Information Processing Letters 112 (2012) 213–217

Contents lists available at SciVerse ScienceDirect

Information Processing Letters

www.elsevier.com/locate/ipl

Fast brief practical DFA minimization

Antti Valmari

Tampere University of Technology, Department of Software Systems, PO Box 553, FI-33101 Tampere, Finland

ARTICLE INFO

Article history:
Received 27 May 2011
Received in revised form 2 December 2011
Accepted 5 December 2011
Available online 7 December 2011
Communicated by J. Torán

Keywords:
Algorithms
Formal languages
Data structures
Deterministic finite automata
Partition refinement

ABSTRACT

Minimization of deterministic finite automata has traditionally required complicated programs and correctness proofs, and taken $O(nk \log n)$ time, where n is the number of states and k the size of the alphabet. Here a short, memory-efficient program is presented that runs in $O(n + m \log m)$, or even in $O(n + m \log n)$, time, where m is the number of transitions. The program is complete with input, output, and the removal of irrelevant parts of the automaton. Its invariant-style correctness proof is relatively short.

1. Introduction

Minimization of a deterministic finite automaton (DFA) is a famous problem in theoretical computer science. It can be solved by successively refining a partition of the states. Let Q be the set of states, Σ the alphabet, "$\xrightarrow{a}$" the (perhaps partial) transition function for each $a \in \Sigma$, $\hat{q}$ the initial state, and F the set of final states. By $q \xrightarrow{a} B'$ where $B' \subseteq Q$ we mean that there is a $q' \in B'$ such that $q \xrightarrow{a} q'$. We say that B is *compatible* with B' if and only if for every $a \in \Sigma$ and q_1 and q_2 in B, if $q_1 \xrightarrow{a} B'$, then also $q_2 \xrightarrow{a} B'$. Partitioning is started with $\{F, Q \setminus F\}$, or just $\{F\}$ or $\{Q \setminus F\}$ if the other set is empty. It is continued until every B in the partition is compatible with every B' in the partition.

It is relatively straightforward to design an algorithm whose running time is $O(kn^2)$, where $n = |Q|$ and $k = |\Sigma|$. In 1971, John E. Hopcroft reduced the running time to $O(nk \log n)$ [1]. His ideas were excellent but, as is often the case with new results, not so easy to read. This led to publications that aimed at clarifying the ideas [2,3]. From the practical point of view, also they left a lot to be desired. The designs were complicated, with many data structures and long correctness arguments. The memory usage was $\Theta(nk)$.

Often in practice, DFAs are *incomplete* in the sense that not every state has an outgoing transition for every label. The number of defined transitions, m, is often much smaller than its theoretical maximum, nk.

In 2008 two independent pairs of researchers succeeded in reducing the running time to $O(n + m \log n)$, either by employing a second refinable partition that stores the transitions [4], or by computing the set of labels of transitions that end in a block and splitting simultaneously with all these labels [5]. The memory consumption of these approaches was $\Theta(m + n + k)$. The former technique was extended to a non-deterministic generalization of the DFA minimization problem [6]. Then a way of omitting some data structures was found, simplifying the algorithm considerably [7,8].

There have been occasional earlier claims that $O(n + m \log n)$ running time can be obtained by straightforward application of Paige's and Tarjan's relational coarsest partition algorithm [9]. In reality, the running time of the suggested algorithms has been $O(n + km \log n)$ or something similar. The claims were either based on treating k as a constant, so that it is not shown in the complexity formula; or under-appreciating the difficulties involved in

E-mail address: Antti.Valmari@tut.fi.
URL: http://www.cs.tut.fi/~ava/.

0020-0190/$ – see front matter
doi:10.1016/j.ipl.2011.12.004

skipping quickly enough those labels that do not label any currently interesting transitions. Please see [8] for more details on this issue.

With the above and some further simplifications, it is possible to implement and prove correct a complete DFA minimization program, including the removal of irrelevant parts (states and transitions that are not reachable from the initial state or from which no final states can be reached), in about 130 non-empty non-comment lines of C++ code. That is the goal of this article.

To do certain subtasks in the promised time, both of [4,5] used a data structure using $\Theta(k)$ memory. It may be big, and when it is not, it is of little benefit. In this publication we leave it out in favor of a more pragmatic solution. This makes the running time $O(n+m\log m)$, which is slightly worse than $O(n+m\log n)$. If one may assume that the transitions are sorted in the input according to their labels, then $O(n+m\log n)$ is obtained by leaving out one line of code.

Section 2 presents the program. The hard part of its correctness proof is given in Section 3.

2. The program

This section presents and discusses the program. To fully understand its main loop, it is necessary to know many details of the low-level operations. Therefore, the section proceeds in the order of the program code.

Fast execution, small memory consumption, and short enough lines to fit in the columns of this publication were favored over programming style recommendations. The version in this publication has been checked by copying its parts from the PDF reader screen, dropping them into a `.cc` file, compiling, and testing. Before compilation, an apostrophe ' was replaced for each right single quotation mark ’, to cancel the accidental replacements in the opposite direction that took place in the process.

The first two lines of Fig. 1 employ the input/output operations of the C++ language and a sorting algorithm. The rest of the figure shows a refinable partition data structure. The program uses two instances of it, one for states and another for transitions.

The variable `z` stores the number of sets in the partition. The sets are numbered from 0 to `z` − 1. The elements of a set s are stored in an unspecified order in $\mathtt{E}[f], \mathtt{E}[f+1], \ldots, \mathtt{E}[p-1]$, where $f = \mathtt{F}[s]$ and $p = \mathtt{P}[s]$. The array names “`F`” and “`P`” stand for “first” and “past”. The location of element e in `E` is stored in `L[`e`]`. So $\mathtt{E}[\mathtt{L}[e]] = e$ and $\mathtt{L}[\mathtt{E}[i]] = i$. The number of the set that e belongs to is in `S[`e`]`.

The procedure `init(`n`)` initializes the data structure to the coarsest partition whose sets together contain $0, 1, \ldots, n-1$. This partition is either empty or consists of one set. The operation `bool(0)` returns 0 and `bool(`n`)` returns 1, if $n > 0$. `init(`n`)` obviously runs in linear time.

Refinement of the partition consists of *marking* some elements by calling `mark(`e`)` for each of them, and then calling `split()`. The number of marked elements of the set s is indicated by `M[`s`]`, and `W` keeps track of *touched* sets, that is, the sets with marked elements. To save memory, `M` and `W` are shared by both instances of the structure. This is why they are defined outside the structure.

```
#include <iostream>
#include <algorithm>

/* Refinable partition */
int *M, *W, w = 0;  // temporary worksets

struct partition{
  int z, *E, *L, *S, *F, *P;

  void init( int n ){
    z = bool( n );  E = new int[n];
    L = new int[n]; S = new int[n];
    F = new int[n]; P = new int[n];
    for( int i = 0; i < n; ++i ){
      E[i] = L[i] = i; S[i] = 0; }
    if( z ){ F[0] = 0; P[0] = n; }
  }

  void mark( int e ){
    int s = S[e], i = L[e], j = F[s]+M[s];
    E[i] = E[j]; L[E[i]] = i;
    E[j] = e; L[e] = j;
    if( !M[s]++ ){ W[w++] = s; }
  }

  void split(){
    while( w ){
      int s = W[--w], j = F[s]+M[s];
      if( j == P[s] ){M[s] = 0; continue;}
      if( M[s] <= P[s]-j ){
        F[z] = F[s]; P[z] = F[s] = j; }
      else{
        P[z] = P[s]; F[z] = P[s] = j; }
      for( int i = F[z]; i < P[z]; ++i ){
        S[E[i]] = z; }
      M[s] = M[z++] = 0;
    }
  }

};
```

Fig. 1. Refinable partition data structure.

The segment in `E` of the set s lists first its marked and then unmarked elements. `mark(`e`)` moves an element from the unmarked part to the marked part, by swapping it with the first unmarked element and incrementing `M[`s`]`. If the set did not contain marked elements before the operation, its number is pushed to `W` so that the set will be split later. `mark(`e`)` runs in constant time.

The procedure `split()` exhausts `W` and splits the sets listed in it. If all elements of the set s are marked, then `split()` unmarks them and skips the rest of the body of the `while`-loop with `continue`. Otherwise `split()` chooses the smaller of the marked and unmarked part, and makes it a new set with a new number `z`. The old set (i.e., the set number `s`) continues existence, but now it only consists of the bigger part. The running time of `split()` is at most proportional to the total number of elements that were marked since the previous call of `split()`. Because the marking operations also took that much time, `split()` runs in amortized constant time.

To the best of our knowledge, all publications before [7,8] always made the new set from the marked part. As will be mentioned in Section 4, our solution makes it possible to avoid a significant complication later in the program.

```
partition
  B,     // blocks (consist of states)
  C;     // cords (consist of transitions)

int
  nn,    // number of states
  mm,    // number of transitions
  ff,    // number of final states
  q0,    // initial state
  *T,    // tails of transitions
  *L,    // labels of transitions
  *H;    // heads of transitions

bool cmp( int i, int j ){
  return L[i] < L[j]; }
```

Fig. 2. General data structures.

```
/* Adjacent transitions */
int *A, *F;
void make_adjacent( int K[] ){
  int q, t;
  for( q = 0; q <= nn; ++q ){ F[q] = 0; }
  for( t = 0; t < mm; ++t ){ ++F[K[t]]; }
  for( q = 0; q < nn; ++q )F[q+1] += F[q];
  for( t = mm; t--; ){ A[--F[K[t]]] = t; }
}
```

Fig. 3. Adjacent transition data structure.

Fig. 2 declares the general data structures of the program, plus one auxiliary operation. The sets of the partition of states are traditionally called *blocks*. The partition of transitions is not traditional. We call its sets *cords*. Transitions are numbered starting from 0. We have $q \xrightarrow{a} q'$ if and only if there is t such that $\mathtt{T}[t] = q$, $\mathtt{L}[t] = a$, and $\mathtt{H}[t] = q'$. The function `cmp` specifies a partial order on transitions that only compares the labels.

Fig. 3 presents a data structure that is sometimes used to store the outgoing transitions of states and sometimes the incoming transitions. The adjacent transitions of state q are $\mathtt{A}[\mathtt{F}[q]], \mathtt{A}[\mathtt{F}[q]+1], \ldots, \mathtt{A}[\mathtt{F}[q+1]-1]$. The procedure `make_adjacent(T)` initializes the data structure to contain the outgoing transitions. It is an immediate application of counting sort, and thus runs in $O(n+m)$ time [10, Section 8.2].

Fig. 4 shows the routines used in the removal of the irrelevant parts of the DFA. It uses set 0 of `B`. First `reach(q)` is called for the start states of the search, and then `rem_unreachable` is called. `rem_unreachable(T,H)` traverses transitions forwards and `rem_unreachable(H,T)` backwards.

The procedure `reach` works otherwise like `mark`, but does not store the number of the set to the workset, and has an extra test to protect against reaching the same state twice. `rem_unreachable` performs breadth-first search [10, Section 22.2]. (In C++, the `for`-loop test `i < rr` uses the current value of `rr`.) Then it compresses the transition data structure, leaving out those whose tail state is not reachable. The ordering of those transitions that remain is preserved. The running time is $O(n+m)$.

The value of `mm` may be reduced by the operation, but `nn` stays the same. This is because `B` uses original state numbers, but `C` has not yet been initialized and will be initialized to use the new transition numbers.

The first half of the main program is shown in Fig. 5. When reading and reaching final states, the program skips

```
/* Removal of irrelevant parts */
int rr = 0;   // number of reached states

inline void reach( int q ){
  int i = B.L[q];
  if( i >= rr ){
    B.E[i] = B.E[rr]; B.L[B.E[i]] = i;
    B.E[rr] = q; B.L[q] = rr++; }
}

void rem_unreachable( int T[], int H[] ){
  make_adjacent( T ); int i, j;
  for( i = 0; i < rr; ++i ){
    for( j = F[B.E[i]];
         j < F[B.E[i] + 1]; ++j ){
      reach( H[A[j]] ); } }
  j = 0;
  for( int t = 0; t < mm; ++t ){
    if( B.L[T[t]] < rr ){
      H[j] = H[t]; L[j] = L[t];
      T[j] = T[t]; ++j; } }
  mm = j; B.P[0] = rr; rr = 0;
}
```

Fig. 4. Removal of irrelevant parts of the DFA.

```
/* Main program */
int main(){

  /* Read sizes and reserve most memory */
  std::cin >> nn >> mm >> q0 >> ff;
  T = new int[ mm ]; L = new int[ mm ];
  H = new int[ mm ]; B.init( nn );
  A = new int[ mm ]; F = new int[ nn+1 ];

  /* Read transitions */
  for( int t = 0; t < mm; ++t ){
    std::cin >> T[t] >> L[t] >> H[t]; }

  /* Remove states that cannot be reached
    from the initial state, and from which
    final states cannot be reached */
  reach( q0 ); rem_unreachable( T, H );
  for( int i = 0; i < ff; ++i ){
    int q; std::cin >> q;
    if( B.L[q] < B.P[0] ){ reach( q ); } }
  ff = rr; rem_unreachable( H, T );

  /* Make initial partition */
  W = new int[ mm+1 ]; M = new int[ mm+1];
  M[0] = ff;
  if( ff ){ W[w++] = 0; B.split(); }

  /* Make transition partition */
  C.init( mm );
  if( mm ){
    std::sort( C.E, C.E+mm, cmp );
    C.z = M[0] = 0; int a = L[C.E[0]];
    for( int i = 0; i < mm; ++i ){
      int t = C.E[i];
      if( L[t] != a ){
        a = L[t]; C.P[C.z++] = i;
        C.F[C.z] = i; M[C.z] = 0; }
      C.S[t] = C.z; C.L[t] = i; }
    C.P[C.z++] = mm;
  }
```

Fig. 5. Main program, part 1.

those that are not reachable from the initial state. The reachable final states enter locations $0, 1, \ldots, \mathtt{rr}-1$ of `B.E`, and their number `rr` is assigned to `ff`.

If the language is empty, the removal of irrelevant parts makes set 0 empty, while it should be {`q0`}. This is not

```
  /* Split blocks and cords */
  make_adjacent( H );
  int b = 1, c = 0, i, j;
  while( c < C.z ){
    for( i = C.F[c]; i < C.P[c]; ++i ){
      B.mark( T[C.E[i]] ); }
    B.split(); ++c;
    while( b < B.z ){
      for( i = B.F[b]; i < B.P[b]; ++i ){
        for(
          j = F[B.E[i]];
          j < F[B.E[i]+1]; ++j
        ){
          C.mark( A[j] ); } }
      C.split(); ++b; }
  }

  /* Count the numbers of transitions
    and final states in the result */
  int mo = 0, fo = 0;
  for( int t = 0; t < mm; ++t ){
    if( B.L[T[t]] == B.F[B.S[T[t]]] ){
      ++mo; } }
  for( int b = 0; b < B.z; ++b ){
    if( B.F[b] < ff ){ ++fo; } }

  /* Print the result */
  std::cout << B.z <<' '<< mo
    <<' '<< B.S[q0] <<' '<< fo <<'\n';
  for( int t = 0; t < mm; ++t ){
    if( B.L[T[t]] == B.F[B.S[T[t]]] ){
      std::cout << B.S[T[t]] <<' '<< L[t]
        <<' '<< B.S[H[t]] <<'\n'; } }
  for( int b = 0; b < B.z; ++b ){
    if( B.F[b] < ff ){
      std::cout << b <<'\n'; } }

}
```

Fig. 6. Main program, part 2.

fixed, because then $\texttt{ff} = \texttt{mm} = 0$ and the further execution skips all statements where the difference matters.

Next the program reserves memory for the temporary worksets. Because unreachable states have been removed, $n \leqslant m + 1$. Clearly $m \leqslant m + 1$. Thus $m + 1$ words suffice. If there are final states, the program calls `split` to make the partition $\{F, Q \setminus F\}$, if $F \neq Q$. Up to this point, the time consumption is $O(n + m)$.

Then the program initializes the transition partition and sorts its elements according to the labels of the transitions. The ordering of the labels is not important, but it is important that transitions with the same label are put next to each other. Sorting takes $O(m \log m)$ time and uses $O(1)$ additional memory, if done with heapsort. If the transitions are given one label at a time in the input, then the sorting could be left out, yielding a program that runs in $O(m \log n)$ time. The program makes each "$\xrightarrow{a}$" a set in the partition and updates the locations that became out of date in the sorting.

The rest of the program is shown in Fig. 6. We say that a block is *ready* if its number is less than `b`, and *unready* otherwise. Similarly cord c is ready or unready depending on whether $c < \texttt{c}$.

If $\texttt{C.z} = 0$, then there are no transitions and there is at most one state. The splitting stage does nothing, which is correct, because the DFA is already minimal or we have the special case mentioned above. Otherwise the stage continues until all blocks and cords are ready.

Cord c is processed by scanning through its elements and marking their tail states. Then `B.split()` is called, resulting in the splitting of all touched blocks so that they become compatible with cord c, that is, either all or none of the states in the block has an outgoing transition in the cord. Next each block that exists and is unready at the time, is processed. Its states are scanned, their incoming transitions are scanned and marked, and the cords that were touched are split. So all cords become compatible with the block. A cord is compatible with a block if either all or no transitions in it end in the block. Why and in what time this yields the correct result is discussed in the next section.

The rest of the program prints the result in $O(m + n)$ time. Transitions between blocks are printed by scanning the (remaining) original transitions, recognizing those whose tail state is the first state in its block, and printing the number of the block, the label, and the number of the block of the head state. The transitions are thus printed one label at a time, if they came one label at a time in the input. Blocks that consist of final states are recognized by the location of their first state. The block that contains the initial state is `B.S[q0]`.

3. Correctness and speed of the splitting stage

The correctness of the splitting stage is proven via a series of lemmas. We assume that every state has a path to a final state, because the opposite case is the empty language case, where splitting is correctly skipped.

Lemma 1. *If the program puts two states in different blocks, then they accept different languages. If it puts two transitions in different cords, then they have different labels or their tail states accept different languages.*

Proof. The situation immediately before the splitting stage clearly satisfies the claim. It remains to be proven that the claim is an invariant of the splitting stage.

If the splitting stage separates two states, then one of them, say q_1, has an outgoing transition $q_1 \xrightarrow{a} q_1'$ that belongs to a cord that contains no outgoing transitions of the other state q_2. If q_2 has no outgoing a-transitions, then q_1 accepts some word $a\sigma$ that q_2 rejects. Otherwise q_2 has an outgoing a-transition in a different cord. The induction assumption yields that q_1 and q_2 accept different languages.

If the splitting stage separates two transitions, then they have the same label a and their head states q_1' and q_2' belong to different blocks. By the induction assumption, either q_1' accepts some word σ that q_2' rejects, or the other way round. As a consequence, the tail state of one of the transitions accepts $a\sigma$ but the tail state of the other rejects it. □

Lemma 2. *Let q_1 and q_2 be states in the same block, t_1 and t_2 transitions in the same cord, and $a \in \Sigma$. If the head states of t_1 and t_2 are in different blocks B_1 and B_2, then at least one of B_1 and B_2 is unready. If q_1 and q_2 both have an outgoing a-transition, but the transitions are in different cords C_1 and C_2,*

then at least one of C_1 and C_2 is unready. If q_1 has an outgoing a-transition but q_2 does not, then the cord of the transition is unready.

Proof. The claim holds initially, because then $\mathtt{b} = 1$ and $\mathtt{c} = 0$, so all cords and all but one block are unready. When the splitting of a block separates the head states of t_1 and t_2 to different blocks, `split` gives the new half-block the number `B.z`, so the new half-block becomes unready. Similarly, when the splitting of a cord separates the outgoing a-transitions of q_1 and q_2, the new half-cord becomes unready. The third situation (i.e., q_1 has but q_2 does not have an outgoing a-transition) cannot become valid inside the loops.

A block or cord becomes ready when `b` or `c` is incremented. Then the block or cord has just been used for splitting, so that t_1 and t_2 are no longer in the same cord or q_1 and q_2 are no longer in the same block. □

Lemma 3. *Let n' and m' be the numbers of states and transitions after the removal of irrelevant parts. After* `make_adjacent`*, the splitting stage runs in $O(m' \log n')$ time.*

Proof. We have $n' \leqslant m' + 1$. The basic operations run in amortized constant time.

If a transition is used anew in `B.mark(T[C.E[i]])`, then it belongs to a cord whose number is greater than the previous time. When splitting a set, `split` gives the new number to the smaller half. Together these imply that the same transition can be used at most $\lfloor \log_2 h \rfloor + 1$ times, where h is the size of the original cord. Because all transitions in the same cord have the same label, we have $h \leqslant n'$. Therefore, each transition is used $O(\log n')$ times, contributing $O(m' \log n')$.

A similar argument shows that each state is used $O(\log n')$ times for splitting cords. This implies that `C.mark(A[j])` marks each transition $O(\log n')$ times. Together these contribute $O((n' + m') \log n')$. □

Theorem 1. *The program outputs the minimal DFA that accepts the same language as the input DFA. It runs in $O(n + m \log m)$ time and uses at most $6n + 11m + O(1)$ words of memory, if the sorting algorithm is heapsort.*

Proof. We first prove by induction that if q_1 and q_2 are in the same block after the splitting stage and q_1 accepts the word $a_1 a_2 \cdots a_h$, then also q_2 accepts the word. If $h = 0$ then this holds, because final and other states were separated after the removal of irrelevant parts. Otherwise there is a q_1' such that $q_1 \xrightarrow{a_1} q_1'$. At the end of the splitting stage all blocks and cords are ready. By Lemma 2, q_2 has a transition $q_2 \xrightarrow{a_1} q_2'$ in the same cord as $q_1 \xrightarrow{a_1} q_1'$. Furthermore by Lemma 2, q_1' and q_2' are in the same block. By induction, they both accept $a_2 a_3 \cdots a_h$. So q_2 accepts $a_1 a_2 \cdots a_h$.

Thus every state in the same block accepts the same language. From here on the proof follows known paths. Also the quotient DFA D printed by the algorithm accepts the same language. By Lemma 1, each state q of D accepts a distinct language. Unreachable states were removed, so q has a word σ_q that leads from the initial state of D to q. If another DFA D' accepts the same language, it must have a state $f(q)$ for each σ_q, because it is a prefix of some accepted word. The state $f(q)$ accepts the same language as q, implying that the $f(q)$ are distinct. So D' has at least the same number of states as D. Therefore, D is minimal.

The correctness and speed of other than the splitting stages are based on well-known results and were briefly discussed in Section 2. By these results and Lemma 3, the running time of the program is $O(n + m \log m)$. The memory consumption of the program can be read directly from the code. □

4. Conclusions

DFAs can be minimized efficiently with a rather short program. Thanks to always giving the new block or cord number to the smaller half, no data structure and complicated tests are needed for keeping track of blocks and cords that have to be used in the future for splitting. This is a significant simplification compared to, as far as we know, all publications prior to [7,8]. The present program is much simpler than the pseudocode in [8], because the latter solves a more general problem. Furthermore, the present program has some small improvements over [7,8].

Because the program has been presented in full and tested, the reader can check that there are no hidden issues, and use it easily.

Acknowledgements

The author thanks Jaco Geldenhuys and the reviewers for helpful comments.

References

[1] J.E. Hopcroft, An $n \log n$ algorithm for minimizing states in a finite automaton, Technical Report STAN-CS-71-190, Stanford University, 1971.
[2] D. Gries, Describing an algorithm by Hopcroft, Acta Informatica 2 (1973) 97–109.
[3] T. Knuutila, Re-describing an algorithm by Hopcroft, Theoretical Computer Science 250 (2001) 333–363.
[4] A. Valmari, P. Lehtinen, Efficient minimization of DFAs with partial transition functions, in: STACS 2008, Symposium on Theoretical Aspects of Computer Science, 2008, pp. 645–656, http://drops.dagstuhl.de/volltexte/2008/1328/.
[5] M.-P. Béal, M. Crochemore, Minimizing incomplete automata, in: Finite-State Methods and Natural Language Processing, Seventh International Workshop, 2008, pp. 9–16.
[6] A. Valmari, Bisimilarity minimization in $O(m \log n)$ time, in: Lecture Notes in Computer Science, vol. 5606, Springer-Verlag, Berlin, 2009, pp. 123–142.
[7] A. Valmari, Kuinka kiltti mutta tarpeeton pikku algoritmi sai tärkeän tehtävän, Tietojenkäsittelytiede 20 (2010) 33–55.
[8] A. Valmari, Simple bisimilarity minimization in $O(m \log n)$ time, Fundamenta Informaticae 105 (3) (2010) 319–339.
[9] R. Paige, R. Tarjan, Three partition refinement algorithms, SIAM Journal on Computing 16 (6) (1987) 973–989.
[10] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Introduction to Algorithms, 3rd ed., The MIT Press, London, 2009.