Simulating a stack by queues

Burton Rosenberg

University of Miami
Department of Mathematics
and Computer Science

Abstract

We explore the complexity relationships between finite automata endowed with various data structures as auxiliary stores. It is shown that a *queue automata*, a finite state machine endowed with one or more queues, are Turing Equivalent. A finite automata with two stacks can simulate a (one) queue automata in constant amortized time per operation simulated. The converse question, simulating a (one) stack machine by a queue machine is not satisfactorily resolved. We give an $O(\,n \log n\,)$ amortized cost solution using a *generalized queue automata*.

Introduction

We explore the relationship between stacks, queues and double-ended queues. In practice, efficient methods exist for simulation of one of these data structures with another. These methods do not treat stacks and queues as primitive objects, and make recourse to auxiliary structures. Our goal is to investigate the necessity of these auxiliary structures. Hence, we consider a different model of computation in which stacks and queues are primitive objects admitting to a certain number of unit cost operations and having no discernible internal structure.

A *finite automata* is a set of states with transition rules depending on the state and sensed events in the environment. A *stack automata* endows a finite automata with a writable stack. The automata can push characters on the stack, pop them off the stack, and sense the stack to control future state transitions. This gives the finite automata greater power, see [1]. We define *queue automata* and *deque automata* as finite automata endowed with queues and deques, respectively. The queue automata can request that a character be placed at the back of the queue or removed from the front of the queue. It can sense the characters at the ends of the queue to control future state transitions. A dequeue is similar, except that items can be placed at and removed from both the front and back of the queue. All three machines can be generalized to having k auxiliary stores: the k-stack automata, the k-queue automata, and the k-deque automata. In addition, we define an automata which can have a queue of queues.

The cost of computation is the number of character operations on these auxiliary storage structures.

Simulating queue automata by stack automata

A finite automata is a directed graph with labeled arrows. A single marker is moved from node to node according to the labels on the arrows and the state of the environment. An arrow's label also directs actions upon the environment. The labels of a *Mealy Machine*, for instance, are pairs of characters (τ, ρ), where τ is the character sensed on the input tape and ρ is the character written on the output tape. Both tapes are advanced with each automata step. Our machines have the option of sensing from an auxiliary store and modifying this store. The labeling of the arrows would be modified to include these possibilities. We will remain vague about the details of the new labeling.

The most general of the auxiliary stores is the deque. We shall simplify terminology by referring to the other two stores as restrictions on a deque. A deque is a list of characters from a finite alphabet Σ which can be sensed, appended to or deleted from on the left or on the right. It is also possible to sense when the deque is empty.

To be completely clear, we can use the terminology,

$$\{ \textit{ left , right } \} \{ \textit{ sense, append, delete } \}$$

to specify an operation. To prevent special cases, operations will be capable of returning a special "empty store" symbol. We shall often use of the vernacular of "push" and "pop" for the operation names. The following table is a translation to our formal definitions. Various definitions can be made for the deque, we choose to enrich our vernacular with the operations *R-push* and *R-pop*.

Vernacular =	Operation
enqueue	right append
dequeue	left delete
push	left append
pop	left delete
R-push	right append
R-pop	right delete

Theorem 1 *Even with a single queue, a queue automata is equivalent in computational power to a Turing machine. Hence a queue automata cannot be simulated by a one stack automata.*

PROOF: We refer the reader to Minsky, Theorem 14,4–1 [2] for a proof that the following machine is Turing equivalent: a finite automata with two semi-infinite, i.e. single-ended, non-readable, non-writable tapes. The automata can however sense when the tape has come to its end.

To simulate such a machine, a single queue must be able to store, increment, decrement and test for zero two integer quantities, n and m. We do this by representing the integer n as the character N being n times on the queue. Similarly, m is represented by the character M being m times on the queue. Throughout the remainder of the proof it will be understood that m can be replaced by n and simultaneously M by N. To increment n, we right append an N onto the queue. To check if n is zero we first check if the queue is empty. If not, we search it for an N. To do this, we first right append a special character E to the queue and then repeatedly left sense, left delete and right append characters until the character E appears on the left. An N will have been sensed before this event if and only if n is non-zero. The E is left deleted, leaving the queue unchanged by this procedure.

To decrement n, we first check that it is non-zero. If so, we repeat the above procedure however we do not right append the first N that is sensed and left deleted. Instead, we throw it away.

Because a one stack automata, a.k.a. a *push-down automata*, is not Turing Equivalent [1], it cannot therefore simulate a queue automata. □

Although one stack is too little to simulate a queue, with two stacks, the simulation is quite efficient. We will also show that a deque automata can be efficiently simulated by a three stack automata. We are concerned with the *amortized efficiency* of the simulation [4]. By this measure, it is acceptable that a single operation of the simulation be expensive, as long as its expense is pre-paid by earlier, cheap operations. We shall show that a sequence of queue or deque operations can be simulated on stack machines with only a constant factor amortized cost.

Theorem 2 *There exists a two stack automata which can simulate a sequence of n queue operations in at most $4n$ stack operations.*

PROOF: Call the two stacks the input stack, S_i, and the output stack, S_o. Initially both are empty. Each enqueue is transformed into a push on S_i; each dequeue is either a pop on S_o, if S_o is non-empty, or a transfer of all of S_i to S_o followed by a pop on S_o. The transfer is effectuated by repeatedly popping an element from S_i and pushing it onto S_o until S_i is empty. It is immediate that the semantics of the simulation are correct.

Suppose that 4 pennies are associated with each pushed element. We use these pennies to pay for all operations on the element. The first penny pays for the push onto S_i, the second for its pop from S_i, the third pays for the push onto S_o, and the fourth for the pop from S_o. □

Theorem 3 *There exists a three stack automata which can simulate a sequence of n deque operations in at most $18n$ stack operations.*

PROOF: Call the three stacks the input stack, S_i, the output stack S_o, and the temporary stack, S_t.

The simulation maps push and pop operations to push and pop operations on the input stack. If S_i is empty when a pop operation is requested, then half of the output stack is transferred to the input stack. Similarly, the R-push and R-pop operations map to push and pop on the output stack S_o. If S_o is empty when an R-pop operation is required, half the input stack is transferred to the output stack. The temporary stack is used only during the stack transfers.

Suppose the transfer is half of S_i to S_o and that the size of S_i is s_i. First the stack is measured by simultaneously transferring S_i to S_t and pushing one element onto S_o for every two elements transferred. This requires $2s_i + \lfloor s_i/2 \rfloor$ operations. Then S_t is transferred back to S_i, requiring $2s_i$ operations. Next S_i is divided in half by simultaneously transferring elements from S_i to S_t while poping S_o, stopping when S_o is empty. Here, $3\lfloor s_i/2 \rfloor$ operations will be required. We finish off by transferring the rest of S_i to S_o and then transferring all of S_t back to S_i, requiring $2s_i$ operations. Hence a total of $6s_i + 4\lfloor s_i/2 \rfloor$ operations are required.

The correctness of the algorithm is immediate. We discuss its amortized run time.

Each element presented to the deque is associated with 18 pennies. These pennies will pay for all operations. To pay for the sloshing back and forth between input and output stacks, we set up a bank account, called the *slosh fund.* Of the 18 pennies, one is used to pay for the push or R-push. A second penny is set aside for the eventual pop or R-pop of the element. The remaining 16 pennies are invested in the slosh fund.

Lemma 1 *If S is the slosh fund balance, s_i is the size of the input stack and*

s_o the size of the output stack, we have the invariant,

$$S \geq 16 \max(\, s_i \,, s_o \,).$$

PROOF: The formula is obviously invariant by the four operations push, pop, R-push and R-pop. We now consider what happens during the stack transfer., Without loss of generality, suppose an R-pop is required on an empty S_o. Then $s_o = 0$ and $S \geq 16 s_i$. If s_i is even, then the slosh fund balance after the transfer is at least,

$$16 s_i - 8 s_i = 16(s_i/2),$$

and the new maximum of the two stack sizes is $s_i/2$. If s_i is odd, then the slosh fund balance after the transfer is,

$$16 s_i - 6 s_i - 2(s_i - 1) = 8 s_i + 2 \geq 16(s_i - 1)/2,$$

and the new maximum of the two stack sizes is $(s_i - 1)/2$, since the odd element has been popped. □

The conclusion is that there are always enough pennies in the slosh fund to pay for the necessary sloshing. This completes the proof of the theorem. □

Simulating stack automata by generalized queue automata

As an interim step to finding optimal algorithms for the simulation of stack automata by queue automata, we introduce the *generalized queue automata.* Briefly stated, a generalized queue automata is a queue automata which is allowed to "grow" additional queues on demand. In order that the finite state store of the automata be able to address an unbounded number of queues, we place the queues themselves on a queue.

More precisely, our model of computation is a k queue automata augmented by a special *queue of queues.* The queue of queues takes the same three operations as a normal queue, but these operate on queues rather than elements from a finite alphabet. It will be sufficient that a certain queue among the k is always the operand for operations on the queue of queues. This distinguished queue will be called the *staging queue.* The *sense* command returns whether the queue of queues is empty or not. The *dequeue queue* command removes a queue from the front of the queue of queues and moves it to the staging queue; the previous contents of the staging queue disappears. If the queue of queues is empty, then the staging queue becomes an empty queue. The enqueue queue command moves the staging queue to the back of the queue of queues and the staging queue becomes empty. If the staging queue was already empty, then an empty queue is enqueued on the queue of queues. These three operations on the queue of queues are unit cost.

The general idea behind our approach is to create a multi-level storage hierarchy. All elements are stored so that pops are simulated with $O(1)$ cost. To this end, new items are placed on short queues while older items are moved to longer queues. Moving items onto longer queues while maintaining the stack-like order will be expensive. However, the expense of the moves will be offset by their infrequency. We first discuss our ideas with reference to the "abstract" infinite data structure introduced below. Later it will be explained how our machine model can efficiently implement an arbitrarily large finite portion of this data structure.

Definition 1 *A* right comb *is defined inductively as follows,*

1. *A node is a right comb.*
2. *A tree whose root's left child is a node and whose's right child a right comb is a right comb.*

An ω-comb *is the* ω *iteration of this construction.*

We give canonical names to the nodes of the comb. The right child with no right child of its own is n_1. The other nodes are named according to the rules:

$$\text{sibling of } n_i = \begin{cases} n_{i+1} & \text{if } i \text{ is odd,} \\ n_{i-1} & \text{if } i \text{ is even.} \end{cases}$$

$$\text{parent of } n_i = \begin{cases} n_{i+2} & \text{if } i \text{ is odd,} \\ n_{i+1} & \text{if } i \text{ is even.} \end{cases}$$

Our structure attaches queues to nodes in an ω-comb, where queue q_i is attached to node n_i. Each queue q_i is kept in *stack order*, that is, the element to dequeue next is always the youngest element in the queue. Also, the queues maintain a time ordering amongst themselves. Across the entire comb, the youngest element is always in the queue of smallest index for any non-empty queue in the comb.

After each simulated push, the comb will be rearranged so as to recover the above constraints. In order that this be done efficiently, certain other constraints will be recovered after each rearrangement. We summarize here these *quiescence conditions*:

- Only nodes which are left children will have non-empty queues attached to them.
- The size of non-empty queues will obey the bound: $\mid q_i \mid \leq 2^{i/2}$.
- All q_i are in stack order.
- All elements of q_i are older than any element of q_j when $j < i$.

To push an element, it is placed by itself in a one element queue attached to node n_1 of the right comb. The quiescence conditions are then recovered using the following algorithm. It uses the operation *coalesce* which concatenates its first queue argument followed by its second queue argument onto a result queue.

```
i = 1
while queue q[i+1] is empty do
  attach to n[i+2] the coalesce of q[i] and q[i+1]
```

```
    i = i + 2
  endwhile
  move queue to n[i+1]
```

To pop an element, we find the queue of smallest index for any non-empty queue in the tree and we dequeue an element from it. To sense an element, we find the queue described by the pop operation and sense it. An empty queue will be detected separately (else we might try searching the entire ω-comb!)

Theorem 4 *A generalized queue automata can simulate a sequence of n push and pop operations using $O(\, n \log n\,)$ queue operations.*

PROOF: The correctness rests on the quiescence conditions. If they are maintained, then the pop and sense commands will be correctly simulated. The quiescence condition is true after the first element is placed in the comb. A pop or sense command clearly leaves the conditions valid, as does a push command except for the bound of queue sizes. An easy induction shows that no queue grows beyond this bound.

The q_i for even i will be stored on the queue of queues such that q_2 generally is at the head, followed by q_4, and so on. Empty queues will have their place held by empty queues. To describe the simulation, we will often refer to rotating through the queue of queues until some event occurs. This means that we dequeue a queue, check the staging queue to see that it is of a certain type, or that the queue of queues is empty, and if so halt. Else, enqueue the queue back onto the queue of queues and repeat.

To sense or pop, a marker is enqueued on the queue of queues, and then the queue of queues is rotated through looking for a non-empty queue or the reappearance of the marker. If the marker reappears first, then the stack we are simulating is empty and we return the special empty queue symbol. Else, the non-empty queue in the staging queue is sensed or dequeued, according to the operation we are simulating, before it is enqueued on the queue of queues. The algorithm continues by rotating through the queue of queues until the marker is found, which is discarded. We leave to the reader's imagination the construction of an appropriate marker.

A push simulation is done as follows. A one element queue Q containing the new element is constructed, and a marker enqueued on the queue of queues. Dequeue a queue. If the staging queue is non-empty, transfer it to the end of Q, leaving the staging queue empty. Enqueue the empty staging queue and repeat. If the staging queue is empty, transfer Q to the staging queue, enqueue it, and rotate through the queue of queues until the marker reappears, which is discarded. If the marker reappears before Q is ever transferred, move Q to the staging queue, enqueue it and halt.

The run time analysis requires that the queue of queues never be too long.

Lemma 2 *During a simulation of a sequence of $n > 0$ stack operations, the number of queues on the queue of queues is always less than or equal to,*

$$\lfloor \log_2 n + 1 \rfloor .$$

Proof: Let $\mathcal{C}$ represent possible configurations of queues in the ω-comb and $[I]$ be the binary representation of an integer I. Define a map ϕ from comb configurations to binary representations via:

$$\begin{array}{rccc} \phi : & \mathcal{C} & \rightarrow & [I] \\ & \{ q_{2i} \mid i \in \mathbf{Z} \} & \mapsto & \iota_k \iota_{k-1} \ldots \iota_1 \end{array}$$

where

$$\iota_i = \begin{cases} 0 & q_{2i} \text{ is empty} \\ 1 & \text{otherwise.} \end{cases}$$

The procedure for updating the comb will coalesce q_{2i} and q_{2i-1} if and only if adding one to I would cause a carry out in the i-th bit. Furthermore, if c' is the result of the update, then $\phi(c') = [I + 1]$. Sense and pop only reduce I, hence we are left with the problem of estimating the number of one bits in any binary number of size n or less. The result follows. $\square$

From this lemma it is immediate that pop and sense commands can be completed in time $O(\log n)$ per command. Push commands require extra time to "promote" elements, that is, move them to queues of higher index. Suppose each time we move an element to a higher indexed queue we charge the element for its enqueues and dequeues. Since an element cannot be promoted more than $\lfloor \log_2 n + 1 \rfloor$ times, its total charges will be always less than $O(\log n)$. Work done during the push command which cannot be cost against elements, that of moving queues to and from the staging queue, and so forth, will be bound by queue length, that is, $O(\log n)$ per command. □

Conclusions

We have begun a study of the complexity relationships between stack automata and queue automata. We have shown that one-stack automata cannot simulate a queue automata, but that two-stack automata can simulate a queue automata in constant amortized time. The simulation of stack automata was approached through the use of generalized queue automata. We have an $O(n \log n)$ algorithm to simulate a sequence of n push and pops on a generalized queue automata. Recent results have shown that a sequence of n push and pops can be simulated on a k queue automata in time $O(n^{1+1/k})$. However, no non-trivial lower bounds have been found.

Bibliography

1. J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation,* Addison-Wesley, 1979.
2. M. Minsky, *Computation: Finite and Infinite Machines,* Prentice-Hall, 1967.
3. R. E. Tarjan, *Data Structures and Network Algorithms,* CMBS 44, Society for Industrial and Applied Mathematics, 1983. Pages 7–8.
4. R. E. Tarjan, "Amortized Computational Complexity," SIAM Journal of Algebraic and Discrete Methods, Vol. 6, No. 2, April 1985. Pp. 306–318.