The Capacity-C Torch Problem

Roland Backhouse* Hai Truong†

February 22, 2015

Abstract

The torch problem (also known as the bridge problem or the flashlight problem) is about getting a number of people across a bridge as quickly as possible under certain constraints. Although a very simply stated problem, the solution is surprisingly non-trivial. The case in which there are just four people and the capacity of the bridge is two is a well-known puzzle, widely publicised on the internet. We consider the general problem where the number of people, their individual crossing times and the capacity of the bridge are all input parameters. We present two methods to determine the shortest total crossing time: the first expresses the problem as an integer-programming problem that can be solved by a standard linear-programming package, and the second expresses the problem as a shortest-path problem in an acyclic directed graph, i.e. as a dynamic-programming problem. The complexity of the linear-programming solution is difficult to predict; its main purpose is to act as an independent test of the correctness of the results returned by the second solution method. The dynamic-programming solution has best- and worst-case time complexity proportional to the square of the number of people. An empirical comparison of the efficiency of both methods is also presented.

This manuscript has been accepted for publication in Science of Computer Programming. The manuscript will undergo copyediting, typesetting, and review of the resulting proof before it is published in its final form. Please note that during the production process errors may be discovered which could affect the content, and all disclaimers that apply to the journal apply to this manuscript.

A definitive version was subsequently published in Science of Computer Programming, 1 May 2015, Vol.102:76–107,
http://dx.doi.org/10.1016/j.scico.2015.01.003.

The (capacity-C) torch problem is as follows.

*School of Computer Science, University of Nottingham, Nottingham NG8 1BB, U.K.

†Faculty of Computer Science & Engineering, HCMC University of Technology, National University of Ho Chi Minh City, 268 Ly Thuong Kiet Street, District 10, Ho Chi Minh City, Vietnam. Work completed whilst at University of Nottingham.

> N people wish to cross a bridge. It is dark, and it is necessary to use a torch when crossing the bridge, but they only have one torch between them. The bridge is narrow and at most C people can be on it at any one time. The people are numbered from 1 to N. Person i takes time t.i to cross the bridge; when a group of people cross together they must all proceed at the speed of the slowest.
>
> Derive an algorithm that will construct a sequence of crossings to get all N people across in the shortest time. Prove that the algorithm does indeed find the shortest time.

The torch problem is a generalisation of a problem involving four people wishing to cross a bridge of capacity two and with specific concrete times. In this form, the problem is believed to have first appeared in 1981. Rote [Rot02] gives a comprehensive bibliography.

The main interest in the torch problem is that what is "obvious" or "intuitive" is often wrong. For example, the "obvious" solution of letting the fastest person repeatedly accompany $C-1$ people across the bridge is wrong. (If $N=4$, $C=2$ and the travel times are 1, 1, 2 and 2, this solution takes time 7 whereas the shortest crossing time is 6[1].) Also, the "obvious" property that the shortest time is achieved when the number of crossings is minimised is incorrect. (If $N=5$, $C=3$ and the travel times are 1, 1, 4, 4 and 4, the shortest time is 8, which is achieved using 5 crossings. The shortest time using 3 crossings is 9.) It is not difficult to determine an *upper bound* on the crossing time, even in the general case. Nor is it difficult to provide counterexamples to incorrect solutions. The difficulty is to establish an irrefutable *lower bound* on the crossing time. A proper solution to the problem poses a severe test of our standards of proof.

We present two practical solutions to the problem: an integer-programming solution and a dynamic-programming solution. The practicality of the solutions is made possible by a transformation of the problem from finding sequences of crossings to finding bags (multi-sets) of crossings. The formal basis for the transformation is outlined in section 1 and developed further in section 2. The significance of the transformation from sequence to bags cannot be underestimated; it is vital to constructing solutions that are of polynomial-time complexity and not exponential-time. The two solution methods, which are not unrelated, are presented in section 4. The best- and worst-case time complexity of the dynamic-programming solution is $O(N^2)$ and its space complexity is $O(\frac{N^2}{C})$. This means that the space requirement is proportional to N^2 for small values of C (for example, C equal to 3) but reduces for larger values of C. Our experience in practice is that the space requirement is the main practical limitation. Section 4.6 sketches the results

[1] Our examples are chosen so that it is easy for the reader to discover the fastest crossing time. Of course, the examples in puzzle books are deliberately chosen to make it difficult.

of a practical comparison of both the integer-programming and dynamic-programming solutions. Both have been tested for large values of N (up to 25000) and of C (up to 50 for the largest values of N); execution times for the dynamic-programming solution are at most seconds and for the integer-programming solution are at most minutes. Space limitations prevented us from testing with larger values.

Henceforth, we assume that N is at least C+1. (When N is at most C, it is obvious that exactly one crossing gives the optimal solution. When N is at least C+1, more than one crossing is required.) We assume that C is at least 2 since, otherwise, the problem is not solvable. We also assume that crossing times are positive (that is, $0 \leq t.i$ for all i) and the people are sorted so that $t.i \leq t.j$ for all i and j such that $1 \leq i \leq j \leq N$. The motivation for the latter assumption becomes apparent in section 3. Abusing English slightly, we say that person i is *faster* than person j (or j is *slower* than i) if $i < j$, even though their crossing times may be equal. Finally, to avoid clutter, we often treat N, C and t as global parameters (constants) on which other notions may implicitly depend. (Occasionally, N is introduced as an explicit parameter for the purposes of an inductive proof or construction.)

In a precursor to this paper, Backhouse [Bac08] proposed a different dynamic-programming solution to the problem. The current algorithm is simpler and its space requirements are also less. We would therefore no longer recommend Backhouse's solution but we have implemented it as a further check on the correctness of our algorithms. Other improvements that we have made are to remove the requirements that all persons have distinct crossing times and that the crossing times have to be strictly positive. Backhouse's paper established the theorem that is fundamental to finding an efficient algorithm (in this paper, theorem 38) but the changes we have made have obliged us to revise all our proofs.

1 Outline Strategy

The torch problem is a classical optimisation problem: it is about designing an algorithm that *constructively* determines the minimum of an *objective* function defined on a domain of *putative* solutions. Formally, the minimum time is expressed by[2]

$$\langle \Downarrow s : PutativeSequence.s : TotalTime.s \rangle \quad ,$$

where the predicate $PutativeSequence$ formulates the requirement that a putative solution is a sequence of crossings that gets everyone across in accordance with the rules, and

[2]The symbols $\Downarrow$ and $\Uparrow$ denote the maximum and minimum quantifiers, respectively; later we also use the symbols $\downarrow$ and $\uparrow$ to denote the binary minimum and maximum operators.

the function $TotalTime$ evaluates the time taken for all people to cross for a given putative sequence. The algorithm must be constructive in the sense that it must explicitly construct a sequence s that realises the minimum time.

1.1 Transforming Optimisation Problems

Our solution involves a series of non-trivial transformations of the problem specification and it is as well to begin by explaining the logical basis of such transformations.

Suppose we are given an optimisation problem in the form

$$\langle \Downarrow s : s{\in}P : T.s \rangle \quad .$$

The set P defines the solution space and the function T is the objective function. A transformation of the problem in its most general form replaces P by some set Q and the objective function T by some function U such that

$$\langle \Downarrow s : s{\in}P : T.s \rangle \quad = \quad \langle \Downarrow r : r{\in}Q : U.r \rangle \quad .$$

The goal of making such a transformation is to improve the efficiency of the calculation of an optimal value, usually by reducing the space of solutions in some way. For example, transforming the space of solutions from the set of permutations of n values to the set of subsets of n values "reduces" the size of the search space from $n!$ to 2^n.

Establishing the correctness of the transformation typically involves an at-most-and-at-least argument. That is, we prove

$$\langle \Downarrow s : s{\in}P : T.s \rangle \quad \leq \quad \langle \Downarrow r : r{\in}Q : U.r \rangle$$

(the at-least argument) and

$$\langle \Downarrow s : s{\in}P : T.s \rangle \quad \geq \quad \langle \Downarrow r : r{\in}Q : U.r \rangle$$

(the at-most argument). The required equality is then inferred from the anti-symmetry of the at-least relation on numbers.

Both inequalities are proved by showing how one optimisation problem is transformed to another. The logical basis for the subsequent proof obligation is the following calculation.

$$\begin{array}{ll} & \langle \Downarrow s : s{\in}P : T.s \rangle \quad \leq \quad \langle \Downarrow r : r{\in}Q : U.r \rangle \\ = & \quad \{ \quad \text{definition of minimum quantification} \quad \} \\ & \langle \forall r : r{\in}Q : \langle \Downarrow s : s{\in}P : T.s \rangle \ \leq \ U.r \rangle \\ = & \quad \{ \quad \text{minimum is a choice operator} \quad \} \end{array}$$

$$
\begin{array}{ll}
 & \langle \forall r : r{\in}Q : \langle \exists s : s{\in}P : T.s \leq U.r \rangle \rangle \\
= & \quad \{ \quad \text{axiom of choice} \quad \} \\
 & \langle \exists \tau : \tau \in Q \rightarrow P : \langle \forall r : r{\in}Q : T.(\tau.r) \leq U.r \rangle \rangle \quad .
\end{array}
$$

Vice-versa,

$$
\begin{array}{ll}
 & \langle \Downarrow s : s{\in}P : T.s \rangle \quad \geq \quad \langle \Downarrow r : r{\in}Q : U.r \rangle \\
= & \quad \{ \quad \text{as above} \quad \} \\
 & \langle \exists \sigma : \sigma \in P \rightarrow Q : \langle \forall s : s{\in}P : U.(\sigma.s) \leq T.s \rangle \rangle \quad .
\end{array}
$$

That is, we have to prove

$$\langle \exists \tau \ : \ \tau \in Q \rightarrow P \ : \ \langle \forall r : r{\in}Q : T.(\tau.r) \leq U.r \rangle \rangle$$

and

$$\langle \exists \sigma \ : \ \sigma \in P \rightarrow Q \ : \ \langle \forall s : s{\in}P : U.(\sigma.s) \leq T.s \rangle \rangle \quad .$$

The functions τ and σ are called *transformations* because they transform one solution space to another.

Publications on optimisation algorithms often neglect to provide full details of the correctness proofs — typically by not giving details of one or other of the transformations τ or σ, let alone establishing the correctness of the transformation. In many cases the transformation and correctness proof are trivial. For example, if the solution space Q is a subset of P and the objective functions T and U are identical, a suitable transformation τ is the identity function on Q and the first proof obligation is trivial; the second proof obligation is then met by establishing that every solution s in P is "subsumed" by a solution $\sigma.s$ in Q. However, the most interesting problems involve non-trivial transformations of the solution space, and objective functions that are not identical.

In our solution to the torch problem some of the steps involve simple transformations but others are decidedly non-trivial because the functions T and U are different and the relation between the sets P and Q is non-trivial. It is important to give explicit details of the transformation τ in order to give a constructive solution to the given optimisation problem. Explicit details of σ are unnecessary for this purpose but, even so, it remains important to provide a rigorous proof of correctness. When the specification of a transformation involves an iterative algorithm (as is the case for several of the transformations we present) it is necessary to establish the correctness of the algorithm. This explains why this paper is perhaps longer than the length of the final java code might at first suggest.

1.2 Putative and Regular Sequences

Recall that a sequence of crossings that gets everyone across in accordance with the rules is called a *putative sequence*. Let us be more precise.

When crossing the bridge, the torch must always be carried. This means that crossings, both of groups of people and of each individual person, alternate between "forward"and "return" "trips". Formally, a *trip* is a non-empty set of people and, given a sequence of trips, a *forward trip* in the sequence is an odd-numbered element of the sequence (the first, the third, the fifth, etc.) and a *return trip* is an even-numbered element of the sequence; the sequence of trips *made by* person p is the subsequence given by those trips of which p is an element. A sequence of trips is *putative* if

(a) the sequence has odd length, and each trip has at least one and at most C elements,

(b) each person is an element of at least one trip,

(c) the sequence of trips made by each person alternates between forward and return trips, and begins and ends with a forward trip.

The time taken by a sequence of trips is the sum of the times taken by each element of the sequence. The time taken by an individual trip is the time taken by the slowest person in the trip, who we call the "leader" of the trip. Given our global assumption that the crossing-time function t is monotonic, the leader of a trip is the highest-numbered person in the trip.

We say that one putative sequence *subsumes* another putative sequence if the time taken by the first is at most the time taken for the second. An *optimal* sequence is a putative sequence that subsumes all putative sequences. The problem is to find an optimal sequence.

A forward trip is *regular* if it is made by *at least two people*, and a return trip is *regular* if it is made by *exactly one person*. A sequence of trips is *regular* if it is a putative sequence that consists entirely of regular forward and return trips. Given a sequence of trips s the bag of trips FwdBag.s is obtained from s by removing the return trips and ignoring the order in which the trips are made. It is important to note that the total time taken by a regular sequence s of trips can always be calculated knowing just FwdBag.s. This is because every person makes one fewer return trips than forward trips and each return trip in a regular sequence is made by just one person; so the total time for the return trips can be deduced.

1.3 Outline Transformations

An outline of our solution is as follows.

1. Lemma 10 reduces the search space to the space of regular sequences.

   This has the consequence that the search space for an optimal solution becomes finite: there are at most N−1 forward trips in any regular sequence. More importantly, the time taken by a regular sequence can be evaluated knowing only which forward trips are made. This suggests the next step.

2. Definition 15 defines a "regular" *bag* of *forward* trips and theorem 30 reduces the search space to the space of such bags. It does so by establishing a formal correspondence between regular sequences of forward and return trips and regular bags of forward trips.

   The step from sequences to bags is the most important in designing an efficient algorithm because it eliminates attributes of sequences that are not relevant to the total time taken whilst still retaining a constructive algorithm in the sense that it is also possible to construct a sequence of crossings that achieves the minimum time.

3. Theorem 38 reduces the search space yet further from the space of regular bags to the space of *ordered* bags, as defined in definition 37.

4. Section 4 presents two methods for calculating an optimal ordered bag of forward trips.

   Both methods exploit the property that a trip in an ordered bag can be represented by two people: the slowest person in the trip (who we call the "leader" of the trip), and the slowest "nomad" in the trip. (A "nomad" is a person who makes at least one return trip.) Section 4.4 shows how to use integer programming to determine the minimum time and section 4.5 shows how to exploit dynamic programming.

   The integer-programming solution has unpredictable complexity; our main purpose in formulating such a solution was to check the results determined by our dynamic-programming solution against the results determined by a standard, open-source linear-programming package. The dynamic-programming solution has best- and worst-case time complexity proportional to the square of the number of people, as shown in section 4.5.1. Section 4.6 summarises an empirical comparison of the efficiency of both solutions.

5. Section 5 reflects on what has been achieved.

1.4 Terminology

Henceforth the numbers N and C and the function t will be treated as global parameters (constants) on which other entities may depend. By a *person* we mean a number in the

range 1..N and by a *trip* we mean a subset of {1..N} that has at least 1 and at most C elements. The *forward* trips in a sequence of trips are the odd-numbered trips (that is, the first, the third, the fifth, etc.); the *return* trips are the even-numbered trips.

Let us suppose a putative sequence of trips is given. By extracting just the forward trips in the sequence and ignoring the order in which they are made, we obtain a bag (multiset) of non-empty sets. We use F to denote such a bag. Note that a bag is a set with multiplicities. We sometimes use a notation exemplified by $\{\!|\{3,4\}, 2*\{1,4\}, 2*\{1,2,4\}|\!\}$ to denote a bag; the example is a bag of trips in which the trip {3,4} occurs once (that is, has implicit multiplicity 1), and the trips {1,4} and {1,2,4} occur twice.

For the purposes of formalising our algorithms, it is useful to equate a set of elements of type S with a function of type S→bool (effectively, its membership relation) and a bag of elements of type S with a function of type S→ℕ. This convention has some notational advantages: if b is a bag of elements of type S, then, for all x in S, the *multiplicity* of x in b is b.x and the set of *elements* of b is defined to be the function e where, for all x in S, $e.x \equiv 0 < b.x$. Also, the size of b, denoted as usual by |b|, is Σb (that is, $\langle \Sigma x : x \in S : b.x \rangle$ where S→ℕ is the type of b).

In this way, if {1..N} identifies the set of people, a trip T is a function of type {1..N}→bool and a bag F of trips is a function of type ({1..N}→bool)→ℕ; also person i is an element of the trip T if T.i and the multiplicity of T in F is F.T. By a slight abuse of notation, we write x∈b and call x an *element* of b if x is an element of the set underlying bag b, that is, if $0 < b.x$. Similarly, we also write i∈T instead of T.i.

For brevity, we sometimes define a bag b of type S→ℕ by specifying b.x only for those x such that b.x is greater than 0. For elements y of S not included in the definition, the convention will be that b.y is 0.

For a given bag of forward trips F, the number of times person i makes a forward trip is given by the function f defined by

(1) $\quad f.F.i \;=\; \langle \Sigma T : i \in T : F.T \rangle \quad .$

The number of times that each person returns is given by the function r; since each person makes one more forward trip than return trip, we let

(2) $\quad r.F.i \;=\; f.F.i - 1 \quad .$

A person who makes a return trip is called a *nomad*:

(3) $\quad nomad.F.i \quad \equiv \quad r.F.i > 0 \quad .$

A *settler* is a person who crosses but does not return:

(4) $\quad settler.F.i \quad \equiv \quad r.F.i = 0 \quad .$

We divide the forward trips into "pure", "nomadic" and "mixed". A *pure trip* is a trip in which everyone is a settler:

(5) $\quad \text{pure.F.T} \quad \equiv \quad \langle \forall j : j \in T : \text{settler.F.i} \rangle \quad .$

A *nomadic trip* is a trip in which everyone is a nomad:

(6) $\quad \text{nomadic.F.T} \quad \equiv \quad \langle \forall i : i \in T : \text{nomad.F.i} \rangle \quad .$

A *mixed trip* is a trip that is neither pure nor nomadic (that is, a trip that involves both settlers and nomads). A *full trip* is a trip that has C elements and a *non-full trip* is one that has less than C elements.

Given our assumption that the crossing-time function t is monotonic, the *leader* of a trip is defined to be the person in the trip with the highest number:

(7) $\quad \text{lead.T} \quad = \quad \langle \Uparrow i : i \in T : i \rangle \quad .$

Mixed and pure trips have multiplicity 1 in the bag F, and each settler is an element of exactly one element of F. It is therefore possible to define a function from settlers to people that identifies the highest-numbered person in the trip made by the settler. Let us call this function boss.F. Then the defining property of boss.F is

(8) $\quad \langle \forall i,T \ : \ \text{settler.F.i} \wedge T \in F \wedge i \in T \ : \ \text{lead.T} = \text{boss.F.i} \rangle \quad .$

For nomads, the function boss.F is undefined.

Example 9 (Nomads and settlers, nomadic, mixed and pure trips) Suppose $N = 9$ and $C = 3$. Let F be the bag

$$\{\!|2 * \{1,2\}, \{3,4,5\}, \{3,6,7\}, \{7,8,9\}|\!\} \quad .$$

Persons 1, 2, 3 and 7 are nomads and persons 4, 5, 6, 8 and 9 are settlers. The trip $\{1,2\}$ is nomadic and non-full and the trips $\{3,4,5\}$, $\{3,6,7\}$ and $\{7,8,9\}$ are mixed and full. There are no pure trips.

The tables below exemplify the different functions. The leftmost table shows the forward and return counts for each person, the middle table shows the lead function for each of the trips and the rightmost table shows the boss function for each of the settlers.

Later (example 31) we show that F is the bag of forward trips defined by a regular sequence of trips that gets 9 people across a bridge of capacity 3.

i	f.F.i	r.F.i
1	2	1
2	2	1
3	2	1
4	1	0
5	1	0
6	1	0
7	2	1
8	1	0
9	1	0

T	lead.T
{1,2}	2
{3,4,5}	5
{3,6,7}	7
{7,8,9}	9

i	boss.F.i
4	5
5	5
6	7
8	9
9	9

A simple example of a bag that does have pure trips is

$$\{\!|2{*}\{1,2\},\{3,4\}|\!\}\ .$$

(The trip $\{3,4\}$ is pure and the trip $\{1,2\}$ is nomadic.) This is also the bag of forward trips defined by a regular sequence of trips, this time to get 4 people across a bridge of capacity 2.
□

2 Regular Sequences Versus Regular Bags

This section is crucial to the rest of the paper. We show in theorem 30 that the search space can be limited to bags of forward trips as opposed to sequences of forward and return trips. To do so, we exploit the "regularity" property. Recall that a "regular" sequence is a sequence in which each forward trip involves at least two people and each return trip involves exactly one person. We begin with a lemma that restricts attention to just the regular sequences. Then in section 2.1 we show how regular sequences of forward and return trips can be reconstructed from a bag of forward trips (with certain minimal properties).

Lemma 10 Every putative sequence is subsumed by a regular sequence.

Proof The proof is similar to the one given by Rote [Rot02, Lemma 1] for the case that the capacity is 2. It differs because we do not assume that crossing times are distinct or strictly positive. However, we do need to assume that crossing times are positive.

We describe an iterative algorithm that initialises variable s to the given putative sequence and terminates when s is regular. At each iteration, s is replaced by a sequence s' that subsumes s. The algorithm makes progress by always reducing the total number

of person-trips (that is, it reduces the sum over persons p of the total number of trips made by p).

Suppose a given putative sequence s contains irregular trips. There are two cases to consider: the first irregular trip is a return trip and the first irregular trip is a forward trip.

If the first irregular trip is a return trip, choose one person, p say, making the return trip. Identify the (regular) forward trip made by p prior to the return trip, and remove p from both trips. Because the crossing times are positive, the result is a putative sequence that subsumes the given sequence; it also has a smaller person-trip count.

Now suppose the first irregular trip is forward. There are two cases to consider: the irregular trip is the very first in the sequence, and it is not the very first.

If the first trip in the sequence is not regular, it means that one person crosses and then immediately returns. These two trips can be removed. Because the crossing times are positive, the result is a putative sequence that subsumes the given sequence; it also has a smaller person-trip count.

If the first irregular trip is a forward trip but not the very first, let us suppose it is person q who crosses, and suppose p is the person who returns immediately before this forward trip. (There is only one such person because of the assumption that q's forward trip is the first irregular trip.) Consider the latest trip that precedes p's return trip and involves p or q. There are two cases: it is a forward trip involving p or it is a return trip involving q.

If it is a forward trip, replace p by q in the trip and remove p's return trip and q's irregular trip. The result is a putative sequence that subsumes the given sequence (since, crossing times are positive and, for any positive x, $t.p\uparrow x + t.p + t.q \geq t.q\uparrow x$) and has a smaller person-trip count.

If it is a return trip, replace q by p in the trip, and remove p's return trip and q's irregular trip. The result is a putative sequence that subsumes the given sequence (since $t.p + t.q + t.p \geq t.q$) and has a smaller person-trip count.

□

Theorem 11

$$\langle \Downarrow s : PutativeSequence.s : TotalTime.s \rangle \;=\; \langle \Downarrow s : RegularSequence.s : TotalTime.s \rangle$$

where the predicates $PutativeSequence$ are $RegularSequence$ are as defined in section 1.3.

Proof As discussed in section 1.1, lemma 10 establishes that the left side of the equation is at least the right side. By definition, a regular sequence is a putative sequence so the identity transformation witnesses the opposite inequality.

□

2.1 Scheduling Forward Trips

In view of theorem 11, we now consider bags of forward trips in the image set of the function $FwdBag$. (Recall that $FwdBag.s$ is obtained from sequence s by removing return trips and ignoring the order of the forward trips.) Suppose s is a regular bag of trips and $F = FwdBag.s$. Because s is regular (and hence also putative), F has a number of properties. First, each person must cross at least once:

(12) $\langle \forall i : 1 \leq i \leq N : 1 \leq f.F.i \rangle$.

In a regular sequence, each forward trip involves at least 2 and at most C people:

(13) $\langle \forall T : T \in F : 2 \leq |T| \leq C \rangle$.

Finally, since the number of forward trips is $|F|$ and each return trip is undertaken by exactly one person,

(14) $|F| = \langle \Sigma i : 1 \leq i \leq N : r.F.i \rangle + 1$.

Properties (12), (13) and (14) play a central role. Their importance is highlighted by the following definition.

Definition 15 (Regular bag) A bag of trips, F, is said to be *regular* if it satisfies (12), (13) and (14) (for given values of N and C).
□

Crucially, given an arbitrary regular bag of trips, F, it is possible to construct a regular sequence s such that $F = FwdBag.s$. To establish this theorem, we first prove several properties relating the number of pure trips, the number of nomads and the number of non-pure trips in F.

We define the functions bnc ("bag nomad count"), tnc ("trip nomad count"), rc ("return count"), pc ("pure-trip count") and npc ("non-pure trip count") as follows. In the definitions, G is an arbitrary bag of trips. That is, G is a function of type $(\{1..N\} \rightarrow bool) \rightarrow \mathbb{N}$. Variable T ranges over elements of G and variable i ranges over $\{1..N\}$. See earlier for the definitions of the functions $nomad$, $settler$, $pure$ and r.

(16) $bnc.G = \langle \Sigma i : nomad.G.i : 1 \rangle$.

(17) $tnc.G.T = \langle \Sigma i : nomad.G.i \wedge i \in T : 1 \rangle$.

(18) $rc.G = \langle \Sigma i : nomad.G.i : r.G.i \rangle$.

(19) $pc.G = \langle \Sigma T : pure.G.T : 1 \rangle$.

(20) $npc.G = \langle \Sigma T : \neg(pure.G.T) : G.T \rangle$.

(Note that pure trips always have a multiplicity of 1.)

Lemma 21 Suppose G is a bag of trips. Then

(22) $bnc.G \leq rc.G$,

(23) $rc.G = \langle \Sigma T :: tnc.G.T \times G.T \rangle - bnc.G$,

(24) $npc.G \neq 1$.

Proof Properties (22) and (23) are straightforward. Property (24) is proved as follows.

$$
\begin{array}{ll}
 & npc.G = 1 \\
= & \quad \{ \quad \text{definition of } npc \quad \} \\
 & \langle \Sigma T : \neg(pure.G.T) : G.T \rangle \ = \ 1 \\
\Rightarrow & \quad \{ \quad \text{definition of } pure \text{ and arithmetic} \quad \} \\
 & \langle \exists j : nomad.G.j : \langle \Sigma T : j \in T : G.T \rangle \leq 1 \rangle \\
\Rightarrow & \quad \{ \quad \text{definition of } nomad \text{ and } f \quad \} \\
 & \langle \exists j : 1 < f.G.j : f.G.j \leq 1 \rangle \\
= & \quad \{ \quad \text{inequalities} \quad \} \\
 & \text{false} \ .
\end{array}
$$

□

Corollary 25 If G is a bag of trips such that $|G| = rc.G + 1$ then

(26) $bnc.G = 0 \ \equiv \ |G| = 1$, and

(27) $\langle \forall T :: \neg(pure.G.T) \rangle \ \Leftarrow \ bnc.G = 1$.

Proof First,

$$
\begin{array}{ll}
 & bnc.G = 0 \\
= & \quad \{ \quad (16), \text{ arithmetic} \quad \} \\
 & \langle \forall i :: \neg(nomad.G.i) \rangle \\
= & \quad \{ \quad \text{definition of } nomad, (18), \text{ arithmetic} \quad \} \\
 & rc.G = 0 \\
= & \quad \{ \quad \text{assumption: } |G| = rc.G + 1 \quad \} \\
 & |G| = 1 \ .
\end{array}
$$

Second,

$$
\begin{array}{ll}
 & \langle \forall T :: \neg(pure.G.T) \rangle \\
\Leftarrow & \quad \{ \quad \text{definition of } pure, \text{ arithmetic} \quad \} \\
 & \langle \forall T :: tnc.G.T = 1 \rangle \\
\Leftarrow & \quad \{ \quad \langle \forall T : T \in G : tnc.G.T \leq 1 \rangle \ \Leftarrow \ bnc.G = 1, \text{ arithmetic} \quad \} \\
 & bnc.G = 1 \quad \wedge \quad \langle \Sigma T :: G.T \rangle \ = \ \langle \Sigma T :: tnc.G.T \times G.T \rangle \\
= & \quad \{ \quad |G| \ = \ \langle \Sigma T :: G.T \rangle, (23) \quad \} \\
 & bnc.G = 1 \quad \wedge \quad |G| \ = \ rc.G + bnc.G \\
= & \quad \{ \quad \text{assumption: } |G| \ = \ rc.G + 1 \quad \} \\
 & bnc.G = 1 \quad .
\end{array}
$$

□

In general, the implication in (27) cannot be strengthened to an equivalence. For example, the bag G equal to $\{\{1,3\},\{1,2,4\},\{2,5\}\}$ satisfies the property that $|G| = rc.G + 1$ and every trip in G is non-pure. However, the set of nomads in G is $\{1,2\}$. That is, $bnc.G \neq 1$.

Lemma 28 Suppose G is a bag of trips satisfying $|G| = rc.G + 1$ and suppose $1 < |G|$. Then G contains a non-pure trip T such that $tnc.G.T - 1 \leq pc.G$.

Proof

$$
\begin{array}{ll}
 & \langle \exists T \ : \ T \in G \wedge \neg(pure.G.T) \ : \ tnc.G.T - 1 \leq pc.G \rangle \\
= & \quad \{ \quad G \text{ is non-empty, property of minimum} \quad \} \\
 & \langle \Downarrow T : T \in G \wedge \neg(pure.G.T) : tnc.G.T \rangle \ \leq \ pc.G + 1 \\
\Leftarrow & \quad \{ \quad \text{pigeon-hole principle (the minimum of a non-empty} \\
 & \qquad \text{bag of integers is at most the average),} \\
 & \qquad (20) \text{ and integer inequalities} \quad \} \\
 & \langle \Sigma T : T \in G \wedge \neg(pure.G.T) : tnc.G.T \times G.T \rangle \ < \ npc.G \times (pc.G + 2) \\
= & \quad \{ \quad (23) \quad \} \\
 & rc.G + bnc.G \ < \ npc.G \times (pc.G + 2) \\
\Leftarrow & \quad \{ \quad (22) \quad \} \\
 & 2 \times rc.G \ < \ npc.G \times (pc.G + 2)
\end{array}
$$

$=$ { by range splitting, $|G| = pc.G + npc.G$; $|G| = rc.G + 1$ }

$2 \times (pc.G + npc.G - 1) \ < \ npc.G \times (pc.G + 2)$

$\Leftarrow$ { arithmetic }

$2 \leq npc.G$

$=$ { (24) }

$0 \neq npc.G$

$=$ { (26) and assumption: $1 < |G|$ }

true .

□

Lemma 29 Suppose G is a bag of trips such that $|G| = rc.G + 1$. Let the set of people P be the set of nomads and settlers in G. That is, $P = \{i \mid nomad.G.i \vee settler.G.i\}$. Then there is a sequence s of forward and return trips with the properties that

(a) the sequence has odd length, and each return trip has exactly one element;

(b) for each person i in P, the subsequence of s consisting of trips made by person i alternates between forward and return trips, and begins and ends with a forward trip;

(c) $G = FwdBag.s$.

Proof By induction on $|G|$. If $|G| = 1$ then $rc.G = 0$. Let T be the unique element of G. Define s to be $[T]$ (the sequence of length one consisting of T). It is easy to verify that the required properties are satisfied.

Now suppose $1 < |G|$. Then, by lemma 28, we can choose a non-pure trip T such that $tnc.G.T - 1 \leq pc.G$. Define the bag G' and the sequence $s0$ as follows. The sequence $s0$ begins with the trip T and has total length $2 \times tnc.G.T$. The trip T is followed in $s0$ by a sequence of alternating return and forward trips, beginning and ending with a return trip. The return trips are made by the $tnc.G.T$ nomads in T, the order being arbitrary; the $tnc.G.T - 1$ forward trips are distinct pure trips taken from G, their choice also being arbitrary. The bag G' is the result of removing all the forward trips in the sequence $s0$ from the bag G. It is straightforward to check that $|G'| = |G| - tnc.G.T$ and $rc.G' = rc.G - tnc.G.T$. It follows that $|G'| = rc.G' + 1$. Moreover $|G'| < |G|$ (since T is a non-pure trip, i.e. $0 < tnc.G.T$); so the induction hypothesis can be applied to G'. Let s' be the sequence constructed from G'. Define s to be $s0 +\!\!+ s'$. It is then straightforward to verify that s satisfies the induction hypothesis.

□

Theorem 30 Suppose F is a regular bag of trips. Then there is a regular sequence of trips s such that $F = FwdBag.s$.

Proof Immediate from lemma 29 with $G := F$ and the definition of a regular sequence (noting that $1 \leq i \leq N \equiv nomad.F.i \vee settler.F.i$).
□

Example 31 (Regular bag) As an example of the algorithm given in lemma 29, let F be the bag

$$\{\!|2*\{1,2\},\{3,4,5\},\{3,6,7\},\{7,8,9\}|\!\} \quad .$$

Recall that this was the bag introduced in example 9. From the data given there, it is easy to verify that F satisfies (12), (13) and (14) when N is 9 and C is 3.

Since the algorithm is non-deterministic, there are typically several different sequences that might be constructed. In this case there are eight valid sequencings of the bag captured by the algorithm. Variable G is initialised to equal F so that the initial number of pure trips in G is 0. As a consequence, the first trip T that is chosen may be $\{3,4,5\}$ or $\{7,8,9\}$. Choosing the former, the trip $\{3,4,5\}$ is scheduled together with a return trip by person 3. Then G still has no pure trips so the next trip T that is chosen may be either $\{3,6,7\}$ or $\{7,8,9\}$. (Note that person 3 is no longer a nomad in G.) Choosing $\{3,6,7\}$, this forward trip is scheduled together with a return trip by person 7. After removal of the trip from G, trip $\{7,8,9\}$ is pure and one of the two occurrences of $\{1,2\}$ must be chosen. In this way, the sequence of trips that is computed is the following.

$$+\{3,4,5\}; -\{3\}; +\{3,6,7\}; -\{7\}; +\{1,2\}; -\{1\}; +\{7,8,9\}; -\{2\}; +\{1,2\} \quad .$$

(Interchanging the trips $+\{3,6,7\}$ and $+\{7,8,9\}$ in the above is also a valid sequencing of the bag. The return trips $-\{1\}$ and $-\{2\}$ may also be interchanged. Similarly, a further four possibilities are obtained by choosing $+\{7,8,9\}$ as the first element in the sequence. There are other valid sequencings not captured by the algorithm. For example, there are valid sequencings that begin with the trip $+\{3,6,7\}$.)
□

3 The Optimisation Problem

Let us summarise the development so far. Theorem 11 proves that the solution space can be restricted to the set of regular sequences of trips. A regular sequence of trips is then easily transformed to a regular bag of trips (by forgetting the return trips and the sequencing) and, vice versa, any such bag can be transformed into a regular sequence of trips (see theorem 30). We have thus transformed the solution space from sequences to

bags but must now show how to transform the objective function. This is straightforward because the time taken is independent of the order of the trips in a sequence, and knowing just the forward trips made by each person in a regular sequence of trips, we can easily calculate the bag of return trips that are made.

This section begins by formulating the objective function for our new optimisation problem. We then introduce theorem 38 which shows how to reduce the solution space yet further to "ordered" bags of trips.

Let T be a trip. The time taken to make the trip T is, by definition, the time of the slowest person in the trip. Determining the time is facilitated if persons are sorted so that $t.i \leq t.j$ if $i \leq j$; the time taken by the trip is then the time taken by the leader of the trip:

(32) $time.T \;=\; t.(lead.T)$.

(Recall that the leader of a trip is the highest numbered person in the trip. See (7).)

Let G be a bag of non-empty trips. (So G has type $(\{1..N\} \rightarrow bool) \rightarrow \mathbb{N}$ and $G.\emptyset = 0$.) We extend the function $time$ to G by adding individual trip times:

(33) $time.G \;=\; \langle \Sigma T : T \in G : G.T \times time.T \rangle$.

Now, suppose a regular sequence of trips is given and suppose F is the bag of forward trips in the sequence. The bag of return trips in the given sequence is calculated from F by the function RetTrips defined by

(34) $RetTrips.F.\{i\} \;=\; r.F.i$.

The total time taken by the sequence is then defined by the function TotTime:

(35) $TotTime.F \quad = \quad time.F + time.(RetTrips.F)$.

Spelling out the definition of $time$ and RetTrips, TotTime.F is equal to

$$\langle \Sigma T \;:\; T \in F \;:\; t.(lead.T) \times F.T \rangle \;+\; \langle \Sigma i :: t.i \times r.F.i \rangle \quad .$$

(Forward trip T takes time $t.(lead.T)$ and has multiplicity F.T, and person i makes r.F.i return trips each of which takes time t.i because the sequence is regular.)

In summary, assuming that persons are sorted so that $t.i \leq t.j$ if $i \leq j$, we have:

Theorem 36

$$\langle \Downarrow s : RegularSequence.s : TotalTime.s \rangle \;=\; \langle \Downarrow F : Regular.F : TotTime.F \rangle$$

where the predicate $Regular$ is given by Definition 15 and the objective function TotTime is given by (35).

□

Our optimisation problem has thus been transformed to a constructive computation of

$$\langle \Downarrow F : Regular.F : TotTime.F \rangle \quad .$$

That is, we have to determine a regular bag F of forward trips that realises the minimum total time. The algorithm described in lemma 29 may then be applied to compute $Seq.F$ giving an optimal putative sequence of forward and return trips.

We continue to use the notion of "subsumption" but now applied to (regular) bags rather than sequences. So regular bag F *subsumes* regular bag G if F's total travel time is at most that of G. A bag is *optimal* if it is regular and subsumes all other bags.

Theorem 38, below, reduces the search space yet further to bags that are also *ordered*. The notion of an "ordered" bag is defined as follows.

Definition 37 (Ordered) We say that a bag of trips, F, is *ordered* if it is regular and it also satisfies the following properties.

(a) For each T in F, the nomads in T are persons 1 thru $tnc.F.T$. That is,

$$\langle \forall i,T : T \in F \wedge i \in T : nomad.F.i \equiv 1 \leq i \leq tnc.F.T \rangle \quad .$$

(b) The function $boss.F$ is monotonically increasing. That is, for all settlers i and j,

$$boss.F.i \leq boss.F.j \Leftarrow i \leq j \quad .$$

(c) All pure trips in F are full. All mixed trips in F are full with the possible exception of the fastest[3] mixed trip.

(d) The *settler count* sc defined by

$$sc.F.T \;=\; \langle \Sigma i : settler.F.i \wedge i \in T : 1 \rangle \quad .$$

is a monotonically increasing function of the leader of the trip. That is, for all trips T and U in F,

$$sc.F.T \leq sc.F.U \Leftarrow lead.T \leq lead.U \quad .$$

For all non-nomadic trips, the function tnc is a monotonically decreasing function of the leader of the trip. That is, for all non-nomadic trips T and U in F,

[3]Two mixed trips may have the same crossing time. However, recall our convention that if people i and j have the same crossing time, i is faster than j if $i < j$.

$$tnc.F.T \geq tnc.F.U \Leftarrow lead.T \leq lead.U \quad .$$

(e) If the fastest mixed trip is not full, it has $bnc.F$ nomads.

In anticipation of later usage, we say that a bag of trips, F, is *semi-ordered* if it satisfies properties (a), (b), (c) and (e) but not necessarily property (d).
□

Theorem 38 Every regular bag of trips is subsumed by an ordered bag of trips.

□

In words, 37(a) expresses the property that the nomads are the fastest, and always make forward trips in a contiguous group[4] which includes person 1. Property 37(b) expresses the property that the trips in a regular bag divide the settlers into contiguous groups. So, in summary, theorem 38 establishes the "intuitively obvious" property that the search for an optimal solution can be restricted to bags of trips in which, in order of increasing travel times, the groups of people are: the nomads, the settlers in a non-full mixed trip, the settlers in full mixed trips and the pure settlers. Moreover, the number of settlers increases as the trips get slower.

To prove theorem 38 we present an algorithm that accomplishes the required transformation. The algorithm is a sequential composition of several iterative algorithms that in turn establish properties 37(a), (b) and (c), followed by (d) and (e). Property 37(a) is established in section 3.1, properties 37(b) and (c) are established in section 3.2 and properties 37(c) and (d) in section 3.3. Of course, it is necessary to ensure that the algorithm to establish one property maintains the properties earlier in the list.

Non-nomadic trips have multiplicity 1 in F. Thus, for non-nomadic trips T, there is no confusion between the trip T and the individual occurrences of T in F. On the other hand, nomadic trips may have multiplicity greater than 1 in F. For such trips, we are careful to make clear whether the transformation is applied to all occurrences of the trip or just one.

3.1 Choosing Nomads

We begin with property 37(a). The transformation algorithm is a combination of two iterative algorithms, presented in lemmas 40 and 41. Both algorithms maintain the regularity property whilst choosing the fastest people to be the nomads in non-pure trips. Accordingly, the measure of progress is

(39) $\langle \Sigma i : 1 \leq i \leq N : f.F.i \times i \rangle \quad .$

We first establish that the nomads are persons 1 thru n, for some n.

[4] A "contiguous" group of people is a set of people of the form $\{i..j\}$ for some i and j.

Lemma 40 Every regular bag of trips is subsumed by a regular bag in which all settlers are slower than all nomads.

Proof Suppose that, within regular bag F, i is the fastest settler and j is the slowest nomad. Suppose i is faster than j (that is, $i<j$ and $t.i \leq t.j$).

Interchange i and j everywhere in F. We get a regular bag, F′. The return time is clearly reduced by at least $t.j - t.i$.

The times for the forward trips in F involving j are not increased in F′ (because $t.i \leq t.j$). The time for the *one* forward trip in F involving i is increased in F′ by an amount that is at most $t.j - t.i$. This is verified by considering two cases. The first case is when j is an element of i's forward trip. In this case, swapping i and j has no effect on the trip, and the increase in time taken is 0. In the second case, j is not an element of i's forward trip. In this case, it suffices to observe that, for any x (representing the maximum time taken by the other participants in i's forward trip),

$$
\begin{array}{ll}
 & t.i \uparrow x \ + \ (t.j - t.i) \\
= & \quad \{ \quad \text{distributivity of sum over max, arithmetic} \quad \} \\
 & t.j \uparrow (x + (t.j - t.i)) \\
\geq & \quad \{ \quad t.i \leq t.j, \text{ monotonicity of max} \quad \} \\
 & t.j \uparrow x \ .
\end{array}
$$

Finally, the times for all other forward trips are unchanged.

The net effect is that the total time taken does not increase. That is, F′ subsumes F. Repeating the process is guaranteed to terminate because the measure of progress (39) is decreased by

$$(1 \times i + f.F.j \times j) - (f.F.j \times i + 1 \times j)$$

and $1 < f.F.j$ and $i < j$; that is, the measure of progress is strictly decreased.
□

Lemma 41 Every regular bag of forward trips is subsumed by a regular bag that satisfies 37(a).

Proof Suppose a regular bag F of forward trips is given. By lemma 40, we may assume that the nomads in F are persons 1 thru bnc.F. (If not, apply the transformation detailed in lemma 40 first.)

Suppose T is a trip in F such that the nomads in T are not persons 1 thru tnc.F.T. (Recall that tnc.F.T is the number of nomads in trip T.) Replace one occurrence of T in the bag F by the trip T′ where

$$T' \ = \ (T \cap \{i \mid settler.F.i\}) \ \cup \ \{i \mid 1 \leq i \leq tnc.F.T\} \ .$$

This replaces F by a bag F′. To see that F′ is regular, we consider the three sets T∩T′, T∩¬T′ and T′∩¬T. We have

$$\langle \forall i : i \in T \cap T' : f.F.i = f.F'.i \rangle \quad ,$$

$$\langle \forall i \ : \ i \in T \cap \neg T' \ : \ f.F.i = f.F'.i + 1 \ \wedge \ tnc.F.T < i \leq bnc.F \rangle \quad ,$$

$$\langle \forall i \ : \ i \in T' \cap \neg T \ : \ f.F'.i = f.F.i + 1 \ \wedge \ 1 \leq i \leq tnc.F.T \rangle \quad .$$

It follows that people that are settlers in F are also settlers in F′ but some people may be nomads in F but settlers in F′. However, the number of forward trips made by each person remains strictly positive (since a nomad makes at least 2 forward trips, by definition). Moreover, $|T| = |T'|$ and $|T \cap \neg T'| = |T' \cap \neg T|$. It follows that (12), (13) and (14) are invariant under the replacement. That is, F′ is regular. Clearly, F′ subsumes F and, by a similar argument to that used in lemma 40, the measure of progress (39) is strictly decreased. If in F′ there is a nomad that is slower than a settler, the construction in lemma 40 can be used to rectify the situation. In this way, repeated application of the transformation is guaranteed to terminate with a bag satisfying 37(a).
□

Property 37(a) of a regular bag F guarantees that the number of nomads in F is at most C because $tnc.F.T \leq C$ for each trip T in F.

From now on, we assume that we are given a regular bag F that satisfies 38(a). The transformations detailed in the lemmas below rely on this assumption and are designed to maintain the property.

Except where otherwise stated, the measure of progress we use in both section 3.2 and 3.3 is

(42) $\langle \Sigma T : T \in F : lead.T \rangle$.

Note that a transformation that guarantees a strict decrease of (42) whilst making no change to the nomads automatically guarantees the subsumption relation because the total time is a monotonic function of the leaders of the trips.

3.2 Permuting Settlers

In this section, we consider properties 37(b) and (c).

Lemma 43 Every regular bag is subsumed by a regular bag that satisfies 37(b).

Proof Suppose F does not satisfy 37(b). Take any two settlers i and j such that $boss.F.i > boss.F.j$ and $i \leq j$. It follows that $i \neq j$ and they must be in different trips, T and U say. Swap boss.F.j in trip U with person i in trip T. Then, using primes to denote

the new trips, $lead.T' = lead.T = boss.F.i$ and $lead.U' < lead.U$ because $boss.F.j$ has been replaced by i and $i < j \leq boss.F.j$. So the measure of progress (42) strictly decreases and F' subsumes F. Repeated application of the transformation is thus guaranteed to terminate with a bag that subsumes the given bag and satisfies 37(b). The nomads are unaffected so 37(a) is maintained.
□

In order to prove 37(c), we consider pure and mixed trips separately. Lemmas 44 and 45 show, respectively, how pure trips and mixed trips are filled.

Lemma 44 Every regular bag is subsumed by a regular bag in which all pure trips are full.

Proof Suppose T is a non-full pure trip in F. Among such trips, choose the one with the fastest leader. By combining property 37(a) with corollary 25, we can choose a non-pure trip with at least 2 nomads, and let U be one occurrence of the trip. (The possibility that $bnc.F = 0$ is excluded because of the assumption that $C < N$.) Add person 1 to T. Remove the slowest nomad from U. Additionally, if the size of U is reduced to 1, remove U altogether from F. This results in a regular bag that has total travel time at most the total travel time for F. (The forward time for T is not changed because of 37(a), the forward time for U does not increase or is eliminated entirely, and the total return time is not increased by the removal of the slowest nomad from U and the addition of person 1 to T.) That is, we have constructed a regular bag that subsumes F and has one fewer non-full pure trips. Repeating the process until there are no non-full pure trips is guaranteed to terminate with a bag that subsumes F.
□

Lemma 45 Every regular bag is subsumed by a regular bag satisfying 37(c).

Proof Suppose F is a regular bag of trips satisfying 37(a) and (b). Apply lemma 44 in order to ensure that all pure trips are full. Sort the mixed trips in F in descending order of their leaders. Now rearrange the settlers so that the mixed trips are filled in order whilst maintaining 37(b). That is, the slowest mixed trip is first filled (if not already full) by shunting settlers from the next slowest mixed trip into the trip in descending order of crossing time. Then the next slowest trip is filled, and so on. For example, if the capacity is 5, the nomads are persons 1, 2, and 3, and the non-nomadic trips in descending order of leader are

$$\{1,12,11,10\}\ ,\ \{1,2,9,8,7\}\ ,\ \{1,2,6,5\}\ ,\ \{1,2,3,4\}$$

replace the trips by

$$\{1,12,11,10,9\}\ ,\ \{1,2,8,7,6\}\ ,\ \{1,2,5,4\}\ ,\ \{1,2,3\}\ .$$

The transformation may result in one or more trips with just one element (inevitably person 1 because of property 37(a)). If so, remove this trip.

The transformation strictly decreases the measure of progress (42) and so does not increase the total travel time. The number of non-full mixed trips is reduced to at most one which is necessarily the fastest.

□

3.3 Monotonicity Properties

We now turn to properties 37(d) and (e). We begin by proving monotonicity of the settler count just for the full non-nomadic trips.

Lemma 46 Every regular bag is subsumed by a regular bag for which for all full non-nomadic trips the settler count sc is a monotonic function of the leader of the trip.

Proof We assume that F satisfies 37(a), (b) and (c) and construct F' to satisfy the lemma whilst maintaining these properties.

Suppose the lemma is not satisfied. Then there must be full trips T and U in F such that $sc.F.T > sc.F.U$ and $lead.T \leq lead.U$. Because $sc.F.T > 0$ and F satisfies 37(a), $lead.T$ is a settler. But settlers are elements of exactly one trip. We conclude that $sc.F.T > sc.F.U > 0$, both T and U have multiplicity 1 in F, and $lead.T < lead.U$.

In order to preserve property 37(b), we must choose T and U so that the settlers in $T \cup U$ form a contiguous group. This is achieved by (for example) ordering the trips by their leaders and looking for the first consecutive pair of trips T and U that violate the monotonicity property.

Rearrange the settlers in T and U so that $sc.F.T$ and $sc.F.U$ are unchanged (thus guaranteeing a regular bag) and the slowest settlers are in T and the fastest settler are in U. Using primes to denote the lowest values of T and U, we have that $lead.T' = lead.U$ (since U is the slowest settler) and $lead.U' \leq lead.T$ (since $sc.F'.U' = sc.F.U < sc.F.T$ and U' contains the fastest settler). It follows that the measure of progress (42) strictly decreases and

$$t.(lead.T') + t.(lead.U') \;\leq\; t.(lead.T) + t.(lead.U) \quad .$$

That is, F' subsumes F and repeated application of the transformation is guaranteed to terminate in a bag with the required property. Properties 37(a), (b) and (c) are clearly maintained; in particular, property 37(c) remains true because the transformation only affects full trips.

□

Lemma 47 Every regular bag F is subsumed by a regular bag such that any non-full mixed trip in F has $bnc.F$ nomads.

Proof By the foregoing lemmas, we may assume that F satisfies properties 37(a), (b) and (c), the settler count for full non-nomadic trips is a monotonic function of the leader of the trip and any non-full mixed trip in F is the fastest mixed trip. Suppose the fastest mixed trip in F is U. Suppose U is not full and $tnc.F.U < bnc.F$. (Recall that tnc.F.U is the number of nomads in U and bnc.F is the number of nomads in F.) There must be at least two trips with strictly more nomads than U. The transformation consists of two steps: the first step ensures that all mixed trips are full or no nomadic trips have bnc.F nomads; the second step ensures that no mixed trip has more nomads than U.

The first step repeats the following transformation while U is non-full, $tnc.F.U < bnc.F$ and there is a nomadic trip with bnc.F nomads.

Choose a nomadic trip T such that $tnc.F.T = bnc.F$. Let n equal bnc.F. Add tnc.F.U+1 to U and remove n from one occurrence of T. If the resulting trip has size 1, remove it from F. The transformed bag is regular and the total time does not increase. It also satisfies properties 37(a) and (b). In the transformed bag of trips, n may be a settler; if, in addition, U remains non-full the transformed bag may not satisfy 37(c). If this is the case then, before the transformation, the trip T has multiplicity 2 in F. Move person n from the second occurrence of T to U; remove the trip from F if its size is now 1. The transformed bag is regular, it satisfies 37(a) and (b), and the total time does not increase. The bag now also satisfies 37(c). If the settler count for full mixed trips is no longer a monotonic function of the leader of the trip, apply the construction in lemma 46 to reinstate the property.

Repeated application of the above transformation is guaranteed to terminate because the size of U strictly increases at each iteration.

On termination of this first step, suppose that U is still non-full and has fewer than bnc.F nomads. There is now no nomadic trip with bnc.F nomads. Because all other non-nomadic trips are full and the settler count for these trips is a monotonic function of the leader of the trip, the bag F contains a subbag of $\{\!\{U_1 \ldots U_k\}\!\}$ of k full mixed trips, for some k at least 2, each of which has bnc.F nomads; moreover, since the settler count for full non-nomadic trips is a monotonic function of the leader of the trip, these are the fastest full mixed trips in F. Let U_0 be U and assume that the trips are indexed in decreasing order of speed. (So, by 37 (c), U_0 is faster than U_1 which is faster than U_2, and so on.) Let n be tnc.F.U. Remove bnc.F from U_k and add n+1 to U_0. Now fill each of U_1 to U_k by moving the leader of U_i into U_{i+1} for each i, $0 \leq i < k$. This rearranges the settlers in the trips so that F remains regular and satisfies properties 37(a), (b) and (c). The number of settlers in U_k increases by 1 but, by choosing the largest possible value for k, this does not invalidate the property that the settler count is monotonically increasing. The number of nomads may remain constant or may be reduced by 1; in the latter case, $k = 2$ and the settler count of U_1 increases by 1. This does not invalidate the monotonicity of the settler count. The size of U remains unchanged; it has one

more nomad and one fewer settler. Repeating the process until U has $bnc.F$ nomads is therefore guaranteed to terminate with a bag of trips that satisfies all the required properties.
□

Corollary 48 Every regular bag that satisfies properties 37(a), (b) and (c) is subsumed by a regular bag that also satisfies 37(d) and 37(e)

Proof Comparing lemma 46 with 37(d), we have to show that the settler count increases monotonically with the leader of the trip for all trips and not just the full trips. We also have to show that the nomad count is a monotonically decreasing function of the leader for all non-nomadic trips.

By lemma 46, the nomad count is a monotonically decreasing function of the leader for all full non-nomadic trips since, if T is full, $sc.F.T + tnc.F.T = C$. Lemma 47 extends this property to all non-nomadic trips because any non-full nomadic trip is the fastest mixed trip (37(c)) and $bnc.F$ is the slowest nomad. It is then an immediate consequence that the settler count is a monotonically increasing function of the leader for all trips. (The settler count for nomadic trips is 0 and the nomadic trips have the fastest leaders by 37(a).)
□

Theorem 49

$$\langle \Downarrow F : Regular.F : TotTime.F \rangle \;=\; \langle \Downarrow F : Ordered.F : TotTime.F \rangle$$

where F ranges over bags of (forward) trips and the predicate $Ordered$ expresses the property that F is ordered.

Proof The lemmas and corollaries from lemma 40 through to corollary 48 establish theorem 38 and, hence, that the left side of the equation is at least its right side. Since ordered bags are by definition regular, the left side is trivially at most the right side.
□

4 Constructing an Optimal Bag of Forward Trips

Theorem 49 has transformed our problem to determining a bag of trips F that realises

$$\langle \Downarrow F \;:\; Ordered.F \;:\; TotTime.F \rangle \quad .$$

A direct way of solving the problem would be to encode some (efficient) procedure for searching the space of all ordered bags of trips whilst evaluating the total time incurred

by the bag. This was the basis of the solution in [Bac08]. But this is not necessarily the most efficient way to solve the problem. Instead of searching the space of ordered bags, an optimal solution can be determined by computing

(50) $\langle \Downarrow F : SemiOrdered.F : TotTime.F \rangle$

for some predicate $SemiOrdered$ such that

(51) $\langle \forall F :: Ordered.F \Rightarrow SemiOrdered.F \Rightarrow Regular.F \rangle$.

In words, an algorithm will correctly solve the optimisation problem if it effectively searches a space of regular bags of trips that is guaranteed to include all ordered bags. Choosing the predicate $SemiOrdered$ is dictated by implementation considerations — such as whether an integer-programming or dynamic-programming solution is desired.

In this section, we present two solutions to the problem of finding an optimal regular bag of trips. The first (section 4.4) is an integer-programming solution. By formulating the problem in this way, it can be solved using a freely available linear-programming package. (We have used the package called lp_solve.) The disadvantage of this solution method is that the time needed to calculate an optimal solution is unpredictable. The second solution formulates the problem as solving an acyclic system of simultaneous equations (a so-called "dynamic programming" problem); because the system of equations is acyclic, the equations can be solved in (best- and worst-case) time proportional to the number of terms in the equations.

Since the effectiveness of an integer-programming solution is unpredictable, our interest in it is as a semi-independent test of the correctness of our implementation of the dynamic-programming solution. The main focus of this section is therefore the dynamic-programming solution and the integer-programming solution is a by-product. We begin in section 4.1 with an explanation of the essence of dynamic programming. In brief, a dynamic-programming solution requires an inductive formulation of the solution space together with a "cost" function that respects this inductive structure. In order to realise these requirements, we introduce an encoding of a trip as a pair of numbers, which we call an "NCL pair", in section 4.2. NCL is an abbreviation for "nomad-count, lead". Section 4.3 then shows how a bag of trips is encoded as a bag of NCL pairs and how such bags are assigned costs. This encoding is the basis for both the integer-programming and dynamic-programming solutions but neither solution is an exact encoding of ordered bags of trips. The integer-programming solution discussed in section 4.4 encodes minimal requirements on "semi-ordered" bags of trips whilst the dynamic-programming solution detailed in section 4.5 encodes more stringent requirements. The key properties are discussed in section 4.3.

4.1 The Essence of Dynamic Progamming

A so-called "dynamic programming" solution to an optimisation problem

$$\langle \Downarrow s : PutativeSoln.s : Cost.s \rangle$$

in its most common form can be briefly described as finding an optimal path from one node to another in a finite, acyclic graph. There are thus three components that are crucial to such a solution method. First, the optimisation problem must be generalised by the introduction of a parameter corresponding to nodes of the graph. Second, since paths in a graph are formed of individual edges and the graph is required to be acyclic (equivalently, there is a well-founded ordering on the nodes of the graph), the (generalised) solution space must be characterised inductively[5]. That is, the set of putative solutions for a given node n in the graph must be formulated as a composite of the edges $n \mapsto n'$ from the node and the set of putative solutions for the nodes n'. Finally, the cost function must have the same inductive structure as the space of putative solutions, thus enabling it to be evaluated inductively in a topological ("bottom-up") search of the nodes of the graph. (The cost function must obey the so-called "principle of optimality".) Note that this last component is important not only to the efficient evaluation of the optimal *cost* but is also crucial to determining the optimal *solution* — it would be pointless to determine, say, the length of a shortest path between two points if the route could not also be found just as efficiently.

These three components are evident in the classic examples of dynamic programming. For example, the ubiquitous matrix-multiplication algorithm (see, for example, [Sed88]) solves the problem of determining how to minimise the number of arithmetic operations when multiplying a sequence of matrices. It does so by generalising the problem to determining how to efficiently multiply subsequences of the given sequence. The inductive structure of the solution space is inherited from the inductive structure of the subsequence relation. Finally, and crucially, the fact that addition distributes over minimum means that the cost function (the total number of arithmetic operations) has the same inductive structure as the subsequence relation.

(The matrix-multiplication algorithm is not a shortest-path algorithm in the sense that the inductive definition of the cost function is non-linear; nevertheless, it does conform to the general structure described above provided a more liberal definition of "path" is understood.)

The dynamic programming solution to the torch problem follows this pattern. The solution space (the set of ordered bags of trips) is first generalised: the number n of

[5] We prefer the word "inductively" to the more common "recursively" because "recursion" can be unrestricted, making formal reasoning difficult, whereas "induction" has a clear mathematical meaning allied to formal proof.

people is an obvious generalisation but, as anticipated in section 4.4, we add an extra parameter e called the "excess". The excess of a bag of forward trips b is the difference between $|b|-1$ and the number of return trips encoded by b; the required solution is the case that the excess is 0 but the solution method extends the solution space to cases when the excess is not 0. Crucially, by choosing to order pairs of numbers n and e lexicographically, the introduction of the excess parameter allows us to formulate the solution space inductively. It is then straightforward to exploit the fact that addition distributes over minimum (equivalently, the "principle of optimality" for shortest-path problems) to also formulate the optimal total travel time inductively — indeed, as a classic shortest-path problem. This completes the solution method: the optimal total travel time is computed in topological order of pairs n and e; the relevant choices are recorded simultaneously so that an optimal solution can be reconstructed on completion of the evaluation.

The running time of a dynamic-programming solution is dependent on the total number of terms in the inductive definition of the cost function. The best- and worst-case running times are the same and proportional to the total number of terms; the space requirements are proportional to the number of nodes in the underlying graph structure. Because the latter is not monotonic in the subset relation on the set of putative solutions, it can be more efficient to explore a larger solution space and less efficient to explore a smaller solution space! This is of relevance here. Instead of formulating the set of ordered bags of trips inductively (which is what we did in [Bac08]), it is better to construct an inductive characterisation of a set of putative solutions that encodes a superset of such bags. These are the NCL-pair bags satisfying the predicate $SemiOrdered.N$ introduced in definition 81.

4.2 Representing Trips As Pairs

If T is a trip in an ordered bag F, the nomads in T are, by 37(a), persons k where $1 \leq k \leq tnc.F.T$; the settlers in T are the settlers k such that $boss.k = lead.T$. Since the $boss$ function is monotonic (37(b)), the settlers are persons k, where $k' < k \leq lead.T$, for some k'. The value of k' is governed by 37(a), (c) and (e). This suggests the representation of a trip T by the pair $(tnc.F.T\ ,\ lead.T)$. Specifically, we define the function fwd of type $\{0..C\}\times\{2..N\} \to Trip$ by, for all $(i,j) \in \{0..C\}\times\{2..N\}$,

$$(52)\quad fwd(i,j) \;=\; \{k \mid 1 \leq k \leq i\} \cup \{k \mid i\uparrow(j-(C-i)) < k \leq j\} \quad .$$

Trip T is represented by the pair $(tnc.F.T,\ lead.T)$ in the following sense:

Lemma 53 Suppose the bag of trips F is ordered. Then

$$\langle \forall T \;:\; T \in F \;:\; T = fwd(tnc.F.T,\ lead.T) \rangle \quad .$$

Moreover, if T is a nomadic trip in F or T is a non-full mixed trip in F,

$$T = fwd(i, lead.T) \Leftarrow 0 \leq i \leq tnc.F.T \ .$$

Proof For all $T \in F$, $lead.T \geq C$ equivales T is full. (Clearly, if $lead.T < C$, T is not full. Vice-versa, by 37(c), if T is not full it is either a nomadic trip or the fastest mixed trip. In the former case, by 37(a), $lead.T < C$. In the latter case, by 37(e), (a) and (b), the people in the trip are the nomads $1\,..\,(bnc.F)$ together with persons $(bnc.F+1)\,..\,(lead.T)$; that is, $T = \{1\,..\,(lead.T)\}$. Hence, $lead.T < C$.)

This suggests a case analysis on the type of trips: pure and full mixed trips, nomadic trips and non-full mixed trips. In the case of a pure trip T, $tnc.F.T = 0$. Since, by 37(c), all pure trips are full, it follows that $lead.T \geq C$ and it is easily checked that $T = fwd(0, lead.T)$. For full mixed trips T, it is again the case that $lead.T \geq C$ and the property follows from properties 37(a) and (b) of an ordered bag.

The case of nomadic trips follows from 37(a). (For nomadic trip T, $tnc.F.T = lead.T$.) Finally, for a non-full mixed trip T, properties 37(a), (b) and (e) guarantee that the trip is $\{k \mid 1 \leq k \leq lead.T\}$, where $lead.T < C$.
□

Lemma 53 allows some flexibility in representing the nomadic trips and a non-full mixed trip which we return to later.

Definition 54 (NCL-pair bag) A function b of type $\{0..C\} \times \{2..N\} \rightarrow \mathbb{N}$ with the property that

$$(55)\ \langle \forall i,j : 0 \leq i \leq C \wedge 2 \leq j \leq N \wedge 0 < b(i,j) : i \leq j \ \wedge \ (i = C \Rightarrow j = C) \rangle$$

is called an *NCL-pair bag*.
□

"NCL" is an abbreviation of "nomad-count, lead". An NCL-pair bag b represents the bag Fwd.b where[6]

$$(56)\ Fwd.b \ = \ \langle \uplus i,j : 0 \leq i \leq C \wedge 2 \leq j \leq N : \{\!| b(i,j) * fwd(i,j) |\!\} \rangle \ .$$

The constraint (55) is necessary to guarantee that person j is the slowest person in trip $fwd(i,j)$. Specifically,

Lemma 57 If b is an NCL-pair bag then

$$\langle \forall i,j \ : \ 0 \leq i \leq C \wedge 2 \leq j \leq N \wedge 0 < b(i,j) \ : \ lead.(fwd(i,j)) = j \rangle \ .$$

[6] $\uplus$ denotes the bag-union quantifer.

It follows that

$$\langle \forall\, i,j : 0 \le i \le C \wedge 2 \le j \le N \wedge 0 < b(i,j) : time.(fwd(i,j)) = t.j \rangle$$

and

$$time.(Fwd.b) \;=\; \langle \Sigma\, i,j : 0 \le i \le C \wedge 2 \le j \le N : b(i,j) \times t.j \rangle \quad .$$

□

The lemma is easily checked by expanding the definitions.

As well as representing a bag of forward trips, an NCL-pair bag also represents a bag of return trips. Specifically, the function $rets$ with domain $\{0..C\}\times\{2..N\}$ and range the power set of the set of return trips is defined by, for all $(i,j) \in \{0..C\}\times\{2..N\}$,

$$(58)\quad rets(i,j) \;=\; \{k{:}\ 1 \le k \le i{:}\ \{k\}\}$$

and the function $Rets$ with domain $\{0..C\}\times\{2..N\} \to \mathbb{N}$ and range $Trip \to \mathbb{N}$, is defined by

$$(59)\quad Rets.b \;=\; \langle \uplus\, i,j,k : 0 \le i \le C \wedge 2 \le j \le N \wedge 1 \le k \le i : \{\!|\, b(i,j) * \{k\} \,|\!\} \rangle \quad .$$

Note that $rets(0,j)$ is the empty set, for all j.

In this way, an NCL-pair bag represents a bag of forward trips and a bag of return trips. The total trip time for the two bags is defined by the function $Cost$ where

$$(60)\quad Cost.b \;=\; \langle \Sigma\, i,j : 0 \le i \le C \wedge 2 \le j \le N : b(i,j) \times cost(i,j) \rangle$$

and

$$(61)\quad cost(i,j) \;=\; t.j + \langle \Sigma k : 1 \le k \le i : t.k \rangle \quad .$$

4.3 Representing Bags of Trips

Lemma 53 allows some flexibility in representing a forward trip by an NCL pair. For example, a nomadic trip with n nomads can be represented by (n,n) or by $(0,n)$; both represent the forward trip $\{k{:}\ 1 \le k \le n{:}\ k\}$. However, they differ in the set of return trips they represent. Specifically, $rets(n,n) = \{k{:}\ 1 \le k \le n{:}\ \{k\}\}$ whilst $rets(0,n)$ is the empty set. We now want to transform our optimisation problem from finding an optimal regular bag of trips to that of finding an NCL-pair bag that optimises the function $Cost$. There is yet more flexibility in the search space that we choose. It is not necessary that NCL-pair bags in the search space encode regular bags of trips, nor that they encode ordered bags of trips. But we do need to impose some conditions on NCL-pair bags in order that the flexibility is limited in a way that guarantees that the transformation is correct.

Until now, the number of people, N, and the capacity, C, have been implicit parameters. In anticipation of our dynamic-programming solution, it is necessary to make the parameter N explicit. The "excess" parameter, discussed earlier, will also be introduced shortly.

For our integer-programming solution, we introduce the predicate $RegularNCL.N$ on NCL-pair bags in definition 66; for our dynamic-programming solution, we introduce the predicate $SemiOrdered.N$ on NCL-pair bags in definition 81. Then, with dummy F ranging over bags of trips and dummy b ranging over NCL-pair bags, we show that

(62) $\langle \Downarrow F : Regular.F : TotTime.F \rangle$,

(63) $\langle \Downarrow b : RegularNCL.N.b : Cost.b \rangle$,

(64) $\langle \Downarrow b : SemiOrdered.N.b : Cost.b \rangle$ and

(65) $\langle \Downarrow F : Ordered.F : TotTime.F \rangle$

are all equal. We do this by an at-most and at-least proof. The proof takes the following form:

$\langle \Downarrow F : Regular.F : TotTime.F \rangle$

$\leq$ { definition 66; lemma 72 and corollary 77 }

$\langle \Downarrow b : RegularNCL.N.b : Cost.b \rangle$

$\leq$ { lemma 82 }

$\langle \Downarrow b : SemiOrdered.N.b : Cost.b \rangle$

$\leq$ { definition 81; lemma 83 }

$\langle \Downarrow F : Ordered.F : TotTime.F \rangle$.

Because we have already established the equality of (62) and (65), the equality of all four quantifications follows by transitivity and anti-symmetry of the at-most relation.

Definition 66 Suppose b is an NCL-pair bag (that is, b has type $\{0..C\} \times \{2..N\} \rightarrow \mathbb{N}$ and satisfies (55)). Then b satisfies the predicate $RegularNCL.N$ iff it satisfies two constraints. The first constraint is that the number of forward trips represented by b is one more than the number of return trips:

(67) $|Rets.b| + 1 = |Fwd.b|$.

The second constraint is that for every person the number of forward trips is one more than the number of return trips:

(68) $crosses.N.b$

where crosses.N.b is the predicate

(69) $\langle \forall k : 1 \leq k \leq N : fwdNo.b.k = 1 + retNo.b.k \rangle$

and fwdNo and retNo are defined by

(70) $fwdNo.b.k = \langle \Sigma i,j : k \in fwd(i,j) : b(i,j) \rangle$,

(71) $retNo.b.k = \langle \Sigma i,j : \{k\} \in rets(i,j) : b(i,j) \rangle$.

As the abbreviations suggest, fwdNo and retNo count the number of forward and return trips of each person.
□

We now show that

$$\langle \exists \tau : \tau \in RegularNCL.N \rightarrow Regular : \langle \forall b :: TotTime.(\tau.b) \leq Cost.b \rangle \rangle \quad .$$

Unsurprisingly, the transformation τ is the function Fwd:

Lemma 72 Suppose b satisfies RegularNCL.N. Then $RetTrips.(Fwd.b) = Rets.b$. Moreover, Fwd.b is a regular bag and, if b satisfies (55), $Cost.b = TotTime.(Fwd.b)$.

Proof First, we have for all k, $1 \leq k \leq N$,

$$\begin{array}{ll} & RetTrips.(Fwd.b).\{k\} \\ = & \quad \{ \quad \text{definition of RetTrips: (34)} \quad \} \\ & \langle \Sigma T : k \in T : Fwd.b.T \rangle - 1 \\ = & \quad \{ \quad \text{definition of Fwd: (56)} \quad \} \\ & \langle \Sigma i,j : k \in fwd(i,j) : b(i,j) \rangle - 1 \\ = & \quad \{ \quad (68) \quad \} \\ & \langle \Sigma i,j : \{k\} \in rets(i,j) : b(i,j) \rangle \\ = & \quad \{ \quad \text{definition of Rets: (59)} \quad \} \\ & Rets.b.\{k\} \quad . \end{array}$$

We conclude that

(73) $RetTrips.(Fwd.b) = Rets.b$.

To show that Fwd.b is a regular bag, we have to show that

(74) $\langle \forall i,j : 0 \leq i \leq C \wedge 2 \leq j \leq N \wedge 0 < b(i,j) : 2 \leq |fwd(i,j)| \leq C \rangle$

(that is, every forward trip involves at least two and at most C people: see (13)),

(75) $|Fwd.b| = |RetTrips.(Fwd.b)|+1$.

(that is, the total number of forward trips is one more than the total number of return trips: see (14)) and

(76) $\langle \forall k : 1 \leq k \leq N : 1 \leq f.(Fwd.b).k \rangle$

(that is, each person crosses at least once: see (12)).

Property (74) is immediate from (52). Property (75) is an immediate consequence of (67) and (73); property (76) is an immediate consequence of (68) and (73). Finally,

$$
\begin{array}{ll}
 & TotTime.(Fwd.b) \\
= & \quad \{ \quad \text{definition of } TotTime \quad \} \\
 & time.(Fwd.b)+time.(RetTrips.(Fwd.b)) \\
= & \quad \{ \quad RetTrips.(Fwd.b)=Rets.b \quad \} \\
 & time.(Fwd.b)+time.(Rets.b) \\
= & \quad \{ \quad \text{lemma 57, definition of } time \text{ and } Rets \quad \} \\
 & \langle \Sigma i,j : 0 \leq i \leq C \wedge 2 \leq j \leq N : b(i,j) \times (t.j + \langle \Sigma k : 1 \leq k \leq i : t.k \rangle) \rangle \\
= & \quad \{ \quad \text{definition} \quad \} \\
 & Cost.b \ .
\end{array}
$$

□

Corollary 77

$$\langle \Downarrow F : Regular.F : TotTime.F \rangle \ \leq \ \langle \Downarrow b : RegularNCL.b : Cost.b \rangle$$

Proof Immediate from lemma 72 and the general theory of transforming optimisation problems discussed in section 1.1.
□

We now turn to the definition of "semi-ordered" NCL-pair bags. Because dynamic programming inevitably involves an inductive definition of the search space, we need to make the inductive parameters explicit. The functions we introduce are parameterised by the number of people n and the so-called "excess". Specifically, suppose b is an NCL-pair bag. Then we define the function $excess$ of type $\mathbb{N} \to (\{0..C\} \times \{2..N\} \to \mathbb{N}) \to Int$ by

(78) $excess.n.b = \langle \Sigma i,j : 0 \leq i \leq C \wedge 2 \leq j \leq n : b(i,j) \times (i-1) \rangle + 1$.

Informally, the value of $excess.n.b$ is the difference between the number of return trips encoded by b and the number of return trips needed to effect the bag of forward trips in $Fwd.b$.

Our definition of semi-ordered is the conjunction of "quasi-ordered" and the excess is 0. In order to defined "quasi-ordered", we define the predicate $service$ by

(79) $service.n.b \;=\; \langle \forall k : 2 \leq k \leq n : 0 \leq excess.k.b \rangle$.

and the predicate $allsettler$ by

(80) $allsettler.n.b \;=\; \langle \forall k : C < k \leq n : fwdNo.b.k \;=\; 1 \rangle$.

Then:

Definition 81 Suppose b is an NCL-pair bag and suppose $2 \leq n \leq N$. Then, we define the predicate $QuasiOrdered.n$ by

$$QuasiOrdered.n.b \;\equiv\; crosses.n.b \wedge service.n.b \wedge allsettler.n.b$$

and the predicate $SemiOrdered.n$ by

$$SemiOrdered.n.b \;\equiv\; QuasiOrdered.n.b \wedge (excess.n.b = 0) \;.$$

(The requirement $service.n.b$ does not guarantee that $Fwd.b$ is an ordered bag but it does guarantee that any pure trips in $Fwd.b$ are "serviced" by trips that involve faster people. That is, all the properties of an ordered bag are satisfied except for the property 37(d) — the number of settlers may not be an increasing function of the leader of a trip.)
□

Lemma 82

$$\langle \Downarrow b : RegularNCL.N.b : Cost.b \rangle \;\leq\; \langle \Downarrow b : SemiOrdered.N.b : Cost.b \rangle \;.$$

Proof We have, for all N and NCL-pair bags b,

$\quad SemiOrdered.N.b$

$\Rightarrow$ $\quad \{$ definition 81, weakening $\}$

$\quad crosses.N.b \wedge (excess.N.b = 0)$

$=$ $\quad \{$ definition 78 $\}$

$\quad crosses.N.b \wedge (\langle \Sigma i,j : 0 \leq i \leq C \wedge 2 \leq j \leq N : b(i,j) \times (i-1) \rangle + 1 = 0)$

$=$ $\quad \{$ $\langle \Sigma i,j : 0 \leq i \leq C \wedge 2 \leq j \leq N : b(i,j) \times i \rangle \;=\; |Rets.b|$,

$$\langle \Sigma\, i,j : 0 \le i \le C \wedge 2 \le j \le N : b(i,j) \times 1 \rangle \;=\; |Fwd.b| \quad \}$$

$$crosses.N.b \wedge (|Rets.b| - |Fwd.b| + 1 = 0)$$

$$= \quad \{ \quad \text{definition 66} \quad \}$$

$$RegularNCL.N.b \quad .$$

That is, $SemiOrdered$ implies $RegularNCL$ everywhere. The lemma follows trivially: the transformation from semi-ordered NCL-pair bags to regular NCL-pair bags is the identity function.

□

We now show that

$$\langle \exists \tau : \tau \in Ordered \to SemiOrdered.N : \langle \forall F :: Cost.(\tau.F) \le TotTime.F \rangle \rangle \quad .$$

It is at this point that the flexibility in the choice of NCL pairs discussed in lemma 53 becomes important.

Lemma 83 Suppose F is an ordered bag. Define the bag b as follows. First, among all the trips with $bnc.F$ nomads choose one that has the smallest $lead$ value. Call this $lasttrip$. (The choice can always be made by definition of $bnc.F$; it is called $lasttrip$ because it is always a candidate for selection as the last trip when executing the algorithm given in the proof of theorem 30 to transform F into a sequence of forward and return trips.) Let G be the result of removing one occurrence of $lasttrip$ from F. Define b by

$$b \;=\; \langle \uplus T : T \in G : \{\!|G.T * (tnc.F.T,\, lead.T)|\!\} \rangle \;\uplus\; \{\!|1 * (0,\, lead.lasttrip)|\!\} \quad .$$

Then b is an NCL-pair bag and satisfies $SemiOrdered.N$. Moreover, $Cost.b = TotTime.F$.

Proof We observe first that

$$(84) \quad lasttrip \;=\; fwd(0,\, lead.lasttrip) \quad .$$

The proof is by a case analysis: $lasttrip$ is either a nomadic trip or a mixed trip. (It can't be a pure trip because $0 < bnc.F$; see (26).) In both cases, we have

$$fwd(0,\, lead.lasttrip)$$

$$= \quad \{ \quad \text{definition of } fwd \quad \}$$

$$\{k \mid 1 \le k \le 0\} \cup \{k \mid 0\uparrow(lead.lasttrip - C) < k \le lead.lasttrip\}$$

$$= \quad \{ \quad 1 \le k \le 0 \;\equiv\; \text{false, set calculus} \quad \}$$

$$\{k \mid 0\uparrow(lead.lasttrip - C) < k \le lead.lasttrip\} \quad .$$

Suppose $lasttrip$ is a nomadic trip. Then

$\{k \mid 0\uparrow(lead.lasttrip - C) < k \leq lead.lasttrip\}$

$=$ { lasttrip is a nomadic trip and F is ordered;
so, by choice of lasttrip and 37(a), $lead.lasttrip = bnc.F \leq C$ }

$\{k \mid 0 < k \leq bnc.F\}$

$=$ { lasttrip has bnc.F nomads and F is ordered; 37(a) }

lasttrip .

Suppose lasttrip is a mixed trip. Then

$\{k \mid 0\uparrow(lead.lasttrip - C) < k \leq lead.lasttrip\}$

$=$ { lasttrip has bnc.F nomads and is the fastest mixed trip,
also F is ordered; so, by 37(b), $lead.lasttrip \leq C$ }

$\{k \mid 0 < k \leq lead.lasttrip\}$

$=$ { arithmetic, set union }

$\{k \mid 1 \leq k \leq bnc.F\} \cup \{k \mid bnc.F < k \leq lead.lasttrip\}$

$=$ { $lead.lasttrip \leq C$, definition of fwd }

$fwd(bnc.F, lead.lasttrip)$

$=$ { lasttrip has bnc.F nomads, lemma 53 }

lasttrip .

We have thus verified (84) in both cases.

Since F is regular, it follows that b has the right type.

We now have to establish properties (55), (67), (68), (79) and (80). That b satisfies (55) is an immediate consequence of property 37(a) of an ordered bag (the nomads in T are persons 1 thru tnc.F.T).

That b satisfies (67) and (68) is a consequence of the fact that $Fwd.b = F$ and $Rets.b = RetTrips.F$. The properties are thus inherited from the properties of the return trips of F. We prove the two equalities as follows.

$Fwd.b$

$=$ { definition }

$\langle \uplus T : T \in G : \{\!| G.T * (tnc.F.T, lead.T) |\!\} \rangle \uplus \{\!| 1 * fwd.(0, lead.lasttrip) |\!\}$

$=$ { lemma 53 and (84) }

$\langle \uplus T : T \in G : \{\!| G.T * T |\!\} \rangle \uplus \{\!| 1 * lasttrip |\!\}$

$=$ { definition of G }

F .

Also,

$Rets.b$

$=$ { definition of b and Rets: (59) }

$\langle \uplus i,j,k : 1 \le k \le i : \langle \uplus T : T \in G \wedge (tnc.F.T, lead.T) = (i,j) : \{\!|G.T * \{k\}|\!\} \rangle \rangle$

$=$ { one-point rule }

$\langle \uplus T : T \in G : \langle \uplus k : 1 \le k \le tnc.F.T : \{\!|G.T * \{k\}|\!\} \rangle \rangle$

$=$ { $lasttrip.F$ has $bnc.F$ nomads }

$\langle \uplus T : T \in F : \langle \uplus k : 1 \le k \le tnc.F.T : \{\!|F.T * \{k\}|\!\} \rangle \rangle$

$-$ $\langle \uplus k : 1 \le k \le bnc.F : \{\!|1 * \{k\}|\!\} \rangle$

$=$ { F is ordered; so, for all trips T, the nomads in T are persons k such that $1 \le k \le tnc.F.T$, definition of f.F: (1) }

$\langle \uplus k : 1 \le k \le bnc.F : \{\!|(f.F.k - 1) * \{k\}|\!\} \rangle$

$=$ { definition of r.F: (2) }

$\langle \uplus k : 1 \le k \le bnc.F : \{\!|r.F.k * \{k\}|\!\} \rangle$

$=$ { definition of RetTrips: (34) }

$RetTrips.F$.

Consequently,

$TotTime.F$

$=$ { (35) }

$time.F + time.(RetTrips.F)$

$=$ { above calculations }

$time.(Fwd.b) + time.(Rets.b)$

$=$ { lemma 57 }

$Cost.b$.

We now show that b satisfies (79).

A brief, informal summary of the proof is that the value of $excess.n.b$ is decreased by pairs $(0, n)$ such that $0 < b(0, n)$ and is increased by pairs (i, n) such that $0 < b(i, n)$

and $2 \leq i$. That is, the excess is decreased by $lasttrip$ and by pure trips and is increased by nomadic and mixed trips. If F is ordered, $lasttrip$ is either a slowest nomadic trip or the fastest mixed trip, and the pure trips are the slowest trips. Hence, the value of $excess.n.b$ is always at least 0.

We split the formal proof into three cases determined by the value of $lead.lasttrip$ and the number of pure trips in F. The first case is $n \leq lead.lasttrip$, the second case is $lead.lasttrip < n \leq N - C \times pc.F$ and the final case is $N - C \times pc.F < n \leq N$.

We first show that $lead.lasttrip \leq N - C \times pc.F$. We use proof by contradiction:

$$
\begin{array}{ll}
 & lead.lasttrip > N - C \times pc.F \\
= & \quad \{ \quad \text{arithmetic} \quad \} \\
 & \langle \exists j \ :: \ N - C \times pc.F < j \leq lead.lasttrip \rangle \\
\Rightarrow & \quad \{ \quad \text{by 37(c) and definition of } lasttrip, \\
 & \qquad j \leq lead.lasttrip \Rightarrow j \in lasttrip \quad \} \\
 & \langle \exists j \ : \ N - C \times pc.F < j \ : \ j \in lasttrip \rangle \\
\Rightarrow & \quad \{ \quad 37(c) \text{ and } (d) \quad \} \\
 & lasttrip \text{ is a pure trip} \\
\Rightarrow & \quad \{ \quad \text{definition of } lasttrip \quad \} \\
 & bnc.F = 0 \\
= & \quad \{ \quad (26) \text{ and } C < N \quad \} \\
 & \text{false} \quad .
\end{array}
$$

Now recall the definition of $lasttrip$: it is a trip in F that has the smallest lead value among those trips with $bnc.F$ nomads. Thus $lasttrip$ is either a nomadic trip or, because of property 37(e), it is the fastest mixed trip. By a simple case analysis, it follows that, for all trips T in F,

(85) $lead.T \leq lead.lasttrip \quad \Rightarrow \quad T = lasttrip \ \vee \ 0 < tnc.T$

and

(86) $b(0, lead.lasttrip) = 1$.

Now we begin our analysis of the three cases identified above. First, assume $n \leq lead.lasttrip$. Then

$$0 \leq i \leq C \ \wedge \ 2 \leq j \leq n \ \wedge \ 0 < b(i,j)$$

$$
\begin{array}{ll}
= & \{\quad \text{definition of b, 37(a)} \quad\} \\
& \langle \exists T : T \in G : i = tnc.F.T \wedge j = lead.T \wedge 2 \leq lead.T \leq n \rangle \\
\vee & (0 = i \wedge j = lead.lasttrip) \\
\Rightarrow & \{\quad \text{assumption: } n \leq lead.lasttrip \quad\} \\
& \langle \exists T : T \in G : i = tnc.F.T \wedge j = lead.T \wedge 2 \leq lead.T \leq lead.lasttrip \rangle \\
\vee & (0 = i \wedge j = lead.lasttrip) \\
\Rightarrow & \{\quad (85) \quad\} \\
& 0 < i \vee (0 = i \wedge j = lead.lasttrip) \\
\Rightarrow & \{\quad (86) \quad\} \\
& 0 \leq i-1 \ \vee \ b(i,j) = 1 \ .
\end{array}
$$

For $lead.lasttrip < n \leq N - C \times pc.F$, abbreviating $lead.lasttrip$ to ll, we have

$$
\begin{array}{ll}
& excess.n.b \\
= & \{\quad \text{definition: (78), range splitting} \quad\} \\
& excess.ll.b + \langle \Sigma\, i,j : 0 \leq i \leq C \wedge ll < j \leq n : b(i,j) \times (i-1) \rangle \\
\geq & \{\quad excess.ll.b \geq 0 \quad\} \\
& \langle \Sigma\, i,j : 0 \leq i \leq C \wedge ll < j \leq n : b(i,j) \times (i-1) \rangle \\
\geq & \{\quad \text{assumption: } ll < n \leq N - C \times pc.F \text{; so, by 37(d),} \\
& \qquad \langle \forall T : T \in F \wedge ll < lead.T \leq n : 0 < tnc.F.T \rangle \quad \text{; definition of b} \quad\} \\
& 0 \ .
\end{array}
$$

and in the case that $N - C \times pc.F < n \leq N$, we have

$$
\begin{array}{ll}
& excess.n.b \\
= & \{\quad \text{definition: (78), range splitting} \quad\} \\
& excess.N.b - \langle \Sigma\, i,j : 0 \leq i \leq C \wedge n < j \leq N : b(i,j) \times (i-1) \rangle \\
= & \{\quad excess.N.b = 0 \text{, negation distributes through summation} \quad\} \\
& \langle \Sigma\, i,j : 0 \leq i \leq C \wedge n < j \leq N : b(i,j) \times (1-i) \rangle \\
= & \{\quad \text{assumption: } n \geq N - C \times pc.F \text{; so, by 37(d),} \\
& \qquad \langle \forall T : T \in F \wedge n < lead.T \leq N : 0 = tnc.F.T \rangle \quad \text{; definition of b} \quad\} \\
& \langle \Sigma\, i,j : 0 \leq i \leq C \wedge n \leq j < N : b(i,j) \rangle
\end{array}
$$

$\geq$ { $b(i,j) \geq 0$ }

0 .

This completes the proof of the fact that b satisfies (79).

Finally, b satisfies (80) because F is ordered and so satisfies 37(a). (Everyone crosses and the nomads —the non-settlers— are persons 1 thru tnc.F, where $tnc.F \leq C$.)
□

Corollary 87 Suppose F is an ordered bag of trips. Then

$$\langle \Downarrow b : SemiOrdered.N.b : Cost.b \rangle \;\leq\; \langle \Downarrow F : Ordered.F : TotTime.F \rangle \quad .$$

Proof Immediate from lemma 83 and the general theory of transforming optimisation problems discussed in section 1.1.
□

Combining corollary 87, lemma 82 and corollary 77 with the fact that (62) and (65) are equal, we conclude:

Theorem 88 The quantifications (62), (63), (64) and (65) are all equal to the minimum time required for N people to cross the bridge under the given constraints.

□

4.4 Integer-Programming Solution

The problem of evaluating $\langle \Downarrow b : RegularNCL.b : Cost.b \rangle$ is already in the form of an integer programming problem. The equations (67) and (68) are both linear constraints on b and the constraint (55) simply reduces the number of variables in the integer-programming problem. The function Cost is also clearly a linear function.

Optionally, a third constraint can be added:

$$\langle \forall i,j \;:\; 2 \leq i \leq C \wedge C < j \leq N \;:\; b(i,j) \leq 1 \rangle \quad .$$

This constraint is implied by (80).

4.4.1 Example

Let us give a simple, concrete example. We take the classic case of 4 people and a bridge of capacity 2. That is, $N = 4$ and $C = 2$. We assume that $0 \leq t.1 \leq t.2 \leq t.3 \leq t.4$. The problem is to compute a function b indexed by pairs (i,j) that satisfy the constraints: $0 \leq i \leq 2$, $2 \leq j \leq 4$, $i \leq j$ and $i = 2 \Rightarrow j = 2$ (cf. (55)). In the table below, the first two columns list values of i and j satisfying these constraints. The remaining columns show

the values of four functions on such pairs, the column header giving the name of the function.

i	j	$fwd(i,j)$	$rets(i,j)$	$time.(fwd(i,j))$	$time.(rets(i,j))$
0	2	{1,2}	$\emptyset$	t.2	0
0	3	{2,3}	$\emptyset$	t.3	0
0	4	{3,4}	$\emptyset$	t.4	0
1	2	{1,2}	{{1}}	t.2	t.1
1	3	{1,3}	{{1}}	t.3	t.1
1	4	{1,4}	{{1}}	t.4	t.1
2	2	{1,2}	{{1},{2}}	t.2	t.1 + t.2

Now the problem is to minimise Cost.b, i.e. to minimise

$$\begin{aligned} & b(0,2)\times(t.2+0)+b(0,3)\times(t.3+0)+b(0,4)\times(t.4+0) \\ + \; & b(1,2)\times(t.2+t.1)+b(1,3)\times(t.3+t.1)+b(1,4)\times(t.4+t.1) \\ + \; & b(2,2)\times(t.2+(t.1+t.2)) \end{aligned}$$

subject to two sets of constraints. The first set is the constraints given by (68). These are given without simplification below. In order, the constraints specify that persons 1, 2, 3 and 4 each make one more forward trip than return trips.

$$b(0,2)+b(1,2)+b(1,3)+b(1,4)+b(2,2) = 1+b(1,2)+b(1,3)+b(1,4)+b(2,2)$$

$$b(0,2)+b(0,3)+b(1,2)+b(2,2) = 1+b(2,2)$$

$$b(0,3)+b(0,4)+b(1,3) = 1$$

$$b(0,4)+b(1,4) = 1$$

(Note that the first equation simplifies to $b(0,2)=1$ and the second then simplifies to $b(0,3)+b(1,2) = 0$.) The second set of constraints has just one element obtained by instantiating the constraint (78). It is too long to write out in full. Simplified, it is the equation:

$$b(0,2)+b(0,3)+b(0,4)-b(2,2)-1 = 0 \quad .$$

Performing the obvious simplifications by hand[7], the problem becomes to minimise

[7] In our implementation, these simplifications are done automatically in order to minimise the number of variables and the total size of the equations in the linear-programming model. See the appendix for an example.

$$t.2 + b(0,4)\times t.4 + b(1,3)\times(t.3+t.1) + b(1,4)\times(t.4+t.1) + b(2,2)\times(t.1+2\times t.2)$$

subject to

$$b(0,4)+b(1,3) = 1$$

$$b(0,4)+b(1,4) = 1$$

and

$$b(0,4) = b(2,2) \quad .$$

This corresponds to the standard solution to the problem [Rot02, Bac11]: the choice is between $b(0,4)=1$ with time $t.2+t.4+(t.1+2\times t.2)$ and $b(1,3)=b(1,4)=1$ with time $t.2+(t.3+t.1)+(t.4+t.1)$. That is, the slowest person, person 4, crosses in a pure trip if $2\times t.2$ is at most $t.3+t.1$ and in a mixed trip (with person 1) if $t.3+t.1$ is at most $2\times t.2$.

Of course, knowing the solution to the integer-programming problem is insufficient: the desired solution is how to get the people across, not how long it will take! This is why the calculation of $Fwd.b$ (and its correctness as expressed in lemma 72) and the algorithm to calculate a regular sequence from a regular bag (see theorem 30) are indispensable.

The appendix shows the lp_solve code for an example with 11 people and a capacity of 3.

4.5 Dynamic-Programming Solution

In this section, we show how to use dynamic programming to solve the torch problem.

Let dummy b range over NCL-pair bags. Then our dynamic-programming solution determines the value of B where

$$B \quad = \quad \langle \Downarrow b : QuasiOrdered.N.b \wedge (excess.N.b=0) : Cost.b \rangle \quad .$$

As shown in section 4.3, B equals the optimal total travel time to get all N people across a bridge of capacity C where their individual crossing times are given by the function t.

Since the "excess" plays a crucial role, we begin by introducing a parameter e in the equations for B:

$$B \quad = \quad s(N,0)$$

where $s(n,e)$ is expressed formally as follows:

$$
\begin{array}{ll}
& s(n,e) \\
= & \langle \Downarrow b \\
& \quad : \quad QuasiOrdered.n.b \\
& \qquad \wedge \quad (excess.n.b = e) \\
& \qquad \wedge \quad \langle \forall i,j : 0 \le i \le C \wedge n < j \le N : b(i,j) = 0 \rangle \\
& \quad : \quad Cost.b \\
& \rangle \ .
\end{array}
$$

The dummy b continues to range over NCL-pair bags. The function s is parameterised by variables n and e where $2 \le n \le N$. The requirement $service.n.b$ in the definition of $QuasiOrdered.n$ means that $s(n,e)$ is only defined for values of e such that $0 \le e$. Moreover, since

$$
\begin{array}{ll}
& excess.n.b \\
= & \quad \{ \quad \text{definition: (78), arithmetic} \quad \} \\
& excess.N.b - \langle \Sigma i,j : 0 \le i \le C \wedge n < j \le N : b(i,j) \times (i-1) \rangle \\
= & \quad \{ \quad excess.N.b = 0, \text{ arithmetic} \quad \} \\
& \langle \Sigma j : n < j \le N : b(0,j) \rangle - \langle \Sigma i,j : 2 \le i \le C \wedge n < j \le N : b(i,j) \times (i-1) \rangle \\
\le & \quad \{ \quad \text{arithmetic} \quad \} \\
& \langle \Sigma j : n < j \le N : b(0,j) \rangle \\
\le & \quad \{ \quad allsettler.N.b \text{ and } j \in fwd(0,k) \equiv k-C < j \le k \quad \} \\
& \left\lfloor \frac{N-n}{C} \right\rfloor
\end{array}
$$

the requirement also implies the upper bound $e \le \left\lfloor \frac{N-n}{C} \right\rfloor$.

The first step is to identify the inductive structure of the range of variable b. Observing that

$$
excess.n.(b \uplus \{\!|(i,n)|\!\}) \ = \ excess.n.b + (i-1)
$$

we infer two equations for the range of dummy b. First for $C < n$ it is clear from (71) that $retNo.b.n = 0$. From (70) it follows that $fwdNo.b.n = 1$. In order to meet this requirement and the requirement in the definition of $s(n,e)$ that $b(i,j)$ is 0 for all j such that $n < j \le N$, it is necessary that

$$
\langle \exists i,b' : 0 \le i < C : b = b' \uplus \{\!|(i,n)|\!\} \rangle \quad .
$$

That is, b must encode a trip led by person n with i nomads, for some i. The number of nomads i must also be at most $(e{+}1)$, otherwise it is impossible to choose b' so that the predicate *service* is satisfied everywhere. Moreover, the remaining $n{-}(C{-}i)$ people must be at least 2 and must be "serviced" by b'. So, we obtain that for $C<n$ and $0\leq e\leq \lfloor\frac{N-n}{C}\rfloor$,

$$
\begin{array}{lll}
& s(n,e) & \\
= & \quad \langle \Downarrow i & : \ 0\leq i\leq (C{-}1)\downarrow(e{+}1) \ \wedge \ 2\leq n{-}(C{-}i) \\
& & : \ \langle \Downarrow b' \\
& & \quad : \quad \mathit{QuasiOrdered}.(n{-}(C{-}i)).b' \\
& & \qquad \wedge \ (\mathit{excess}.(n{-}(C{-}i)).b' = e{+}1{-}i) \\
& & \qquad \wedge \ \langle \forall k,j \ : \ 0\leq k\leq C \ \wedge \ n{-}(C{-}i)<j\leq N \ : \ b'(k,j)=0\rangle \\
& & \quad : \quad \mathit{Cost}.(b' \uplus \{\!|(i,n)|\!\}) \\
& & \rangle \\
& \rangle & .
\end{array}
$$

By using the equality $\mathit{Cost}.(b' \uplus \{\!|(i,n)|\!\}) = \mathit{Cost}.b' + \mathit{cost}(i,n)$ and factoring out the term $\mathit{cost}(i,n)$, we thus obtain that, for $C<n$ and $0\leq e\leq \lfloor\frac{N-n}{C}\rfloor$,

$$
\begin{array}{lll}
s(n,e) \ = & \langle \Downarrow i & : \ 0\uparrow(C{+}2{-}n) \leq i \leq (C{-}1)\downarrow(e{+}1) \\
& & : \ s(n{-}(C{-}i)\,,e{+}1{-}i) \ + \ \mathit{cost}(i,n) \\
& \rangle & .
\end{array}
$$

(It is necessary to check that $e{+}1{-}i\leq \left\lfloor\frac{N-(n-(C-i))}{C}\right\rfloor$ if $e\leq \lfloor\frac{N-n}{C}\rfloor$ and $0\leq i$. This is easily done.) Now we formulate equations for the case that $n\leq C$. In this case, the *allsettler* constraint in the definition of *QuasiOrdered* is no longer relevant. If the excess is 0 and there are at most C people left, there is only one possible choice for the NCL-pair bag b, namely $\{\!|(0,n)|\!\}$. So, for $n\leq C$,

$$s(n,0) \ = \ \mathit{cost}(0,n) \quad .$$

If the excess is greater than 0 and n is at most C, any NCL-pair bag b in the definition of $s(n,e)$ must include a pair (i,n) where $2\leq i$ (so that the excess is decreased) and $i\leq e{+}1$ (so that the excess remains positive). In order to guarantee the predicate *crosses*, the number i of return trips becomes the number of people remaining to cross again. So, for $2\leq n\leq C$ and $0<e\leq \lfloor\frac{N-n}{C}\rfloor$,

$$s(n,e) \ = \ \langle \Downarrow i : 2\leq i\leq (e{+}1)\downarrow n : s(i\,,e{-}i{+}1)+\mathit{cost}(i,n)\rangle \quad .$$

The above three inductive equations constitute the dynamic-programming solution to the torch problem.

4.5.1 Solving the Equations

Finding the total travel time incurred by an optimal bag of forward trips is achieved by solving the equations in s. First, the $(N-1)\times(C+1)$ values of the $cost$ function are tabulated using $N\times(C+1)$ additions. Then, for n in the range $2..C$, the $(C-1)\times(\lfloor\frac{N-n}{C}\rfloor+1)$ values of $s(n,e)$ can be computed in order of increasing n and increasing e. This requires at most $(C-1)\times(\lfloor\frac{N-2}{C}\rfloor+1)\times C$ additions/comparisons. Next, $\lfloor\frac{N-n}{C}\rfloor+1$ values of $s(n,e)$ must be computed for each n greater than C. This requires at most $(N-C)\times(\lfloor\frac{N-2}{C}\rfloor+1)\times C$ additions/comparisons. In this way, the optimal solution can be determined with $O(N^2)$ additions/comparisons. The space requirements are dictated by the need to store values of the function s: in total, at most $O(\frac{N^2}{C})$ storage locations are required. For small values of C (for example, C equal to 2) the space requirement is $O(N^2)$ but it is less when C is proportional to N.

A bag of pairs that realises the minimum time is calculated at the same time as the equations are solved: whenever $s(n,e)$ is evaluated, for some n and e, a value of i that realises the minimum quantification in the equation for s is recorded. Once the value of $s(N,0)$ has been calculated, the bag of pairs can be accumulated by retracing the sequence of choices.

Java code for the complete algorithm is given in the appendix. The code uses the array `nmds` (short for "nomads"), indexed by the number of people and the excess, to record a value of i that realises the minimum in the equations for s given above. The code for computing a sequence of crossings (rather than just the bag of forward trips) is also included.

The evaluation of $s(N,0)$ is equivalent to a shortest-path problem. The nodes in the graph are pairs (n,e) where $2\le n\le N$ and $0\le e\le\lfloor\frac{N-n}{C}\rfloor$. There is also a single terminal node. The start node is the node $(N,0)$. The edges are defined by $cost$ terms in the equations. There is an edge of length $cost(0,n)$ from each node $(n,0)$, where $n\le C$, to the terminal node. For pairs (n,e) where $C<n$, there is an edge of length $cost(i,n)$ from the node (n,e) to the node $(n-(C-i),\,e+1-i)$ for each i such that $0\uparrow(C+2-n)\le i\le(C-1)\downarrow(e+1)$. For pairs (n,e) where $n\le C$ and $0<e$ there is an edge of length $cost(i,n)$ from the node (n,e) to the node $(i,\,e+1-i)$ for each i such that $2\le i\le(e+1)\downarrow n$. That s is inductively defined is equivalent to the graph being acyclic. The algorithm sketched above is a topological search algorithm with best- and worst-case time complexity proportional to the number of edges in the graph, and best- and worst-case space complexity proportional to the number of nodes.

4.6 Evaluation

In order to evaluate our dynamic-programming solution, we have implemented it and the earlier, more complicated solution [Bac08] in Java; we have also implemented the integer-programming solution. A substantial number of test cases was used to evaluate the efficiency of the different solutions and the results compared. For some tests we forced the number of pure trips to be at least a certain number by choosing extremely large crossing times for a certain proportion of the people. (To force at least p pure trips, the crossing times for $p \times C$ people are chosen to be, say, $10^3 \times C$ whilst the crossing times for the remaining people are chosen randomly but much smaller — for example, in the range 1 to 10^2.) Further details of the methodology used for the comparisons are in the second author's MSc dissertation [Tru11]. The results of the comparison in [Tru11] are, however, now out of date because we have since improved both the dynamic-programming algorithm and the generation of the integer-programming model.

The primary goal of implementing the integer-programming solution was to have a semi-independent test of the correctness of the dynamic-programming solution. The test is independent in the sense that we used a publicly available package. However, it cannot be regarded as completely independent because both solutions rely on the same representation of a solution as a bag of pairs of numbers. Our implementation generates from given input values an integer-programming problem expressed in the syntax of lp_solve. See the appendix for an example. The number of variables in the lp_solve program is approximately $N \times C$.

This element of the evaluation was successful: all three methods returned the same value for the optimal value of the total crossing time in all tests (more precisely, in all tests where all three methods successfully ran to completion).

We implemented two versions of the program to generate the integer-programming model. The first version, discussed in [Tru11], made no attempt to minimise the number of variables; a second version identified variables that are necessarily zero in order to minimise the number of variables and the size of the equations in the generated model. For the first version, the execution time of lp_solve was dependent on the individual crossing times and somewhat unstable. With small values of the capacity (less than 5) the execution time was commensurate with the execution time of the dynamic-programming solution; when the capacity was increased to 50 the execution time for 2000 people was variable up to a maximum of about 30 minutes. The time taken to generate the lp_solve file is not included; it was insignificant. For larger values of the capacity and/or number of people, the unpredictability of the execution time hindered further tests. We were able to test the second version on the same input data as used for the dynamic programming solution, i.e. up to 25000 people and with capacity ranging from 2 to 50. Beyond these numbers, space limitations on the program to generate the model prevented further tests.

The execution time of lp_solve was slower than the dynamic-programming solution and, for the largest problems, two to three minutes. Even so, we were very surprised by the size of the problems that could be successfully solved.

The dynamic-programming solution is more efficient than the integer-programming solution, a major factor being that the best- and worst-case execution times of a dynamic-programming solution are always the same. Our earlier algorithm [Bac08] was comparable to the integer-programming solution but its significantly larger space requirements were prohibitive for large values of N and/or C. The algorithm presented in this paper is also limited by space: we exceeded the available heap storage with N equal to 50000 and C equal to 50 but solutions were found (in seconds) for N equal to 25000. In general, in spite of the quadratic complexity of the algorithm, execution times were always seconds rather than minutes.

Because the problem reduces to a shortest-path problem, it is possible to ignore the fact that the graph is acyclic and use an alternative shortest-path algorithm. Such an algorithm may have better best-case performance. We haven't investigated the relative merits of such a solution method. The greatest potential benefit is to improve the best-case space requirements (although it is unlikely that the worst-case space requirements would be improved).

5 Conclusion

Puzzles have long been used to inspire further learning and to test problem-solving skills. The classic torch problem of 4 people and a bridge of capacity 2, usually formulated with specific crossing times like 1, 2, 5 and 10, is an example that has apparently been used in job interviews for major companies. But isolated examples have little long-term educational value. Once the solution has been seen, the classic torch problem is quickly dismissed. By introducing input parameters for the crossing times, the problem becomes an interesting introduction to conditional statements. (See section 4.4.1.)

A much bigger and more challenging step is to also parameterise the number of people. The general capacity-2 problem offers a very good example of algorithmic problem solving. The problem is easily understood but finding an efficient solution is very challenging. Most importantly, it demonstrates that "obvious" solutions may be incorrect. See [Rot02] for publications and web links. Obtaining a correct solution demands particular attention to the avoidance of unnecessary detail. Like the capacity-C problem, the solution is obtained by focusing on just the forward trips[8]. The problem is sometimes used in a course on algorithmic problem solving [Bac11] for entry-level Computer Science students at the University of Nottingham.

[8]The importance of bags versus sequences is mentioned by Rote [Rot02] in a footnote.

The capacity-C problem appears to be much more difficult than the capacity-2 problem. By far the greatest part of this paper has been devoted to deriving an algorithm that is correct by construction; the final (Java) implementation of the dynamic programming algorithm is quite short. This is not atypical. The late Edsger W. Dijkstra [Dij76] observed that the derivation of an algorithm from its (formal) specification is typically an order of magnitude longer than the algorithm itself. This is seldom reflected in current textbooks on algorithm design but it is indeed our own experience.

That the capacity-C problem can be solved in time proportional to the square of the number of people appears to be new. The problem is mentioned on some (non-peer-reviewed) websites but none offers a practical solution. Our solution could be used in advanced courses on algorithm design and/or operations research as a detailed illustration of the complexities of optimisation problems and their solutions. A number of facets of the solution are educational: the reduction in size of the search space by a focus on bags of forward trips rather than sequences of forward and return trips, the choice of a suitable representation of bags of forward trips, the formulation as an integer-programming problem, and the various components essential to a dynamic-programming solution. It is disappointing that we have not been able to discover a linear-time or (more importantly as it turns out) linear-space algorithm. When the capacity is 2, our theorem 49 reduces to Rote's [Rot02] theorem 2 (albeit formulated using different terminology). A greedy algorithm, discussed by Rote [Rot02], is then easily derived since the pure trips are in (1–1) correspondence with the nomadic trips. Indeed, a sub-linear, binary search can be used to determine the optimal number of pure trips once people have been sorted in order of crossing time [Bac11]. This simplification of the algorithm is reflected in a very regular structure of the graph underlying the dynamic-programming algorithm. For capacities greater than 2, it seems plausible that similar structural properties can be exploited in order to obtain a more efficient algorithm but such an algorithm has so far eluded us. Note that we do not expect other well-known shortest-path algorithms to improve on the worst-case efficiency of our dynamic-programming solution although they may offer some improvement in the best case.

Acknowledgements We wish to express our thanks to the following people: Diethard Michaelis for stressing the importance of regularity, for suggestions on naming, for reading and commenting on drafts of the conference paper and for his efforts to improve on notation; Arjan Mooij for helping prove that any regular bag of forward trips can be transformed to a putative sequence; João Ferreira and Arjan Mooij for suggesting improvements to some calculations; Tom Verhoeff and Günter Rote for providing bibliographic information; the anonymous referees (of the original conference paper and this journal paper) for their careful reading and numerous suggestions which also helped to remove some minor errors.

References

[Bac06] Roland Backhouse. Regular algebra applied to language problems. *Journal of Logic and Algebraic Programming*, (66):71–111, 2006.

[Bac08] Roland Backhouse. The capacity-C torch problem. In Philippe Audebaud and Christine Paulin-Mohring, editors, *Mathematics of Program Construction, 9th International Conference, MPC2008, Marseille, France*, volume LNCS 5133, pages 57–78. Springer, 2008.

[Bac11] Roland Backhouse. *Algorithmic Problem Solving*. John Wiley & Sons, 2011.

[Dij76] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[Rot02] Günter Rote. Crossing the bridge at night. *Bulletin of the European Association for Theoretical Computer Science*, 78:241–246, October 2002.

[Sed88] Robert Sedgewick. *Algorithms*. Addison-Wesley Publishing Company, Inc, second edition, 1988.

[Tru11] Hai Truong. Capacity-C torch problem: Novel solution. MSc in Advanced Computing Science, School of Computer Science, University of Nottingham, 2011.

Appendix

Integer-Programming Solution

This is an example of the lp_solve file constructed for an instance of the torch problem with N equal to 11 and C equal to 3. The numbers 35, 46, etc. in the objective function are (random) crossing times for the individual people. The variables beginning with the letter X represent the multiplicity of pairs in the constructed bag of trips. For example, X1_7 is the multiplicity of the pair $(1, 7)$. The variables beginning with the letters r and f represent numbers of return trips and numbers of forward trips, respectively. For example, r1 is the number of times person 1 makes a return trip. The variable pc is the number of pure trips. Variables that are necessarily zero are not included. For example, there is no variable X0_10 because person 10 can never lead a pure trip. In fact, person 10 can never lead any trip and so there is no constraint generated by person 10. In this way, the total size of the model is reduced as much as possible.

```
// lpsolve citation data
// ---------------------
// Description     : Open source (Mixed-Integer) Linear Programming system
// Language        : Multi-platform, pure ANSI C / POSIX source code,
//                   Lex/Yacc based parsing
// Official name   : lp_solve (alternatively lpsolve)
// Release data    : Version 5.0.0.0 dated 1 May 2004
// Co-developers   : Michel Berkelaar, Kjell Eikland, Peter Notebaert
// Licence terms   : GNU LGPL (Lesser General Public Licence)
// Citation policy : General references as per LGPL
//

// objective function
min: 35X0_2 + 46X0_3 + 53X0_5 + 104X0_8 + 148X0_11 + 51X1_4 + 53X1_5
+ 66X1_6 + 79X1_7 + 104X1_8 + 138X1_9 + 148X1_11 + 35X2_2 + 46X2_3
+ 51X2_4 + 53X2_5 + 46X3_3 + 13r1 + 35r2 + 46r3 ;

// Each person j where j>C makes exactly one forward trip
X0_11 + X1_11 = 1 ;
X0_11 + X1_9 = 1 ;
X0_8 + X1_8 + X1_9 = 1 ;
X0_8 + X1_7 + X1_8 = 1 ;
X0_8 + X1_6 + X1_7 = 1 ;
X0_5 + X1_5 + X1_6 + X2_5 = 1 ;
```

```
X0_5 + X1_4 + X1_5 + X2_4 = 1 ;
// Return trips
X3_3 = r3 ;
X2_2 + X2_3 + X3_3 + X2_4 + X2_5 = r2 ;
X2_2 + X2_3 + X3_3 + X1_4 + X2_4 + X1_5 + X2_5 + X1_6 + X1_7 + X1_8
+ X1_9 + X1_11 = r1 ;
// Forward Trips: (Potential) Nomads
X0_3 + X0_5 + X1_4 + X2_3 + X3_3 = f3;
X0_2 + X0_3 + X2_2 + X2_3 + X2_4 + X2_5 + X3_3 = f2;
X0_2 + X0_3 + X1_4 + X1_5 + X1_6 + X1_7 + X1_8 + X1_9 + X1_11 + X2_2
+ X2_3 + X2_4 + X2_5 + X3_3 = f1;
// Relation between individual return- and forward-trip counts
r1 = f1 - 1 ;
r2 = f2 - 1 ;
r3 = f3 - 1 ;
// Relation between total return- and forward-trip counts
pc = X0_2 + X0_3 + X0_5 + X0_8 + X0_11  ;
pc = X2_2 + X2_3 + X2_4 + X2_5 + 2*X3_3 + 1 ;

// Declarations
bin X0_2, X0_3, X0_5, X0_8, X0_11, X1_4, X1_5, X1_6, X1_7, X1_8, X1_9
, X1_11, X2_3, X2_4, X2_5;

int X2_2, X3_3;
```

Java Implementation of Dynamic-Programming Solution

This appendix contains the implementation (in Java) of the dynamic-programming algorithm presented in section 4.5.

Execution of the implementation is limited by storage requirements (which increase as C decreases).

```
class DynamicProgramming{
    int N = CommonProperties.numPeople;
    int C = CommonProperties.capacity;
    int maxNoPureTrips = (int)Math.floor((double)(N-2)/C);
    int[][] m = new int[N][maxNoPureTrips+1];
    int[][] nmds = new int[N][maxNoPureTrips+1];
    /* m[n-1][e] is optimal time for n people to cross with excess e */
    /* thus m[n-1][e] = s(n,e) */
    /* where the function s is as defined in the paper */
    /* nmds[n-1][e] is the number of nomads in the trip with lead n  */
    /* in an optimal solution to n people crossing with excess e */

    public int cost(int i0, int j0){
           int t = CommonProperties.travellingTime[j0-1];
           for (int i = 1; i <= i0; i++){
                t += CommonProperties.travellingTime[i-1];
                }
                return t;
           }

    public void run(){
      /* assert(2<=C && C<=N); */

      // Calculate optimal travel time

      /* At most C people*/
      /* excess equals 0 */
      for (int n= 2; n <= C; n++){
              m[n-1][0] = cost(0,n);
              nmds[n-1][0] = 0;
          };

      /* excess e>0 */
```

```
for (int n= 2; n <= C; n++){
    for (int e= 1; e <= maxNoPureTrips; e++){
        int i= Math.min(e+1, n);
        int min= cost(i,n) + m[i-1][e-i+1];
        nmds[n-1][e]= i;
        i--;
        while (2 <= i){
           int temp;
           temp= cost(i,n) + m[i-1][e-i+1];
           if (temp < min){
              min= temp; nmds[n-1][e]= i;
           };
           i--;
       }
       m[n-1][e] = min;
    }
};

/* n>C */

/* n=C+1 */
for (int e= 0; e <= maxNoPureTrips; e++){
  /* no pure trip: mixed trip with at least one nomad */
  int min = cost(1,C+1) + m[1][e];
  nmds[C][e]= 1;
  int temp, i;
  i= 2;
  while (i <= Math.min(e+1, C-1)){
    temp= cost(i,C+1) + m[i][e-i+1];
    if (temp < min){
      min= temp;
      nmds[C][e]= i;
    };
    i++;
  }
  m[C][e] = min;
}

/* n>C+1 */
```

```
   for (int n= C+2; n <= N; n++){
     int maxExcess= (int)Math.floor(((double)N-n)/C);
     for (int e= 0; e <= maxExcess ; e++){
         int min= cost(0,n) + m[n-C-1][e+1];
         nmds[n-1][e]= 0;
         int temp;
         for (int i= 1; i <= Math.min(e+1, C-1); i++){
           temp= cost(i,n) + m[n-(C-i)-1][e-i+1];
           if (temp < min){
               min= temp;
               nmds[n-1][e]= i;
           }
         }
         m[n-1][e] = min;
     }
   };

System.out.print("Optimal time = "); System.out.println(m[N-1][0]);

// compute the sequence of crossings
int [] leadPureTrips = new int[maxNoPureTrips] ;
           /* stack of pure trips waiting to be scheduled */
int p= 0, pureTrips= 0;
           /* p is the stack index */
           /* pureTrips counts total number of pure trips */
int n= N, e= 0;
while (n > C){
  int i= nmds[n-1][e];
  if (i == 0){/* stack pure trip with lead n*/
      leadPureTrips[p]= n; p++; pureTrips++;
      n= n-C; e++;
  }
  else {System.out.print("Forward Trip:  nomads 1..");
        System.out.print(i); System.out.print(" ; settlers ");
        System.out.print(n-(C-i)+1); System.out.print("..");
        System.out.println(n);
        e= e-(i-1);  n= n-(C-i);
        System.out.print("Return Trip:  nomad ");
        System.out.println(i); i--;
```

```
            /* unstack and schedule pure trips */
            while (i>0){
              System.out.print("Forward Trip: settlers ");
              System.out.print(leadPureTrips[p-1]-C+1);
              System.out.print("..");
              System.out.println(leadPureTrips[p-1]); p--;
              System.out.print("Return Trip: nomad ");
              System.out.println(i); i--;
            }
      }
    };
    /* n =< C */
    while (n != 0){
      int i= nmds[n-1][e];
      System.out.print("Forward Trip: 1.."); System.out.println(n);
      n= i;
      /* assert(((i==0)==(e==0)) && ((i==0)||(i>=2}})) */
      if (i>0){
          e= e-i+1;
          System.out.print("Return Trip:  nomad ");
          System.out.println(i); i--;
          /* unstack and schedule pure trips */
          while (i>0){
            System.out.print("Forward Trip: settlers ");
            System.out.print(leadPureTrips[p-1]-C+1);
            System.out.print("..");
            System.out.println(leadPureTrips[p-1]); p--;
            System.out.print("Return Trip: nomad ");
            System.out.println(i); i--;
         }
      }
    }
  }
}
```