Chapter 5

Simulations

In this chapter, we present computer simulations that we did to study the average-case performance of transitive closure algorithms and representations. First, in section 5.1 we describe the method that we used in our experiments. The presentation of the actual simulation experiments is divided into three sections: in section 5.2, we study memory requirements of different transitive closure representations, in section 5.3 the amount of work needed to construct the successor sets in different algorithms and representations, and in section section 5.4 the execution times of different transitive closure algorithms.

5.1 Method

In this section, we describe the method that we used in our performance experiments. We first explain why we decided to use computer simulations instead of mathematical analysis as the performance evaluation technique. Then we discuss the issues that have to be considered in computer simulations and describe the inputs that we used in the simulations. We end the description of the method by discussing the overall design of performance evaluation studies.

5.1.1 Mathematical algorithm analysis

Mathematical analysis of the worst-case execution time is the most common method for studying the performance of algorithms. It is usually easy, since we only have to determine the set of possible inputs, decide which inputs cause the algorithm to spend most time, and compute the corresponding execution time. The problem with the worst-case analysis is that it often gives overly pessimistic results. The algorithm may run much faster with typical inputs than in the worst case. Comparing the efficiency of two algorithms only by their worst-case execution times may yield misleading conclusions. For instance, many transitive closure algorithms have approximately the same worst-case time, but these algorithms behave differently with typical inputs.

To get a more realistic picture of the performance of an algorithm with typical inputs, we can analyze its average-case performance. Unfortunately, mathematical average-case analysis is much more difficult than worst-case analysis. We have to create a more sophisticated model of inputs. Determining the set of possible inputs is not enough: we have to assign each input a probability of occurrence. It is not obvious what these probabilities should be. To make the processing of probabilities easier, we may have to choose an overly simplified model of inputs that may differ greatly from the inputs in practical applications. Given an algorithm and an input model, computing the average-case execution time is usually much more difficult than computing the worst-case execution time. As a general rule, average-case analysis using realistic models of input is beyond the limits of the current analytical methods [88]. The average-case analysis is often easier for probabilistic algorithms, i.e., algorithms that use the distribution of the underlying input model in some way in their behavior. If the inputs of the algorithms are simple combinatorial structures like binary trees or sequences of integers, a divide-and-conquer strategy may be possible.

In analyzing the average-case performance of transitive closure algorithms, we face all these problems. The selection of the input model is not obvious. We discuss this problem in the next section. The algorithms that we have presented are not probabilistic. The inputs are not simple combinatorial structures. In addition to the number of vertices and edges, the topology of the graph and the order in which the vertices and edges are processed affect the performance of the algorithms.

The average-case execution time of a transitive closure algorithm is analyzed in only a few previous articles. Bloniarz et al. [18] analyzed the average-case execution time of a simple transitive closure algorithm that traverses the input graph n times starting from each vertex in turn. The average-case bound of this algorithm is $O(n^2 \log n)$. Schnorr [106] presented a probabilistic algorithm that computes the transitive closure in $O(n + e^+)$ average-case time, where e^+ is the number of edges in the transitive closure. The algorithm has high constant costs, since it traverses the input graph several times to obtain both the vertices adjacent from and the vertices adjacent to each vertex and to sort these vertices lexically. Karp [79] presented another probabilistic transitive closure algorithm. When the expected outdegree of the graph is slightly greater than one, the asymptotic average-case execution time of the algorithm is $O((n \log n)^{4/3}\omega(n))$, where $\omega(n)$ is any non-decreasing function. Otherwise, the average-case execution time is $O(n)$. Also Karp's algorithm has high constant costs, since it consists of three separate algorithms that are executed simultaneously step-by-step until one of them completes. The algorithm needs a constant time access both to the vertices adjacent from and the vertices adjacent to each vertex. If only the vertices adjacent from each vertex are available, as we generally have expected in this thesis, Karp's algorithm needs an additional $\Theta(n + e)$ time to compute the vertices adjacent to each vertex. Both Schnorr's and Karp's algorithms depend heavily on the distribution of the underlying input model. It is not clear how well the algorithms behave with other kinds of inputs.

5.1.2 Performance evaluation by simulation

Computer simulation is another method for studying the average-case performance of an algorithm [88]. The technique seems to be straight-forward: we implement the algorithm augmented with commands to collect performance measures, then we apply the program on randomly generated inputs, and finally we compute the average of the performance measures. The benefit of this approach is that it can be used even when mathematical average-case analysis fails. It also yields more accurate results than mathematical analysis. The deficiency of this approach is that the results are valid only in that area of the space of inputs from which we have selected the inputs. Simulations give no asymptotic results. After a simulation study, we cannot say that the run time of the algorithm is $O(f(n))$ in the average case. Note also that a simulation study usually takes more time than mathematical analysis (when the latter is possible).

In selecting the input model for a simulation study, we face partially the same problems as in mathematical average-case analysis. We must choose the set of possible inputs and assign a probability of occurrence to each input. A benefit of the simulation approach is that we can study many input models using the same simulation program. All we need is a new procedure for generating the new inputs and more simulation runs. In simulation studies, we can usually apply more complicated input models than in mathematical analysis. However, in simulation studies, the generation of inputs must be fast whereas in analytical studies we have no such requirement. We consider the selection of the input model and the generation of inputs for our simulation studies in the next subsection.

The measurement data of a simulation study must be properly treated to get reliable results. Since the inputs of the simulation are usually controlled by random variables with some distribution, also the outputs are usually random variables with some other distribution. In many studies presented in the literature, much effort is put in the implementation of a sophisticated simulation program, but the program is applied to only a few arbitrary selected inputs and the mean of the measured values is presented as the average-case performance metric. The problem is that this approach gives no information on the accuracy of the results. As Pawlikowski [99] said, these kinds of studies are merely programming exercises.

The theory of estimation in statistics gives us tools for analyzing the measurement data. We present here only those parts of estimation theory that we have used in our simulations, see any textbook of statistics, e.g., [35], for a more thorough presentation. Our presentation is based on references [35, 83, 84, 99].

In estimation, we expect that a random sample $(x_1, x_2, \ldots, x_n)$, for instance a series of measurement data, is the realization of a sequence of random variables $X_1, X_2, \ldots, X_n$, which have the same distribution $F(x)$ and are independent. The goal is to find values of properties that describe distribution F. In our study, we want to know μ_x, the expected value of the distribution. The sample mean

$$\bar{x} = \frac{1}{n} \sum_{i=1}^{n} x_i \tag{5.1}$$

can be used as the estimator of μ_x.

A standard statistical technique to assess the accuracy of an estimate is to construct a *confidence interval* for the estimated parameter. A $100(1-\alpha)\%$ confidence interval for a parameter θ is an interval $[L_1, L_2]$ that contains the actual value of θ with probability $(1-\alpha)$. The confidence interval is constructed using the sample $(x_1, x_2, \ldots, x_n)$. If the random variables X_i are normally distributed, then the confidence interval of the expected value is $[\bar{x} - \Delta_x, \bar{x} + \Delta_x]$, where Δ_x can be computed using Student's t-distribution and the estimator $\hat{\sigma}^2[\bar{x}]$ of the variance of X as follows:

$$\Delta_x = t_{n-1,1-\alpha/2}\hat{\sigma}[\bar{x}] \tag{5.2}$$

$$\hat{\sigma}^2[\bar{x}] = \frac{1}{n(n-1)} \sum_{i=1}^{n} (x_i - \bar{x})^2 \tag{5.3}$$

Given a sample, we can compute different confidence intervals by changing the confidence level $1-\alpha$. Obviously, the higher the confidence level is, the wider the confidence interval.

These equations can also be used when random variables X_i are not normally distributed or when we do not know that they are. The necessary requirement is that the random variables are independent and have an identical distribution. When n, the size of the sample, increases, the random variable $X = X_1 + X_2 + \ldots + X_n$ tends to the normal distribution and the estimator of the confidence interval is again valid. In practice, the estimator gives good approximation when $n > 30$.

Equations 5.2 and 5.3 can be used after the simulation to compute a $100(1-\alpha)\%$ confidence interval of the expected value. The problem is that before the simulation, we do not know what kind of a confidence interval the simulation will yield. If we fix the number of simulation inputs, i.e., the size of the sample, beforehand, we may either produce confidence intervals that are too wide or too uncertain to be useful or waste simulation time for processing unnecessary inputs. This problem can be solved by using *sequential procedures* [83, 84, 99]. In a sequential procedure, the length of the simulation run is increased until the desired confidence interval is reached. Before the simulation starts, we only have to describe the confidence interval that we would like to have. The simulation procedure produces new inputs and applies the simulation program to them until the desired confidence interval is reached.

We can describe the desired confidence interval, and thus the stopping condition of the simulation, either in absolute or in relative terms. In the former case, we usually state that $|\Delta_x| \leq \delta_{\max}$, i.e., the error of our estimate must be at most $\delta_{\max}$. In the latter case, we have several possibilities. In the methods presented in [83, 84, 99], the $100(1-\alpha)\%$ confidence interval produced by the simulation should be such that the ratio of its half-length to its mid-point does not exceed a fixed constant, i.e., $|\Delta_x/\bar{x}| = \epsilon \leq \epsilon_{\max}$. A shortcoming in these methods is that they do not directly limit the relative error $|(\bar{x}-\mu_x)/\mu_x|$ of the expected value μ_x. Pawlikowski [99] showed that if $[\bar{x}-\Delta_x, \bar{x}+\Delta_x]$ is a $100(1-\alpha)\%$ confidence interval produced in this way, then

$$1-\alpha \leq P[\frac{|\bar{x}-\mu_x|}{|\mu_x|} \leq \frac{\epsilon}{1-\epsilon}] \tag{5.4}$$

We can use this inequality to get a direct limit for the relative error of μ_x. If we replace $\epsilon_{\max}$ by $\epsilon_{\max}/(1+\epsilon_{\max})$ in the stopping condition, then the relative error $|(\bar{x}-\mu_x)/\mu_x|$ is at most $\epsilon_{\max}$. The length of the simulation increases slightly, but according to our experience this causes no problems.

Our sequential simulation procedure is presented in Figure 5.1. As inputs, the simulation procedure needs values for parameters α and $\epsilon_{\max}$ that control the confidence interval. In addition, it needs a procedure A that implements the algorithm which is studied, a procedure G that generates a random input for procedure A, and $p_1, p_2, \ldots, p_k$, the values of parameters that control the generation of the inputs, e.g., the number of vertices in the generated random graph. The random inputs that procedure G yields must be independent. Procedure A applies the algorithm which is studied to the given input and computes the associated performance measure.

The output of the sequential simulation procedure is the estimate $\bar{x} = s/n$ of the expected value μ_x of the performance measure. The relative error of $\bar{x}$ to μ_x is at most $100\epsilon_{\max}\%$ with

$s := 0;$
$n := 0;$
repeat
    $I := G(p_1, p_2, \ldots, p_k);$
    $x := A(I);$
    $s := s + x;$
    $n := n + 1;$
until $|\Delta_x / \bar{x}| \leq \epsilon_{\max}/(1 + \epsilon_{\max})$

FIGURE 5.1: A sequential simulation procedure.

probability $1 - \alpha$. Ganguly et al. [44] used a similar procedure in their study of transitive closure computation.

Since we did not know that the performance measures are normally distributed, we generated at least 30 inputs in each simulation. When the expected value of the measure was near zero in some parts of the input space and large in other parts of the input space, the fulfillment of the relative stopping condition required long simulation runs. Knowing that the expected value is near zero in some part of the input space is usually sufficient when the expected value is large in other parts of the input space. Thus, in these experiments we used a stopping condition that is a combination of a relative and an absolute condition. The simulation was stopped when either the relative error or the absolute error became sufficiently small.

5.1.3 Inputs

Since we had no particular application in mind when choosing the input model, we first decided to use the model of directed random graphs $G(n, p)$ that is mostly used in the literature [19, 79]. The model is defined as follows:

Definition. The model $G(n, p)$ of random graphs is a probability space that contains all directed graphs having the same vertex set $V = \{1, 2, \ldots, n\}$ and an edge set $E \subseteq V \times V$. Each possible edge of a graph in $G(n, p)$ exists with probability p independently of the existence of other edges; thus, the probability of a given graph G with $e = |E|$ edges is $p^e(1 - p)^{n^2 - e}$.

Although all possible directed graphs can be generated in this model, the graphs drawn from $G(n, p)$ are usually similar in some respects. Karp [79] showed that when n is large and np (the expected outdegree) is equal to a constant c greater than one, it is likely that the graph contains one large component, called the *giant component*, and the other components are small. When n is constant and np grows, the percentage of vertices in the giant component increases rapidly. Our experiments showed that also when n is small, e.g., $n = 100$, the giant component grows rapidly when np grows. In Figure 5.2, we present the results of a simulation that measured the expected value of the percentage of vertices in the largest component of the graph. Parameter n varied between 100 and 50000, and np varied between 0.0 and 8.0. In these simulations, $\alpha = 0.05$ and $\epsilon_{\max} = 0.05$. As we see, the proportion of vertices that were in the giant component was almost independent of n. When np was less than one, no giant component existed. When np was two, about 60% of the vertices were in the giant component, and when np was greater than four, almost all vertices were in the giant component. This

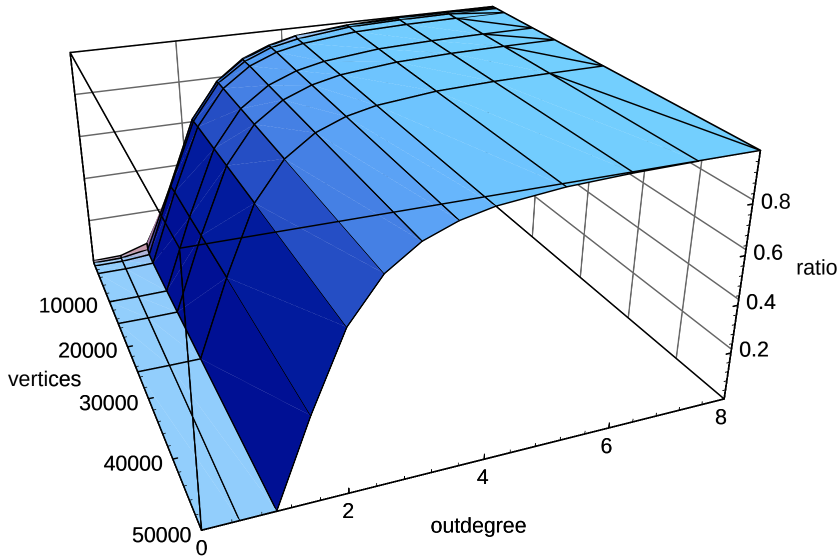

FIGURE 5.2: The percentage of vertices in the largest component in model $G(n,p)$.

phenomenon has a well-known counterpart in the theory of undirected random graphs [19]: also undirected random graphs $G(n,p)$ usually contain one large connected component and the other components contain only a few vertices.

Karp [79] showed also that a component that has more than one vertex, but that is not the giant component, is unlikely. In Figure 5.3, we present the percentage of vertices that are outside the giant component and the trivial components. As we see, almost all vertices are either in the giant component or in trivial components.

If we would have used $G(n,p)$ as our only simulation model, we would have gained no information on graphs with many nontrivial components. To overcome this problem, we decided to also use another model of directed random graphs $G(n,p,l)$, defined as follows:

Definition. The model $G(n,p,l)$ of random graphs is a probability space that contains all directed graphs having the same vertex set $V = \{1,2,\ldots,n\}$ and an edge set $E \subseteq \{(i,j) \mid i,j \in V \wedge j \in [(i-l) \mod n, \ldots, (i+l) \mod n]\}$. Each possible edge of a graph in $G(n,p,l)$

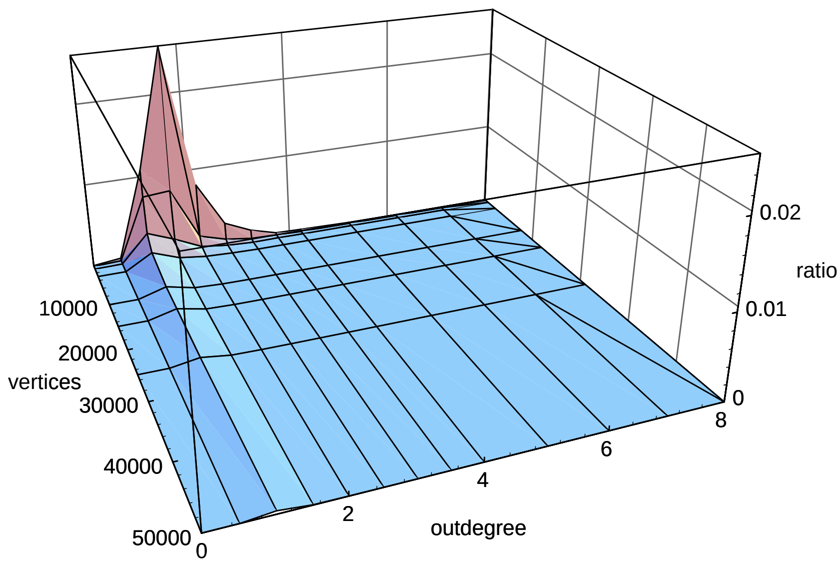

FIGURE 5.3: The percentage of vertices outside the largest component and the trivial components in model $G(n,p)$.

exists with probability p independently of the existence of any other edge in the graph; thus, the probability of a given graph $G = (V, E)$ with e edges is $p^e(1-p)^{n(2l+1)-e}$.

This means that edges leaving a vertex i can only go to a vertex j that is between $(i - l) \bmod n$ and $(i + l) \bmod n$ in the modular arithmetic. Parameter l is called the *locality*.

In Figure 5.4, we present the expected percentage of vertices outside the largest component and the trivial components when the graphs were drawn from $G(n, p, l = 5)$. When the expected outdegree $(2l + 1)p$ was between two and four, most vertices were in nontrivial components other than the giant component. The behavior depended only on the outdegree, not on the number of vertices. When the expected outdegree approached five, the graphs became strongly connected. We got similar results with other small values of l, e.g., $l = 10$. When l increased, $G(n, p, l)$ started to resemble $G(n, p)$. Thus, we can consider $G(n, p)$ to have the greatest possible locality. It seems that $G(n, p, l)$ with a small value of l is a good model for studying sparse graphs with many nontrivial components. Ioannidis et al. [64] used a similar model in their simulations.

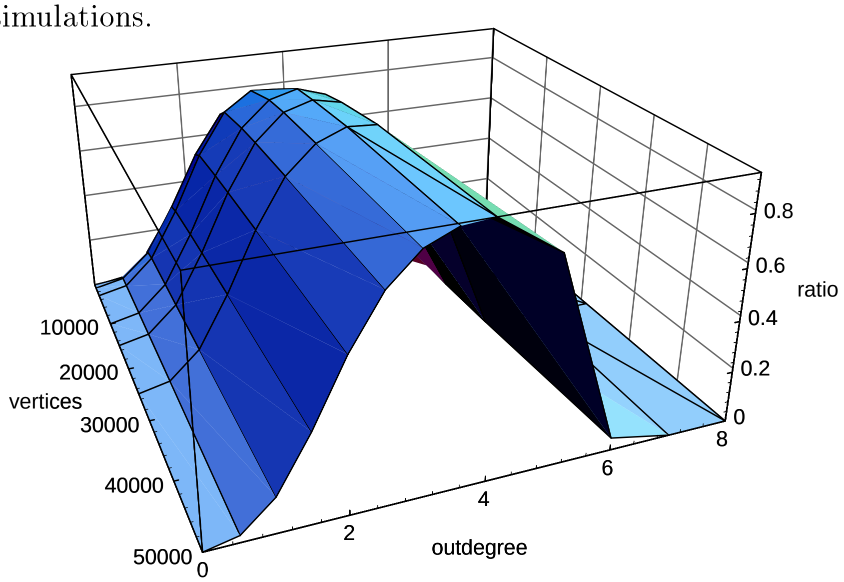

FIGURE 5.4: The percentage of vertices outside the largest component and the trivial components in model $G(n, p, l = 5)$.

Generating inputs

Since the algorithms that we study are efficient, generating the inputs may dominate the execution time in the simulations. We describe below how random graphs of model $G(n, p)$ can be efficiently generated. Random graphs of model $G(n, p, l)$ can be generated with only slight modifications.

By definition, we could generate a random graph $G = (V, E)$ of $G(n, p)$ as follows: for each possible edge (i, j), draw a uniformly distributed random number u, $0 \le u \le 1$. If $u \le p$, then insert (i, j) into E. The weakness in this method is that we always need $\Omega(n^2)$ time, even when the generated graph has $O(n)$ edges.

A simple probability theoretic observation makes it possible to use only $O(n + e)$ time to generate G [78]. In generating the edges, we do n^2 independent 0–1 trials, where the probability of 1 is p in each trial. By definition, the distance between two consecutive trials that yield 1 is geometrically distributed with parameter p. Thus, we generated the edges of G by using the algorithm presented in Figure 5.5.

```
x := 0;
while true do begin
    x := x + geom(p);
    if x > n^2 then
        return
    else
        insert (⌈x/n⌉, x mod n + 1) into E
end
```

FIGURE 5.5: An algorithm for generating the edges of a random graph of model $G(n, p)$.

The procedure draws $e+1$ geometrically distributed random numbers and generates one new edge per each of these random numbers except the last. Since the edge generation procedure above yields the edges in ascending order, we randomized the order of edges in the adjacency lists after the generation.

We generated geometrically distributed random numbers efficiently by drawing a uniformly distributed random number u, $0 \leq u \leq 1$, and converting it into a geometrically distributed random number x by an inverse transformation [100]

$$x = \lceil \ln(u)/\ln(1-p) \rceil \tag{5.5}$$

In generating the uniformly distributed random numbers, we used a linear-congruential generator

$$u_i + 1 = 7^5 u_i \bmod (2^{31} - 1) \tag{5.6}$$

The same generator is used, for instance, in IBM's SIMPL/1 [59] simulation package and the mathematical library IMSL [60]. This generator was highly recommended by Park and Miller [98] in their study of random number generators.

5.1.4 Performance evaluation design

Above we have discussed three important issues in a performance evaluation study: selecting the evaluation technique, coping with the variance of performance metrics, and selecting the inputs. We discuss now some other important issues that must be considered in designing a performance evaluation study.

Jain presented in his book [69] an overall approach to performance evaluation studies that consists of ten steps. We list these steps below and explain how the steps apply to our experiments.

1. *State the goals of the study and define the system boundaries.* If this step is missed, it is unlikely that any useful results are obtained in the study. Our goals were simple, e.g., to compare the number of operations in different algorithms. The system boundaries were mostly obvious.

2. *List services and possible outcomes.* A computer system that is evaluated offers a set of services. For instance, a computer network allows its users to send packets to specified destinations. Each service has several possible outcomes, some desirable and some undesirable. For instance, the packets may be delivered to the correct destination in the computer network or may get lost. In our experiments, the evaluated system offered a single service, computing the transitive closure of the given graph, and the service had only desirable outcomes.

3. *Select performance metrics.* The performance metrics are the criteria to compare the performance. We used several different metrics: the number of specific operations, the size of a data structure in number of elements, and the execution time in seconds.

4. *List system and workload parameters.* All parameters that affect the performance of the evaluated system should be identified. Workload parameters characterize the inputs that are given to the system. System parameters characterize the software and the hardware. We have above described the input models that we used in our experiments and listed the parameters used in generating the inputs. Most system parameters were related to the software, e.g., the algorithms and data structures that were used.

5. *Select factors and their levels.* Those parameters that are varied in the experiment are called factors. The values of the factors are called levels.

6. *Select the evaluation technique.* The three categories of evaluation techniques are analytical modeling, simulation, and direct measuring of a real system. We explained above why we selected simulation as our main evaluation technique. In the execution time experiments, we used direct measuring.

7. *Select workload.* The workload consists of a list of service requests to the evaluated systems. Our workloads were random graphs drawn from $G(n, p)$ and $G(n, p, l)$.

8. *Design experiments.* We should select a sequence of experiments that offer maximal information with minimal effort. In our experiments, we first ran a few initial simulation runs with inaccurate confidence intervals to get an overall picture of the behavior of the system. Then we selected appropriate levels for the input factors and ran the simulations again with more accurate confidence intervals. Thus, the selection of the levels of the input factors and the experimental design were connected to each other.

9. *Analyze and interpret data.* Since we used the sequential simulation procedure, no additional statistical analysis of the result was needed. In some studies, we used a regression model to fit curves to the data.

10. *Present results.* We present the results of the experiments in graphic form using two or three dimensional plots.

5.2 The size of transitive closure representations

5.2.1 Experimental setting

Our goal was to compare the average size of different transitive closure representations. We wanted to find out which representations need the smallest and the largest memory space on the average in different input models. We also wanted to find out how the average memory requirements vary with different input parameter levels. The representations were

1. Successor lists that contain vertices when components are not detected, i.e., each vertex has a separate successor list. We call this representation the *$v \times v$ -representation*, since its size corresponds to the size of the transitive closure relation represented as pairs of vertices.

2. Successor lists that contain vertices when components are detected, i.e., each component has a single successor list and the list contains vertices. We call this representation the *$c \times v$ -representation*, since its size corresponds to the size of the transitive closure relation represented as component–vertex pairs.

3. Successor lists that contain components when components are detected, i.e., each component has a single successor list and the list contains components. We call this representation the *$c \times c$ -representation*, since its size corresponds to the size of the transitive closure relation of the condensation graph represented as pairs of components.

4. Interval representation, presented in section 4.2.

5. Chain representation, presented in section 4.3.

We used the number of required memory words as the performance metric. We assumed that in a list representation one vertex or component takes one word of memory space. A pair in the chain representation or an interval in the interval representation takes two words. No other system parameters were used.

We computed the sizes of the different representations by first computing the transitive closure using algorithm STACK_TC and the interval representation and then computing the sizes of the $v \times v$ -representation, $c \times v$ -representation, and $c \times c$ -representation by using the interval representation and the strong components. Since the interval representation is time and space efficient, this approach made it possible to use large input graphs. Unfortunately, the size of the chain representation cannot be computed directly from the interval representation. Therefore, we also had to compute the transitive closure using the chain representation.

The workload consisted of randomly generated sparse graphs drawn from $G(n, p)$ and $G(n, p, l)$. We did not study dense inputs, since the behavior of each representation is obvious when the inputs are dense: if the strong components are not detected, we usually need quadratic memory space, and if components are detected, we usually need linear or smaller than linear memory space.

The workload factors were the number of vertices n, the expected outdegree d, and the locality l. The edge probability p of the input model was computed from d using n and l. After some initial simulation runs, we selected the following levels for the factors: factor n

had seven different levels 100, 300, 1000, 3000, 10000, 25000, and 50000, factor d had levels ranging from 0.0 to 6.0 by 0.1, and factor l had levels 5, 10, and 20. We could not use vertex counts $n = 25000$ and $n = 50000$ in model $G(n, p)$ with the chain representation, since the representation required too much memory.

The simulation control parameters were $\alpha = 0.1$ and $\epsilon_{\max} = 0.1$ or $\delta_{\max} = n/100$, i.e., we computed a 90% confidence interval I such that either the relative error of I is at most 10% or the absolute error of I is at most $n/100$. This combined stopping condition was needed, since the sizes of some representations are small in some parts of the input space and large in other parts of the input space.

5.2.2 Results

We study how the average size varied in different representations when the input model and the parameter levels were varied. We present some results using three dimensional plots with the number of vertices n and the expected outdegree d on the x- and y-axis. The average size is presented on the z-axis as a *representation size ratio*, i.e., the average number of memory words required divided by the number of vertices. Examine carefully the scale of the z-axis; since the maximum average size varied much in different representations and also in different input models with the same representation, we cannot use the same scale on the z-axis in all graphs. We also describe the dependency of the maximum average sizes on the number of vertices n. We estimated these dependencies from the simulation results using a regression model.

The average size of the $v \times v$ -representation

We study first the $v \times v$ -representation. In each input model, the size of the $v \times v$ -representation increased monotonically as the expected outdegree d grew. The growth was fastest at those levels of d where large components started to emerge. When the size of components increased, the size of the representation rapidly approached n^2 words. In model $G(n, p)$, this happened between outdegrees one and two (Figure 5.6), in model $G(n, p, l = 20)$ between outdegrees two and three, in model $G(n, p, l = 10)$ between outdegrees three and four, and in model

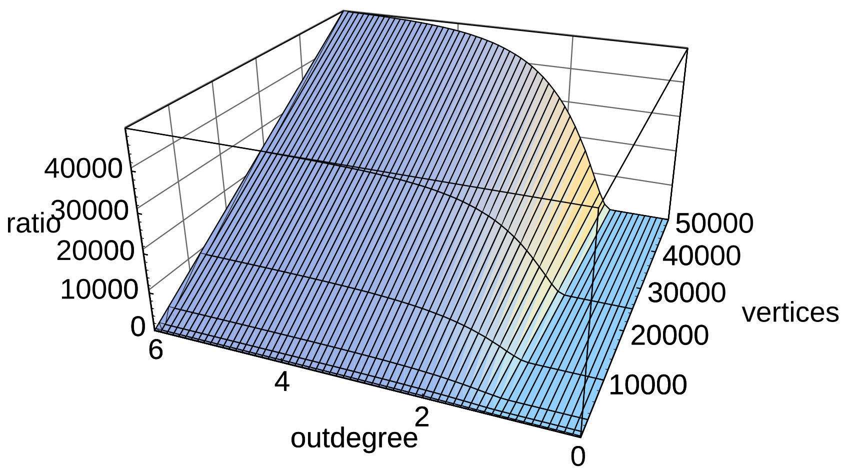

FIGURE 5.6: The representation size ratio of the $v \times v$ -representation in model $G(n, p)$.

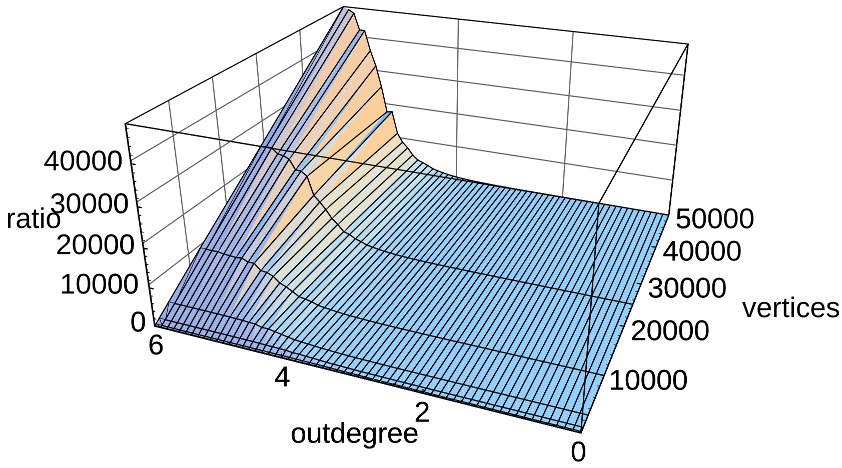

FIGURE 5.7: The representation size ratio of the $v \times v$ -representation in model $G(n, p, l = 5)$.

$G(n, p, l = 5)$ between outdegrees five and six (Figure 5.7). Thus, the rapid growth started earlier with greater values of locality l.

The average size of the $c \times v$ -representation

The size of $c \times v$ -representation first increased monotonically as d grew and then decreased monotonically. The difference between the behavior in different models was that the greater the locality was, the greater the maximum average size and the earlier the maximum was reached. In model $G(n, p, l = 5)$, the maximum average size was about $1.2n^{1.3}$ words (reached at $d \approx 5$), in model $G(n, p, l = 10)$ about $0.40n^{1.7}$ words (reached at $d \approx 4$), and in model $G(n, p, l = 20)$ about $0.15n^{1.9}$ words. In model $G(n, p)$, the maximum average size was about $0.15n^2$ words (reached at $d \approx 1.7$ (Figure 5.8)). Thus, the dependency of the maximum average size on the number of vertices seemed to become quadratic when the locality grew.

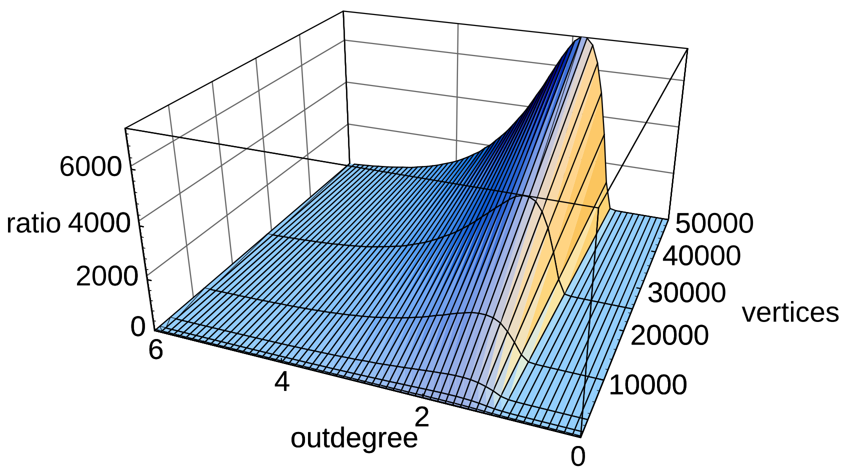

FIGURE 5.8: The representation size ratio of the $c \times v$ -representation in model $G(n, p)$.

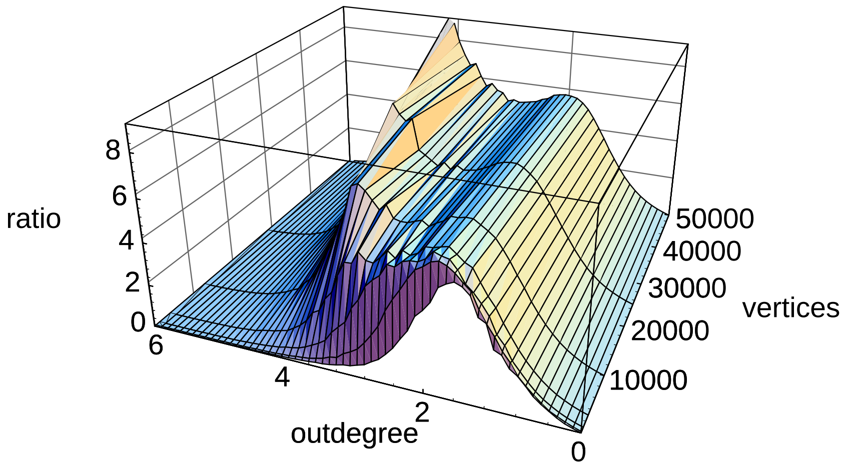

FIGURE 5.9: The representation size ratio of the $c \times c$ -representation in model $G(n, p, l = 10)$.

The average size of the $c \times c$ -representation

The $c \times c$ -representation behaved like the $c \times v$ -representation in models $G(n, p)$ and $G(n, p, l = 20)$, except that the maximum average sizes were smaller (about $0.063n^2$ words in $G(n, p)$ and about $0.11n^{1.7}$ words in $G(n, p, l = 20)$) and were reached at lower levels of d. In model $G(n, p, l = 10)$, the behavior was more complicated (Figure 5.9). With each number of vertices n, the average size had two local maxima, at $d \approx 2$ and at $d \approx 4$. The latter maximum was the global maximum, about $0.61n^{1.3}$ words. In model $G(n, p, l = 5)$, the behavior was simpler: the average size increased when the outdegree grew from $d = 0$ to $d \approx 2$ and then decreased (Figure 5.10). The maximum average size was about $2.8n$ words.

The average size of the chain representation

The chain representation behaved like the $c \times c$ -representation in all models, except that the average sizes were mostly about twice as large as in the $c \times c$ -representation.

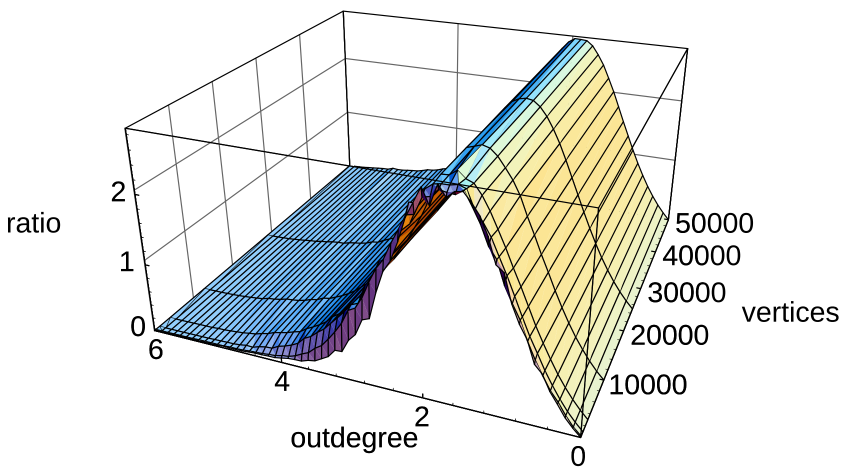

FIGURE 5.10: The representation size ratio of the $c \times c$ -representation in model $G(n, p, l = 5)$.

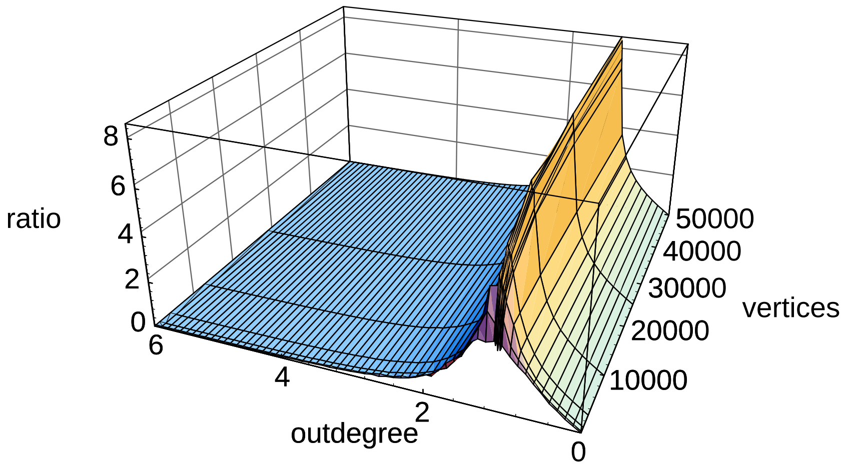

FIGURE 5.11: The representation size ratio of the interval representation in model $G(n,p)$.

The average size of the interval representation

The interval representation behaved in models $G(n,p,l)$ with all three values of l like the $c \times c$ -representation in model $G(n,p,l=5)$. The average size first increased when the outdegree grew and then decreased, and the maximum average size grew linearly to the number of vertices. The maximum average size was about $2.2n$ words in $G(n,p,l=5)$, about $2.6n$ words in $G(n,p,l=10)$, and about $3.3n$ words in $G(n,p,l=20)$. In model $G(n,p)$, the behavior was otherwise similar to the behavior in models $G(n,p,l)$ except that the maximum average size did not grow linearly to the number of vertices, but was about $0.55n\log n$ words (Figure 5.11). The maximum average size was also reached with smaller values of d.

Table 5.1 shows the maximum average sizes of the representations at levels $n = 10000$, $n = 25000$, and $n = 50000$. In addition, it shows the estimated growth of the maximum average size as a function of n.

5.2.3 Discussion

In all input models, the maximum average size was largest in the $v \times v$ -representation and smallest in the interval representation. The $v \times v$ -representation required quadratic memory space in all input models and in most expected outdegrees. Hence, the $v \times v$ -representation can only be used with small inputs. Strong components must be used if larger inputs are processed. As expected, the $c \times v$ -representation required less memory than the $v \times v$ -representation, but more than the $c \times c$ -representation. Contrary to our initial expectations, the chain representation saved no space compared to the $c \times c$ -representation. Thus, the chain representation seems to be useful only with acyclic graphs. The interval representation was the clear winner. Its maximum average size was linear to the number of vertices in models $G(n,p,l)$ and approximately $0.55n\log n$ in model $G(n,p)$. The maximum average sizes of all other representations were quadratic to the number of vertices at least in one of the input models. The interval representation was usually smaller than the input graph.

Representation	Locality	$n = 10000$	$n = 25000$	$n = 50000$	Growth
$v \times v$	none	1.0×10^8	6.3×10^8	2.5×10^9	n^2
	$l = 20$	1.0×10^8	6.3×10^8	2.5×10^9	n^2
	$l = 10$	1.0×10^8	6.3×10^8	2.5×10^9	n^2
	$l = 5$	1.0×10^8	6.3×10^8	2.5×10^9	n^2
$c \times v$	none	1.5×10^7	9.2×10^7	3.7×10^8	$0.15n^2$
	$l = 20$	6.5×10^6	3.2×10^7	1.2×10^8	$0.15n^{1.9}$
	$l = 10$	2.1×10^6	8.8×10^6	2.9×10^7	$0.4n^{1.7}$
	$l = 5$	$250,000$	$820,000$	2.1×10^6	$1.2n^{1.3}$
$c \times c$	none	6.3×10^6	3.9×10^7	1.6×10^8	$0.063n^2$
	$l = 20$	$590,000$	2.5×10^6	7.9×10^6	$0.11n^{1.7}$
	$l = 10$	$65,000$	$200,000$	$460,000$	$0.61n^{1.3}$
	$l = 5$	$29,000$	$71,000$	$140,000$	$2.8n$
Chains	none	6.9×10^6	–	–	$0.069n^2$
	$l = 20$	$930,000$	4.0×10^6	1.3×10^7	$0.13n^{1.7}$
	$l = 10$	$110,000$	$350,000$	$820,000$	$1.1n^{1.3}$
	$l = 5$	$28,000$	$70,000$	$140,000$	$2.8n$
Intervals	none	$73,000$	$200,000$	$430,000$	$0.55n \log n$
	$l = 20$	$33,000$	$82,000$	$160,000$	$3.3n$
	$l = 10$	$27,000$	$66,000$	$130,000$	$2.6n$
	$l = 5$	$21,000$	$54,000$	$110,000$	$2.2n$

TABLE 5.1: The maximum average sizes of the representations.

5.3 Successor set construction

5.3.1 Experimental setting

Our goal was to detect the average complexity of constructing the successor sets in different algorithms and representations. The algorithms and representations were:

1. EBERT [36] and unordered list representation.
2. EBERT and interval representation.
3. SCHMITZ [105] and unordered list representation.
4. SCHMITZ and interval representation.
5. CR_TC, presented in section 3.3, and unordered list representation.
6. CR_TC and interval representation.
7. STACK_TC, presented in section 3.4, and unordered list representation.
8. STACK_TC and interval representation.

We chose EBERT of those algorithms that compute the successor sets during the strong component detection, since it does at most as many unions as EKS [40], and it is easier to implement than GDFTC [64], which does the same number of unions as EBERT. EBERT constructs the successor sets from vertices instead of components. Hence, we had to modify the interval representation to use vertex exit numbers instead of component numbers in EBERT. Since at most one successor set is under construction at any moment during the execution of STACK_TC and SCHMITZ, we used a bit vector in these algorithms for duplicate elimination. Since many successor sets are simultaneously under construction in CR_TC and EBERT, we did not use bit vectors in CR_TC and in EBERT. For the same reason, the interval representation stored the intervals as binary trees in EBERT and in CR_TC, but as arrays in SCHMITZ and in STACK_TC.

To find out the savings gained by topological sorting in STACK_TC, we also used a version of STACK_TC, called STACK_TC_N, which does not sort the components in the component stack.

The performance metric was the total number of basic elements accessed in successor set operations. A basic element was a node of a list in the list representation and an interval in the interval representation. We call the average of the total number of accessed basic elements the *access count*. In addition to the access count, we counted separately the numbers of basic elements accessed in unions, insertions, membership tests, and set initializations.

The workload consisted of randomly generated sparse graphs drawn from $G(n,p)$ and $G(n,p,l=10)$. The workload factors were the number of vertices n and the expected outdegree d. After some initial simulation runs, we selected the following levels for the factors: factor n had five different levels 1000, 3000, 10000, 25000, and 50000, and factor d had levels ranging from 0.0 to 10.0 by 0.1. Since the list representation required too much memory, we could not use levels of n above $n = 10000$. With EBERT, we could not even use level $n = 10000$.

The simulation control parameters were $\alpha = 0.05$ and $\epsilon_{\max} = 0.05$ or $\delta_{\max} = n/100$, i.e., we computed a 95% confidence interval I such that either the relative error of I is at most 5% or the absolute error of I is at most $n/100$.

5.3.2 Results

We present the results using three dimensional plots with the number of vertices n and the expected outdegree d on the x- and y-axis. The access count divided by the number of vertices is presented on the z-axis. Note that the scale of the x-axis and the scale of the z-axis are different in different plots. We also describe the dependency of the maximum access count on the number of vertices n. Note that the maximum access count is the *greatest average value* at a fixed level of n, not the greatest value detected in all simulation runs.

Ebert's algorithm and the list representation

In both models, EBERT behaved almost as in the worst case. The access count was quadratic to the number of vertices and linear to the expected outdegree. In $G(n,p)$, the access count was about $0.42n^2d$ (Figure 5.12). In $G(n,p,l=10)$ the access count was about $0.17n^2d$. Most of these accesses occurred in insertions. Since the memory requirements were quadratic to the number of vertices, we could not use vertex counts greater than $n = 3000$.

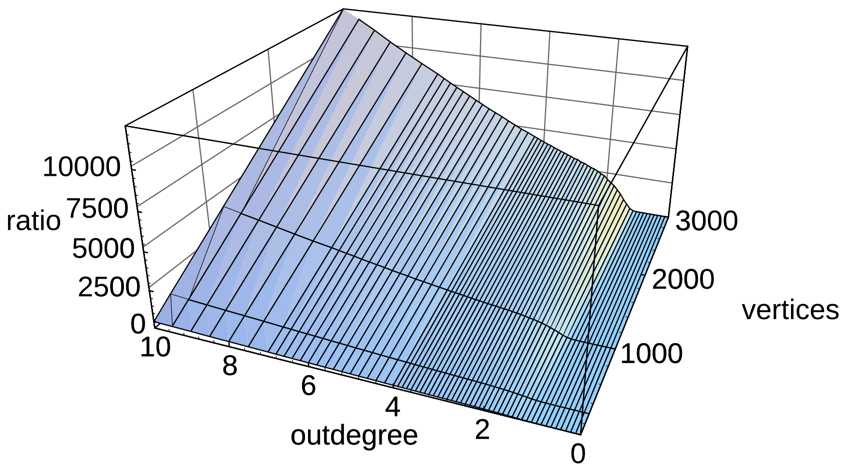

FIGURE 5.12: The access count divided by the number of vertices in EBERT with the list representation: model $G(n, p)$.

Ebert's algorithm and the interval representation

We encountered memory space problems here also and could not use vertex counts above $n = 3000$. The memory requirements and the access counts were here too quadratic to the number of vertices. In model $G(n, p)$, the access count increased between outdegrees $d = 0.0$ and $d \approx 2.0$ and then decreased (Figure 5.13). The maximum access count, reached at $d \approx 2.0$, was about $0.25n^2$. If we had used greater values of d, the access count would have increased again, since the access count in insertions seemed to grow linearly to the expected outdegree. In model $G(n, p, l = 10)$, the access count increased rapidly from $d = 0.0$ to $d \approx 5.0$ and then increased slowly between $d \approx 5.0$ and $d = 10.0$ (Figure 5.14). At $d = 10.0$, the access count was about $0.23n^2$. In both models, most accesses occurred in unions.

Algorithm CR_TC and the list representation

In both models, the access count first increased as the expected outdegree grew and then decreased. In $G(n, p)$, the maximum access count was about $0.14n^2$ and it was reached at $d \approx 1.5$ (Figure 5.15). In $G(n, p, l = 10)$, the maximum access count was reached at $d \approx 3.6$

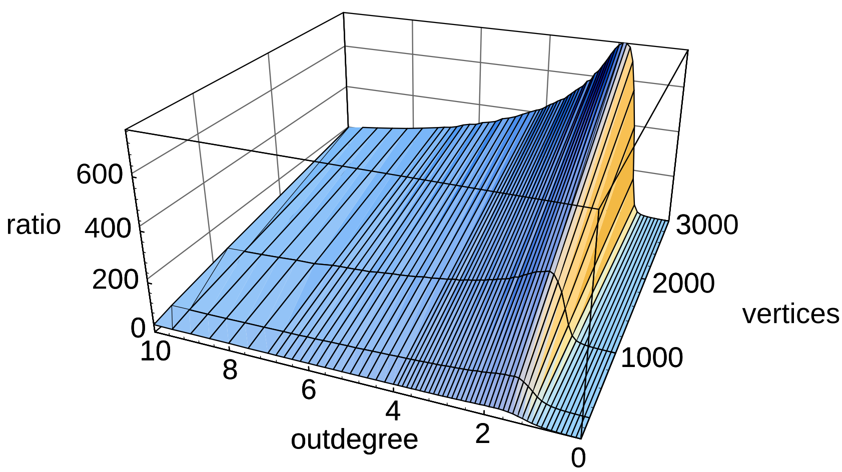

FIGURE 5.13: The access count divided by the number of vertices in EBERT with the interval representation: model $G(n, p)$.

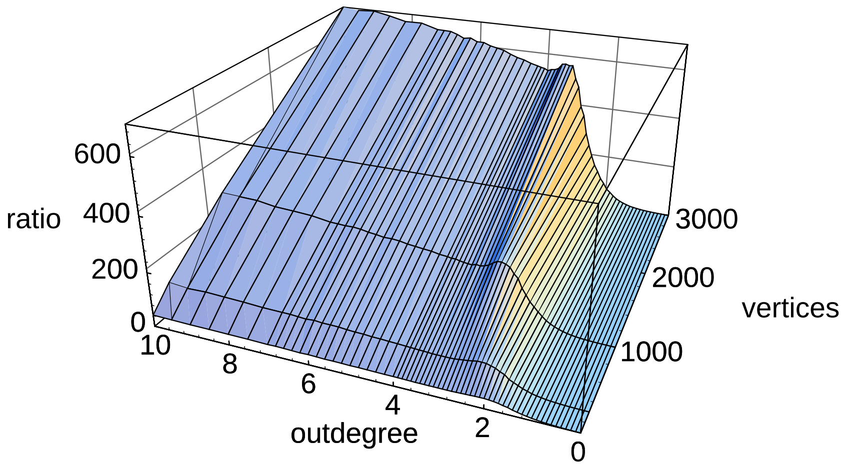

FIGURE 5.14: The access count divided by the number of vertices in EBERT with the interval representation: model $G(n, p, l = 10)$.

(Figure 5.16). The maximum access count seemed to be neither linear nor quadratic to the number of vertices. In both models, the number of accesses in unions and insertions were roughly equal.

Algorithm CR_TC and the interval representation

In model $G(n, p)$, CR_TC behaved with the interval representation as with the list representation except that the maximum access count was much smaller, about $1.4 \times 10^{-3}n^2$. Most accesses occurred in unions. In model $G(n, p, l = 10)$, the maximum access count was $9.6 \times 10^{-5}n$, i.e., linear to the number of vertices (Figure 5.17). The number of accesses in unions and insertions were roughly equal.

Schmitz's algorithm and the list representation

In model $G(n, p)$, the access count increased as the expected outdegree grew from $d = 0.0$ to $d \approx 1.5$ and then decreased (Figure 5.18). Although it cannot be seen in Figure 5.18, after $d \approx 4.0$ the access count started to grow again. This growth was linear to the expected

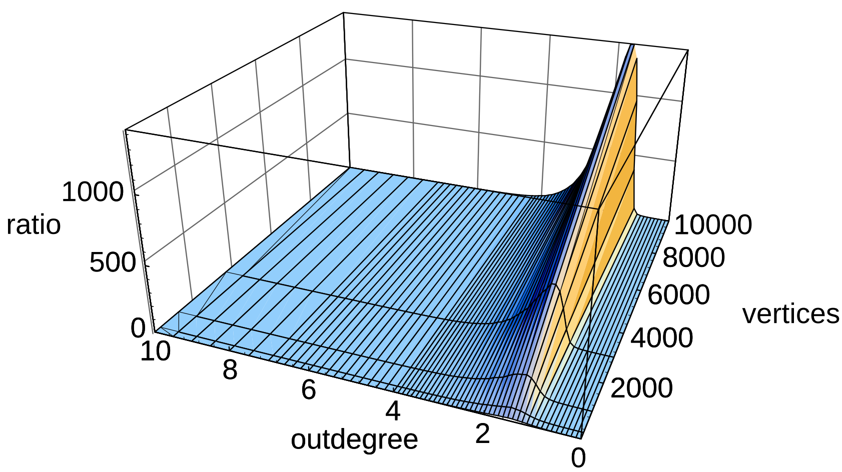

FIGURE 5.15: The access count divided by the number of vertices in CR_TC with the list representation: model $G(n, p)$.

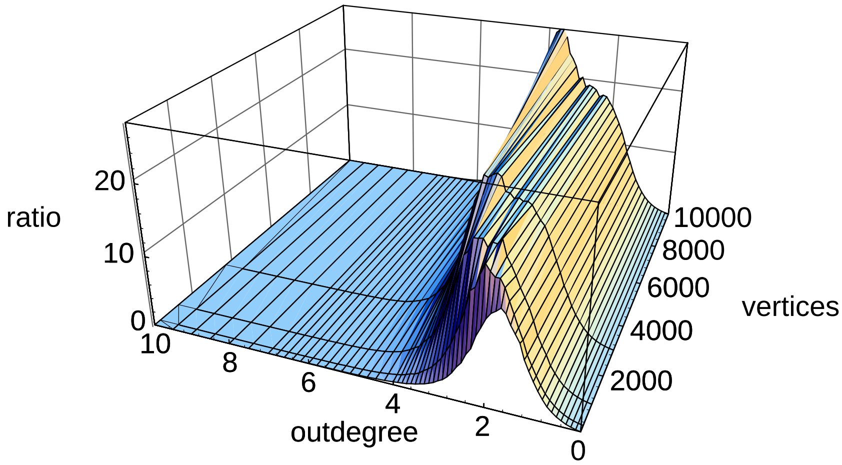

FIGURE 5.16: The access count divided by the number of vertices in CR_TC with the list representation: model $G(n, p, l = 10)$.

outdegree and was caused by insertions. The local maximum access count at $d \approx 1.5$ was about $0.082n^2$ and it was caused by unions. In model $G(n, p, l = 10)$, a similar local maximum appeared between $d = 2.0$ and $d = 3.8$; it was neither linear nor quadratic to the number of vertices (Figure 5.19).

Schmitz's algorithm and the interval representation

In model $G(n, p)$, SCHMITZ behaved with the interval representation roughly as with the list representation except that the local maximum access count at $d \approx 2.0$ was much smaller, about $1.6 \times 10^{-3}n^2$. In model $G(n, p, l = 10)$, the access count mostly increased between outdegrees from $d = 0.0$ to $d \approx 5.0$ and then decreased, but started to increase again at $d \approx 8.0$ (Figure 5.20). The (local) maximum at $d \approx 5.0$ was neither linear nor quadratic to the number of vertices.

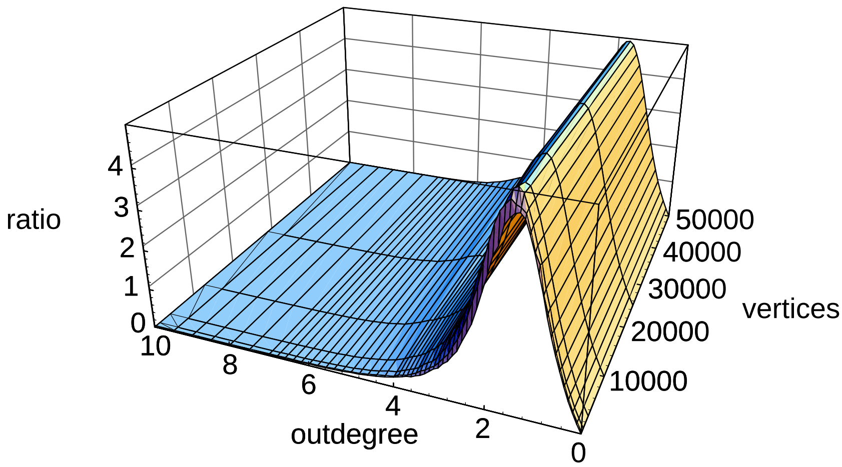

FIGURE 5.17: The access count divided by the number of vertices in CR_TC with the interval representation: model $G(n, p, l = 10)$.

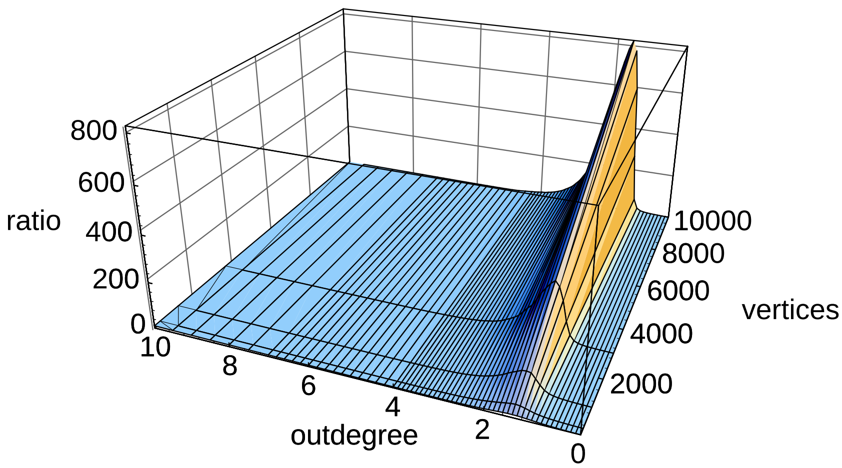

FIGURE 5.18: The access count divided by the number of vertices in SCHMITZ with the list representation: model $G(n, p)$.

Algorithm STACK_TC and the list representation

In model $G(n, p)$, STACK_TC behaved like CR_TC with the list representation. However, the maximum access count, reached at $d \approx 1.5$, was smaller, about $0.076n^2$. In model $G(n, p, l = 10)$, STACK_TC behaved roughly like CR_TC except that the maximum access count, reached between $d \approx 2.0$ and $d \approx 3.8$, was smaller and did not grow as fast as in CR_TC (Figure 5.21). In both models, most accesses occurred in unions. STACK_TC_N, the version of STACK_TC that does not sort the component stack, required only slightly more accesses than STACK_TC in both models.

Algorithm STACK_TC and the interval representation

In model $G(n, p)$, STACK_TC behaved with the interval representation as with the list representation except that the maximum access count was much smaller, about $6.8 \times 10^{-4}n^2$. The maximum access count of STACK_TC_N was about $1.4 \times 10^{-3}n^2$. In model $G(n, p, l = 10)$, the maximum access count of both algorithms was about $9.0 \times 10^{-5}n$, i.e., linear to the number of vertices (Figure 5.22). In both models, the number of accesses in unions and insertions were roughly equal.

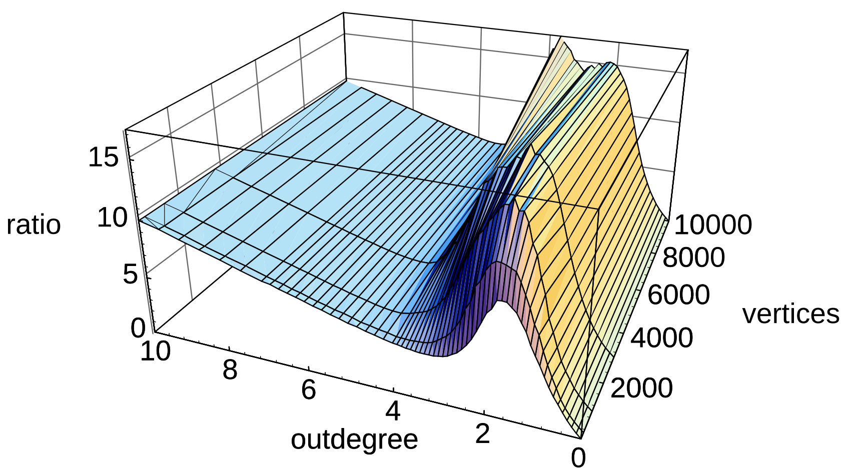

FIGURE 5.19: The access count divided by the number of vertices in SCHMITZ with the list representation: model $G(n, p, l = 10)$.

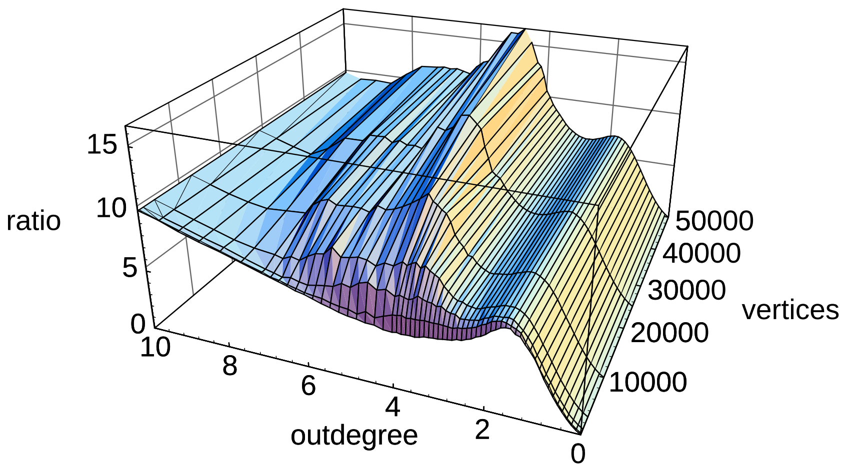

FIGURE 5.20: The access count divided by the number of vertices in SCHMITZ with the interval representation: model $G(n, p, l = 10)$.

Table 5.2 lists the maximum access counts at levels $n = 3000$, $n = 10000$, and $n = 25000$ when the expected outdegree was between 0.0 and 10.0. In addition, it shows the estimated growth of the maximum access count as a function of n.

5.3.3 Discussion

STACK_TC was the clear winner. In both models and with both representations, STACK_TC had the smallest maximum access count. STACK_TC_N usually did more accesses than STACK_TC, but the difference was significant only in model $G(n, p)$ with the interval representation. SCHMITZ was not much worse than STACK_TC and STACK_TC_N. Note, however, that when the expected outdegree grew, the access counts of STACK_TC and STACK_TC_N tended to zero, whereas the access count of SCHMITZ grew linearly. Note also that the maximum access counts grew linearly to the number of vertices in CR_TC, STACK_TC, and STACK_TC_N, but faster than linearly in SCHMITZ. Thus, the difference between SCHMITZ and our algorithms should become greater with greater inputs. EBERT was very inefficient compared to the other algorithms in both models and with both representations.

Other experiments that we have done also indicate that the maximum access counts of our

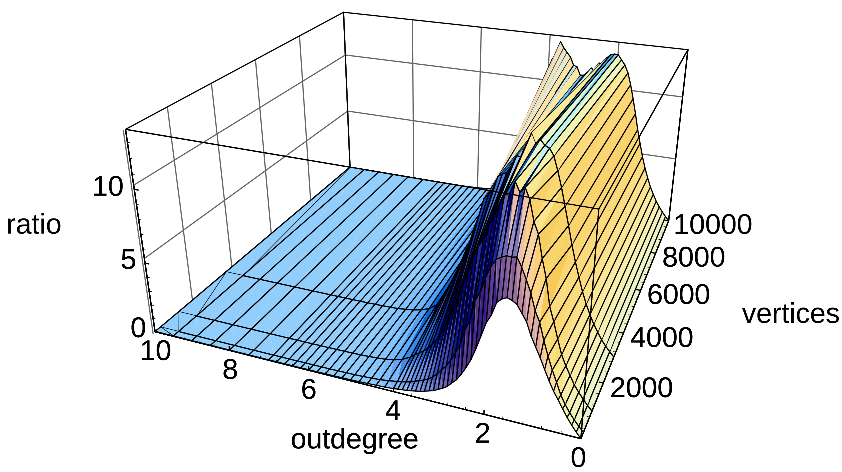

FIGURE 5.21: The access count divided by the number of vertices in STACK_TC with the list representation: model $G(n, p, l = 10)$.

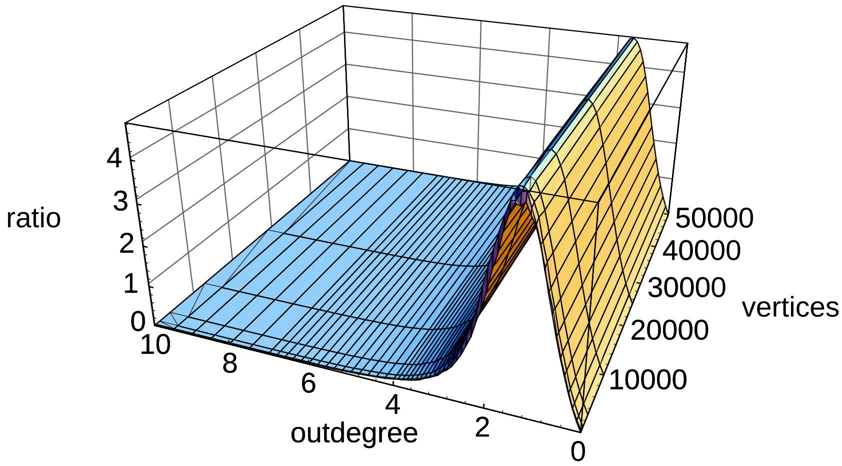

FIGURE 5.22: The access count divided by the number of vertices in STACK_TC with the interval representation: model $G(n, p, l = 10)$.

algorithms are linear to the number of vertices in all models $G(n, p, l)$ when l is small. Since the other operations in the algorithms take $O(n + e)$ time, these results indicate that in model $G(n, p, l)$ with a small value of l, our new algorithms should require a time linear to the size of the input graph on the average. In model $G(n, p)$, the algorithms should also require a linear time except when the expected outdegree is slightly greater than one. Then, the average execution time should be quadratic to the number of vertices.

Algorithm	Repr.	Loc.	$n = 3000$	$n = 10000$	$n = 25000$	Growth
EBERT	list	none	3.8×10^7	–	–	$0.40n^2$
	list	$l = 10$	1.5×10^7	–	–	$1.7n^2$
	interval	none	2.3×10^6	–	–	$0.25n^2$
	interval	$l = 10$	2.1×10^6	–	–	$0.23n^2$
CR_TC	list	none	1.3×10^6	1.4×10^7	–	$0.14n^2$
	list	$l = 10$	64,000	280,000	–	$3.0n \log n$
	interval	none	27,000	150,000	800,000	$1.2 \times 10^{-3}n^2$
	interval	$l = 10$	15,000	49,000	120,000	$4.9n$
SCHMITZ	list	none	770,000	8.3×10^6	–	$0.083n^2$
	list	$l = 10$	50,000	170,000	–	$1.9n \log n$
	interval	none	42,000	250,000	1.2×10^6	$1.7 \times 10^{-3}n^2$
	interval	$l = 10$	30,000	120,000	350,000	$1.5n \log n$
STACK_TC	list	none	710,000	7.7×10^6	–	$0.077n^2$
	list	$l = 10$	41,000	130,000	–	$1.4n \log n$
	interval	none	25,000	110,000	490,000	$7.0 \times 10^{-4}n^2$
	interval	$l = 10$	14,000	48,000	120,000	$4.8n$
STACK_TC_N	list	none	760,000	8.2×10^6	–	$0.082n^2$
	list	$l = 10$	44,000	140,000	–	$1.5n \log n$
	interval	none	27,000	190,000	1.0×10^6	$1.5 \times 10^{-3}n^2$
	interval	$l = 10$	15,000	51,000	130,000	$5.1n$

TABLE 5.2: The maximum access counts; $0.0 \leq d \leq 10.0$.

5.4 Execution time

5.4.1 Experimental setting

Our goal was to compare the execution times of different transitive closure algorithms. We were mainly interested in the relative efficiency of the algorithms. The algorithms were SCHMITZ, CR_TC, STACK_TC, and STACK_TC_N. Since EBERT was very inefficient compared to the other algorithms in the measurements presented in section 5.3, we decided not to use it.

We used only the interval representation, since our goal was to compare algorithms and not representation. Note that in [92], we presented a similar comparison between SCHMITZ and STACK_TC_N using unordered lists augmented with a bit vector as the representation.

The performance metric was the execution time measured using the timing facilities provided by the computer. The simulations were executed in single-user mode to eliminate measurement error caused by other processes. Note that here the method of evaluation was direct measurement and not simulation.

Although the computer that is used in these simulations is clearly a system parameter that affects the results, we assumed that the relative efficiency of the algorithms does not depend much on the computer. Therefore, we used only a single computer, a Sun Sparcstation-10/20 having 96 megabytes of main memory. The operating system was SunOS 4.1.3. We implemented the algorithms and the data structures in C++. An implementation parameter that affects the efficiency of the algorithms is whether or not recursion is removed. Deep recursion seemed to be inefficient in the Sparcstation architecture and in some situations dominated the execution times in our initial simulation runs. Therefore, we removed recursion from the algorithm implementations.

The workload consisted of randomly generated sparse graphs drawn from $G(n,p)$ and $G(n,p,l)$. The workload factors were the number of vertices n, the expected outdegree d, and the locality l. After some initial simulation runs, we selected the following levels for the factors: factor n had three levels 3000, 10000, 25000, factor d had levels ranging from 0.0 to 10.0 by 0.5, and factor l had levels 5 and 10.

The simulation control parameters were $\alpha = 0.05$ and $\epsilon_{\max} = 0.05$ or $\delta_{\max} = n/100$, i.e., we computed a 95% confidence interval I such that either the relative error of I is at most 5% or the absolute error of I is at most $n/100$.

5.4.2 Results

Since all algorithms behaved rather similarly in models $G(n,p,l=5)$ and $G(n,p,l=10)$, we present only the results in $G(n,p,l=10)$. For each vertex count n, we present the results using two dimensional plots having the expected outdegree in the x-axis and the execution time in the y-axis (Figures 5.23–5.28).

5.4.3 Discussion

STACK_TC_N was the overall winner. It was always faster than SCHMITZ and CR_TC. In model $G(n,p,l=10)$ with $d < 6$, STACK_TC_N was slightly faster than STACK_TC. In model $G(n,p)$, STACK_TC was slower than STACK_TC when n was small, but slightly faster than STACK_TC_N when n grew. This is consistent with the results of section 5.3; the maximum access count grew significantly faster in STACK_TC_N than in STACK_TC in model $G(n,p)$. Thus, the topological sorting in STACK_TC seems to become useful in model $G(n,p)$ when the graphs are large.

SCHMITZ behaved like STACK_TC between $d = 0.0$ and $d = 1.5$, but was slower with greater values of d. With greater values of d, SCHMITZ required twice or three times as much time as STACK_TC and STACK_TC_N. The extra time in SCHMITZ is caused by two factors. First, SCHMITZ always scans the input graph twice. Second, as the simulations in section 5.3 showed, SCHMITZ needs more successor set accesses than STACK_TC and STACK_TC_N.

CR_TC was considerably slower than the other algorithms between $d = 0.0$ and $d = 1.5$, but became faster than SCHMITZ when d was greater than three. It seems that the strategy of creating the successor sets during the strong component detection is inferior to the strategy of creating the successor set immediately after the detection of the component.

Interestingly, the execution times grew slightly faster than linearly to the number of vertices when they should have grown linearly. The explanation is that with smaller values of n, a greater portion of data fits into the processor cache. We confirmed this hypothesis by applying Tarjan's algorithm to the same inputs; the execution time of Tarjan's algorithm also grew faster than linearly to n.

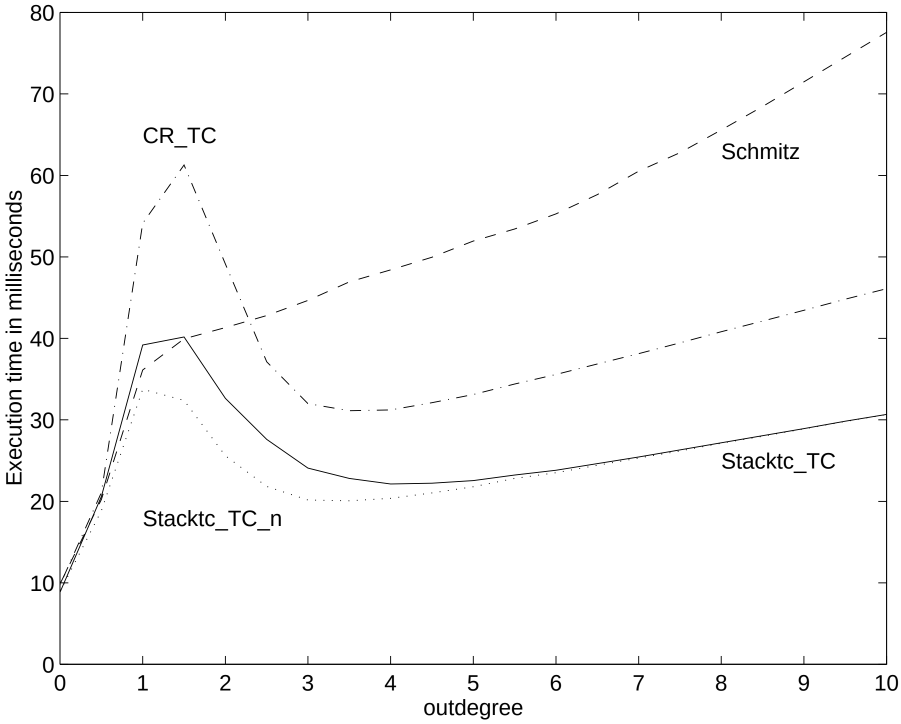

FIGURE 5.23: Execution times in model $G(3000, p)$.

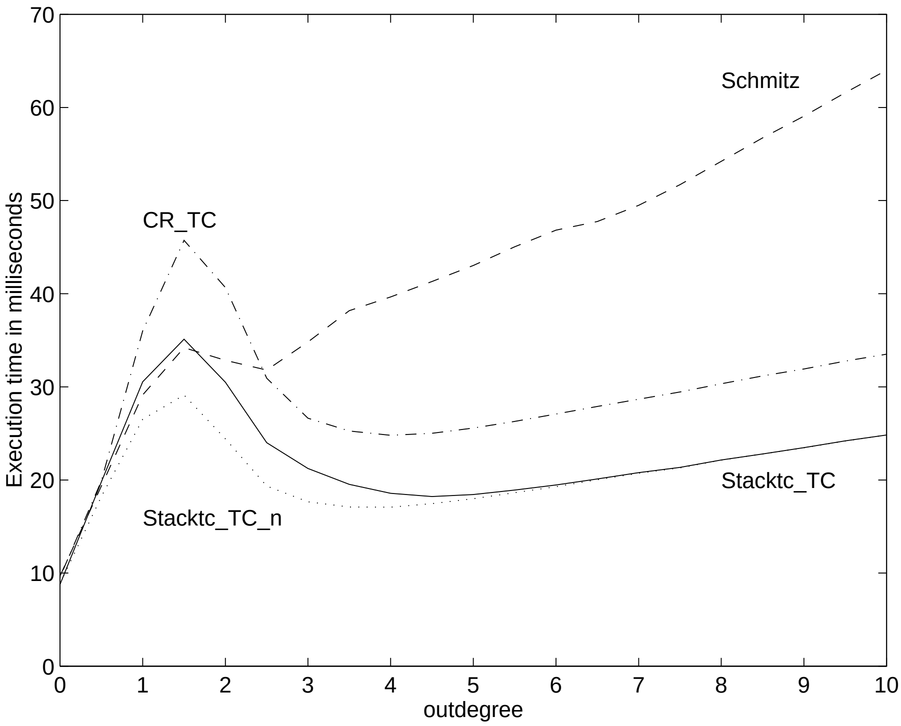

FIGURE 5.24: Execution times in model $G(3000, p, l = 10)$.

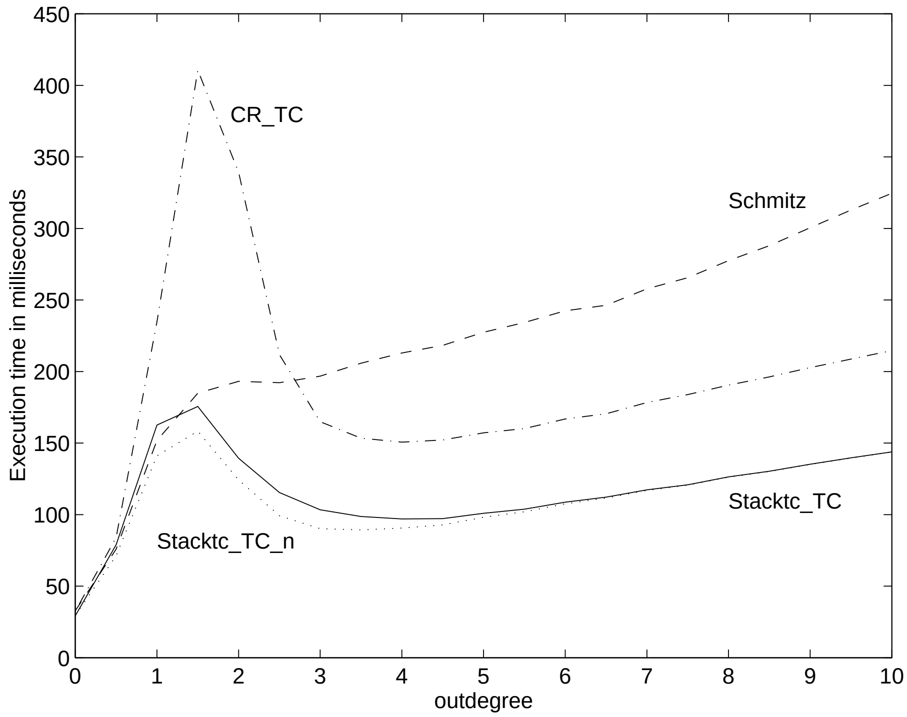

FIGURE 5.25: Execution times in model $G(10000, p)$.

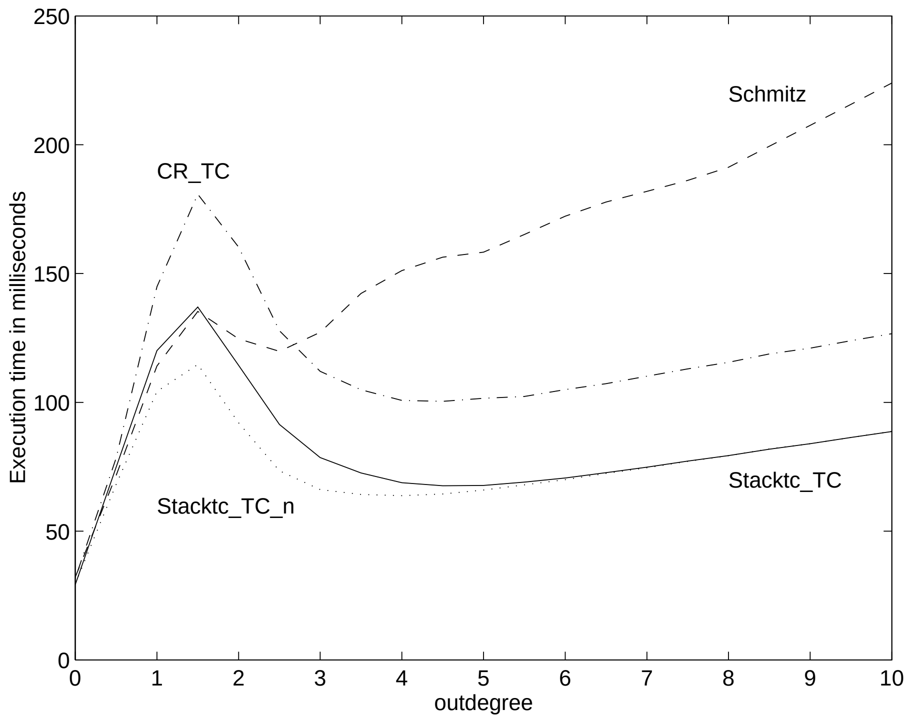

FIGURE 5.26: Execution times in model $G(10000, p, l = 10)$.

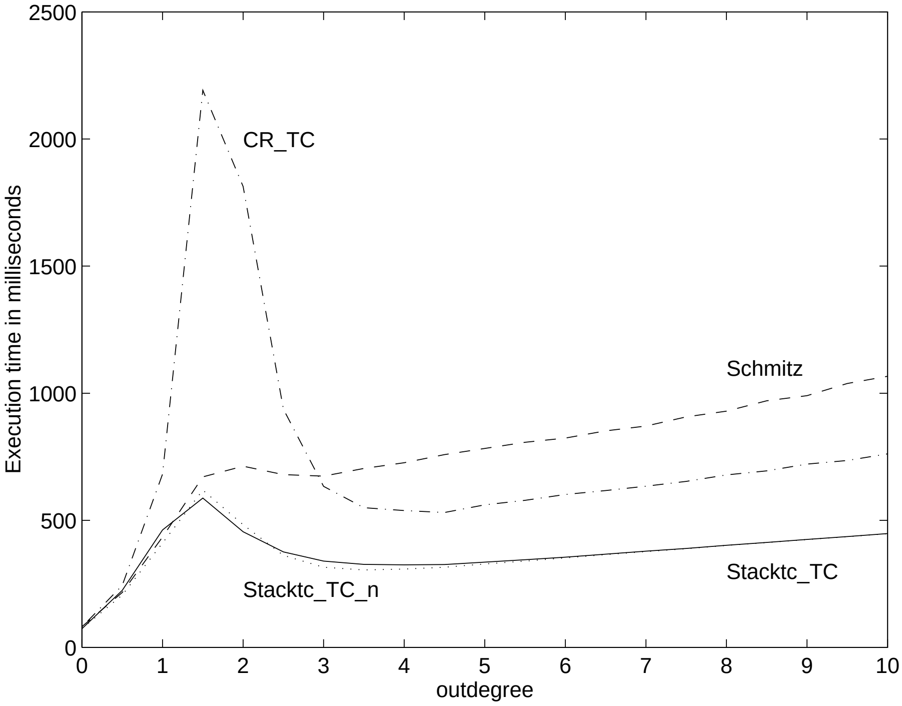

FIGURE 5.27: Execution times in model $G(25000, p)$.

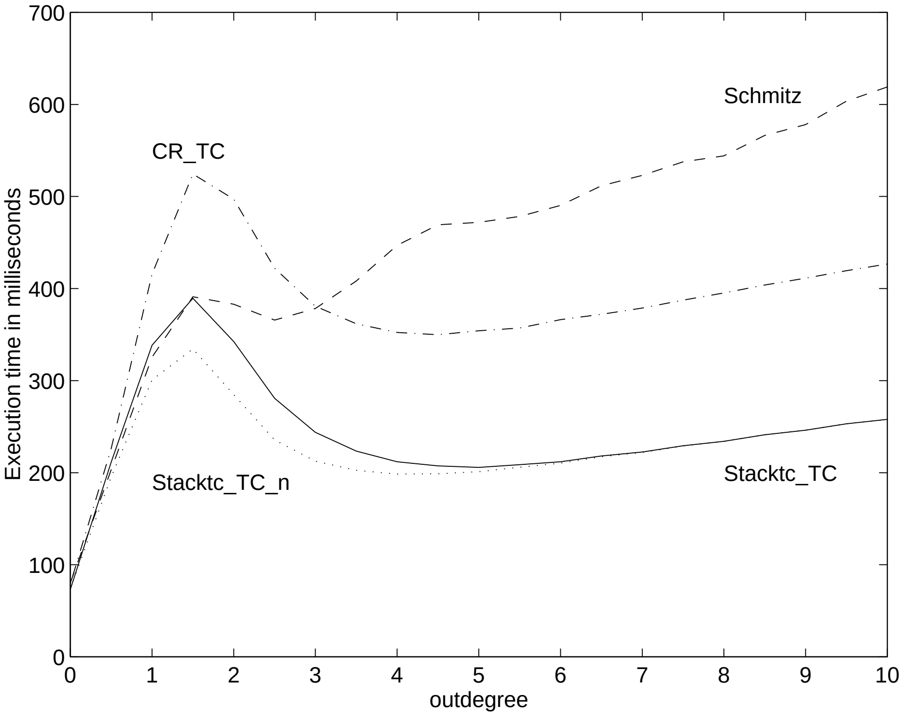

FIGURE 5.28: Execution times in model $G(25000, p, l = 10)$.