Data Structures and Cost-bounded Petri Nets

Daniel D. Sleator

Department of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

January 2, 1992

Abstract

Consider a collection of particles of various types, and a set of reactions that are allowed to take place among these particles. Each reaction is defined by an input linear combination of particles, and an output linear combination of particles. This framework (which is a Petri net) is shown to model the cost of updating several standard data structures, the amortized cost of counting in various number systems and the space consumption of persistent data structures. A proof that the system of reactions is guaranteed to terminate gives a bound on the cost of the corresponding data structure problem. I show how linear programming can be used to analyze these systems.

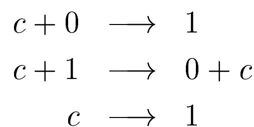
1. Introduction

Starting from zero, a binary counter is incremented up to n. The total cost of the process is defined to be the total number of bits that change. Consider the problem of bounding the total cost of the sequence of increments.

It is easy to see that $2n$ is a bound on the total cost: The low order bit (the ones bit) changes n times, the twos bit changes $\lfloor \frac{n}{2} \rfloor$ (at most $\frac{n}{2}$) times, the fours bit changes at most $\frac{n}{4}$ times, etc. Thus the total number of bit changes is bounded by $2n$.

An alternative approach to the problem is revealed by examining on a more microscopic level what happens when a number is incremented. This process can be described as follows: A carry enters the number from the right, and runs into the low order bit. If this bit is a 0, it is changed to a 1, and the increment operation terminates. If the low order bit is a 1, it changes to a 0, and a new carry propagates into the twos position. This process is repeated until the carry reaches a 0.

If we imagine a number to be a collection of 0s and 1s, then incrementing corresponds to adding a "c" (a carry) to this collection, and applying a sequence of the following reactions:



$$\begin{aligned} c+0 &\longrightarrow 1 \\ c+1 &\longrightarrow 0+c \\ c &\longrightarrow 1 \end{aligned}$$

(The semantics of these reactions is that the collection of things on the left hand side is replaced by the collection of things on the right hand side.)

The process of incrementing a number n times starting from zero corresponds to starting with n cs and nothing else, and applying these reactions until all of the cs are gone. Thus, if we obtain an upper bound B on the number of times these reactions can fire starting from that initial state, B is also an upper bound on the cost of the increments. Note that while any sequence of increments corresponds to a sequence reaction firings, the converse is not true; a sequence of reaction firings does not always correspond to a sequence of increment operations.

Let N_0, N_1, and N_c be the number of 0s, 1s and cs. Consider the following function:

$$\Phi = 2N_c + N_1$$

Applying any one of the reactions causes Φ to decrease by 1. Since the initial value of Φ is $2n$, and Φ is always non-negative, we infer that the total number of times these reactions can fire is at most $2n$. This gives an alternative derivation of the bound on the cost of incrementing.

This paper has two parts. In the first part I show how the approach demonstrated above can be used to analyze several standard data structures: 2-3 trees, 2-3-4 trees, persistent data structures and fibonacci heaps. These results on data structures are not new.

The reaction systems that arise from these data structure problems are known as *Petri nets* [7, 9, 10]. Each reaction (called a *transition* in the Petri net literature) has an non-negative integer linear combination of particles on its left side and on its right side. At any point in the process there is a population of particles (called the *marking*) to which one of the transitions is applied. Such a transition is said to have *fired*. A transition can only fire if the requirements of its left side are satisfied by the current population. Applying the transition removes the particles specified on the left side, and replaces them by those of the right side. The order in which the transitions fire is arbitrary subject to these constraints.

In order to obtain useful bounds on the cost of data structure operations, it is necessary to bound some behavior (such as the number of particles that can arise, or the number of reactions that fire) of the corresponding Petri net. In the second part of the paper I explore this problem.

Starting in a particular initial state, a Petri net is said to be *terminating* if the total number reactions that can possibly fire is finite. The net is said to be *bounded* if the number of particles of each type remains finite. (A terminating Petri net is bounded, but the converse is not necessarily true.)

Theorem 1 says that the problem of determining if a Petri net (starting from a particular initial condition) is bounded or terminating is NP-hard.

In a *priced Petri net* each reaction has a fixed cost. The problems of determining termination and boundedness of a Petri net can be reduced (by the appropriate choice of cost

vectors) to the problem of determining if a priced Petri net has bounded cost. It therefore follows from Theorem 1 that the problem of determining if the Petri net has bounded cost for a particular initial state is also NP-hard.

It is much easier to determine if a priced Petri net has bounded cost for *all* initial states. Theorem 2 shows that this problem is equivalent to the feasibility of a certain set of linear inequalities. One interpretation of the theorem is that if the Petri net has bounded cost for all inputs, then there must be a linear potential function (such as the one in the counting example) that proves this fact. This potential can be found with linear programming.

The work in this paper is closely related to other work on *amortized analysis of algorithms* [14, 15]. This type of analysis seeks to bound the performance of an algorithm which is required to perform a sequence of tasks. An amortized analysis is one which bounds the cost incurred by the algorithm over such a sequence. We can assign an *amortized cost* to each of the tasks in a sequence. These amortized costs are arbitrary subject to the provision that they sum to the total cost of the sequence.

One characteristic of most amortized analyses is that a potential function mysteriously appears to complete the proof. There has been very little work on the problem of automatically deriving potential functions. The only other work is that of Nelson [11], who showed how to derive the potential required for analyzing a snoopy caching problem [6]. This paper removes the mystery from several other amortized analyses.

2. Representing data structures as Petri nets

2.1. Counting up and down

When a binary number is decremented, a borrow, which I will denote by "b" combines with the low order bit of the number and propagates to the left. If we allow both increments and decrements, the five transitions that govern the situation are:

$$\begin{aligned} b+0 &\longrightarrow 1+b \\ b+1 &\longrightarrow 0 \\ c+0 &\longrightarrow 1 \\ c+1 &\longrightarrow 0+c \\ c &\longrightarrow 1 \end{aligned}$$

Consider the initial configuration: $\{0, c, b\}$. The first and the fourth transition can be alternately applied forever. Although the actual cost of applying n increments, and m decrements to a number (that is never allowed to be negative) is at most $(m+n)\log(n)$, we do not obtain this bound. The added generality of the Petri net allows an infinite repetition, when no such occurrence is possible in the original situation.

There are counting schemes in which the cost of both an increment and a decrement is constant in the amortized sense. One example is a ternary representation in base 2. The

three digits are $^-1$, 0, and 1. (To increment the number, a carry is injected into the low order trit. This trit is increased by one. If its new value is 2, it is changed to 0, and a carry is propagated to the next trit. The decrement is analogous.) The transitions governing the increment and decrement operations are:

$$\begin{aligned} b + {}^-1 &\longrightarrow 0 + b \\ b + 0 &\longrightarrow {}^-1 \\ b + 1 &\longrightarrow 0 \\ b &\longrightarrow {}^-1 \\ c + {}^-1 &\longrightarrow 0 \\ c + 0 &\longrightarrow 1 \\ c + 1 &\longrightarrow 0 + c \\ c &\longrightarrow 1 \end{aligned}$$

Let the potential function Φ be

$$\Phi = 2N_c + N_1 + 2N_b + N_{-1},$$

where N_c, N_b, N_1 and N_{-1}, are the number of cs, bs, 1s, and $^-1$s respectively. An easy verification shows that Φ decreases by at least one as a consequence of any of the transitions. If we start with n cs, m bs, and any number of zeros, the initial potential is $2m + 2n$. This proves that the amortized cost of an increment or decrement is bounded by 2.

2.2. Insertions in 2-3 trees

A popular data structure for representing ordered list of items while allowing efficient searching, inserting, deleting, splitting, and joining is the 2-3 tree [1]. This data structure represents the list as a tree. Every internal node of the tree has either two or three children, and each path from the root to an external node has the same length. The *degree* of a node is the number of children it has, and a node with d children is said to be a d-node. For the purposes of this discussion we regard the items as being stored in the external nodes of the tree.

Insertions in 2-3 trees are carried out in two phases. The first phase finds the internal node y to which the new external node x is to be attached. In the second phase the tree is modified to accommodate the new external node. This is carried out as follows: x is made a child of y. If y now has four children, it is split into two nodes, each with two children. Both of these nodes are attached to the parent of y, which may now have four children. This process of splitting iterates up the tree until either a node with two children is reached, or the root splits in two, and a new root with two children is introduced.

The cost of an insertion is defined to be the number of nodes that change as a result of the operation[1]. Starting from a tree with one internal node and two leaves, a sequence of n

[1]As you'll see later, the techniques here will work even when each operation has an arbitrary cost. This allows, for example, the running time of each section of the program to be used as its cost.

insertions is applied. What is the amortized cost of these insertions?[2]

If we let "2" denote a node with two children and "3" denote a node with three children, and "c" denote the pending increase of the degree of a node by one, then we can capture the restructuring of the tree with a sequence from the following set of transitions:

$$\begin{aligned} c+2 &\longrightarrow 3 \\ c+3 &\longrightarrow 2+2+c \\ c+3 &\longrightarrow 2+2+2 \end{aligned}$$

The last of these transitions corresponds to the case when the root node splits in two. One of these transitions is fired each time the tree changes during the sequence of n insertions. Setting $\Phi = 2N_c + N_3$, it is easy to see that each transition causes the potential to decrease by at least one. The initial population of particles is $N_c = n$, $N_2 = 1$, $N_3 = N_1 = 0$. The initial potential is therefore $2n$, and the cost of the sequence of insertions is at most $2n$. This shows that the amortized cost of an insertion is 2.

Incorporating deletions into this analysis leads to a Petri net that is non-terminating as long as there is at least one insertion and one deletion. The situation is analogous to that occurring during a sequence of increments and decrements of a binary number, as in section 2.1.

2.3. Insertions/deletions in 2-4 trees

To allow both insertions and deletions to be performed efficiently in the amortized sense, the tree must be allowed more flexibility. A natural generalization is to allow a node to have up to four children. Such a data structure is known as a 2-4 tree. In it every internal node has two, three, or four children, and the distance from the root to each external node is the same.[3]

Insertions are performed almost as in 2-3 trees. The only difference is that when a 5-node is created, it is split into a 2-node and a 3-node, and when a 3 or 4-node is created the process stops.

Deletions are performed as follows. Let x be the node to be deleted. Let y be the parent of x. After x is deleted from y, the degree of y is one, two, or three. If the degree is two or three the process terminates. If the degree is one, then we must consider a node z that is a sibling of y adjacent to y. If the degree of z is three or four, then one of its children is removed and made a child of y. If the degree of z is two, then y and z are merged to make a 3-node. This decreases the degree of the parent of y, which is repaired in the same manner

---

[2]This cost measure ignores the first phase of the insertion — finding the place at which to put the node. The running time of that phase is easily seen to be be $\Theta(\log n)$ per operation. In many applications the location at which to insert the new node is known without searching, and only the second phase is necessary.

[3]For a more complete discussion of these trees, isomorphisms between 2-4 trees and other data structures, and algorithms for manipulating these trees see [15].

that y was repaired. The process continues up the tree until either a node of sufficiently high degree (three or four) is reached. If the root is reached and is changed into a 1-node, it is deleted. I define the cost of a deletion to be the number of nodes that change.

Below is a set of transitions that describe the situation. Each transition corresponds to a unit cost operation. The symbol "b" denotes the pending decrease in degree of a node. The column labeled $\Delta\Phi$ is the change in the potential function caused by the transition. The potential used is $\Phi = 3N_1 + N_2 + 2N_4 + 3N_c + 3N_b$

$$
\begin{array}{lcll}
 & & & \Delta\Phi \\
b+2 & \longrightarrow & 1 & -1 \\
b+3 & \longrightarrow & 2 & -2 \\
b+4 & \longrightarrow & 3 & -5 \\
1+4 & \longrightarrow & 2+3 & -4 \\
1+3 & \longrightarrow & 2+2 & -1 \\
1+2 & \longrightarrow & 3+b & -1 \\
c+2 & \longrightarrow & 3 & -4 \\
c+3 & \longrightarrow & 4 & -1 \\
c+4 & \longrightarrow & 2+3+c & -1 \\
c+4 & \longrightarrow & 2+3+2 & -3
\end{array}
$$

For simplicity we have assumed that the tree always has at least one internal node. (The initial tree consists of one 2-node and two external nodes.) The form of the potential function, and the fact that the potential decreases by at least one as a result of each transition proves that doing n insertions and m deletions costs at most $3(n+m)$. The amortized cost of an insertion or deletion is 3.

2.4. Persistent data structures

Persistence is another example of a dynamic data structure problem that can be analyzed with these techniques.

Consider a data structure that allows update operations (which change the data represented), and query operations (which answer queries about the data). Over time, a sequence of different versions of the data structure exists. A new version is created (and the old one destroyed) each time an update operations occurs. Such a data structure is called *ephemeral*. A *persistent* extension of such a structure maintains all of the versions that ever existed, in the sense that any query operation can be performed on any version of the structure.

Driscoll, Sarnak, Sleator, and Tarjan [12, 3, 4] gave techniques to automatically transform certain types of ephemeral data structures into persistent ones, at the cost of only a small additive term in the time, and using constant amortized extra space per update. A *fully persistent* data structure [3, 4] allows both queries and updates of past versions, for a particular notion of what it means to "change the past."

The analysis of both the space and the time bounds of these techniques for making data structures persistent can be done using the Petri net method described in this paper.

2.5. Fibonacci heaps

A fibonacci heap [5] is a data structure allows the manipulation of a collection of sets of numbers (keys). Specifically, it allows the union of two sets to be formed (melding), new keys to be inserted into a set, and a key in a set to decreased (decrease-key) all in constant amortized time. The operation of deleting of a specific key and deleting the minimum key (delete-min) can be performed in $O(\log n)$ time in a set with n keys. This data structure has many applications in efficient graph algorithms.

The decrease-key algorithm in fibonacci heaps involves a process called a *cascading cut*. Its analysis can be performed using the Petri net approach described here.

3. Bounding the cost in general

First I should make the definitions a little more precise. Each transition has an non-negative integer linear combination of particles on its left side and on its right side. At any point in the process there is a population of particles to which a sequence of transitions is applied. A transition can only be applied if the requirements of the left side of the transition are satisfied by the current population. Applying the transition removes the particles specified on the left side, and replaces them by those of the right side.

Our work on data structures motivates the following questions: Under what conditions is the cost of a priced Petri net bounded? When it is bounded, how can a bound on its cost be computed (as a function of the net and the initial population)? We have partial answers to these questions.

The reachability problem for Petri nets is the following: Given a Petri net, an initial distribution of particles, and a goal distribution, is there a firing sequence that leads from the initial distribution to the goal distribution? Lipton [8] proved that the problem is exponential-space-hard, and Mayr [9] showed it is decidable. I am interested in the problem of deciding if a Petri net is has bounded cost for a particular initial distribution of particles. I was not able to show the equivalence between this problem and the reachability problem, but I was able to show that the problem is NP-complete.

Theorem 1 *Given a priced Petri net and an initial state the problem of determining whether it is terminating (or bounded) is NP-hard.*

Proof. I will show how to reduce an instance of 3-SAT to this problem. Let the variables of the 3-SAT instance be $x_1, x_2, \ldots, x_n$, and the clauses be $C_1, C_2, \ldots, C_m$. The Petri net I construct will have a particle type for both truth values of each variable in each clause. These particles will be denoted T_{ij} (variable x_i in clause C_j), and F_{ij}. There will also be a particle type for each clause P_j for $1 \leq j \leq m$.

The initial configuration consists of one of each of the T particles. The transitions will ensure that no T particles or F particles for the same variable will simultaneously exist.

There is (for each variable x_i) a transition whose purpose is to complement that variable. This transition is:

$$T_{i1} + T_{i2} + \cdots + T_{im} \longrightarrow F_{i1} + F_{i2} + \cdots + F_{im}$$

For each clause there are three transitions (one for each literal of the clause). The left side of each of these corresponds to one of the variables, and the right side is just the P corresponding to this clause. For example, if $C_7 = (x_1 \vee x_2 \vee \overline{x_3})$ then the three transitions for this clause would be:

$$\begin{array}{rcl} T_{1,7} & \longrightarrow & P_7 \\ T_{2,7} & \longrightarrow & P_7 \\ F_{3,7} & \longrightarrow & P_7 \end{array}$$

Notice that once one of these has fired the variable corresponding to its left side cannot change (the variable assignment transition cannot fire).

Finally, there is a *checking transition* whose left side has all of the Ps, and whose right side has two copies of each of these. If this transition fires, it can fire infinitely often.

This Petri net is non-terminating for the specified initial population if and only if the 3-SAT formula is satisfiable. The following paragraphs show this.

Suppose that the formula is satisfiable. For each variable that is false in the satisfying assignment the truth assignment transition is fired. Now there must exist for each clause a particle which will allow one of the clause transitions to fire (creating a P type particles for that clause). Once all of the P particles have been produced, the checking transition can be fired infinitely often.

Conversely suppose that the Petri net is non-terminating. The only cyclic path in the Petri net involves the checking transition. Therefore we know that at least one copy of each of the P type particles must have been created. This in turn means that one of the literals in it must be true. This proves that the formula is satisfiable.

□

More constructive results can be obtained by asking the more general question of whether a priced Petri net's cost is bounded for *all* initial populations.

To state these results we need some notation. Let n be the number of different particle types, and let m be the number of transitions. The *transition matrix* A of a Petri net is an n by m integral valued matrix. Each of the m columns corresponds to a transition, and a_{ij} is the change in the count of particle type i caused by an application of transition j. The *cost vector* c is a vector of m costs, where c_j is the cost of transition j. (The following is a generalization of a theorem of Memmi and Roucairol to the case of an arbitrary cost vector [10, theorem 1].)

Theorem 2 *Given a priced Petri net with transition matrix A and cost vector c, the following three are equivalent:*

(1) *The Petri net has bounded cost for all initial states.*

(2) *There is no solution s (a real vector of size n) to the following system of inequalities:*

$$\begin{aligned} s &\geq 0 \\ c^t s &> 0 \\ As &\geq 0 \end{aligned}$$

(3) *There is a solution p (a real vector of size m) to the following system of inequalities:*

$$\begin{aligned} p &\geq 0 \\ p^t A &\leq -c^t \end{aligned}$$

Proof.

(3) $\Rightarrow$ (1). (The solution p to (3) corresponds to the potential in the examples.) We rewrite the inequality as

$$p^t A + c^t \leq 0.$$

Consider a firing of the jth transition. Let the population vector before and after the firing be v and v' respectively. The jth of the above inequalities can be expressed as:

$$p^t(v' - v) + c_j \leq 0$$

If we let v_0 be the initial population vector, and v_k be one at time k, and C_k be the cumulative cost for the first k steps, then we get:

$$p^t(v_k - v_0) + C_k \leq 0$$

or

$$C_k \leq p^t v_0 - p^t v_k \leq p^t v_0$$

It follows that the cost of the Petri net is bounded in each initial state.

(1) $\Rightarrow$ (2): Actually I will show the contrapositive; if (2) does not hold, then neither does (1). So assume that there is a solution s to the inequalities in (2). I have to show that there is an initial state of the Petri net which can lead to unbounded cost.

We may assume without loss of generality that s is integral. (If there is a real solution, then there must be a rational solution. Multiplying s by the product of the denominators gives an integral solution without violating any of the inequalities.) There is some finite number of particles sufficiently large that from it we can apply the jth transition s_j times (for all j). Because of the inequality $As \geq 0$ no particle type decreased in population as a result of all these transitions. Therefore this sequence can be repeated as often as desired.

The cost of all these transitions ($c^t s$) is greater than zero. This shows that the cost of this particular initial state can be unbounded.

(2) $\Rightarrow$ (3): Let A and c be fixed. Consider the following linear programming problems (the variables are s and p):

$$\begin{array}{lrcl} \text{maximize} & c^t s & & \\ \text{subject to} & As & \geq & 0 \\ & s & \geq & 0 \end{array} \qquad \begin{array}{lrcl} \text{minimize} & 0 & & \\ \text{subject to} & p^t A & \leq & -c^t \\ & p & \geq & 0 \end{array}$$

These are dual linear programs [2, p.60]. The linear program on the left has the feasible solution $s = 0$. By (2) we know that the maximum value of this LP is 0. By the duality theorem [2] we infer that the LP on the right is also feasible. This completes the proof.

□

If the conditions of Theorem 2 hold, then we can use linear programming to give a specific bound on the cost that can be incurred in any initial state as follows. My proof shows that ${v_0}^t p$ is a bound on the ultimate cost of initial state v_0. Thus, the linear program on the right below gives a bound (the tightest this method can give) on the total cost. Furthermore, by duality its solution equals that of the linear program in the left.

$$\begin{array}{lrcl} \text{maximize} & c^t s & & \\ \text{subject to} & As & \geq & -v_0 \\ & s & \geq & 0 \end{array} \qquad \begin{array}{lrcl} \text{minimize} & {v_0}^t p & & \\ \text{subject to} & p^t A & \leq & -c^t \\ & p & \geq & 0 \end{array}$$

4. Conclusions

This paper represents a small step in the direction of making amortized analysis more systematic. Can these methods be generalized to apply to more complicated data structures, such as splay trees [13]?

References

[1] Aho, A. V., J. E. Hopcroft, J. D. Ullman, *The Design and Analysis of Computer Algorithms,* Addison-Wesley, Reading, Massachusetts, 1974.

[2] Chvátal, V., *Linear Programming,* W. H. Freeman, New York. 1983.

[3] Driscoll, J. R., N. Sarnak, D. D. Sleator, and R. E. Tarjan, “Making data structures persistent,” *Proceedings of the Eighteenth Annual Symp. on the Theory of Computing* 1986, 109–121,

[4] Driscoll, J. R., N. Sarnak, D. D. Sleator, and R. E. Tarjan, "Making data structures persistent," *Journal of Computer and System Sciences.*, Vol. 38, No. 1, February 1989.

[5] Fredman, M., R. E. Tarjan, "Fibonacci heaps and their uses in improved network optimization algorithms," *Journal of the ACM*, 34(3), 1987, 209-221.

[6] Karlin, A. R., M. S. Manasse, L. Rudolph, D. D. Sleator, "Competitive snoopy caching," *Algorithmica*, (1988)3, 79–119.

[7] Lien, Y. E., "Termination properties of generalized petri nets," *SIAM J. Computing*, 5(2), June 1976, 251–265.

[8] Lipton, R., "The reachability problem is exponential-space-hard," Dept. of Comp. Sci. Rep. 62, Yale Univ., New Haven, CT, 1976.

[9] Mayr, E. W., "An algorithm for the general Petri net reachability problem," *SIAM J. Computing*, 13(3), August 1984, 441–460.

[10] Memmi, G., G. Roucairol, "Linear algebra in net theory," Lecture Notes in Computer Science No. 88, Springer Verlag, 1980, 213–223.

[11] Nelson, C. G, "Systematic snoopy caching," In *Beauty is Our Business, A Birthday Salute to Edsger W. Dijkstra* (eds. W. Feijen, N. van Gasteren, D. Gries, J. Misra), Springer-Verlag, New York, 1990.

[12] Sarnak, N. and R. E. Tarjan, "Planar point location using persistent search search trees," *Comm. ACM*, 29(1986), 669–679.

[13] Sleator, D. D. and R. E. Tarjan, "Self-adjusting binary search trees," *Journal of the ACM*, 32(3), 1985, 652–686.

[14] Sleator, D. D. and R. E. Tarjan, "Amortized efficiency of list update and paging rules," *Comm. ACM*, 28(2), February 1985, 202–208.

[15] Tarjan, R. E. "Amortized computational complexity," *SIAM J. Algebraic and Discrete Methods*, 6(1985), 306–318.