Case 4: Computing probabilities and expected costs of different terminations

Antti Valmari
Tampere University of Technology

File \LaTeX\ed October 29, 2009
1 Background

1.1 An abstract formulation of the problem

Given is a big sparse rooted directed multigraph whose each edge has a weight and cost. A random walker starts at the root and walks from one vertex to another by choosing the next edge at random weighed according to the edge weights. We must find out, for each terminal strong component, the probability with which the walker ends up in that component, and (if the costs of the edges in the component are zero) what is the expected cost of the path taken.

A directed multigraph is otherwise like a directed graph, but there may be many edges in the same direction between the same nodes

- can be defined, e.g., as the tuple \((V, E, t, h)\), where \(t : E \rightarrow V\) gives the tail and \(h : E \rightarrow V\) gives the head of the edge
- now path of length \(n\) must be defined as a single vertex \(v_0\) \((n = 0)\) or as a sequence of edges \(e_1, e_2, \ldots, e_n\) \((n > 0)\) such that \(\forall i ; 2 \leq i \leq n : t(e_i) = h(e_{i-1})\)
“Big” means that there are many vertices

**Sparse** means that many fewer pairs of vertices than $|V|^2$ have edges

- usually this is written as “there are many fewer edges than $|V|^2$”, why not now?
- “there are $o(|V|^2)$ pairs of vertices that have edges” (little $o$!) would be a precise definition, but it only applies to families of inputs

**Rooted** means that there is a vertex named *root* from which there are paths to all vertices

\[
\forall v \in V : v_{\text{root}} \rightarrow^* v
\]

- $u \rightarrow v \iff \exists e \in E : t(e) = u \land h(e) = v \iff$ there is an edge from $u$ to $v$
- $u \rightarrow^* v \iff \exists k \geq 0 : \exists v_0, \ldots, v_k : v_0 = u \land v_k = v \land \forall i ; 1 \leq i \leq k : (v_{i-1} \rightarrow v_i) \iff$ there is a path from $u$ to $v$

The **weights** are represented as a function $W : E \rightarrow \mathbb{R}^+$

- $\mathbb{R}^+$ denotes the positive real numbers
- that is, each edge has an associated positive real number

$\Rightarrow$ a weight cannot be 0
The costs are represented as a function \( C : E \rightarrow \mathbb{R}^+ \cup \{0\} \).

If the output edges of vertex \( v \) are \( e_1, \ldots, e_n \), then the random walker continues from \( v \) via the edge \( e_i \) with the probability

\[
p(e_i) = \frac{W(e_i)}{\sum_{1 \leq j \leq n} W(e_j)}
\]

A maximal strongly connected component is a maximal set of vertices such that from each of them there are paths to all of them

- that is, \( \forall u \in U : \forall v \in U : u \rightarrow^* v \) and \( \forall u \in U : \forall v \in V \setminus U : u \not\rightarrow^* v \lor v \not\rightarrow^* u \)
- also known by the shorter name strong component

A strong component is terminal, if and only if it cannot be exited

- that is, \( \forall u \in U : \forall v \in V \setminus U : u \not\rightarrow v \)
- we will later use the shorter name terminal component
- a special case: a vertex that has no output edges
A random walker that has entered a terminal component will stay there forever.

In this kind of a system the random walker eventually ends up in some terminal component with probability 1.

- It means that if $p(n)$ is the probability that the random walker is not yet in any terminal component after $n$ steps, then $\lim_{n \to \infty} p(n) = 0$.

- The results that are shown later will make this obvious.

- A simplified explanation: the probabilities eventually make the random walker try all possibilities that are still available, and at least one of them leads to a terminal component.

- Notice: with probability 1 it is not certain.

Let $v_0$ or $e_1, e_2, \ldots, e_n$ be a path.

- Its probability is the product of the probabilities of the choices made along the path:
  $$\prod_{1 \leq i \leq n} p(e_i)$$

- Its cost is, of course, the sum of the costs of the edges of the path:
  $$\sum_{1 \leq i \leq n} C(e_i)$$
The probability of ending up in a certain terminal component is, of course, the sum of the probabilities of all paths that lead there from the root

- the path ends in the first vertex that belongs to the terminal component
- it may be possible to go around cycles along the path

⇒ typically there are infinitely many paths

⇒ to be more precise, the probability is the limit of the sum
  – the limit exists, because the sum grows as paths are added, but cannot grow beyond 1

The expected value of the cost of going into a given terminal component is the weighed average of the costs of all paths that lead there from the root

- the cost of each path is weighed by the probability of the path
The expected value of the cost was only asked if the costs of the edges in the terminal component are zeros

- guarantees that continuing the path inside the terminal component does not increase its cost
  
  \[ \Rightarrow \text{takes away the need to investigate, when precisely the path enters the terminal component} \]

  \[ \Rightarrow \text{simplifies designing the algorithm} \]

  - terminal components can be found and the costs of their internal edges replaced by zeros in \( O(|V| + |E|) \) time, see e.g., Cormen–Leiserson–Rivest–Stein
  
  \[ \Rightarrow \text{when necessary, we can solve with it and our algorithm also problems that do not obey the assumption} \]

A directed graph with probabilities on output edges is a Markov process

- our task (without costs) is basically the problem of finding the probability of ending up in a given absorbing state in a given Markov process
1.2 What was the solution of this problem needed for?

The original goal was to investigate

- with what probability and costs certain tests reveal design errors in telecommunication protocols, especially errors that are hard to detect
- how these probabilities and costs depend on such factors as the error probabilities of channels and the relative speeds of processors

Only limited results may be obtained by experimenting with implementations of the protocols

- detecting small probabilities (say, $10^{-3}$) would require very many experiments
  - on the other hand, errors of that probability occur during the use of systems
- investigating the dependency on background factors increases the number of experiments
- investigating the dependency requires that the background factors can be controlled

$\Rightarrow$ A numerical solution that is based on a model of the situation was desired
The problem is too difficult to be solved with high accuracy

- the results depend on probabilities concerning the run-time environment, and they are not known well
- computing the result precisely is extremely laborious

⇒ The goal was to facilitate at least some experiments that could provide at least some information
1.3 Reactive and concurrent systems

Telecommunication protocols are a good example of reactive concurrent systems

- **reactive** = interacts simultaneously in at least two directions
- **concurrent** = consists of many parts that have activity of their own

The simplest form of interaction is *batch processing*

- the input is ready at start time, or at least the input does not depend on the output
- the main object of research of classical theory of computability and computational complexity
- an example: the compilation of a program
- more complicated forms of interaction often have individual batch processings as their parts
  - an example: the effect of pressing the key "A" in a word processing program in the writing mode
At some point in time, *interactive* programs became common

- interacts with one user
- offers the user alternatives, waits for the user action (e.g., pressing a key), and reacts to it
- an example: word processor

Reactivity adds to this the existence of more than one peer at the same time

- e.g., a mobile phone interacts both with its human user and the network
- interactions with different peers may interleave, but they must still succeed

⇒ unlike an interactive system, a reactive system cannot be based on determining by itself who can talk at each time

Two concepts describe simultaneous computation by more than one unit

- “parallelism”: simultaneous computation is used to speed up computations
- “concurrency”: the problem is of such a nature that co-operation between two or more autonomous parts is necessary
Co-operation is often easy in parallelism

- e.g., all parts execute the same instructions, but with different inputs

In concurrency, organizing the co-operation is often challenging

- the parts are typically reactive

The co-operation in telecommunication systems is of the concurrency type
1.4 On the behaviour of reactive concurrent systems

Correct behaviour of a reactive system is difficult to specify

- the behaviour of a batch processing system is a (partial) function (or more generally a relation) from the input to the output

- in the case of a reactive system, individual sequences of events (e.g., incoming phone call, writing and sending a text message) can interleave and affect each other

In practice, we only define (and test)

- individual sequences of events: *use cases*

- the most common interleavings that an individual user can observe

- many simultaneous users, whose use cases are interleaved but do not affect each other
The co-operation of the parts of a concurrent system may introduce errors that do not occur in other kinds of systems

- **deadlock**: each part waits that some other part does something first
- **starvation**: some part never gets some resource, because it is always given to some other part
- it is possible that two parts disagree on what to do next
  - e.g., the computer in a bank charges the account, although the bank-teller machine does not give money

The behaviour of a concurrent system may change drastically depending on subtle timing details

- e.g., who gets the last seat on a flight

⇒ The result depends on factors that cannot be known
⇒ Concurrent systems must be considered *nondeterministic*

- the same state and input may yield different results
Nondeterminism adds to the difficulty of testing and tracking errors

- a feature that has been tried once may behave differently next time
1.5 State space as a model of the behaviour

The *global state* of a concurrent system is obtained by listing the internal states of all of its parts (including telecommunication channels, etc.)

- often the global state is simply called the *state*

The *initial state* is the state of the system when it is started

- in reality, a system may have many alternative initial states e.g., because of uninitialized memory
- this is seldom essential for analysis

⇒ most of the time, only one initial state is used with state spaces

An *atomic event* means that a single part, or some parts together, does so small a sequence of events that its analysis in parts is not necessary

- the choice of atomic events depends on the level on which we want to analyse the system: a phone call ↔ writing on the memory ↔ the moving of an electron from one place to another
The *state space* is a directed graph that is obtained by starting at the initial state and trying, in each state found, all atomic actions that are possible in that state, until new states are no more found

- the vertices of the graph are the states that can be found in this way, that is, *reachable states*

- the edges of the graph correspond to atomic events

- the initial state is a root of the graph

Actions performed by concurrent parts are forced sequential in a state space, even if they are simultaneous in reality

- a clearly imprecise model of the system
Researchers have developed as an alternative to state spaces also so-called *truly concurrent* models of the behaviour of systems

- they are, however, much more complicated than state spaces in terms of both their theory and their use

- excluding some issues whose significance is small, the predictions that state spaces yield of the behaviour of systems are the same as those produced by truly concurrent models

⇒ the use of truly concurrent models may be “correct”, but usually it is not sensible

The state space of a real system is typically far too big to be processed in a computer

⇒ in the analysis, simplified models are used that only contain those features that are essential for concurrency and reactivity

- despite abstraction, the state space is easily far too big
Many errors can be easily detected from the state space, in particular if the model of the system is built in a suitable way

- deadlock $\sim$ a vertex that has no output edges
- a wrong received message $\sim$ a vertex in which a certain variable has a certain value
1.6 An example

Customer 1
1: do_something_else; goto 1
[] reserve(printer)
2: reserve(file)
3: print; release(file); release(printer); goto 1

Customer 2
1: do_something_else; goto 1
[] reserve(file)
2: reserve(printer)
3: print; release(printer); release(file); goto 1

The state space of each customer and the system as a whole

- \( vk \) = reserve printer
- \( vt \) = reserve file
- other labels have been omitted for clarity
1.7 The weights and costs of the edges in the model

The cost has a simple interpretation

- the cost of the atomic operation that corresponds to the edge

By modifying the costs we may investigate, for instance

- how many losses of messages are needed, on the average, for the error to occur
  - the cost of a loss $= 1$
  - the cost of other operations $= 0$

- how many steps by customer 1 are needed, on the average, for the error to occur
  - the cost of an operation by customer 1 $= 1$
  - the cost of other operations $= 0$

Choosing appropriate weights is difficult

That edges are alternative represents two different things

- alternatives within the same part, e.g., reserve $\leftrightarrow$ do_something_else
- which part is executed next
The relative weights of the alternative choices within the same part are easy to select, in principle

- the ratio between the weights $= \text{the ratio between the probabilities}$
- on the other hand, the probability of an alternative is not always known
- on the other hand, the dependency of the result on the probabilities of the alternatives is one of the issues that we wish to be able to investigate

The relative weights between the alternatives that correspond to executing different parts are more difficult to choose

- they roughly correspond to the relative execution speeds of the parts
- the execution speeds become, however, a statistical (= noisy) entity
- that they are noisy is actually good for the goals of this work, but it is unclear how well the distribution of execution speeds produced by this model corresponds to the reality
- as a matter of fact, it is unclear how the execution speeds behave in reality

$\Rightarrow$ There is a significant arbitrary component in the selection of the weights
On the other hand, if some result holds for a wide variety of weights, it probably also holds in the real situation.
1.8 The problem as a graph problem

A typical way of processing Markov processes is as sets of simultaneous equations that are solved using matrix operations.

Now that does not work, however, because

- the time consumption of the use of matrices is typically cubic in the number of vertices
- there are very many vertices—the bigger graph we can process, the better
- there are few edges compared to the square of the number of vertices

⇒ We want a method that works with a big sparse graph.
2 The basic idea of the algorithm

2.1 Inspiration

The inspiration came from the theorem in theoretical computer science that says “for every finite automaton there is a regular expression that defines the same language” Its proof contains an algorithm that transforms a finite automaton to a corresponding regular expression

- in intermediate stages generalised finite automata are used, whose edge labels are regular expressions
- the algorithm is based on transformations, with which the automaton is simplified one step at a time such that the language that it accepts does not change
- eventually the automaton only contains the initial state, one final state and one edge between them

⇒ the name of the edge is the desired regular expression

We now design a similar algorithm for our Markov process problem
2.2 Basic transformations

*Merging double edges:* if there are two edges between the same vertices in the same direction, it is easy to combine them without changing the desired results.

\[ w = w_1 + w_2 \]
\[ c = \frac{w_1 c_1 + w_2 c_2}{w_1 + w_2} \]

- when we later say that merging of double edges is applied to a vertex, the tail vertex of the edges is meant.
**Removal of a self-loop:** if a vertex has precisely one self-loop and at least one other output edge, the self-loop is easy to remove without changing the desired results.

\[
\begin{align*}
\sum w_j &= w_1 + w_2 + \cdots + w_n \\
\text{let} \quad w_i' &= w_i \\
\text{with the probability} \quad \frac{w}{w + \sum w_j} \text{ a (new) iteration of the loop is taken, and with the probability} \quad \frac{\sum w_j}{w + \sum w_j} \text{ some exit edge is taken} \\
\Rightarrow & \quad x = 0 \cdot \frac{\sum w_j}{w + \sum w_j} + (c + x) \cdot \frac{w}{w + \sum w_j} \\
\Rightarrow & \quad x = \frac{cw}{\sum w_j} \quad \text{and} \\
& \quad c_i' = c_i + \frac{cw}{\sum w_j}
\end{align*}
\]
**Removal of a vertex**: if a vertex is not the root, it has at least one output edge, and it has no self-loops, it can be removed without changing the desired results.

\[
    w_{i,j} = \frac{w_i w_j'}{\sum w_k'} \\
    c_{i,j} = c_i + c_j'
\]

- vertices at the left side of the picture may have edges also elsewhere than to the removed vertex

⇒ the total weight of the descendants of the edge that leads to the removed vertex must be the same as the weight of the original edge

⇒ division with the expression \( \sum w_k' \)
2.3 The main algorithm

Basic transformations are made until no basic transformation applies to any vertex.

Each basic transformation

- either reduces the number of vertices (the number of edges may grow)
- or reduces the number of edges while keeping the number of vertices the same

⇒ Basic transformations cannot be done forever

- sooner or later $|V|$ stops decreasing
  - it cannot become negative
- when $|V|$ has reached its final value, at most as many basic transformations can take place as $|E|$ is at that moment

The ordering

$((v_1, e_1) < (v_2, e_2) : \iff v_1 < v_2 \lor v_1 = v_2 \land e_1 < e_2$ is called a well-ordering

- each descending sequence of elements terminates
To prove that a loop terminates, it suffices to find a well-ordering such that every iteration of the loop strictly reduces the value.

⇒ If basic transformations are made as long as possible, eventually a situation is reached where no basic transformation is possible.

In that situation:

- there are nowhere two edges in the same direction between the same vertices
- if a vertex has an output edge other than a self-loop, it does not have self-loops
- every vertex either is the initial vertex (root), has no output edges, or has only one output edge and that is a self-loop

⇒ The graph is of one of the following forms:

1. a single initial vertex without edges
2. a single initial vertex with a self-loop
3. the initial vertex and a set of other vertices such that the initial vertex has one edge to each other vertex and no other output edges, and each other vertex has at most one output edge and it is a self-loop.
The remaining vertices are original vertices

- the algorithm does not create new vertices, but only removes original ones

At most one vertex from each strong component has been preserved

- if at least two remain, then at least one of them is not the initial vertex
- it has an output edge that is not a self-loop, because it has a path to the other vertex in the same strong component
- this is in contradiction with the properties of the remaining vertices that were reasoned above

Precisely one vertex from each terminal component has been preserved

- when only one vertex of a terminal component remains, it has no other output edges than perhaps a self-loop
  \[ \Rightarrow \text{it and its possible self-loop cannot be removed} \]

Every remaining vertex is the initial vertex or from a terminal component

- we reasoned above that the output edges of other vertices than the initial vertex are self-loops
The other vertices of the result than the initial vertex correspond to the terminal components of the original graph

- the weights and costs of the edges that lead to them tell directly the probability and expected cost of ending up in each terminal component
- the probability is evaluated as \( \frac{w_i}{\sum w_j} \)

The initial vertex corresponds to a terminal component if it is the only vertex that remains

The question remains how an applicable basic transformation is found and implemented efficiently
3 An efficient implementation

3.1 Where are the memory and time consumed?

It is useful to analyse the issue on three levels

- the cost of executing a single basic transformation
- the cost of finding an applicable basic transformation
- the effect of the order in which the basic transformations are applied

Sometimes in this kind of an algorithm the effect of the application order is small

⇒ it is worthwhile to concentrate on the first two

We will see that in this case the effect of the application order is very big!

- one trap in algorithm design is to optimise individual steps and forget to analyse whether all those steps are necessary

The cost of a single basic transformation is, however, the most concrete starting point

⇒ Let us analyse the issue starting from it
3.2 On executing a single basic transformation

Computing the weights and costs during the basic transformation is not a problem

- always when \( \sum w_j \) is needed, a proportional amount of other work is done

\[ \Rightarrow \text{computing the sum is not an important factor in the algorithm as a whole} \]

However

- divisions cannot be removed by scaling the numbers suitably, because they would become too big

\[ \Rightarrow \text{floating-point numbers are needed} \]

\[ \Rightarrow \text{the risk of rounding errors must be taken into account} \]

- everything is not okay, if the sum of the probabilities of ending up in each terminal component is far from one!

- in practice, rounding errors were not much of a problem
Normalisation of the weights brings small benefits

• the weights of the output edges of the vertices are replaced according to the formula
  \[ w'_i = \frac{w_i}{\sum w_j} \]

⇒ the sum of the weights becomes 1

⇒ the weights are probabilities as such

• in the removal of a self-loop, the weight of the result must be
  \[ w'_i = \frac{w_i}{1-w} \]

• computing the above-mentioned sum is avoided

• the numbers remain in the range 0…1 even in long computations

⇒ the risk of a numeric overflow is reduced

Edges have to be removed and added and vertices removed in the basic transformations

⇒ The cost of a basic transformation depends on the chosen data structures

The data structures also affect the cost of finding an applicable vertex

⇒ We do not choose them now, but later
3.3 On finding an applicable basic transformation

The execution time of an algorithm of this kind easily grows by a factor of $O(|V|)$, if a big number of vertices is often unnecessarily investigated when seeking for an applicable basic transformation

⇒ It may be a good idea to divide the vertices to two categories

1. potentially transformable
2. certainly impossible to transform

Originally every vertex is in category 1

A vertex that has just been investigated is in category 2

A vertex moves to category 1, when its environment changes

• merging double edges may make the removal of a self-loop of the same vertex possible
• removal of a self-loop may make the removal of the same vertex possible
• the removal of a vertex may introduce a double edge or a self-loop for another vertex
We may try to exploit this

- by investigating the vertices in this order, or
- maintaining the set of vertices in category 1 in a suitable data structure

Did we list all situations, where a basic transformation may become possible?

- only the removal of a vertex may introduce edges

⇒ only it can introduce new double edges and self-loops

- the removal of a self-loop may in principle become possible in three different ways
  - an output edge that leads elsewhere is introduced to a vertex with a self-loop
  - one self-loop is introduced to a vertex that has an elsewhere-output edge
  - the number of self-loops of a vertex that has an elsewhere-output edge decreases to one

- the first two require creating an edge

- merging double edges may cause the third one

⇒ both the merging of double edges and the removal of a vertex may make the removal of a self-loop possible
A potentially useful observation

- the removal of a vertex does not introduce output edges that lead elsewhere to a vertex that does not already have one

⇒ it suffices to investigate the situation where a self-loop is introduced

The condition “is not the initial vertex” of the removal of a vertex never changes its truth value

The same holds for the condition “has at least one output edge”

- if a vertex that is not removed by a basic transformation has output edges, each basic transformation leaves at least one output edge

- no basic transformation introduces output edges to a vertex that does not already have one

⇒ The removal of a vertex may become possible only so that its self-loops are removed
It looks worthwhile to process each vertex in the following order

- merge double edges (if there are any)
- remove self-loop (if there is any)
- remove the vertex (if possible)

These *basic transformation triplets* of different vertices may be interleaved, if that is advantageous

When does a vertex enter the start of a basic transformation triplet?

- when it is no more certain that the vertex does not have a double edge or a self-loop
  ⇒ that is, when output edge(s) have been added to the vertex
  ⇒ that is, when the vertex at the other end of an output edge has been removed
  ⇒ The removal of a vertex moves its input vertices to category 1
Vertices that have no output edges may be left out of the data structures that are used for finding applicable vertices

- no basic transformation ever applies to them

Can the initial vertex be left out of the search data structures?
3.4 How many basic transformations must be made?

Worthwhile to estimate, to concentrate the attention to the essential

The number of the removal of vertices is easy to estimate

• at most $|V| - 1$ times

• an assumption was that the graph is very big

• usually there are very few terminal components

$\Rightarrow$ approximately $|V| - 1$ times

It is difficult to estimate the number of the removal of edges

• the removal of a vertex may increase the number of edges

• as a matter of fact, it increases whenever a vertex has $\geq 2$ input edges and $\geq 3$ output edges or the other way round
Does the growth of the number of edges have a reasonably small upper bound?

- if double edges are not merged, then every acyclic path that starts at the initial vertex eventually becomes an edge of its own

⇒ it is easy to invent an example that demonstrates that the number of edges may grow exponentially

⇒ The consumption of memory (and time) grows exponentially

But a graph cannot have more edges than $|V|^2$?

- remember, this is a multigraph!
3.5 An idea: merge double edges immediately when created

Restricts the number of edges to $\leq |V|^2$

$\Rightarrow$ Prevents the above-mentioned exponential growth

• an important benefit!

An additional advantage: no need to process merging of edges as a basic transformation

Implementation

• find an edge from the given vertex to the given vertex
• if it exists, modify its weight and cost according to the formulas above
• if it does not exist, add it
Does the number of edges now stay small enough?

- we assumed that the graph is sparse
- we want to be able to process bigger graphs than the adjacency matrix representation allows

⇒ the number of edges must stay much smaller than $|V|^2$

⇒ Not necessarily
3.6 An idea: remove first the vertices with few adjacent edges

If the number of vertices is made small before the number of edges starts to grow quickly, then there is no room for the number of edges to grow too big

- it is at most the square of the current number of vertices

Vertices that cannot be reached from the initial vertex can be removed with a simple efficient graph search before other work

- no basic transformation introduces more vertices that cannot be reached from the initial vertex

⇒ the algorithms may be designed assuming that every vertex (except perhaps the initial vertex) has an input edge from elsewhere

- depending on the source of the problem it may be that other vertices without input edges than the initial vertex are not there to start with

It has been pointed out above that basic transformations cannot be applied to vertices that have no output edges that lead elsewhere

- they can be kept apart from other vertices in the data structures
If a vertex has precisely one input or precisely one output edge from/to elsewhere, its removal reduces the total number of edges.

A heuristic: always remove next a vertex $s$ that has the smallest possible

$$(|\bullet s| - 1)(|s\bullet| - 1)$$

- $\bullet s$ is the set of the input vertices of $s$ and $s\bullet$ of the output vertices.
- $|\bullet s||s\bullet|$ and $|\bullet s| + |s\bullet|$ would cause the removal of the case (2,3) before the case (1,7).

Another way to think of the above heuristic: minimise the growth in the number of edges, that is,

$$|\bullet s||s\bullet| - (|\bullet s| + |s\bullet|) = (|\bullet s| - 1)(|s\bullet| - 1) - 1$$
The classification of the vertices to those that potentially have and those that certainly do not have self-loops becomes unnecessary

- a self-loop of a vertex does not affect the number of input and output edges of other vertices

⇒ the removal of a self-loop of a vertex can be postponed until the vertex is chosen for removal

- the self-loop is ignored when evaluating the value of the heuristic

\[(|\bullet s| - 1)(|s \bullet| - 1)\]

- the self-loop of the initial vertex can be removed at the end, if it has any
  - why not in the beginning?

Instead of choosing the basic transformation and the vertex, the only problem is to choose the vertex to be removed
3.7 The implementation of the heuristic

The vertex to be removed next can be found efficiently with the aid of a heap

- the smallest at the root
- the key is \((|s| - 1)(|s| - 1)\)
- the heap only contains an index to the information proper
- the information proper has an index back to the heap
- the implementation of this has already been seen in the context of the sliding median

Initially other vertices than the initial vertex and vertices that lack output edges that lead elsewhere are put to the heap

The removal of a vertex begins with taking its index from the top of the heap, moving the last of the heap into the top, and fixing the heap

Then the self-loop of the vertex is removed, if it exists
Then the input and output edges of the vertex are replaced by edges that bypass the vertex

- the place in the heap of each input and output vertex of the removed vertex is fixed to match the new number of its input and output edges with the usual algorithms for moving an item up or down in a heap

- the heap must be fixed after the change of the information of each individual vertex, because otherwise the changed information in another vertex that is not yet in its right place may stop the moving prematurely

Now the vertex is isolated from the others and can be removed
3.8 The remaining data structures

The removal of a vertex investigates edges forward and backward
⇒ there must be links in both directions (in one way or another)

It must be possible to find, add, and remove edges in an arbitrary order

A vertex has a small number of edges on the average, but altogether there are many edges
⇒ The use of a complicated structure for representing the input or output edges of each vertex consumes a lot of memory, and its benefits are questionable

In the implemented program

- each vertex had the set of its output edges as a sorted array, and similarly with the input edges
- the weight and the cost (and the head vertex) were stored in the output edge
- the input edge only consisted of the number of the (tail) vertex
It was assumed that there will be many mergings of double edges

- if altogether only a small number of new edges is created, then there will be no performance problems

- if many new edges are created but there is only a small number of mergings, then many edges will remain, so the program will be exhausted in any case

- only a limited number of new edges can be new self-loops

⇒ thus the case is important, where many double edges are merged

⇒ The set of the output edges was kept sorted, to find double edges quickly

Merging a double edge does not require adding or removing an edge, because one of them is a new edge that has not yet been given a place in the data structures

⇒ altogether a cheap operation
The program could be used in the year 2004 up to about 12 000 vertices

- corresponds to an adjacency matrix with more than one hundred million elements

- 74 s in a workstation

3.9 To ponder

When putting an edge to its place, it is easy to check whether it is a self-loop

- would it make sense to remove the self-loop immediately?
- would it make sense to have a bit in each vertex telling whether it has a self-loop?
- what if self-loops were not represented similarly to other edges, but as fields “weight_of_self-loop” and “cost_of_self-loop” in the vertex?

Would it be worthwhile to put the other edges into a more efficient data structure?

—— End of case 4 ——