto 1,\; \mathit{val} \mapsto \text{“y”}],\; [\mathit{key} \mapsto 1,\; \mathit{val} \mapsto \text{“x”}],\; [\mathit{key} \mapsto 2,\; \mathit{val} \mapsto \text{“x”}] \\
\quad [\mathit{key} \mapsto 2,\; \mathit{val} \mapsto \text{“y”}],\; [\mathit{key} \mapsto 3,\; \mathit{val} \mapsto \text{“y”}],\; [\mathit{key} \mapsto 3,\; \mathit{val} \mapsto \text{“x”}] \;\rangle
\end{array}
$$

You can read TLC's output more easily by copying it (click on the *Value* region and type Control+A) and pasting it into a text editor.

TLC should have taken just a second or two to perform the evaluation. Now have it evaluate $Sorting(TestSet(4))$. It probably takes about 30 seconds. Now try $Sorting(TestSet(5))$. It will take TLC several hours. (You can hit *Cancel* if you get bored waiting.) What's going on?

To evaluate a set of the form $\{x \in T \,:\, P(x)\}$, TLC enumerates all the elements of T and keeps only those elements x satisfying $P(x)$. The number of elements in a set of the form $[D \rightarrow R]$ is $Cardinality(D)^{Cardinality(R)}$. The set $TestSet(n)$ contains $2n$ elements. To evaluate $Sorting$ on it, TLC must enumerate a set of $2n^{2n}$ functions. For $n = 5$, that's 10 billion functions—600 times more than for $n = 4$. Although it's not a large number of elements for computers these days, TLC is quite slow at such a task compared to a program written especially for that task. For the $n = 4$ case, the 30 seconds it takes TLC to enumerate the 16.7 million functions to find the sortings of this 8-element set is perhaps 50 times longer than it would take a hand-coded program to perform that same calculation.

?
←
→
C
I
S

For candidate ranking, we need to define an operator $SortSet$ so that $SortSet(S)$ is some single sorting of S. The simplest way to define it is with the CHOOSE□ operator:

$$SortSet(S) \;\triangleq\; \text{CHOOSE } seq \in Sorting(S) \,:\, \text{TRUE}$$

This defines S to be an arbitrarily chosen element of $Sorted(S)$. While this definition is fine in principle, in practice it won't do. With this definition, computing $SortSet(S)$ requires computing $Sortings(S)$, which for a set of 10 candidates would take billions of years. We need a definition that TLC can compute more efficiently.

Here is the simplest practical algorithm I know for computing a sorting seq of a set S:

- Let T equal S and i equal 1.
- While T is nonempty:
  - Set $seq[i]$ to be any element x of T with a minimal value of $x.key$.
  - Increment i by 1.
  - Remove x from the set T.

It's easy to write this algorithm in PlusCal. However, we're writing a definition, not an algorithm. The value computed by an iterative computation like this can be defined inductively□. In TLA$^+$, you describe the value inductively using a recursive operator□ or recursive function□ definition. This will seem quite natural if you've used a functional programming language.

We define $SortSet(S)$ inductively by:

- Defining $SortSet(\{\,\})$ to equal $\langle\,\rangle$, the empty sequence.

- Defining *SortSet*(S), for a nonempty set S, to equal the sequence that begins with an element s of S having a minimal value of $s.key$ and ends with *SortSet*$(S \setminus \{s\})$, where $S \setminus \{s\}$ is the set obtained from S by removing the element s.

To write this definition in TLA$^+$, we use the CHOOSE operator□ as follows to describe an element of S with minimal value of $s.key$:

?
←
→
C
I
S

$$\text{CHOOSE } ss \in S : \forall t \in S : ss.key \leq t.key$$

The complete definition is as follows, where $\circ$ is sequence concatenation□ and $\setminus$ is the set-difference operator:

ASCII version

```
RECURSIVE SortSet(_)
SortSet(S) ≜
  IF S = {} THEN ⟨ ⟩
            ELSE LET s ≜ CHOOSE ss ∈ S : ∀ t ∈ S : ss.key ≤ t.key
                 IN  ⟨s⟩ ∘ SortSet(S \ {s})
```

Since the definition of *SortSet* is recursive, it is preceded by a RECURSIVE declaration.

Copy the definition of *SortSet* into module *CandidateRanking*, and let's test it. Have TLC evaluate *SortSet*(*TestSet*(3)). It should give the value

$$\begin{array}{l}\langle\ [key \mapsto 1,\ val \mapsto \text{“x”}],\ [key \mapsto 1,\ val \mapsto \text{“y”}],\ [key \mapsto 2,\ val \mapsto \text{“x”}] \\ \quad [key \mapsto 2,\ val \mapsto \text{“y”}],\ [key \mapsto 3,\ val \mapsto \text{“x”}],\ [key \mapsto 3,\ val \mapsto \text{“y”}]\ \rangle\end{array}$$

We can do better than just testing the definition of *SortSet* on a few individual arguments. The value of *SortSet*(S) is correct iff it is an element of *Sortings*(S). We can check that *SortSet*(S) is correct for all sets S in some set Σ of sets by having TLC check that the following formula equals TRUE:

$$\forall S \in \Sigma : SortSet(S) \in Sortings(S)$$

Try this for Σ equal to the set of all subsets of *TestSet*(3). Using the TLA$^+$ SUBSET operator□, this set is written SUBSET *TestSet*(3). TLC quickly reports the value TRUE.

Checking it on all 2^6 subsets of this 6-element set gives us quite a bit of confidence in the correctness of the definition of *SortSet*. Just to be safe, let's check it on the larger set SUBSET *TestSet*(4). Just evaluating *Sortings* on *TestSet*(4) $[key : 1 \,..\, 4,\ val : \{\text{“x”}, \text{“y”}\}]$ took 30 seconds, and now we must evaluate the expression *SortSet*$(S) \in$ *Sortings*(S) for all subsets S of this set (which includes *TestSet*(4) itself). We expect this to take quite a bit of time, but let's start it and see what happens.

Surprise! TLC reports the answer immediately. Try it with *TestSet*(5). Still an immediate answer. Try it on *TestSet*(6), *TestSet*(7), and so on until it starts taking more than a couple of seconds. On my computer, only for *TestSet*(9) does it take 15 seconds. This is a set with 18 elements. To compute its set of sortings, TLC must enumerate a set containing 18^{18} elements—a task that should take it billions of years. What's going on?

?
←
→
C
I
S

You should always be suspicious of success when using TLC, especially if it reports success more quickly than you expect. You should make sure that TLC is checking what you think it is. When TLC reports no error, it's a good idea to insert an error and make sure that TLC catches it. For example, you can replace $\leq$ by $\geq$ in the definition of *SortSet*. In this case, you will find that TLC really is checking the correctness of *SortSet* as we expect it to.

TLC is so fast because it is evaluating $SortSet(S) \in Sortings(S)$ without computing the set $Sortings(S)$. To compute all the elements of a set of the form $\{x \in T \ : \ P(x)\}$, TLC must enumerate the elements of T. However, the expression $e \in \{x \in T \ : \ P(x)\}$, is equivalent to $(e \in T) \land P(e)$, and TLC usually does not have to enumerate the elements of a set T to determine if $e \in T$ is true. (Indeed, TLC can evaluate $e \in T$ for some infinite sets T such as *Nat*.) If T is the set $TestSet(9)$, then $e \in T$ is true iff e is a record with only *key* and *val* fields, $e.key$ is in $1 \,..\, 9$, and $e.val$ equals "x" or "y". TLC can check this quickly, and the evaluation of $SortSet(S) \in Sortings(S)$ for each S takes almost no time. TLC takes 15 seconds to evaluate the entire formula because it must perform that evaluation for each of the 2^{18} subsets S of $TestSet(9)$.

A little thought reveals that the time needed to compute $SortSet(S)$ for an n-element set S is on the order of n^2. There are faster sorting algorithms. However, a bit of testing shows that TLC can perform the computation essentially instantaneously for a 100-element set, so this definition of *SortSet* is good enough for us.

We have defined sorting a set of records according to their *key* values, which we have assumed to be integers. We could just as easily have defined sorting of a set according to an arbitrary relation by using higher-order operators$^{\square}$.

Let's define an operator *GeneralSortings* so that, if *KeyLeq* is defined by

$$KeyLeq(s, t) \ \triangleq \ s.key \leq t.key$$

then $GeneralSortings(S, KeyLeq)$ equals $Sortings(S)$, for any set S. This is easily done by modifying the definition of *Sortings* to be:

$$\begin{array}{l}
GeneralSortings(S,\ LEQ(\_,\ \_)) \ \triangleq \\
\quad \text{LET } D \triangleq 1 \,..\, Cardinality(S) \\
\quad \text{IN } \ \{seq \in [D \rightarrow S] : \ \land\ S \subseteq \{seq[i] : i \in D\} \\
\qquad\qquad\qquad\qquad\qquad\quad\ \land\ \forall\, i,\ j \in D : (i < j) \Rightarrow LEQ(seq[i],\ seq[j])\}
\end{array}$$

Copy the source of this definition into the *CandidateRanking* module and let's make sure that $Sortings(S)$ does equal $GeneralSortings(S,\ KeyLeq)$, when *KeyLeq* is defined as above. Check that this is true for all subsets of $TestSet(3)$ by letting TLC evaluate:

$$
\begin{array}{l}
\textsc{let}\ \ \mathit{KeyLeq}(s,\, t)\ \triangleq\ s.\mathit{key} \leq t.\mathit{key} \\
\textsc{in}\ \ \ \forall\, S \in \textsc{subset}\ \mathit{TestSet}(3) : \mathit{Sortings}(S) = \mathit{GeneralSortings}(S,\, \mathit{KeyLeq})
\end{array}
$$

We can also write this expression without explicitly defining *KeyLeq* by using a LAMBDA expression□:

$$
\begin{array}{l}
\forall\, S \in \textsc{subset}\ \mathit{TestSet}(3) : \\
\quad \mathit{Sortings}(S) = \mathit{GeneralSortings}(S,\ \textsc{lambda}\ s,\, t : s.\mathit{key} \leq t.\mathit{key})
\end{array}
$$

?
←
→
C
I
S

Either way we write it, TLC reports that its value is TRUE.

Question 9.2 We generalized *Sortings* to *GeneralSortings* by giving it an operator argument. Generalize *SortSet* in the same way by defining an operator *GeneralSortSet* in terms of the *SortSeq* operator□ defined in the standard *TLC* module. Use TLC to check the correctness of your definition on all subsets of *TestSet*(7).

ANSWER

9.3 Votes

In both Borda and Condorcet elections, each voter ranks the candidates in order of preference. To write the spec, I had to decide how to represent the voters' ballots. The obvious way to represent a single voter's ballot is by a sequence whose i^{th} element is the name of the candidate the voter ranks number i. An obvious way to represent all the votes is as a set of such sequences. However, that's not right because two voters can cast identical ballots, and there is no concept of a set having "two copies" of an element. Here are three reasonable ways to represent the collection of votes:

- With a set whose elements are records or tuples with one component being the ranking and the other identifying the voter (with a randomly chosen identification if the vote is to be anonymous).
- With a bag (multiset)□ of rankings.
- With a sequence of rankings, arranged in an arbitrary order.

The first two methods are more elegant than the third, which imposes an unnecessary ordering on the rankings. However, I decided to use the third because it's a little simpler.

Instead of making the collection of votes an argument of an operator that describes a voting scheme, it's more convenient to make it a parameter of the module. Let's call it *Votes* and add the following declaration to module *CandidateRanking*:

$$\textsc{constant}\ \ \mathit{Votes}$$

```
CONSTANT Votes
```

We can declare the set of candidates also to be a parameter, but there's no need to; we can extract the set of candidates from the value of *Votes*. Since *Votes* is a sequence of rankings, and each ranking lists all the candidates, we can define the set *Cand* of all candidates to be the set of all candidate names in *Votes*[1], the first voter's ranking. That ranking is a sequence, its i^{th} element being *Votes*[1][i]. The set of candidates is therefore the set

?
←
→
C
I
S

$$\{\mathit{Votes}[1][1],\ \mathit{Votes}[1][2],\ \ldots,\ \mathit{Votes}[1][\mathit{Len}(\mathit{Votes}[1])\}$$

where $\mathit{Len}(s)$ is defined in the *Sequences* module to be the length of a sequence s. We can write this set formally (without the "...") as:

$$\mathit{Cand}\ \triangleq\ \{\mathit{Votes}[1][i] : i \in 1\,..\,\mathit{Len}(\mathit{Votes}[1])\}$$

ASCII version

This uses the TLA$^+$ notation that $\{e(x) : x \in S\}$ is the set of all elements of the form $e(x)$ for x an element of the set S.

When we have TLC evaluate the definition of a voting scheme to determine an outcome, we should check that *Votes* is a correct collection of votes—namely, that it is a sequence of rankings of the candidates. A ranking r is a sequence containing every candidate exactly once, which is true iff the length of r equals the number of candidates, and the set of all elements of r is the set of all candidates. Correctness of the collection of votes is therefore expressed by the following formula.

$$\begin{aligned}
&\land\ \mathit{Votes} \in \mathit{Seq}(\mathit{Seq}(\mathit{Cand}))\\
&\land\ \forall j \in 1\,..\,\mathit{Len}(\mathit{Votes}) :\\
&\qquad \land\ \mathit{Len}(\mathit{Votes}[j]) = \mathit{Cardinality}(\mathit{Cand})\\
&\qquad \land\ \{\mathit{Votes}[j][i] : i \in 1\,..\,\mathit{Len}(\mathit{Votes}[j])\} = \mathit{Cand}
\end{aligned}$$

ASCII version of ASSUME statement

Make this an assumption of the spec by adding it as an ASSUME statement. TLC checks a specification's assumptions, and it will report an error if this formula is not true for the value a model assigns to the constant parameter *Votes*.

Question 9.3 Create a new model for the specification. In the What is the model? section of the Model Overview page, assign this value to *Votes*. Run TLC on the model. It will report that the assumption is false. Add print statements□ to the assumption to help locate the errors, and correct them. (It will help to start by commenting out the first conjunct of the assumption.)

9.4 The Borda Ranking

In the Borda ranking, a candidate gets a score computed as follows from the rankings. A ranking of N candidates assigns to the first-ranked (highest-ranked) candidate $N-1$ points, to the the second-ranked candidate $N-2$ points, ..., assigning 0 points to the last-ranked candidate. The score of a candidate is the sum of the points assigned to him or her by all the rankings.

Let N be the number of candidates, let V be the number of voters, and let $RankBy(c, i)$ be the ranking of candidate c by the i^{th} voter (which is a number in $1 \,..\, V$). The score of candidate c is written mathematically as

$$\sum_{i=1}^{V} N - RankBy(c, i)$$

?
←
→
C
I
S

It would be nice if we could define an operator *Sigma* that allowed us to write this formula as

$$Sigma(i, 1, V, N - RankBy(c, i))$$

However, the i in the summation is a bound identifier, and TLA$^+$ has no mechanism for defining an operator that takes a bound identifier as an argument. Instead, we define an operator that sums a sequence of numbers and apply it to the sequence:

$$\langle N - RankBy(c, 1),\ N - RankBy(c, 2), \ldots,\ N - RankBy(c, V)\rangle$$

The standard way to write an inductive definition of an operator *Op* whose argument is a finite sequence is:

$$\begin{aligned} Op(s) \ \triangleq\ & \textsc{if}\ \ s = \langle\,\rangle\ \ \textsc{then}\ \ \ldots \\ & \phantom{\textsc{if}\ \ s = \langle\,\rangle\ \ }\textsc{else}\ \ \text{some function of } Head(s)^{\square} \text{ and } Op(Tail(s)^{\square}) \end{aligned}$$

Using this pattern, we define the operator *SumSeq* that sums a sequence of integers as follows:

ASCII version

$$\begin{aligned} &\textsc{recursive}\ SumSeq(\_) \\ &SumSeq(s) \ \triangleq\ \textsc{if}\ \ s = \langle\ \rangle\ \ \textsc{then}\ 0 \\ &\phantom{SumSeq(s) \ \triangleq\ \textsc{if}\ \ s = \langle\ \rangle\ \ }\textsc{else}\ \ Head(s) + SumSeq(Tail(s)) \end{aligned}$$

Add this definition to module *CandidateRanking*, and use TLC to check that it correctly sums a sequence of numbers.

Question 9.4 (a) Define an operator *SumFcn* so that if f is an integer-valued function with a finite domain $\{d_1, \ldots, d_n\}$, then $SumFcn(f)$ equals $f[d_1] + \cdots + f[d_n]$. Use TLC to check that $SumFcn(s)$ equals $SumSeq(s)$ for finite sequences s of integers.

(b) Define a higher-order operator *SumOp* so that if $Op(d)$ is an integer for all d in the finite set D, then $SumOp(Op, D)$ equals the the sum of $Op(d)$ for all d in D.

The score of candidate c is the sum of the sequence of numbers:

$$\langle N - \mathit{RankBy}(c,\ 1),\ N - \mathit{RankBy}(c,\ 2),\ \ldots,\ N - \mathit{RankBy}(c,\ V)\rangle$$

This is a sequence of length V whose i^{th} element is $N - \mathit{RankBy}(c,\ i)$. Using TLA$^+$'s notation for writing functions□, we can write this sequence as:

$$[i \in 1 \mathrel{..} V \mapsto N - \mathit{RankBy}(c,\ i)]$$

?
←
→
C
I
S

We can therefore define the score $\mathit{Score}(c)$ of candidate c by

$$\mathit{Score}(c) \ \triangleq\ \mathit{SumSeq}([i \in 1 \mathrel{..} V \mapsto N - \mathit{RankBy}(c,\ i)])$$

Before we can add this definition to the spec, we must define RankBy. Remember that $\mathit{RankBy}(c,\ i)$ is the ranking (a number in $1 \mathrel{..} N$) that the i^{th} vote assigns to candidate c. In other words, it is the value r in $1 \mathrel{..} N$ such that the r^{th} element of the i^{th} vote is c. We express "the value r in $1 \mathrel{..} N$ such that …" with the CHOOSE operator□ as

$$\textsc{choose}\ r \in 1 \mathrel{..} N\ :\ \ldots$$

Thus, we define:

$$\mathit{RankBy}(c, i) \ \triangleq\ \textsc{choose}\ r \in 1 \mathrel{..} N\ :\ \mathit{Votes}[i][r] = c$$

We can represent the ranking of a candidate as a record with two fields: the name of the candidate and his or her score. Let's put the name in the record's *name* field and, because we want to sort the rankings, let's put the score in the record's *key* field. The set of rankings is the set of all elements

$$[\mathit{name} \mapsto c,\ \mathit{key} \mapsto \mathit{Score}(c)]$$

with c a candidate—a set we write

$$\{[\mathit{name} \mapsto c,\ \mathit{key} \mapsto \mathit{Score}(c)]\ :\ c \in \mathit{Cand}\}$$

We can then define *Borda* as follows to be the sequence of all elements in this set, sorted by score:

$$\mathit{Borda} \ \triangleq\ \mathit{SortSet}(\{[\mathit{name} \mapsto c,\ \mathit{key} \mapsto \mathit{Score}(c)]\ :\ c \in \mathit{Cand}\})$$

Add these definitions to the spec. Create a new model, copying and pasting this value for the constant *Votes*. (It is a corrected version of the value given in Question 9.3.) Have TLC compute the value of *Borda*. It should produce something like:

```
<< [key |-> 9, name |-> "bacon"],
   [key |-> 14, name |-> "boyle"],
   [key |-> 18, name |-> "faust"],
   [key |-> 19, name |-> "romeo"],
   [key |-> 24, name |-> "green"],
   [key |-> 44, name |-> "brown"],
   [key |-> 48, name |-> "smith"],
   [key |-> 48, name |-> "jones"] >>
```

?
←
→
C
I
S

This is OK, but it would be better to sort the results with the highest ranking candidate first. We could change the definition of *SortSet* to sort the elements in descending order of *key*. However, let us instead change the name of this operator from *Borda* to *ReverseBorda* and define *Borda* to be the sequence obtained by reversing the sequence *ReverseBorda*. A little thought (or a little less thought and some experimenting with TLC) shows the correct definition to be:

$$Borda \;\triangleq\; [i \in 1\,..\,N \mapsto ReverseBorda[N - i + 1]]$$

ASCII version

Problem 9.5 In addition to seeing the total scores, the award committee for which I was doing this was used to seeing how many voters ranked each candidate first, second, etc. Define an operator *BordaDetails* that produces approximately the following output when evaluated with the value of *Votes* used above, where the first line indicates that Jones was ranked first by 3 voters, second by 2 voters, etc.

```
<< <<48, "jones", 3, 2, 3, 0, 0, 0, 0, 0>>,
   <<48, "smith", 5, 1, 0, 1, 1, 0, 0, 0>>,
   <<44, "brown", 0, 4, 4, 0, 0, 0, 0, 0>>,
   <<24, "green", 0, 1, 1, 2, 0, 2, 1, 1>>,
   <<19, "romeo", 0, 0, 0, 2, 3, 1, 0, 2>>,
   <<18, "faust", 0, 0, 0, 1, 1, 5, 1, 0>>,
   <<14, "boyle", 0, 0, 0, 1, 2, 0, 4, 1>>,
   <<9, "bacon", 0, 0, 0, 1, 1, 0, 2, 4>> >>
```

9.5 The Condorcet Ranking

We say that a candidate c *dominates* a candidate d, and write $c \succ d$, iff $c \neq d$ and more voters prefer c to d than prefer d to c. In a Condorcet voting scheme, a candidate that dominates all other candidates is the winner. However, there may not be any candidate that dominates all others. An equal number of voters might prefer c to d as prefer d to c, so neither $c \succ d$ nor $d \succ c$ holds. More interestingly, there could be three candidates c, d, and e for whom $c \succ d$, $d \succ e$, and $e \succ c$ hold.

?
←
→
C
I
S

Define a nonempty set D of candidates to be a *dominating set* iff every candidate in D dominates every candidate not in D. Define the set of *Condorcet winners* to be the smallest dominating set. A Condorcet voting scheme is one in which the winner is some element of the set of Condorcet winners. There are various ways of choosing which element should win an election. However, my goal was not to choose a winner but to help the committee make its decision. Therefore, I wanted to determine the complete set of Condorcet winners. For this set to exist, there must be a smallest dominating set; and it's not obvious that such a set always exists. To show that it does, we must first prove:

C1. If D and E are dominating sets, then $D \subseteq E$ or $E \subseteq D$.

Proof

The set of all candidates is a dominating set (assuming it is nonempty) and it is finite, so there is a finite, nonempty set of dominating sets. Therefore, property C1 implies that there is a (unique) dominating set that is a subset of all other dominating sets. We define the set of Condorcet winners to be this smallest dominating set. It can be written mathematically as:

$$\begin{array}{l}\textsc{choose}\ D \in \textsc{subset}\ \mathit{Cand} : \\ \quad \land\ \mathit{IsDominatingSet}(D) \\ \quad \land\ \forall\, E \in \textsc{subset}\ \mathit{Cand} : \mathit{IsDominatingSet}(E) \Rightarrow (D \subseteq E)\end{array}$$

where *IsDominatingSet* is defined by

$$\mathit{IsDominatingSet}(D) \ \triangleq\ \begin{array}{l}\land\ D \neq \{\} \\ \land\ \forall\, d \in D : \forall\, e \in \mathit{Cand} \setminus D : d \succ e\end{array}$$

Instead of just computing the set of Condorcet winners, I wanted a ranking of all the candidates. To define this ranking, observe that the ballots cast for all the candidates define an election for any subset C of the set of candidates. The ballots of this election are obtained from the ballots $\mathit{Votes}[i]$ by simply removing all the candidates not in C. I defined the *Condorcet ranking* to be the sequence $\langle C_1, \ldots, C_m \rangle$ of sets of candidates such that:

- C_1 is the set of Condorcet winners of the election for all candidates.
- C_2 is the set of Condorcet winners of the election for all candidates not in C_1.

  $\vdots$
- C_{m-1} is the set of Condorcet winners of the election for all candidates not in $C_1 \cup C_2 \cup \ldots \cup C_{m-2}$.
- Every candidate in C_m is a Condorcet winner in the election for all candidates in C_m, where C_m is the set of all candidates not in $C_1 \cup \ldots \cup C_{m-1}$.

You should now be able to complete the definition of the Condorcet ranking yourself.

Question 9.6 Add to module *CandidateRanking* the definitions of $\succ$ and *CondorcetRanking*, the Condorcet ranking. Test them using this value for the parameter *Votes*. TLC should produce this Condorcet ranking.

ANSWER

```
<< {"smith"}, {"jones"}, {"brown"}, {"green"},
   {"faust", "romeo"}, {"boyle"}, {"bacon"} >>
```

?
←
→
C
I
S

Checking the definition on one example isn't very satisfactory. Since this is the only definition we have for the Condorcet ranking, there is nothing to check it against. The best we can do is to try it on examples and see if it produces the correct answer. Typing values to substitute for the parameter *Votes* would be tedious. Instead, let's generate random values of *Votes*. (Since we have to examine them to check the value of *CondorcetRanking*, we can try them one at a time.)

The trick to generating random test data is to use the *RandomElement* operator from the *TLC* module. For a nonempty set S, $RandomElement(S)$ equals an arbitrary element of S. If S is finite, TLC computes its value by pseudo-randomly choosing an element of S, with each element chosen with the same probability. Let's first define $RandomRanking(S)$ to be a randomly chosen single voter's ranking of the candidates in the set S. Remember that a a ranking of candidates is a sequence containing each candidate exactly once. The definition is easy:

Warning: *RandomElement* is not mathematics.

$$\begin{array}{l}
\textsc{recursive}\ RandomRanking(\_) \\
RandomRanking(S) \triangleq \textsc{if}\ S = \{\}\ \textsc{then}\ \langle\,\rangle \\
\qquad\qquad\qquad\qquad\qquad\qquad \textsc{else}\ \ \textsc{let}\ e \triangleq RandomElement(S) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textsc{in}\ \ \langle e\rangle \circ RandomRanking(S \setminus \{e\})
\end{array}$$

ASCII version

The value of *Votes* is a sequence of rankings, one from each voter. We can therefore define $RandomVotes(n, S)$ as follows to be a value of *Votes* with n voters and a set S of candidates.

$$RandomVotes(n, S) \ \triangleq\ [i \in 1\,..\,n \mapsto RandomRanking(S)]$$

ASCII version

Create a new TLC model for the *CandidateRanking* spec and add these definitions to the Additional Definitions section of the model's Advanced Options page. Have the model set *Votes* to $RandomVotes(4, \{$"a", "b", "c", "d"$\})$, and have TLC evaluate the pair $\langle Votes, CondorcetRanking\rangle$ Verify that (unless you're very unlucky), each execution of TLC uses a different value of *Votes*. Check a few of the results to make sure that they contain the correct Condorcet ranking.

The definition of the Condorcet ranking we have developed is the mathematically nicest one. It's not very efficient since evaluating it requires TLC to examine each of the 2^N subset of *Cand* for every level of recursion. (Remember that N is the number of candidates.) But TLC can do it in a few seconds for $N = 13$, which is good enough.

However, this is not how I originally wrote the definition of the Condorcet ranking. I wanted to execute the definition, not explain it to others. So, I wrote a more algorithmic definition in terms of the transitive closure of a relation—in part because I had already written a definition of transitive closure in another specification. The transitive closure of a relation is a very useful mathematical concept in computer science, so let's define it.

?
←
→
C
I
S

9.6 Transitive Closure

9.6.1 A Mathematical Definition

The transitive closure $\succ^+$ of a relation $\succ$ is defined by letting $r \succ^+ s$ hold iff there are values $t_1, \ldots, t_n$ such that

$$r \succ t_1 \succ \; \cdots \; \succ t_n \succ s$$

where this is an abbreviation for the formula

$$(r \succ t_1) \land (t_1 \succ t_2) \land \ldots \land (t_n \succ s)$$

We allow the possibility $n = 0$, in which case this formula becomes $r \succ s$, so we define $r \succ^+ s$ to be true if $r \succ s$ is.

Mathematicians represent a relation as a set of ordered pairs, taking $r \succ s$ to be an abbreviation of $\langle r, s \rangle \in \; \succ$. In the syntax of TLA$^+$, the symbol $\succ$ is the name of an infix operator, so it cannot be used as the name of a set. Let's therefore switch to using letters like R rather than symbols like $\succ$ as names of relations. We can still use $r \, R \, s$ as an abbrevation, but we have to write this formula as $\langle r, s \rangle \in R$ in an actual TLA$^+$ specification.

To understand the transitive closure, it is perhaps best to think of a relation R as a directed graph, where we take $r \, R \, s$ to mean that there is an edge from node r to node s. The transitive closure R^+ of R is then the directed graph in which there is an edge from r to s iff there is a path from r to s in the graph R. To define R^+ in this way, we need to define what a path in R is.

There are two natural ways to represent a path mathematically: as a sequence of its nodes or a sequence of its edges. I tend to prefer the representation as a sequence of nodes. So, let's first define the set *NodesOf*(R) of nodes of a relation R. A node of R is an element x such that $x \, R \, y$ or $y \, R \, x$ holds for some y, which means that x is either the first or second element of some pair in R. A pair r is a sequence of length 2 whose two elements are $r[1]$ and $r[2]$. Hence, we can define the set of nodes of R by:

$$\mathit{NodesOf}(R) \;\triangleq\; \{r[1] : r \in R\} \;\cup\; \{r[2] : r \in R\}$$

A path in R is a sequence p of nodes of R such that there is an edge from each of its nodes to the next. The i^{th} node of p is $p[i]$, so p is a path iff $\langle p[i], p[i+1]\rangle$ is in R, for all i with $1 \leq i < Len(p)$. We want to restrict paths to ones with at least one edge, which means at least two nodes. Hence, we can define the set $Paths(R)$ of all paths in R by:

$$\begin{aligned} Paths(R) \;\triangleq\; \{p \in Seq(NodesOf(R)) :& \\ & \land\; Len(p) > 1 \\ & \land\; \forall\, i \in 1 \,..\, (Len(p) - 1) : \langle p[i],\, p[i+1]\rangle \in R\} \end{aligned}$$

?
←
→
C
I
S

There is a path from one node in R to another iff they are the first and last nodes in a path in R. Hence, we define the transitive closure of R, which we write $TC(R)$, by:

$$TC(R) \;\triangleq\; \{\langle p[1],\, p[Len(p)]\rangle \;:\; p \in Paths(R)\}$$

This is a fine mathematical definition, but TLC can't evaluate it. To evaluate $TC(R)$, TLC must calculate the set $Paths(R)$, which may be infinite. (Even if it is finite, to compute its value TLC must compute the set $Seq(NodesOf(R))$, which is infinite if R is nonempty.)

To compute the transitive closure, we don't need to compute the set of all paths in R. There is a path from one node to another in R iff there is a path that contains each node at most once. Hence, for a finite relation, it suffices to consider paths of length at most equal to the cardinality of the set of nodes. Is this correct? A sequence of nodes of R of length j is an element of the set of functions□ $[1 \,..\, j \rightarrow NodesOf(R)]$. We can therefore define the set $PathsOfLen(R, j)$ of paths in R of length j by:

$$\begin{aligned} PathsOfLen(R, j) \;\triangleq\; \{p \in [1 \,..\, j \rightarrow NodesOf(R)] :& \\ & \forall\, i \in 1 \,..\, (j - 1) : \langle p[i],\, p[i+1]\rangle \in R\} \end{aligned}$$

Let's next define $ShortPaths(R)$ to be all paths with length between 2 and the cardinality of $NodesOf(R)$. This is the set:

$$PathsOfLen(R, 2) \cup PathsOfLen(R, 2) \cup \ldots \cup PathsOfLen(R,\, Cardinality(NodesOf(R)))$$

We can write this union of sets with the UNION operator□, defining $ShortPaths$ by:

$$ShortPaths(R) \;\triangleq\; \text{UNION}\;\{PathsOfLen(R, j) \;:\; j \in 2 \,..\, Cardinality(NodesOf(R))\}$$

We can then define the transitive closure operator TC as above, except using $ShortPaths$ instead of $Paths$. Copy and paste these definitions into the spec.

Let's check our definition of TC. It's a good idea to start small. The smallest non-trivial example is a relation with 3 nodes, say the nodes 1, 2, and 3, whose graph is $1 \to 2 \to 3$. The corresponding relation R is $\{\langle 1,2\rangle, \langle 2,3\rangle\}$. Have TLC evaluate TC applied to this set. It should produce

$$\{\langle 1,2\rangle,\ \langle 1,3\rangle,\ \langle 2,3\rangle\}$$

which is correct. Now lets add the edge $3 \to 1$ to the graph (adding $\langle 3,1\rangle$ to the set). TLC finds the transitive closure to be

$$\{\langle 1,2\rangle,\ \langle 1,3\rangle,\ \langle 2,1\rangle,\ \langle 2,3\rangle,\ \langle 3,1\rangle,\ \langle 3,2\rangle\}$$

?
←
→
C
I
S

This is not right. The graph has a path between every pair of nodes, so the transitive closure should contain all nine possible pairs, not just six. Missing are the pairs $\langle 1,1\rangle$, $\langle 2,2\rangle$, and $\langle 3,3\rangle$.

The problem is that our definition does not include long enough paths. If there is a path from any node x to any *different* node y, then there is a path from x to y in which each node appears at most once. However, in a path from a node to itself, that node must appear twice. Hence, we must also include paths of length $\mathit{Cardinality}(\mathit{NodesOf}(R))+1$ in $\mathit{ShortPaths}(R)$. Correct the definition of $\mathit{ShortPaths}$ and check that TLC now computes the correct transitive closure. Further checking will reveal no errors; the definition is now correct.

9.6.2 A Definition TLC Can Execute Faster

With the definition above, TLC can compute the transitive closure, but can TLC compute it fast enough? To evalute it, TLC must compute the set of all paths in R of length $\mathit{Cardinality}(\mathit{NodesOf}(R))+1$. The number of paths of some length j increases exponentially with j. A little experimentation reveals that with 8 nodes, TLC starts taking a long time. (Even for some relations with only 7 nodes, the set $\mathit{ShortPaths}(R)$ can be too large for TLC to handle.) Since there could be more than 8 candidates, we need a definition that TLC can compute more efficiently.

Searching the Web reveals that the transitive closure of a relation R is often defined to equal

$$R \ \cup\ R\cdot R\ \cup\ R\cdot R\cdot R\ \cup\ \ldots$$

where $\cdot$ denotes *relation composition*. The composition $R\cdot S$ of relations R and S is defined by letting $x\, R\cdot S\, y$ be true iff there is a z such that $x\, R\, z$ and $z\, S\, y$ hold. Mathematicians often write R^n for the composition $R \cdot R \cdot\ \ldots\ \cdot R$ of a relation R with itself n times. With this notation, the transitive closure of R is

$$R^1 \cup R^2 \cup R^3 \cup \ldots$$

For a finite relation R, we can stop after a finite number of terms, so the transitive closure equals

$$R^1 \cup R^2 \cup \ldots \cup R^k$$

for a sufficiently large k. A little throught reveals that we can take k to be the number of nodes of R. This leads immediately to a simple recursive definition of the transitive closure. Let's write that definition.

First, we must define relation composition. TLA$^+$ does not provide $\cdot$ as a user-defined operator, so let's write $**$ instead. Since a relation is a set of ordered pairs, a mathematician might define $**$ by

? ← → C I S

$$R ** S \;\triangleq\; \{\langle x, y\rangle \;:\; \exists\, z : (\langle x, z\rangle \in R) \land (\langle z, y\rangle \in S)\}$$

However, the right-hand side of this definition is not a legal TLA$^+$ expression. TLA$^+$ provides two ways to write a set in terms of the conditions satisfied by its elements: $\{x \in T \,:\, P(x)\}$ and $\{e(x) \,:\, x \in T\}$. Let's use the second way.

The definition above suggests that we want T to be the set of all pairs of pairs of the form $\langle\, \langle x, z\rangle,\ \langle z, y\rangle\,\rangle$ with $\langle x, z\rangle \in R$ and $\langle z, y\rangle \in S$. That's the subset of $R \times S$ consisting of all pairs $\langle r, s\rangle$ with $r[2] = s[1]$. We can define that set T by

See the definition of ×.□

$$T \;\triangleq\; \{rs \in R \times S \,:\, rs[1][2] = rs[2][1]\}$$

For $R ** S$ to have the form $\{e(x) \,:\, x \in T\}$, what should e be? If rs equals $\langle\, \langle x, z\rangle,\ \langle z, y\rangle\,\rangle$, then $\langle x, y\rangle$ equals $\langle rs[1][1],\ rs[2][2]\rangle$. This leads to the following definition

$$\begin{array}{l} R ** S \triangleq \text{LET } T \triangleq \{rs \in R \times S : rs[1][2] = rs[2][1]\} \\ \qquad\qquad\;\; \text{IN } \;\; \{\langle x[1][1],\ x[2][2]\rangle : x \in T\} \end{array}$$

ASCII version

Question 9.7 Write an alternative definition of $**$ using a set construction of the form $\{x \in T \,:\, P(x)\}$.

ANSWER

We can now define the transitive closure of R to be

$$R \;\cup\; R ** R \;\cup\; \ldots \;\cup\; \overbrace{R ** \;\ldots\; ** R}^{Cardinality(NodesOf(R)) \text{ times}}$$

Calling the operator $SimpleTC$, we write it as:

$$\begin{array}{l} SimpleTC(R) \;\triangleq \\ \quad \text{LET RECURSIVE } STC(\_) \\ \qquad\quad STC(n) \triangleq \text{IF } n = 1 \text{ THEN } R \\ \qquad\qquad\qquad\qquad\qquad\quad \text{ELSE } STC(n-1) \cup STC(n-1) ** R \\ \quad \text{IN } \;\; STC(Cardinality(NodesOf(R))) \end{array}$$

ASCII version

Add this definition to module *CandidateRanking*. Let's now test it by comparing it with the definition TC of the transitive closure. A relation is just a set of ordered pairs, so let's check it on all possible relations on a set of three elements by checking it on all subsets of $1\,..\,3 \times 1\,..\,3$. Have TLC evaluate:

$$\begin{array}{l}\forall S \in \textsc{subset}\ ((1\,..\,3) \times (1\,..\,3)) : \\ \quad SimpleTC(S) = TC(S)\end{array}$$

```
\A␣S␣\in␣SUBSET␣((1..3)\X␣(1..3))␣:
␣␣␣␣␣SimpleTC(S)␣=␣TC(S)
```

TLC reports the error:

> This was a Java StackOverflowError. It was probably the result of an incorrect recursive function definition that caused TLC to enter an infinite loop when trying to compute the function or its application to an element in its putative domain.

?
←
→
C
I
S

What's the problem? On what value of S is the definition failing? Let's add a print statement□ to find out. Have TLC evaluate

$$\begin{array}{l}\forall S \in \textsc{subset}\ ((1\,..\,3) \times (1\,..\,3)) : \\ \quad PrintT(S)\ \land\ SimpleTC(S) = TC(S)\end{array}$$

The only value printed is $\{\,\}$. (TLC prints the value twice because it re-evaluates the expression when determining its error message.) When S equals the empty set, $NodesOf(S)$ is the empty set, which has cardinality 0. The evaluation of $STC(0)$ never terminates.

Modify the definition to check for the special case of the empty relation, whose transitive closure is the empty relation

```
$SimpleTC(R) \triangleq$
  LET RECURSIVE $STC(\_)$
      $STC(n) \triangleq$ IF $n = 1$ THEN $R$
                                    ELSE $STC(n-1) \cup STC(n-1) ** R$
  IN  IF $R = \{\}$ THEN $\{\}$ ELSE $STC(Cardinality(NodesOf(R)))$
```

Have TLC again check that $SimpleTC$ equals TC when applied to all subsets of $(1\,..\,3) \times (1\,..\,3)$. This time, immediately TLC reports that they are equal. It takes TLC a few minutes to check all subsets of $(1\,..\,4) \times (1\,..\,4)$. There's no point trying it on the 2^{25} subsets of $(1\,..\,5) \times (1\,..\,5)$, which would take more than 2^9 times as long.

Although we haven't tried comparing these definitions on any relation with more than four nodes, we have tried it on *all* relations with at most four nodes. This kind of complete testing on small relations is much more effective at finding errors than is testing on randomly chosen large relations. We can be quite confident that the two definitions are equivalent on finite relations. Since the two definitions are so different, it's unlikely that we've made an error in formalizing the definition of transitive closure in TLA$^+$.

9.6.3 Warshall's Algorithm

Searching for *transitive closure* on the Web reveals that the standard method of computing it is called *Warshall's Algorithm* or sometimes the *Floyd-Warshall*

?
←
→
C
I
S

Algorithm. The algorithm is usually described in code consisting of nested iterative loops. It's easy to write such code as a PlusCal algorithm, but we want to define a TLA$^+$ operator *Warshall* for which TLC evaluates $Warshall(R)$ by executing Warshall's algorithm. We could do it by directly translating the iterative loops into a recursively defined operator. As a simple example of how that's done, consider this code for computing a value v.

```
$i := 1$ ;
$v := v0$ ;
while $(i \leq n)$ { $v := F(v, i)$ ;
                    $i := i + 1$;    }
```

The value v it computes can be defined as follows (assuming $n > 0$):

$$\begin{array}{lll} v \triangleq & \text{LET} & \text{RECURSIVE } vr(\_) \\ & & vr(i) \triangleq \text{ IF } i = 0 \text{ THEN } v0 \text{ ELSE } F(vr(i-1),\, i) \\ & \text{IN} & vr(n) \end{array}$$

Note that $vr(i)$ is defined to be the value of v computed by the i^{th} iteration of the code's loop.

Instead of trying to mimic a particular coding of an algorithm, it's best to first understand the algorithm and then express your understanding in a mathematical definition. Let's return to our definition of the transitive closure in terms of the graph of a relation. Let $\mathcal{N}$ be the set of nodes of a relation R. The transitive closure $TC(R)$ of R is the set of all pairs $\langle r, s \rangle$ in $\mathcal{N} \times \mathcal{N}$ such that there is a path

$$r \rightarrow t_1 \rightarrow \cdots \rightarrow t_k \rightarrow s$$

in R. By eliminating loops, we can choose this path so that all the t_i are distinct. So, let's consider only such paths. Let $W(\mathcal{M})$ be the set of all such pairs $\langle r, s \rangle$ for which the path can be chosen with all the t_i in $\mathcal{M}$. Observe that $W(\mathcal{N})$ equals $TC(R)$, and $W(\{\})$ equals R. Warshall's algorithm computes $TC(R)$ by recursively computing $W(\mathcal{N})$. The key observation is that for any node n in $\mathcal{N}$, the pair $\langle r, s \rangle$ is in $W(\mathcal{M} \cup \{n\})$ iff it is in $W(\mathcal{M})$ or there is a path

$$r \rightarrow t_1 \rightarrow \cdots \rightarrow t_j \rightarrow n \rightarrow t_{j+1} \rightarrow \cdots t_k \rightarrow s$$

in R with all the t_i in $\mathcal{M}$. Thus, $\langle r, s \rangle$ is in $W(\mathcal{M} \cup \{n\})$ iff it is in $W(\mathcal{M})$ or else $\langle r, n \rangle$ and $\langle n, s \rangle$ are in $W(\mathcal{M})$. In other words:

$$\begin{array}{l} W(\mathcal{M} \cup \{n\}) \;=\; W(\mathcal{M}) \;\cup\; \{rs \in \mathcal{N} \times \mathcal{N} \;: \\ \qquad\qquad (\langle rs[1], n \rangle \in W(\mathcal{M})) \land (\langle n, rs[2] \rangle \in W(\mathcal{M}))\} \end{array}$$

Substituting $\mathcal{L}$ for $\mathcal{M} \cup \{n\}$ in this relation, we get the following relation, where n is any element of $\mathcal{L}$:

$$\begin{array}{l} W(\mathcal{L}) \;=\; W(\mathcal{L} \setminus \{n\}) \;\cup\; \{rs \in \mathcal{N} \times \mathcal{N} \;: \\ \qquad\qquad (\langle rs[1], n \rangle \in W(\mathcal{L} \setminus \{n\})) \land (\langle n, rs[2] \rangle \in W(\mathcal{L} \setminus \{n\}))\} \end{array}$$

Remembering that $W(\{\})$ equals R and $W(\mathcal{N})$ equals $TC(R)$, this leads to the following definition of *Warshall*:

```
$Warshall(R) \triangleq$
  LET $NR \triangleq NodesOf(R)$
      RECURSIVE $W(\_)$
      $W(L) \triangleq$ IF $L = \{\}$
                    THEN $R$
                    ELSE LET $n \quad \triangleq$ CHOOSE $node \in L :$ TRUE
                             $WM \triangleq W(L \setminus \{n\})$
                         IN  $WM \cup \{rs \in NR \times NR :$
                                    $(\langle rs[1], n \rangle \in WM) \land (\langle n, rs[2] \rangle \in WM)\}$
  IN  $W(NR)$
```

ASCII version

?
←
→
C
I
S

Use TLC to check that the operators *Warshall* and *SimpleTC* are equivalent definitions of the transitive closure on all subsets of $1 \,..\, 4 \times 1 \,..\, 4$, which it should do in a fraction of a minute.

Question 9.8 The time taken by Warshall's algorithm to compute the transitive closure of a relation with n nodes is proportional to n^3. Use TLC to evaluate *Warshall* on the relation with graph

$$0 \rightarrow 1 \rightarrow 2 \rightarrow \ldots \rightarrow n-1 \rightarrow 0$$

for different values of n. How does the time taken by TLC vary with n? Why?

ANSWER

9.7 The Condorcet Ranking Revisited

Let us return now to the problem of computing the Condorcet ranking, defined in Section 9.5 above. This requires computing the set of Condorcet winners of an election. Let $\succ$ be the domination relation, meaning that $c \succ d$ iff more voters prefer c to d than prefer d to c. Recall that a set D of candidates is a *dominating set* iff $c \succ d$ holds for any c in D and any candidate d not in D. The set $\mathcal{CW}$ of Condorcet winners is the smallest dominating set.

Let c be in $\mathcal{CW}$ and suppose $c \not\succ d$ for some candidate d. By definition of a dominating set, this implies that d is also in $\mathcal{CW}$. Similarly, if $d \not\succ e$ for some candidate e, then e is also in $\mathcal{CW}$. Let $\succeq$ be the relation defined by letting $x \succeq y$ hold iff $\neg(y \succ x)$ holds. Then $c \in \mathcal{CW}$ and $d \succeq c$ imply $d \in Cset$, and $e \succeq d$ then implies $e \in \mathcal{CW}$. A simple induction argument shows that if $\succeq^+$ is the transitive closure of the $\succeq$ relation, then $c \in \mathcal{CW}$ and $d \succeq^+ c$ imply $d \in \mathcal{CW}$.

The observation that $d \in \mathcal{CW}$ and $c \succeq^+ d$ imply $c \in \mathcal{CW}$ suggests that the reason a candidate c must be in $\mathcal{CW}$ is that $c \succeq^+ d$ holds for some candidate d that must be in $\mathcal{CW}$. This line of thinking leads us eventually to the following property:

C2. $c \succeq^{+} d$ holds, for any elements c and d of $\mathcal{CW}$.

PROOF

It follows from the definition of $\succ$ that $c \succeq d$ holds iff either $c = d$ or the number of voters that prefer c to d is greater than or equal to the number of voters that prefer d to c. Therefore, $c \succ d$ implies $c \succeq d$, as does $c = d$.

Property C2 asserts that any candidate c in $\mathcal{CW}$ satisfies $c \succeq^{+} d$ for all $d \in \mathcal{CW}$. Because $\mathcal{CW}$ is a dominating set, $c \succ d$ for all candidates d not in $\mathcal{CW}$. Since $c \succ d$ implies $c \succeq d$, which implies $c \succeq^{+} d$, we see that every c in $\mathcal{CW}$ satisfies $c \succeq^{+} d$ for all candidates d. We now show that this condition characterizes the elements of $\mathcal{CW}$.

?
←
→
C
I
S

C3. $\mathcal{CW} = \{c \in \mathit{Cand} : \forall d \in \mathit{Cand} : c \succeq^{+} d\}$

PROOF

It's easy to modify the definition *CondorcetRanking* of the Condorcet ranking that you wrote in Question 9.6 to use the formula of C3 for the set of Condorcet winners. The following defines *CRanking* to be equivalent to *CondorcetRanking*. It assumes the definition of $\succ$ from the answer to Question 9.6; *DomEq* is the relation $\succeq$ and *DomEqPlus* is its transitive closure computed using *SimpleTC*.

$$\begin{array}{l}
\mathit{CRanking} \triangleq \\
\quad \textsc{let } \mathit{DomEq} \triangleq \{r \in \mathit{Cand} \times \mathit{Cand} : \neg(r[2] \succ r[1])\} \\
\quad\quad\quad \mathit{DomEqPlus} \triangleq \mathit{SimpleTC}(\mathit{DomEq}) \\
\quad\quad\quad \mathit{CWinners}(C) \triangleq \{c \in C : \forall\, d \in C : \langle c,\, d\rangle \in \mathit{DomEqPlus}\} \\
\quad\quad\quad \textsc{recursive } \mathit{CRanking}(\_) \\
\quad\quad\quad \mathit{CRanking}(C) \triangleq \textsc{if } C = \{\} \textsc{ then } \langle\rangle \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \textsc{else } \textsc{let } \mathit{CW} \triangleq \mathit{CWinners}(C) \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \textsc{in } \langle \mathit{CW} \rangle \circ \mathit{CRanking}(C \setminus \mathit{CW}) \\
\quad \textsc{in } \mathit{CRanking}(\mathit{Cand})
\end{array}$$

ASCII version

Add this definition to module *CandidateRanking*.

Now that we have two definitions of the Condorcet ranking, we can check them by comparing the two. In module *CandidateRanking*, we have made their inputs (the collection of votes) the parameter *Votes*. We can therefore compare the definitions of *CondorcetRanking* and *CRanking* on only one input for each run of TLC. Had we instead made the votes an argument of the definitions, then it would have been easy to compare them on a set of inputs.

By using the TLA^{+} INSTANCE construct, we can compare the two definitions on a set of inputs without without having to rewrite the definitions. The statement

$$\textsc{instance } \mathit{CandidateRankings} \textsc{ with } \mathit{Votes} \leftarrow e$$

in a module M imports into M all the definitions from the *CandidateRankings* module, except with the expression e substituted for *Votes* in all those definitions. For example, it defines *Borda* in module M to be the Borda ranking for the collection e of votes. We can parameterize the imported definitions by instead using this form of INSTANCE statement:

$$CR(vt) \;\triangleq\; \text{INSTANCE } \mathit{CandidateRanking} \text{ WITH } \mathit{Votes} \leftarrow vt$$

? ← → C I S

This defines $CV(e)!\mathit{Borda}$ to be the definition of *Borda* from *CandidateRanking* with e substituted for *Votes*. (It's a "deep" substitution, meaning that the substitution is made as well in all the other definitions in *CandidateRanking* on which the definition of *Borda* depends.) Of course, the same applies to all definitions in *CandidateRanking*, so $CV(e)!\mathit{RankBy}(c, i)$ equals the result of substituting e for *Votes* in $\mathit{RankBy}(c, i)$.

What does "!" mean?

Create a new specification with a root module named *CheckRankings*. We will need the operators of the *Integers* and *Sequences* modules, so import them with an EXTENDS statement. Then add the parameterized INSTANCE statement above.

As we have seen with our definitions of the transitive closure, a good way to check that two definitions are equivalent is to test them on all possible inputs of a certain size. So, let's check them on all sets of N candidates and V voters. Add a CONSTANTS statement declaring N and V to be constant parameters. (There is no name conflict with the definitions of N and V in module *CandidateRankings*, because those operators are renamed to $CV(\_)!N$ and $CV(\_)!V$.)

We now have to define the set of all values of *Votes* with N candidates and V voters. Let's start by defining the set of all rankings of N candidates by a single voter. Such a ranking is a sequence of candidates containing each candidate exactly once. From the discussion in Section 9.2 we can see that the set of all such rankings for an N-element set *Cand* of candidates is:

$$\{seq \in [1 \,..\, N \to \mathit{Cand}] \;:\; \mathit{Cand} \subseteq \{seq[i] \;:\; i \in \mathit{Cand}\}\}$$

Since the identities of the candidates doesn't matter, we can let the set of candidates be $1 \,..\, N$. This leads to the definition:

$$\begin{array}{lll} \mathit{VoterRankings} \;\triangleq & \text{LET} & \mathit{Cand} \;\triangleq\; 1 \,..\, N \\ & \text{IN} & \{seq \in [\mathit{Cand} \to \mathit{Cand}] : \\ & & \qquad \mathit{Cand} \subseteq \{seq[i] : i \in \mathit{Cand}\}\} \end{array}$$

The set of all sequences of rankings by V voters is then

$$\mathit{AllVotes} \;\triangleq\; [1 \,..\, V \to \mathit{VoterRankings}]$$

The assertion that our two definitions of the Condorcet rankings are equivalent is then expressed by the following assumption.

$$\text{ASSUME } \forall v \in \mathit{AllVotes} \;:\; CR(v)!\mathit{CondorcetRanking} = CR(v)!\mathit{CRanking}$$

Add these definitions and this assumption to module *CheckRankings*. Create a model with 3 candidates and 4 voters and run TLC on it. TLC checks assumptions, so it will report an error if this assumptions is violated. It should run for a couple of seconds and not report any error. TLC can check the assumption for 4 candidates and 4 voters in about 15 minutes. However, with 5 candidates and 4 voters there are about 200 million values to check, which would take TLC days.

?
←
→
C
I
S

The checks for 4 or fewer candidates that TLC can do provide some confidence in the equivalence of the definitions. However, there could be strange cases that occur only with more candidates or voters. It would be nice to do some further checking. We can't check all possible elections with more candidates and voters, but we can check randomly chosen ones. Recall the definitions above of *RandomRanking* and *RandomVotes*, where $RandomVotes(n, S)$ is a randomly chosen value of *Votes* with n voters and S the set of candidates. Add these definitions to module *CheckRanking*. Also, add a declaration of a new constant *Trials* that is the number of random values of *Votes* to check. The following strange definition is evaluated to be a set of (usually) *Trials* randomly chosen values for *Votes*.

$$SetOfRandomVotes \triangleq \{RandomVotes(V, 1\,..\,N) : x \in 1\,..\,Trials\}$$

ASCII version

This definition is strange because the set expression has the form $\{e(x) : x \in S\}$ where the value of $e(x)$ is independent of x. Mathematically, this should define *SetOfRandomVotes* to be a set consisting of one element (assuming $Trials > 0$). However, because TLC evaluates the set expression by evaluating $RandomVotes(V, 1\,..\,N)$ for each value of x in $1\,..\,Trials$, it obtains a set containing *Trial* elements (unless two different executions of $RandomVotes(V, 1\,..\,N)$ happen to obtain the same value).

Add the definition of *SetOfRandomVotes* and the following assumption to module *CheckRanking*:

ASCII version

$$\textsc{assume}\ \forall\, v \in SetOfRandomVotes : CR(v)!CondorcetRanking = CR(v)!CRanking$$

Use TLC to check the equivalence of the two definitions of the Condorcet ranking for some values of V and N. First, set *Trial* to 1 to see how long it takes TLC to check equivalence for a single value of *Votes*. For V and N equal to 10, it should take a couple of seconds per trial. (Comment out the previous assumption before doing this, otherwise TLC will spend forever trying to check it.)

After some amount of checking, we will decide that module *CandidateRanking* is correct. However, we are not done yet. I am assuming we wrote the specification for someone to use. It would be difficult for a user to understand the specification just from the formulas—especially if she were not used to reading TLA$^+$ specs. (It would also be difficult for us a year from now.) We need to add comments to explain the spec. Here is what the module might look like□ after adding some comments. It contains what I regard to be a fairly minimal set of comments, suitable for a reasonably sophisticated reader who is acquainted with TLA$^+$. The members of the committee for which I wrote my original specification were mathematically sophisticated but knew nothing of TLA$^+$. The comments that I wrote for them therefore explained all the TLA$^+$ notation whose meaning I felt they would not find obvious.

Problem 9.9 There is a more efficient method for computing the Condorcet ranking than by using property C3. The sets of candidates that occur in the Condorcet ranking are the connected components of the graph of $\succeq$. There are algorithms that compute the set of connected components of a graph in time approximately proportional to the sum of the number of nodes and the number of edges in the graph. Look up these algorithms and use one as the basis for another definition of the Condorcet ranking. (Needless to say, you should use TLC to check the equivalence of this definition with *CRanking*.)

?
←
→
C
I
S

Problem 9.10 Write a new specification that allows voters to indicate that they have no preference among certain candidates. In other words, voters should be able to rank some sets of candidates as equivalent. Choose a convenient method of representing the votes, and define the Borda and Condorcet rankings on this representation.