AVERAGE-CASE ANALYSIS OF GRAPH-SEARCHING ALGORITHMS

Sarantos Kapidakis

(Thesis)

CS-TR-286-90

October 1990

AVERAGE-CASE ANALYSIS OF GRAPH-SEARCHING ALGORITHMS

SARANTOS KAPIDAKIS

A DISSERTATION
PRESENTED TO THE FACULTY
OF PRINCETON UNIVERSITY
IN CANDIDACY FOR THE DEGREE
OF DOCTOR OF PHILOSOPHY

RECOMMENDED FOR ACCEPTANCE
BY THE DEPARTMENT OF
COMPUTER SCIENCE

October 1990

Average-case analysis of graph-searching algorithms – Abstract

Sarantos Kapidakis

Advisor – Professor Robert Sedgewick

We estimate the expected value of various search quantities for a variety of graph-searching methods, for example depth-first search and breadth-first search. Our analysis applies to both directed and undirected random graphs, and it covers the range of interesting graph densities, including densities at which a random graph is disconnected with a giant connected component.

We estimate the number of edges examined during the search, since this number is proportional to the running time of the algorithm. We find that for hardly connected graphs, all of the edges might be examined, but for denser graphs many fewer edges are generally required. We prove that any searching algorithm examines $\Theta(n \log n)$ edges, if present, on all random graphs with n nodes but not necessarily on the complete graphs.

One property of some searching algorithms is the maximum depth of the search. In depth-first search, this depth can be used to estimate the space needed for the recursion stack. For random graphs of any density, even for disconnected graphs, we prove that this space is $\Theta(n)$. On the other hand, the depth of breadth-first search is $\Theta(\log n/ \log(pn))$, where p is the probability of the existence of an edge. The size of the data structure needed by any searching algorithm is proved to be $\Theta(n)$. If the search terminates at a particular node, any searching algorithm needs a data structure of size $\Theta(n)$, and examines only $\Theta(n)$ edges.

Finally, we derive similar results for variants of the above searching algorithms, more general classes of searching algorithms, and for random graphs with multiple edges.

These results are verified through simulations. The techniques employed to permit simulations of big graphs (few millions of nodes) and the results obtained are of general interest, especially to those performing similar experiments.

Dedicated to

the memory of my father Ioannis Kapidakis

and to my mother Eleni Kapidaki

Acknowledgments

I would like to thank my advisor, Robert Sedgewick, for his encouragement, constructive comments, support and advice during my stay at Princeton. I would also like to thank my readers, Bernard Chazelle and Andrew Yao for their helpful comments and suggestions on my dissertation.

I am thankful to David Hanson for his comments on my writing style, and Leonidas Palios and especially Christos Polyzois, for proofreading my dissertation. Their corrections and comments were greatly appreciated.

Many thanks to Robert Tarjan for his thoughts on my dissertation, to Sharon Rodgers, for continuous support all these years, and to the computer staff, for keeping our computers running.

I would also like to thank the rest of the faculty, students, and staff of the Computer Science department and my friends, for being there when I needed them. Special thanks to my officemates, housemates, and to Dimitris and Giota Doukas and Christos Fotopoulos.

Sarantos Kapidakis

Princeton, New Jersey

September, 1990

Contents

Chapter 1

Preliminaries

Two fundamental graph algorithms, depth-first search and breadth-first search, are simple, efficient and easy to implement. They are commonly used alone or as components in other algorithms. In the worst case, they are linear in the size of the data; each edge is examined at most once in a directed graph and at most twice in an undirected graph. However, they have not yet been fully analyzed. In this thesis, we study the behavior of depth-first search, breadth-first search and related searching processes in more detail. Our work concentrates on the average-case analysis of searching methods on random graphs.

Algorithm behavior varies with the input data. In the general case, this behavior depends not only on the size of the data, but on the particular data as well. For a graph-searching algorithm, both the size of the graph and its shape and representation are important. Consequently, we must first specify a mathematical model for the probabilistic distribution of our input, i.e. graphs, and then analyze the average behavior of the algorithms for a random input distributed according to the model. This may not be a realistic model for the input in a particular application, but it is a natural way to begin analyzing these algorithms.

We use *random graph* [Bo85] as our graph model. A random graph $G_p(n)$ is a graph with n nodes in which any two nodes are joined by an edge with probability p independently of the existence of any other edge in the graph. If a random graph $G_p(n)$ has fewer than $n/2$ edges, all nodes belong to small components with probability one, none of which contains more than $\log n$ nodes [Bo85]. If, instead, it has at least $n/2$ edges, there is a unique big component, the *giant component*, that has linear size with probability one, and the remaining nodes are grouped into components of no more than $\log n$ nodes each [Bo85]. If the graph has $n \log n/2$ edges, or $p = \log n/n$, the graph is connected with constant probability, while for larger values of p, the graph is always connected. See Chapter 2 in this thesis or Chapter V of Reference [Bo85] for more details.

In random graphs that are expected to be connected, all searching algorithms reach all nodes. Otherwise, the search examines only a portion of the graph. When the graph is very sparse, most nodes belong to small isolated trees, and the search stops shortly. The interesting case is when the giant component is formed containing a constant fraction the total number of nodes in the graph. The search is performed on the giant component of the graph with probability at least a constant, since at least a constant portion of the nodes belong to that component, and the starting node can be any node of the graph, with the same probability. In this case, we are interested in the *accessible subgraph* only, the part of the graph that can be reached from the starting node by following edges, even if unvisited nodes remain after the search.

1.1 Section Outline

In Chapter 1 we define our notation, we describe graph-searching and the motivation for average-case analysis, and we conclude with some basic lemmas.

In Chapter 2 we discuss random graphs, their representation, their properties and related algorithms. We consider the relation between directed and undirected random graphs and their properties and we mention lemmas for expected values averaging over all random graphs.

In Chapter 3 we analyze node-based searching algorithms and calculate their expected running time, size of the data structure and depth.

In Chapter 4 we continue with the analysis of other common searching algorithms that do not fall into the node-based searching model, and in particular with depth-first search.

In Chapter 5 we study walk-first search, a depth-first search variant, which is much different from the algorithms examined previously.

In Chapter 6 we concentrate on common properties of all searching algorithms, and we compare these algorithms.

In Chapter 7 we examine special cases of searching, like searching in graphs that need a lot of time to be searched, searching in multigraphs and forests, or terminating the search earlier.

In Chapter 8 we present our principles and techniques for deriving our simulation results, as well as algorithm modifications and different kinds of simulation results.

We conclude with a summary of the results, applications, open problems and list of references.

The appendix contains more simulation results.

1.2 Notation

It is customary for $f(n) = O(g(n))$ to mean that there exists a real constant c and a positive integer n_o that is in general dependent on c such that for all positive integers $n \geq n_o$, $f(n)$ never exceeds $cg(n)$.

We use the notation $f(n) > \Theta(g(n))$ to mean that for all real constants c there exists a positive integer n_o that is in general dependent on c such that for all positive integers $n \geq n_o$, $f(n)$ is always greater than $cg(n)$. In other words, $f(n)$ is asymptotically greater than any function that is $O(g(n))$.

The notations we are going to use, in summary, are:

$$f(n) = O(g(n)) \Leftrightarrow \exists c, n_o(c) \text{ such that } \forall n \geq n_o \quad f(n) \leq cg(n).$$

$$f(n) = \Omega(g(n)) \Leftrightarrow \exists c, n_o(c) \text{ such that } \forall n \geq n_o \quad f(n) \geq cg(n).$$

$$f(n) = \Theta(g(n)) \Leftrightarrow \exists c_1, c_2, n_o(c_1, c_2) \text{ such that } \forall n \geq n_o \quad c_1 g(n) \leq f(n) \leq c_2 g(n).$$

$$f(n) > \Theta(g(n)) \Leftrightarrow \forall c \; \exists n_o(c) \text{ such that } \forall n \geq n_o \quad f(n) > cg(n).$$

The notations $f(n) > \Theta(g(n))$ and $f(n) = \Omega(g(n))$ are not equivalent, as the latter includes all functions $f(n) = \Theta(g(n))$, which are the functions that grow linearly with $g(n)$.

We are interested in asymptotic behavior for large graphs, so all limits and asymptotic values mentioned in the following chapters, unless explicitly defined otherwise, imply that the convergence occurs as $n \to \infty$.

We shall say that a property exists with *probability one* if the property exists with probability asymptotically 1, i.e., $1 - o(1)$.

1.3 Graph-searching

A *graph* is a mathematical model that can represent binary relations between simple objects. The objects are the *nodes* of the graph, and the relations are the *edges*.

If the relations are symmetric, the graph is *undirected*, otherwise it is *directed*. An undirected graph can be regarded as a special case of an directed graph by substituting each undirected edge by two directed edges with opposite directions. When independent probabilities are associated with all edges, a random undirected graph cannot be obtained in the same fashion from a random directed graph. In most cases, however, it is easier to assume first a directed graph and to extend later for an undirected graph.

Graphs have *static* (*structural*) properties that depend only on the existence of edges, and *dynamic* (*algorithmic*) properties that depend not only on the presence of various edges, but on the order in which this presence is revealed, as well. Thus, the representation of the graph must be taken into consideration when studying the algorithms.

A possible representation of a graph, the *adjacency matrix*, uses a two-dimensional array to denote the presence or absence of an edge between any two nodes of the graph. Using this representation, the existence of a particular edge can be checked in constant time, but going through all edges incident to a node requires time proportional to the number of nodes. As a result, graph traversal always requires quadratic (on the number of nodes) time, even when the total number of edges is significantly smaller. Another disadvantage of this representation is that the order in which we scan the edges of different nodes is always the same, it is the order in which the nodes appear in the adjacency matrix.

In a different representation, the *adjacency list* form, the record of each node is a list of its incident edges. Going through all edges incident to a node requires time proportional to the number of these edges, but checking for the existence of a particular edge may require scanning all edges incident to the node. Adjacency lists are the most compact representation for sparse graphs and permit linear (on the number of edges) traversal of the graph.

The order in which we scan the edges on one adjacency list is independent from the order on another adjacency list. Rearrangement of the edges on the adjacency lists produces a different *dynamic representation* of the same static graph: although all structural properties of the graph are preserved, properties of algorithms like graph traversal may not be identical, because the order in which edges are processed is different.

Depending on the way the adjacency lists are created, the order of edges on them may not be independent on different lists. For example, the edges may be sorted by their target node. In undirected graphs, all edges may be considered in order, and they may appear on the adjacency lists of their nodes in that order; as each edge appears in two adjacency lists, not all possible rearrangement may be produced. The order in which we scan the edges on one adjacency list can be independent from the order on another adjacency list; when they are not, we can shuffle the edges on all adjacency lists in order to consider the lists in random order.

Graph-searching on a graph G is the process of visiting all of the nodes of G starting from a given node of G and moving along its edges.

Graphs and graph-searching have a long history. For example, Reference [Ma] describes mazes treated as modern graphs in ancient years, and algorithms for their traversal based on keeping information at the intersection points (nodes) about the paths (edges) used so far to decide which path to follow. Many books explore the various graph algorithms [Ev] [GM] [Ha] [Bog] and they have many applications [CGM]. A special case of graphs, *trees*, where there exists at most one path joining any two nodes, have many applications too [AS] [OSW] [Ya78], and have been studied a lot [Br] [KP] [EP] [Ke] [MM] [Pro].

Problems similar to graph-searching have been studied too, like the s-t connectivity, that decides if a node t is accessible from a node s [BKRU], random walks [GJ], traversal sequences [AKLLR], or parallel searching [KUW].

In undirected graphs, when we count the number of edges used, we count the edges we use in both directions twice. The total number of edges counted may be up to twice the number of existing edges, but the same expression will be appropriate for both directed and undirected graphs in most cases.

The particular searching algorithm used by an application may depend on the application itself; if, for example, the application needs to reach these nodes in a restricted order, then only some of the searching methods may be used. Otherwise, a choice based on the differences in the behavior of these algorithms must be made.

When a node is first encountered, its incident edges are examined only once. An edge is incident on no more than two nodes, so the running time of the search need never exceed twice the number of edges.

During the search, we keep a flag for each node, that indicates whether or not the node has been visited. Once a node is marked as visited, it remains marked until the end of the search. The actions of the searching algorithm are usually different for edges that lead to already visited nodes than for edges that lead to unvisited nodes.

We also maintain a data structure that records the visited nodes whose edges are not yet all used, and data to indicate the edges yet to be scanned. The search terminates when all nodes are visited, when the data structure is empty (we cannot reach any more nodes), or when some search objective is met (the desired node has been found, for example).

The size of the data structure never exceeds the number of nodes in the graph, since there is at most one entry per node. The contents of the data structure, how elements are inserted and deleted, and the processing of these elements are choices that define various searching algorithms. We consider a number of algorithms that represent specific instances of this general paradigm.

The generic searching algorithm can be described by Program 1.1.

The rearrangement step can also be incorporated in the node selection step: instead of rearranging, we select nodes with a different algorithm so that all nodes are selected in the same order and so that exactly the same edges are scanned in the same order as in the original version. However, the complexity of the resulting algorithm may be different, and in all cases we consider, both the selection and the rearranging steps are simple enough to complete in constant time.

```
procedure search($s$ : node)
begin
    data structure:= empty
    insert $s$ into the data structure
    $visited[s]$ := true
    while the data structure is not empty
    begin
        select an element $v$ from the data structure
        if there exists $w$ such that there is an edge from $v$ to $w$ then
            if not $visited[w]$ then
            begin
                insert $w$ to the data structure
                $visited[w]$ := true
                if all nodes are visited then return
            end
            else possibly rearrange the elements of the data structure
        else remove $v$ from the data structure
    end
end
```

Program 1.1: The generic graph-searching algorithm.

Although arbitrary graph-searching algorithms can be used, applications tend to use the more efficient and practical algorithms. These algorithms take constant-time actions for selection and rearranging. Many of these algorithms have common properties and we classify them in models of searching algorithms.

One model that encompasses many commonly used graph-searching algorithms is the *node-based search model* where the data structure contains only nodes for which no edges have been used yet. Whenever a node from the data structure is processed, all its edges are scanned and all its unvisited neighbors are appended to the data structure. This model can be obtained from the generic graph-searching algorithm by selecting the same node continuously until all of its incident edges have been scanned. The program that follows the node-based search model, based on the generic searching procedure of Program 1.1, can be found in Chapter 3.

Another model that encompasses many commonly used graph-searching algorithms is the *new-node search model* that includes all algorithms that ignore edges leading to visited nodes. The ignored edges affect only the running time and do not change any local memory or the data structure.

Other searching models can be used. For example, we could define the *scan-k-edges model* where at each node, we scan exactly k edges (if present) before returning the node back to the data structure. Scan-n-edges is thus identical to the node-based search model. The use of models helps us to concentrate on important issues common to some searching algorithms and enables us to study the properties of many searching algorithms at the same time.

When we mention a property of a model, we imply that all algorithms that fall under this model have this property. When we say that all algorithms have a property, we imply that all algorithms that fall under any model have this property.

All searching algorithms not encompassed by the node-based searching model must be able to keep nodes with only some of their edges scanned into the data structure. Thus, we have to store data into the data structure to indicate which of the edges are still not examined for each node, and not just store the nodes themselves.

An algorithm may fall under more than one model. Different algorithms use different criteria to select a node from the data structure. For example, the node-based search model with a first-in, first-out queue as data structure is breadth-first search, while with a stack as data structure describes the stack-based search. Breadth-first search, stack-based search, and depth-first search can all be described by the new-node search model.

The number of edges scanned during graph-searching indicates (within a constant factor) the time spent on the searching, since the edge-processing operations dominate the inner loop of the algorithm. Multiple edges are not allowed so the number of edges is $O(n^2)$.

Lemma 1.1. *All algorithms searching the same graph G starting from the same initial node v visit the same number of nodes.*

Proof: The number of nodes searched is the number of nodes reachable from the initial node v, i.e., the nodes that belong to the component that contains v. This number does not depend on the order in which these nodes are visited. Thus, all searching algorithms visit the same nodes in any graph, possibly in a different order. ∎

Since all searching algorithms visit exactly the same nodes, the expected number of visited nodes is the same for all algorithms. In contrast, the time (edges traversed) necessary to visit the nodes and the order in which the nodes are visited may be different.

Lemma 1.2. *All algorithms searching the same disconnected graph G starting from the same initial node v examine the same number of edges.*

Proof: Any searching algorithm scans all edges that are accessible from the graph component that the search started, i.e., all edges incident on any of the nodes on that component, because the searching algorithm always tries to find a path to the rest of the nodes, without knowing that some nodes will never be reached. ∎

If the graph is connected, the search may stop earlier when it reaches the last unvisited node. In dense graphs, the search can be much cheaper than the worst case because many edges may never be examined.

Lemma 1.3. *When we search a connected graph G with n nodes and m edges, the number of edges used and the running time of the searching algorithm are $\Omega(n)$ and $O(m)$ respectively.*

Proof: Since the graph is connected, we must reach all n nodes and scan at least $n - 1 = \Omega(n)$ edges. On the other hand, we never scan an edge twice, so we never scan more than $O(m)$ edges. ∎

We use the notation $T(G)$ to denote the number of edges used for searching a graph G. The searching method in use will be mentioned explicitly.

1.4 Average-case analysis

It is common to measure algorithm quality by its worst-case performance. As a result, for most algorithms, only the worst-case behavior has been explored. If we use an algorithm many times, we are interested in the total time of all executions, and therefore, in its average-case behavior. The algorithm with the best worst-case behavior does not guarantee the best average-case behavior. Algorithms with the best worst-case behavior are not always used in practice. Instead, they are substituted by others that have better average-case behavior. The latter usually run faster, although they can be much slower in particular situations. Classical examples are the simplex [KT] and the quicksort algorithms [Se].

Worst-case analysis is easier to perform, since it deals only with bounding the estimated quantity, and it is easier to work with inequalities than with exact quantities. It usually suffices to find an input sequence that causes the worst-case behavior, and compute the appropriate bounds on the inequalities. Worst-case performance may be needed in real-time systems where an unexpected large delay may be unacceptable. For these systems, we sacrifice algorithm efficiency on all executions to ensure an acceptable guaranteed response time.

The inadequacy of worst-case analysis to provide a practical measure of an algorithm's performance is reflected by a recent shift to using *amortized analysis*, which is actually the worst case over a set of operations. Although a specific operation can be particularly expensive, only few such operations will be executed throughout the algorithm's execution, even in the worst case. The proofs are more complex than the worst-case proofs, because they account for past history, i.e., previous operations and the status of the data structures as they are affected by the operations already performed. Operations that are expensive change the data structures in a way that makes many of the other operations much cheaper, so the total cost is not very high.

A next step towards more representative performance measures is average-case analysis, which is much more difficult. It needs to take into account all possible inputs and their distribution. When the measured quantity cannot be computed directly, it can be bounded, but the approximations should be close to reality. We are interested in both upper and lower bound, and we want them to be close to each other for the average-case analysis. The worst-case bound can be used as a first approximation of the upper bound, but it may not be close enough to the final result.

It was long after the analysis of algorithms was introduced [Kn72] that average case analysis began. Many graph algorithms have been analyzed on the average-case [HL] [GS][Mc][Mot].

Since the average-case analysis depends on the input distribution, it is not the same for all kinds of problems. If we view the parameters of a problem as random variables, their distributions usually do not match any of the known distributions exactly and may not be independent. If our data model is close to a distribution for a specific application, it is worth studying the algorithm behavior under that distribution, since our results will be close to the expected behavior. When we are not using a given input distribution for studying an algorithm's behavior, we must use the random distribution, which is the average over all distributions. The inability to model the input is a major weakness in average-case approach, and the main reason why average-case analysis has not been done for graphs. Even so, average-case analysis under uniform distribution is an instructive way to capture algorithms with results that generally find application. Also, analysis based on the uniform distribution is normally the easiest to perform, and is a good starting point.

Chapter 2

Random graphs

As mentioned in the previous chapter, our goal is to average the performance of an algorithm over all possible inputs, where each input is taken into consideration according to a pre-specified probability distribution. In particular, we are interested in representing a variety of graphs in a more general way. Our model is the random graph, whose definition is drawn from Reference [Bo85].

Definition 2.1. *A random graph $G_p(n)$ with n nodes and probability for each edge p, is a graph where every edge exists in the graph with probability p, independently of the existence of any other edge in the graph. In a random graph $G_p(n)$, q denotes the probability of the absence of some edge, so that $q = 1 - p$.*

Random graphs may be directed or undirected. Directed random graphs, $G_p^D(n)$, can have up to $n(n-1)$ directed edges, all of them independent. Undirected random graphs, $G_p^U(n)$, can have up to $n(n-1)/2$ independent undirected edges, or $n(n-1)$ directed edges; In the latter case, each undirected edge corresponds to two directed edges with opposite direction. When we refer to a random graph $G_p(n)$, the result under discussion applies to either directed or undirected graphs.

An undirected graph with n nodes and m (independent) edges is in $G_p^U(n)$ with probability $p^m q^{\binom{n}{2}-m}$, and a directed graph with n nodes and m edges is in $G_p^D(n)$ with probability $p^m q^{n(n-1)-m}$.

Since we are interested in graph-searching, rearranging the order in which we scan the edges affects some algorithmic properties. Thus, we have to choose a graph representation to study all algorithmic properties. We assume that a graph is kept in *adjacency list* form.

The adjacency list representation has the property that the order of the edges that originate from each node, and thus the scanning order, is independent of the order at any other node. Algorithms applied on sparse graphs run usually more efficiently when the graph is in adjacency list form. We assume the *random adjacency list representation of random graphs*, simply referred to as *random graphs* in subsequent chapters.

Although adjacency lists are irrelevant for static (structural) properties, they are crucial for dynamic (algorithmic) properties. For every random graph $G_p(n)$, we consider all distinct orderings of the edges on the adjacency lists as equally likely. For example, an undirected graph with n nodes, m edges, node degrees d_1, d_2, $\ldots$, d_n, and given adjacency lists will represent the model $G_p^U(n)$ with probability

$$\frac{p^m q^{\binom{n}{2}-m}}{d_1!d_2!\ldots d_n!}.$$

In searching we never need to use an edge twice, keeping our position on the adjacency list is enough to identify the next edges to be used. Edges that have appeared on an adjacency list, do not reappear when we scan subsequent edges on the adjacency list.

If there is at least one edge on the list, it can lead to any node with the same probability. Furthermore, after some edges on an adjacency list have been processed, any remaining edges can lead to any of the other nodes with the same probability.

We are interested in unvisited nodes during searching. If we find an edge to a node already visited, while scanning an adjacency list, the probability of finding an unvisited node from a subsequent edge on the same list increases. We have no indication of edges not on the adjacency list until the adjacency list is exhausted.

On undirected random graphs we find out the existence of an edge the first time we use it. We learn of the existence of the edge on the adjacency list of the other node incident to the edge: For every edge from a node v to a node w in an undirected graph, an edge from w to v exists on w's adjacency list, but we don't know its position on the list. Furthermore, when v's adjacency list is exhausted without finding an edge to w, we know that w cannot have an edge that leads to v.

In any random graph, all unvisited nodes are interchangeable because there are no identifying data associated with any unvisited nodes.

The larger the value of p, the more edges the random graph contains. There is an equivalent model for random graphs with parameters the number of nodes and the number of edges, but most of our results are expressed more easily using the $G_p(n)$ model.

2.1 Known properties and algorithms

Random graphs were introduced and initially studied by Erdös and Rényi [ER59] and [ER60], and have been studied extensively thereafter [EP] [GJ] [Karó][BT]. A collection of interesting and useful results on random graphs can be found in Reference [Bo85]. To better understand random graphs, we summarize some of those results below. In most cases, the properties of random graphs that have been studied in the literature assume an undirected random graph and are independent of the graph representation. Many of them can be generalized easily for directed random graphs.

- The degree of a node has binomial distribution with parameters $n-1$ and p. Furthermore, the distribution of the random variables denoting the degrees of different nodes are almost independent. If, for fixed k, we have $pn^{1+1/k} \to \infty$ and $pn^{1+1/(k+1)} \to 0$, the maximum node degree is asymptotically k with probability one. The minimum node degree is asymptotically at least k with probability one if and only if $p = (\log n + (k-1)\log\log n + \omega(n))/n$ for some $\omega(n) \to \infty$. Also, if $pn/\log n \to \infty$, the maximum node degree is asymptotically $(1+o(1))pn$ with probability one. The degree of the nodes have been studied more than all other properties of random graphs, see Reference [Bo82], Reference [Iv] and Reference [Palk].
- On a random graph with m edges, when $m/n^{(k-2)/(k-1)} \to \infty$ and $m/n^{(k-1)/k} \to 0$ the largest component of the graph has asymptotically size k with probability one. If $m < n/2$, all nodes belong to small components, none of which contains more than $\log n$ nodes with probability one.

- When $m > n/2$, there is a unique largest component of the graph, the *giant component*, that has about linear size, ϵn, where $\epsilon > 0$ depends only on p, with probability one. The remaining nodes are divided into components that do not contain more than $\log n$ nodes.
- When $p = (\log n + c + o(1))/n$ (or $m = (\log n + c + o(1))n/2$) the graph is connected with a constant probability $e^{-e^{-c}}$. See also Reference [Tu].
- If p is large enough to ensure that a random graph has minimum degree at least 1 with probability one, then the graph is connected and, if n is even, the graph has a perfect matching with probability also one.
- For some constant c, almost every graph with n nodes and at least $m = cn \log n$ edges is Hamiltonian. Even when $p = c/n$ for some constant $c > 1$, the graph contains a path of length $\Theta(n)$. See also Reference [Po].
- Let d be a natural number, and let $p^d n^{d-1} = \log(n^2/c)$ where $pn/(\log n)^3 \to \infty$; then the probability that the diameter of the graph is d is asymptotically $e^{-c/2}$ and the probability that the diameter of the graph is $d+1$ is asymptotically $1 - e^{-c/2}$. See also Reference [Sp].

Randomized algorithms, algorithms that make some random decisions, have been developed for many graph problems, too [Ya87][Bo86]. Their input, behaves much more randomly, and avoids all pathologically bad cases with probability one. Examples of such algorithms follow:

- The algorithm of Angluin and Valiant that finds Hamilton cycles and matchings and will fail only with probability $O(n^{-2})$ [AV].
- The algorithms of Dyer and Frieze that address the NP-complete problems of graph coloring, minimum cut, and graph partitioning [DF].
- The algorithm of Broder, Karlin, Raghavan and Upfal that detects s-t connectivity in limited space. [BKRU].
- The algorithm of Cherigan and Hagerup that finds the maximum flow in $O(nm)$ average time [CH].

Some algorithms use properties of random graphs and produce results that are valid for almost all graphs, and these algorithms run much faster than their deterministic counterparts [Ti]. Examples of such algorithms are:

- Polynomial-time algorithms developed by Karp with expected near-optimal solution on random graphs for the classical NP-complete problems Euclidian traveling-salesman problem, hamilton circuits, graph coloring, and independent sets of vertices [Karp].
- The algorithm of Babai, Erdös, and Selkow that tests if two graphs of size n are isomorphic in time $O(n^2)$, a problem known to be NP-complete. The algorithm is based on the fact that the degree sequence of almost every graph contains many gaps [BES].
- The algorithm of Karp and Sipser that finds matchings in sparse graphs very close to the maximum ones in linear time [KS].
- The algorithm of Bollobás, Fenner and Frieze that finds Hamilton cycles in an undirected random graph [BFF].

  Many other properties of random graphs have been studied in the literature [Fe] [SS].
- The algorithm of Moffat and Takaoka that finds the all pairs shortest path in expected running time $O(n^2 \log n)$ [MT].

2.2 Searching random graphs

To study the properties of searching algorithms in random graphs, we need the technical details on the structure of random graphs that have been developed in the classical literature.

Theorem 2.1(Bollobás). *Let $c > 1$ be a constant, $m = \lfloor cn/2 \rfloor$ and any function $\omega(n) \to \infty$. Then almost every random graph G with m edges is the union of the giant component, the small unicyclic components and the small tree components. There are at most $\omega(n)$ vertices on the unicyclic components. Let $L_i(G)$ represent the size of the ith component of G, sorted by size, where $L_i(G)$ is 0 if less than i components exist. The size of the giant component satisfies*

$$|L_1(G) - (1 - t(c))n| \leq \omega(n)n^{1/2},$$

where

$$t(c) = \frac{1}{c} \sum_{k \geq 1} \frac{k^{k-1}}{k!} (ce^{-c})^k, \tag{2.1}$$

and for every fixed $i \geq 2$

$$|L_i(G) - \frac{1}{a}(\log n - \frac{5}{2} \log \log n)| \leq \omega(n),$$

where $a = c - 1 - \log c$.

Proof: See Theorem 6.11 in Reference [Bo85]. ■

Figure 2.1 depicts the function $t(c)$, as given by Equation 2.1, for different values of $c = pn$ (on the x-axis). When c is close to 1, the giant component contains a very small portion of the nodes; this portion increases rapidly and soon nearly all nodes belong to the giant component.

Figure 2.2 depicts the number of accessible nodes on a graph starting from a random node. These numbers were found running depth-first search on directed graphs of $n = 2,000,000$ nodes. The x-axis represents constant values for the product $r = pn$. This curve is complementary to $t(c)$.

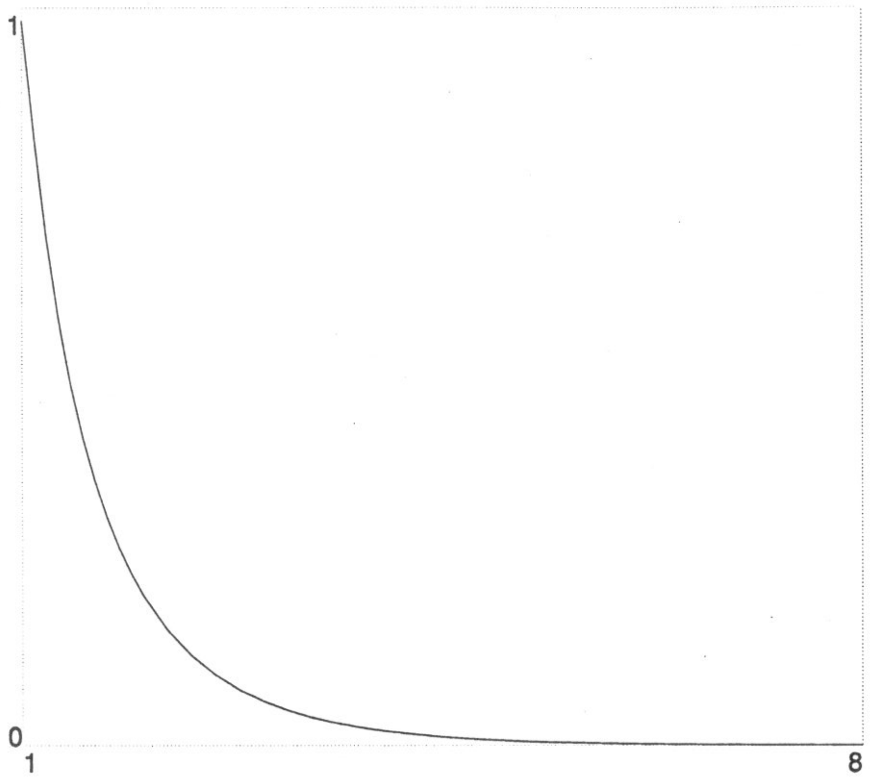

Figure 2.1: The function $t(c)$.

If a random graph is connected, $p = \Omega(\log n/n)$ whereas all disconnected graphs have $p = O(\log n/n)$.

When $p = 0$, the graph consists of isolated nodes and no edges. So we can always assume that $p > 0$, or that $q < 1$. Similarly, there is only one graph with $p = 1$, the complete graph; there is interest in this graph, which is examined separately in many cases, because we have different representations of this graph depending on the order of the edges on the adjacency lists.

For any quantity $f(G)$ associated with a graph G, we use the notation $E(f(G))$ to denote the average value of the quantity $f(G)$ over all random graphs G. An equivalent notation is $f(G_p(n))$.

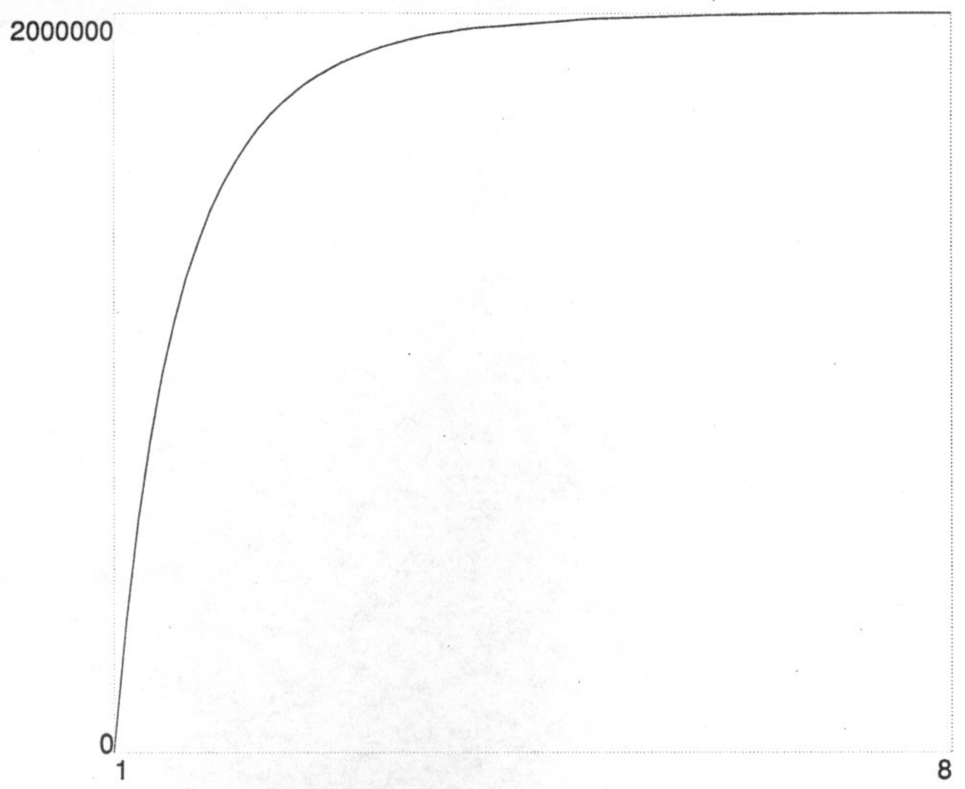

Figure 2.2: The accessible nodes on a graph.

Lemma 2.1. *When a random graph $G_p(n)$ is searched, with $p > 1/(n-1)$, the expected number of nodes visited is $\Theta(n)$.*

Proof: According to Lemma 1.1, this number is independent of the searching algorithm used. In connected graphs, all nodes are visited. Otherwise, the giant component is expected to contain at least a linear number of nodes, say αn for a positive constant $\alpha < 1$. In that case, the probability that the starting node belongs to the giant component is at least α, so the expected number of nodes visited is at least $\alpha^2 n$, which is $\Omega(n)$. Since this number cannot exceed n, it is $\Theta(n)$. ■

Theorem 2.1(Bollobás) helps to estimate the coefficient for the number of nodes visited, bounding the size of the giant component $L_1(G_p(n))$ when $p > cn$:

$$\begin{aligned}
& |L_1(G_p(n)) - (1-t(c))n| \le \omega(n)n^{1/2} \\
\Rightarrow\quad & (1-t(c))n - \omega(n)n^{1/2} \le L_1(G_p(n)) \le (1-t(c))n + \omega(n)n^{1/2} \\
\Rightarrow\quad & (1-t(c))^2n^2 + \omega(n)^2n - 2(1-t(c))n^{3/2}\omega(n) \le L_1(G_p(n))^2 \\
& \le (1-t(c))^2n^2 + \omega(n)^2n + 2(1-t(c))n^{3/2}\omega(n) \\
\Rightarrow\quad & \alpha^2 = (1-t(c))^2 + O(\omega(n)n^{-1/2}).
\end{aligned}$$

So, the coefficient for the number of nodes visited is at least $1 - 2t(c)$, where $t(c)$ is given by Equation 2.1. This value is close to one and since no more than n nodes exist, it cannot exceed one.

We use the size of the graph components frequently. In most cases, the giant component specifies the extent of the search. We know the expected size of that component, for undirected random graphs, and we want to find the same quantity for directed random graphs.

In directed graphs, we do not have the equivalent of a component of an undirected graph because connectivity is not symmetric. For the purpose of searching, however, we can find an equivalence: in the undirected graph, the size of the component that contains the starting node is the number of nodes accessible from the starting node, which is the only interesting number.

We prove that this number is expected to be the same for directed graphs and undirected graphs. The method employed in the proof can be used for extending other properties of undirected random graphs to directed random graphs. For example, the degrees of the nodes of an undirected random graph have the same distribution as the in-degrees and the out-degrees of a directed random graph.

Figure 2.3 depicts the fraction of nodes visited using depth-first search on disconnected directed graphs, i.e., the number of nodes visited divided by n. The dependence of the number of visited nodes on both n (on the x-axis) and the product $r = pn$ (on the y-axis) can be observed; the shape of the curve seems totally independent of n. The picture for undirected graphs is identical.

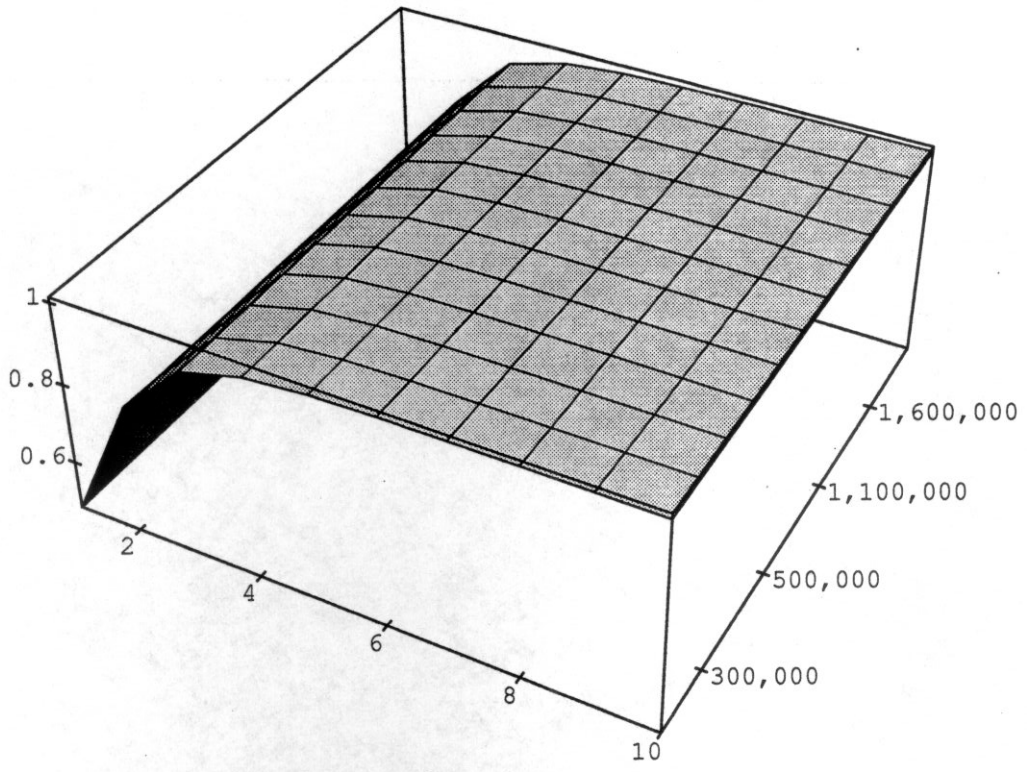

Figure 2.3: The number of accessible nodes on graphs.

The fast growth of the number of nodes visited observed for small values of p is the result of the evolution of the giant component. Compare this figure with Figure 2.2.

Although undirected random graphs have been studied extensively, only few results have been published for directed random graphs [Ms] [JS] [AF]. Fortunately, most undirected graph properties can be extended to directed random graphs.

Theorem 2.2. *The expected number of accessible nodes in a random graph $G_p(n)$ is the same for directed and undirected graphs.*

Proof: The expected number of accessible nodes in a random graph $G_p(n)$ is independent of the model representing the graph, and thus of the order of the edges on the adjacency lists. Thus, we concentrate on the different graphs and ignore the adjacency lists and other representation details.

The expected number of accessible nodes on a random graph $G_p(n)$ is the average value of the number of accessible nodes over all graphs G with n nodes, using as weight the probability $P(G)$ that the graph G represents the model $G_p(n)$, and starting the search from any node v, with the same probability; this holds for both directed and undirected graphs, separately, using different probability functions for the undirected and directed cases, $P^U(G^U)$ and $P^D(G^D)$ respectively.

For each undirected graph G^U we search, and starting the search from a node v, we number the nodes from 1 to n in the order we visit them. Although different searching algorithms and graph representations may yield different numberings of the nodes, any of these numberings can be used, as long as we use the same searching algorithm and graph representation for all searches mentioned in the current proof. Unreachable nodes get the highest numbers. If the graph G^U has m edges, $P^U(G^U) = p^m q^{\binom{n}{2}-m}$.

Using undirected graphs with numbered nodes, we generate all directed graphs: for each graph G^U, we assign direction to its edges from the lower numbered node to the higher numbered one and we generate all possible edge configurations from higher numbered nodes to lower numbered nodes. We create k new edges with probability $p^k q^{\binom{n}{2}-k}$ because each of these edges is independent of all others, and exists with probability p, so that $P^D(G^D) = P^U(G^U)p^k q^{\binom{n}{2}-k} = p^{m+k} q^{n(n-1)-(m+k)}$.

First, we show that all possible directed graphs are created this way, each with probability the same as the probability that it represents the directed random graph model. Each of the derived graphs has any edge with probability p, and independent of any other edge; this is true for the edges from lower numbered nodes to higher ones since they were edges of an undirected random graph, and is true for the rest of the edges, because of their selection. Furthermore, there is a unique way to create a directed random graph with this method: we number the nodes in the order we visit them, we delete the edges from higher numbered nodes to lower ones and we remove the direction from the remaining edges to construct the *underlying undirected graph*. We can create our directed graph back starting from this undirected graph. The underlying undirected graph of a directed graph depends only on the directed graph and the numbering of the nodes; the numbering depends only on the searching algorithm and the graph representation.

We observe that all directed graphs generated from an undirected graph have the same set of accessible nodes with the underlying undirected graph G^U, if the search starts from node v (which is node 1). Let $j = j(G^U)$ be the last node accessible from 1 in the undirected graph. This means that exactly $j(G^U)$ nodes are accessible from node 1, (the nodes from 1 to $j(G^U)$) and all other nodes belong to other, isolated components. According to the numbering of the nodes and the direction on the edges, the same $j(G^U)$ nodes are still accessible from node 1 in the directed graph. We assign the next available number the first time we visit a node, so the edge used to visit a node for the first time leads from a lower numbered node to a higher one; all such edges can still be used in the directed graph. Furthermore, no additional nodes are accessible from node 1, because no new edges have been created that lead from any node accessible from 1 (nodes 1 to $j(G^U)$) to any other node (nodes $j(G^U)+1$ to n). The k edges that are added and lead from higher to lower numbered nodes do not affect the number of accessible nodes $j(G^D) = j(G^U)$.

Finally, the expected number of accessible nodes on a directed random graph is

$$\begin{aligned}
&\sum_{0\le m\le \binom{n}{2}} \sum_{0\le k\le \binom{n}{2}} j(G^D)P^D(G^D) \\
&= \sum_{0\le m\le \binom{n}{2}} \sum_{0\le k\le \binom{n}{2}} j(G^U)P^U(G^U)p^k q^{\binom{n}{2}-k} \\
&= \sum_{0\le m\le \binom{n}{2}} j(G^U)P^U(G^U) \sum_{0\le k\le \binom{n}{2}} p^k q^{\binom{n}{2}-k} \\
&= \sum_{0\le m\le \binom{n}{2}} j(G^U)P^U(G^U)
\end{aligned}$$

which is the expected number of accessible nodes on an undirected random graph. ∎

Note that the proof does not depend on the exact nature of the quantity $j(G^U)$, so the proof can be applied to any quantity that is the same on a directed graph and its underlying undirected counterpart.

When a random subset contains a fraction ϵ of the nodes, the edges between these nodes are a fraction ϵ^2 of the edges; each edge connecting any two points is equally likely. The giant component is a subset of the nodes that has correlations with other properties of the nodes: the higher the degree of a node, the higher the probability the node belongs to the giant component. Hence, the giant component contains much more edges than a typical random subset of the nodes of the same size. Since there are no edges between the nodes in the giant component and those out of the giant component, only $\epsilon^2 + (1-\epsilon)^2$ possible edges exist in the graph. Therefore, an estimation of the fraction of the edges in the giant component is: $\epsilon^2/(\epsilon^2 + (1-\epsilon)^2)$, which is at least a constant. A different estimation is used later.

Lemma 2.2. *When a disconnected random graph $G_p(n)$, with $p > 1/(n-1)$, is searched, the expected number of edges examined is $\Theta(pn^2)$.*

Proof: According to Lemma 1.2, this number is independent of the searching algorithm used. In a disconnected graph, the search scans all edges in the component where the search starts. The expected number of edges in the graph is $np(n-1)$ for a directed graph and $np(n-1)/2$ for an undirected graph, which is $\Theta(pn^2)$ in both cases. Since the graph contains the giant component with a linear number of nodes, say at least αn, we start searching in the giant component with probability at least α and at least a constant portion of the total number of edges in the graph are in this component, say β (and also $\beta \geq \alpha^2$, see the discussion above). Thus, the number of edges used during the search is expected to exceed $\alpha\beta pn^2$, which is $\Omega(pn^2)$. The total number of edges is $\Theta(pn^2)$, so the number of scanned edges is $\Theta(pn^2)$. ■

The size of the giant component $L_1(G_p(n))$ in an undirected random graph can be bounded using Theorem 2.1(Bollobás), thus we can use $\alpha = L_1(G_p(n))/n$. The expected number of edges in the graph is $pn(n-1)/2$, and we need to know how many of these edges are in the giant component. Since the other components are trees and unicyclic components, they cannot have more edges than they have nodes. This means that the expected number of edges used over pn^2 is at least

$$\frac{L_1(G_p(n))}{n}\frac{p\frac{n(n-1)}{2}-n+L_1(G_p(n))}{pn^2}=\frac{L_1(G_p(n))}{n}\left(\frac{1}{2}-\frac{1}{2n}-\frac{1-\frac{L_1(G_p(n))}{n}}{pn}\right)$$

which converges to $(1-t(c))/2$.

In an undirected random graph each edge is used twice, so the coefficient in the above lemma is expected to be at least $1-t(c)$, and cannot exceed one.

In a directed random graph, we first examine the edges in the underlying undirected graph: we number the nodes in the order we visit them, we ignore the edges from higher to lower numbered nodes and we remove direction from the remaining edges. In the resulting undirected graph, we can access the same nodes as in the directed graph, and the graph has about $pn^2(1-t(c))/2$ edges, as estimated before. We want to know the number of edges in the directed graph with direction from higher numbered nodes to lower ones. These edges constitute a random graph, independent from the previous undirected random graph; we have no information about any of its edges since we never used any of them. The giant component of the previous undirected graph is a random subset of the nodes in this graph, and is expected to contain $p\,(1-t(c))\,n\,((1-t(c))n-1)\,/2$ edges, or more than $pn^2\,(1-2t(c))\,/2$. This means that the coefficient in the above lemma is expected to be at least $(1-t(c))\,/2+(1-2t(c))\,/2=1-3/2t(c)$, and cannot be more than one.

Since the graph is disconnected, i.e., $p=O(\log n/n)$, the number of edges is bounded by $O(n\log n)$. The only edges that are not scanned, are the edges belonging to isolated components, which are usually a very small fraction of the total number of edges, since the search usually starts on the giant component of the graph.

Figure 2.4 depicts the number of edges used by depth-first search on disconnected directed graphs. The dependence of the number of visited nodes on both n (on the x-axis) and the product $r=pn$ (on the y-axis) can be observed; the shape of the curve seems totally independent of n. The number of edges displayed has been divided by n, so that for all graphs the value drawn does not exceed r.

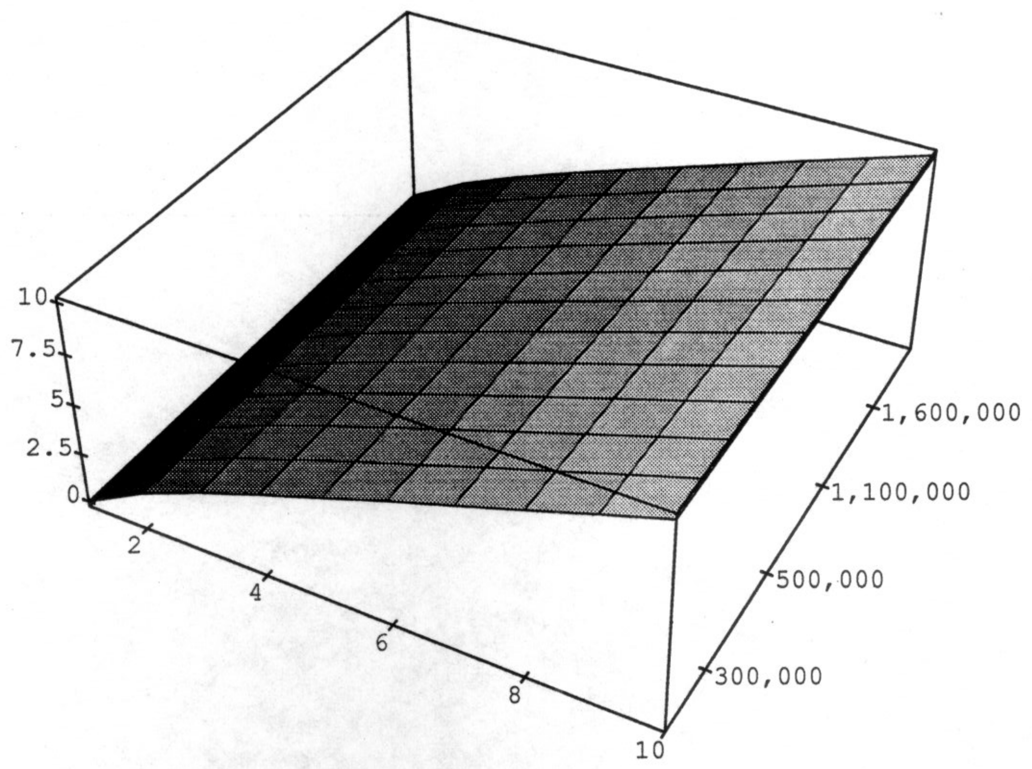

Figure 2.4: The number of edges used on a disconnected graph.

At the beginning, while the size of the giant component is still growing, the number of edges examined grows faster than the number of edges in the graph; afterwards, it grows linearly, examining almost all edges. Undirected graphs are expected to give the same picture.

The next lemma is important and is used in subsequent proofs. It is consequence of the fact that the degree of a node has binomial distribution with parameters $n-1$ and p, as mentioned before (see also Chapter III of Reference [Bo85]).

Lemma 2.3. *In a random graph $G_p(n)$, the nodes of the graph have expected degree $p(n-1)$. If $p > (1+\epsilon)/(n-1)$ for any constant $\epsilon > 0$, the degrees of almost all nodes (except for at most a constant number of them) fall within distance $\lambda p(n-1)$ (i.e., λ times their expected degree) from the expected value with probability one, for some constant $0 < \lambda < 1$. Furthermore, if $p > \Theta(1/n)$ the degrees of almost all nodes fall within distance*

$\lambda p(n-1)$ *(i.e., λ times their expected degree) from the expected value with probability one, for any constant* $0 < \lambda < 1$.

Proof: Since the degree of a node is a random variable which is following binomial distribution with parameters $n-1$ and p, the mean is $\mu = p(n-1) \geq 1+\epsilon$ and the variance is $\sigma^2 = qp(n-1) = q\mu$. Then the probability that the value of the random variable X differs from its mean value by at least k is $P(|X-\mu| \geq k) \leq \sigma^2/k^2$, using Chebyshev's inequality.

If $p > (1+\epsilon)/(n-1)$, we use k to be $k = \lambda\mu$ for any constant $0 < \mu^{-1/4} < \lambda < 1$ so that $P(|X-\mu| \geq \lambda\mu) \leq \sigma^2/(\lambda\mu)^2 = q/(\lambda^2\mu) = c$ for some constant $c < (1+\epsilon)^{-1/2} < 1$, as $\lambda^2\mu > \mu^{1/2} > (1+\epsilon)^{1/2} > 1$. The probability that $f(n)$ nodes have degree different from their mean by at least $\lambda\mu$ is less than $c^{f(n)}$, which for any increasing function $f(n)$ will be zero.

If $p > \Theta(1/n)$, we use k to be $k = \lambda\mu$ for any constant $0 < \lambda < 1$ so that

$$P(|X-\mu| \geq \lambda\mu) \leq \frac{\sigma^2}{(\lambda\mu)^2} = \frac{q}{(\lambda^2\mu)} \to 0$$

because $\mu > \Theta(1)$. ■

The nodes that deviate from the majority in Lemma 2.3 do not affect the results of the theorems that use the lemma by more than a constant term, and are ignored in the proofs. Also, on all connected random graphs we have $p > \Theta(1/n)$, and they fall into the latter category.

2.3 Undirected v.s. directed random graphs

An undirected graph is a special case of a directed graph, since any undirected edge is equivalent to two directed edges with opposite directions that connect the same nodes. Whenever we have an undirected graph, we can always convert it to directed, and process it with any procedures we may have for directed graphs. Furthermore, directed and undirected random graphs seem to have common properties. According to Theorem 2.2, directed and undirected random graphs have the same number of accessible nodes. We further explore the relationship of the two kinds of graphs, to reveal other common properties.

A random undirected graph does not produce a random directed graph, since only half of the resulting edges are independent. In this case, we do not necessarily expect the same behavior of our algorithm as on random graph input. The running time of our algorithm may be different, since we expect to scan a different set of edges, but other parameters of the scanning process, like the depth we reach and the size of the data structure, may not be affected in some searching algorithms.

Searching an undirected random graph $G_p^U(n)$ is very similar to searching a directed random graph $G_p^D(n)$, since we expect to find each edge with probability p, except that some edges will certainly be present or absent. These are the edges that definitely exist, if we have encountered the reverse edge, or definitely do not exist, if we have scanned the entire adjacency list of the target node without finding the reverse edge. In all searching algorithms we only scan edges originating from visited nodes; the edges of known presence or absence all lead to visited nodes, and all edges to unvisited nodes exist with probability p.

Lemma 2.4. *Random directed graphs $G_p^D(n)$ and random undirected graphs $G_p^U(n)$ are equivalent for new-node searching algorithms, except with respect to the number of edges used and the running time.*

Proof: All edges of known existence lead to visited nodes, so they are ignored by the new-node searching algorithms and do not affect the search in any way, except in the number of edges examined and, subsequently, in the running time.

In other words, if we visit a node v in an undirected graph and from v we follow an edge to node w, whenever we reach w, we do not care if we encountered the edge from w to v, since v is visited; if we encountered this edge, we ignore it. Similarly, in a directed graph, we are only interested in edges to unvisited nodes, and we ignore all other edges. Only the running time is affected, since the existence of an edge to a visited node increases the time we need to scan the adjacency list of w. ∎

We use the notation $T(G)$ to denote the number of edges used for searching a graph G, we can express the next lemma:

Lemma 2.5. *For new-node searching algorithms*

$$T(G_p^U(n)) \leq T(G_p^D(n)) + n - 1.$$

Proof: We prove that for every individual directed graph G^D and its underlying undirected graph G^U we expect $T(G^U) \leq T(G^D) + n - 1$. The inequality for the average values follows directly.

In the undirected graph, we number the nodes in the order we visit them, from 1 to n. Only half of the directed edges are independent: When we visit a node i we find out about the existence of edges to nodes j, where $i < j \leq n$; the existence of edges to all other nodes has been discovered before. The cost of searching the edges to all nodes j, where $i < j \leq n$, is the same as in the directed graph.

The edges to nodes j, where $1 \leq j < i$, are not independent of the search so far. One of these edges, the edge we followed to visit node i, exists with probability 1. Although the destination of the other edges depends on the search of previous nodes, we are only interested in their cardinality, because any new-node searching algorithm ignores them, and this number is still expected to be the same as in the directed graph, because each possible edge still exists with probability p.

The $n - 1$ edges of known existence, which may be scanned in the opposite direction, constitute the potential extra cost. ∎

The $n-1$ known edges may also exist in the directed graph with probability p, so the extra cost is actually expected to be at most $q(n-1)$.

Lemma 2.6. *For any searching algorithm* $T(G_p^U(n)) \leq 2T(G_p^D(n))$.

Proof: For new-node searching algorithms, Lemma 2.4 gives a stronger result. For algorithms that do not fall into the new-node searching model, edges that lead to visited nodes can be followed. In an undirected graph we know that for each edge found, the reverse edge also exists.

If the reverse edges are used, they at most duplicate the number of edges used, but they do not affect our calculations. When the reverse edges leaving a node are used, the probability of finding an unvisited node does not drop: The next edge can always lead to a certain unvisited node with probability p; if we follow the reverse edge, at least one edge to a visited node may no longer be used from that node.

In other words, the probability of the existence of an edge from the current node to an unvisited node is always p, while not fewer edges to visited nodes can be used next. ■

In the case of a disconnected graph, all edges in the component we are searching will be used anyway. Otherwise, some of the edges of known existence may not be used, and the total cost may be lower.

Notice that we only give an upper bound for the case of the undirected graph. Under certain circumstances the searching of undirected graphs can be cheaper: The only edges that lead to a node that has been deleted from the data structure are the edges we have encountered in their other direction; the probability of finding a new unvisited node on adjacency lists that do not contain these edges increases.

Chapter 3

Node-based search

In the node-based search model, whenever a node is removed from the data structure and processed, its adjacency list is scanned entirely and all its unvisited neighbors are appended to the data structure. The name of this class of algorithms comes from the fact that only nodes are kept in the data structure. The searching procedure, displayed in Program 3.1, can be obtained by modifying the generic searching procedure of Program 1.1:

```
procedure node-based search($s$ : node)
begin
    data structure:= empty
    insert $s$ into the data structure
    $visited[s]$ := true
    while the data structure is not empty
    begin
        select and remove an element $v$ from the data structure
        foreach $w$ such that there is an edge from $v$ to $w$
            if not $visited[w]$ then
            begin
                insert $w$ to the data structure
                $visited[w]$ := true
                if all nodes are visited then return
            end
    end
end
```

Program 3.1: The node-based searching algorithm.

Various node-based searching algorithms use different criteria to select a node from the data structure.

Lemma 3.1. *The average number of edges used in the node-based searching of a directed random graph $G_p^D(n)$ is independent of the particular method used.*

Proof:

When we process a node, we search a subgraph with the same (in all methods) expected number of unvisited nodes, the same probability p for the existence of an edge connecting the current node to any particular new node and the same probability p of reaching a particular visited node. Furthermore, the order of the edges on the adjacency list is random, and every permutation has the same probability.

Similarly, every other factor (parameter) that affects the search is independent of the next node visited, so all nodes in the data structure are interchangeable. Thus, we expect to use the same number of edges, visit the same number of previously unvisited nodes, and start a new step or end the search, no matter which node we select. The total number of edges used by all these steps is expected to be the same, as well. ∎

Node-based searching algorithms differ only in the selection of the next node from the data structure. Different searching algorithms may produce different searching trees and may scan a different number or set of edges on a particular graph. However, the average number of scanned edges over all possible graphs is the same.

We cannot use the same argument for undirected graphs. For every node v on the data structure we know that we have an edge to a node w from which we first visited v. Since all nodes have one such edge, the nodes could still be considered interchangeable. However, we also know that any node processed before w cannot have an edge leading to v; if it did, it would have been the first node from which to visit v, a contradiction. The nodes in the data structure have not all been visited from the same node, so the number of edges that cannot appear on the adjacency list is not the same for all nodes. Thus, the nodes are not interchangeable.

Theorem 3.1. *In a connected, but not complete, random graph $G_p(n)$ the edges in $O(\log n/\log(1/q))$ adjacency lists are sufficient for visiting all nodes. Furthermore, the edges in $\Omega(\log n/\log(1/q))$ adjacency lists are necessary for visiting all nodes.*

Proof: By Lemma 2.3, each adjacency list scanned contains between $(1-\lambda)p(n-1)$ and $(1+\lambda)p(n-1)$ edges, for any constant λ. Every time a new adjacency list is fully scanned, a fraction between $(1-\lambda)p$ and $(1+\lambda)p$ of the previously unvisited nodes remain unvisited. After using k adjacency lists, there are between $((1-\lambda)p)^k(n-1)$ and $((1+\lambda)p)^k(n-1)$ unvisited nodes.

The search terminates when no nodes remain unvisited, or when the expected number of unvisited nodes is c, for c a constant less than one (e.g. $1/2$), or even a decreasing function of n (e.g. $1/\log n$). This happens after using k adjacency lists if $(n-1)(1-(1-\lambda)p)^k \leq c$ which implies $k \geq (\log c - \log(n-1))/\log(1-(1-\lambda)p)$ and that $k = \Theta(\log n/\log(1/q))$, since $1-(1-\lambda)p = (1-\lambda)q + \lambda = \Theta(q)$ whenever $(1-(1-\lambda)p) \neq 0$ (and therefore $p \neq 1$, the complete graph) and $(1-(1-\lambda)p) \neq 1$ (and therefore $p \neq 0$, the graph without edges).

The search continues after using k adjacency lists when the expected number of unvisited nodes is c, for c a constant greater than one (e.g. 2), or even an increasing function of n (e.g. $\log n$). Then $(n-1)(1-(1+\lambda)p)^k \geq c$ implies $k \leq (\log c - \log(n-1))/\log(1-(1+\lambda)p)$ and $k = O(\log n/\log(1/q))$, since $1-(1+\lambda)p = (1+\lambda)q - \lambda = \Theta(q)$.

We conclude that less than $O(\log n/\log(1/q))$ adjacency lists are expected to be inadequate to visit all nodes. ■

We can get a coefficient in the Θ notation as close to 1 as we want, by choosing λ as small as we want and c a constant or a small function of n.

In the worst case, when the graph is hardly connected and $p = \Theta(\log n/n)$, we need to scan $\Theta(n)$ adjacency lists. Also, if the graph is disconnected, we scan all adjacency lists.

3.1 Size of the data structure needed by node-based search

Theorem 3.2. *The average size of the data structure needed for the search a random graph $G_p(n)$ by any of the node-based searching algorithms is $\Theta(n)$ when $p > (1+\epsilon)/(n-1)$ for any constant $\epsilon > 0$. In particular, if $p > \Theta(1/n)$, it is asymptotically n.*

Proof: The upper bound on the size of the data structure is always n. We only need an expected lower bound.

We study the search until a constant fraction α of the nodes remains unvisited. We define the value of α that we use for best results later, but the analysis is valid for any value of α. We use the value of λ defined in Lemma 2.3, on the subgraph with the current node and the (at least αn) unvisited nodes, which is a random graph. Every node inserts at least $p(1-\lambda)\alpha n$ nodes into the data structure, on the average. The $(1-\alpha)n$ nodes visited by this process, need at most $\frac{(1-\alpha)n}{p(1-\lambda)\alpha n}$ steps; this number is also the number of nodes which are taken out of the data structure and whose neighbors are inserted into the data structure. After $\frac{(1-\alpha)n}{p(1-\lambda)\alpha n}$ steps the data structure has at least $(1-\alpha)n - \frac{(1-\alpha)n}{p(1-\lambda)\alpha n}$ nodes, which is a lower bound on the size of the data structure q_n:

$$q_n \geq (1-\alpha)n - \frac{(1-\alpha)n}{p(1-\lambda)\alpha n} = (1-\alpha)n(1 - \frac{1}{p(1-\lambda)\alpha n}), \tag{3.1}$$

and $q_n \leq n$. The right hand side of Equation 3.1 has its maximum value when $\alpha = ((1-\lambda)pn)^{-1/2}$, so that $q_n \geq \left(1 - ((1-\lambda)pn)^{-1/2}\right)^2 n$.

From this result we see that if $p = O(1/n)$ (and since we are only considering cases where $p = \Omega(1/n)$, also $p = \Theta(1/n)$), then $q_n = \Omega(n)$ and therefore $q_n = \Theta(n)$. We still try to estimate the coefficient, and how close to 1 it is.

Consider the search until a constant fraction α of the nodes remains unvisited. Instead of stopping when αn nodes remain unvisited, we repeat the same process with these αn nodes, i.e., until $\alpha^2 n$ nodes remain unvisited, and so on. In each of these steps the formula giving the change to the size of the data structure is similar to Equation 3.1 for the first step, since the nodes already visited and the edges to and from these nodes affect neither the nodes inserted into the data structure nor the size of the data structure; however, instead of n, we have to use the number of nodes with which we started at each step.

We must choose the value of α for each step. Although we may get a better bound with different values of α for each step, it is simpler to use the same value of α for all steps. The number of unvisited nodes in step i is $\alpha^i n$, where the numbering of the steps starts at 0. If we stop after j steps, there are $\alpha^j n$ unvisited nodes and the size of the data structure q_n is the sum of the increments incurred by these steps:

$$\begin{aligned} q_n &\geq \sum_{0\leq i<j} (1-\alpha)\alpha^i n(1-\frac{1}{p(1-\lambda)\alpha\alpha^i n}) \\ &= \sum_{0\leq i<j} (1-\alpha)n(\alpha^i - \frac{1}{p(1-\lambda)\alpha n}) \\ &= n(1-\alpha^j - \frac{j(1-\alpha)}{p(1-\lambda)\alpha n}). \end{aligned} \tag{3.2}$$

When $p > \Theta(1/n)$, we can take α to be any constant and j to be any increasing function less than pn. For example, $j = (pn)^{1/2}$. Then, the coefficient of the term n will be one for $n \to \infty$, and we have the tightest bound possible. ∎

When $p = \Theta(1/n)$, the coefficient of n is a constant less than one; we can look for a better coefficient on the lower bound. The problem is still open, and a possible approach follows:

Taking derivatives we see that the quantity in Equation 3.2 has its maximum value when $\alpha = ((1-\lambda)pn)^{-1/(j+1)}$. We can get the same value for α in another, more intuitive way: The expected number of nodes inserted by each node decreases as the process goes on, since it is proportional to the number of unvisited nodes at each moment. When the expected number of nodes to be inserted in a single step becomes one, the size of the data structure has reached its maximum (one node is removed, at most one is inserted). So we stop when $p(1-\lambda)\alpha^{j+1}n = 1$.

In this case Equation 3.2 gives:

$$q_n \geq n(1-(j+1)\alpha^j + j\alpha^{j+1}) = n\left(1-(j+1)\left((1-\lambda)pn\right)^{-j/(j+1)} + j\left((1-\lambda)pn\right)^{-1}\right). \tag{3.3}$$

In principle, one can take derivatives to find the value of j that maximizes the right hand side of Equation 3.3 and obtain the best possible results in this approach, but the solution for j seems too complicated, even for symbolic manipulation packages. Nevertheless, the maximum value of the quantity in Equation 3.3 may not be a rigorous lower bound to the size of the data structure.

3.2 Time needed by node-based search

Theorem 3.3. *The expected number of edges used by any node-based searching procedure on a random graph $G_p(n)$ is: $n-1$ for the complete graph, $\Theta(n\log n)$ for all connected graphs and $\Theta(pn^2)$ in all other cases.*

Proof: For disconnected graphs, Lemma 2.2 applies and gives us the desired result, $\Theta(pn^2)$. In a complete graph, the search reaches all nodes by scanning the $n-1$ edges of the first node.

In connected graphs, we have $\Omega(n\log n)$ edges. We examine a directed random graph first. In the node-based searching algorithms, the number of edges scanned is between $(1-\lambda)p(n-1)$ and $(1+\lambda)p(n-1)$ times the nodes scanned, since, on the average, each node is incident upon between $(1-\lambda)p(n-1)$ and $(1+\lambda)p(n-1)$ other nodes, for any constant λ, (Lemma 2.3).

According to Theorem 3.1, we only need to use $\Theta(\log n/\log(1/q))$ adjacency lists, which means between $\Theta\left((1-\lambda)p(n-1)\log n/\log(1/q)\right)$ and $\Theta\left((1+\lambda)p(n-1)\log n/\log(1/q)\right)$ edges, which is $\Theta(n\log n\, p/\log(1/q))$ edges.

Selecting a very small value for λ (for both this λ and the one in Theorem 3.1) we can get a coefficient for $n\log n\, p/\log(1/q)$ as close to 1 as we want. In order to calculate the coefficient of $n\log n$ we need to estimate $p/\log(1/q)$. For $p=\Theta(1)$ (a constant, different from 0 and 1), $p/\log(1/q)$ is a constant too. If $p\to 0$ as $n\to\infty$, then $\log q=\log(1-p)=-p-p^2/2+O(p^3)$ so that $p/\log(1/q)=-p/\log(1-p)=1/(1+p/2+O(p^2))\to 1$.

Thus, the running time is always $\Theta(n \log n)$, with coefficient $p/\log(1/q)$ for p a constant, and as close to one as we want otherwise.

In an undirected connected graph, we still need to use m adjacency lists where $m = \Theta(\log n/\log(1/q))$. In a directed graph, these adjacency lists contain $\Theta(n \log n)$ edges, $n-1$ of which visit new nodes; the remaining $\Theta(n \log n)$ edges lead to already visited nodes. In an undirected graph, these $\Theta(n \log n)$ edges are not independent. For each adjacency list used (except the first one), we know the reverse of the edge followed to visit the node is always present. Thus, the search scans $\Theta(\log n/\log(1/q))$ more edges. Since this quantity is $O(n \log n)$, it does not change the upper bound.

For each node v in the data structure, we know that there is no edge to any of the nodes whose adjacency list was exhausted when v was inserted into the data structure.

Since at most m nodes have their adjacency list exhausted, each node has at least $(n - m)$ possible edges that are independent of the edges used so far, and is incident on at least $(1 - \lambda)p(n - m)$ edges. The total number of edges scanned on all adjacency list is at least $m(1 - \lambda)p(n - m)$.

When $n - m = \Theta(n)$, the number of edges scanned is $\Theta(n \log n)$, as in directed graphs. Otherwise, when m is $m = \Theta(n)$ and very close to n as well, the previous approximation does not yield the same bound. Since $\Theta(\log n/\log(1/q)) = \Theta(n)$, this case corresponds to $p = \Theta(n \log n)$.

When $m/4$ of the adjacency lists have been used, at most $x = ((1 - \lambda)p)^{m/4}(n - 1)$ nodes are out of the data structure. Since $((1 - \lambda)p)^m (n - 1) < 1$ we conclude that $x < (1/(n-1))^{1/4}(n-1) = (n-1)^{3/4} < n/4$. From this point and until $m/2$ of the adjacency lists have been used, as $m < n$, each of the $m/2 - m/4 = m/4$ adjacency lists that are scanned can have edges to at least $(n - x) - m/2 > n/4$ nodes that are independent of any edges found so far. This implies that the cost is at least $p(m/4)(n/4) = \Theta(n \log n)$.

We conclude that in all undirected connected, but not complete, random graphs we need $\Theta(n \log n)$ edges, as in directed random graphs. ∎

We always examine the entire adjacency list of all nodes that we remove from the data structure, except possibly the last one. Changing the order of the edges can only change the number of edges scanned on the last node, which is too small to affect the calculation (at most $n-1$ and with expected value $p(n-1)$).

Figure 3.1 depicts the time needed to search directed graphs using a node-based searching algorithm for different values of n (on the x-axis) and p (on the y-axis). The value displayed is the number of edges examined divided by $n \log n$. The values of p are large enough so that all graphs are normally connected, and the method used for these experiments was breadth-first search.

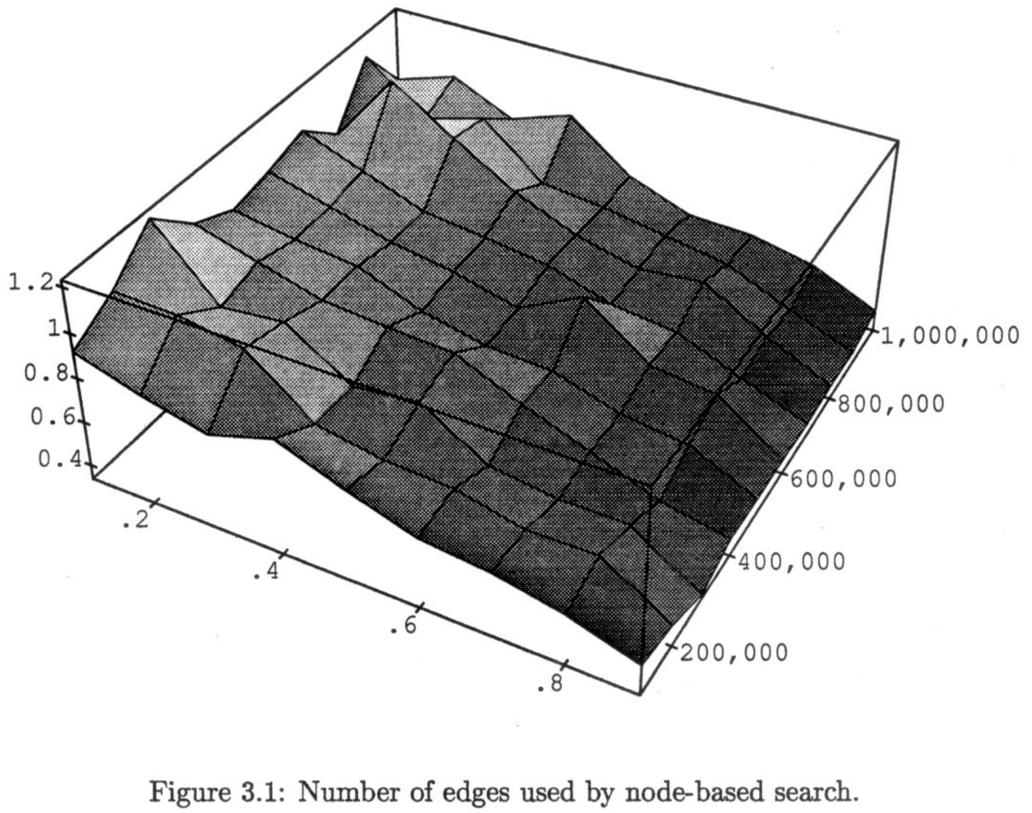

Figure 3.1: Number of edges used by node-based search.

We observe that the number of edges examined over $n \log n$ is decreasing in denser graphs, while it does not seem to vary either monotonically or significantly with n. The variance seems small too.

3.3 Stack-based search

Using the node-based searching model, we can perform a search following the depth-first search philosophy, i.e., using a stack as the data structure. Whenever we select a node from the data structure, we take one from those inserted last and located on the deepest level, so that we can go even deeper.

From Theorem 3.3, and the proof for the case of undirected graphs we obtain the following corollary:

Corollary 3.1. *Stack-based search is the fastest among all node-based searching algorithms when applied to an undirected random graph $G_p^U(n)$.*

Proof: In directed random graphs, all node-based searching algorithms need the same time. In undirected graphs all node-based searching algorithms use the same number of edges that are known not to lead to new nodes: each adjacency list scanned is known to contain one edge to the node that visited the current node.

Searching undirected graphs is not necessarily more expensive than searching directed graphs. All nodes that have their adjacency list processed completely are known not to have any edges from unvisited nodes that lead to them.

In stack-based search we have the maximum information about edges not present on the adjacency list of a node: As the current node does not have an edge to any of the nodes that were completely processed before this node was inserted into the data structure, selecting the last node from the data structure (or any node visited from the same node), gives us the node with the most known absent edges to visited nodes.

Any node from the data structure, that had been inserted into the data structure by a node w different from the last node in the data structure, v, may additionally have edges to nodes processed after w, up to and including v. ■

Stack-based search will go deeper than all other node-based searching algorithms, because it always tries to use the longest path and append nodes to that path.

Theorem 3.4. *Stack-based search is expected to reach $\Theta(1/\log(1/q))$ depth in a random graph $G_p(n)$ with $p > (1+\epsilon)/(n-1)$ for any constant $\epsilon > 0$.*

Proof: We first examine the case where $p = \Theta(1/n)$. By Lemma 2.3, no node inserts into the data structure more than a constant number of nodes. Thus, when the data structure attains its maximum size, which is linear by Theorem 3.2, the current depth is $\Theta(n)$ because nodes inserted by $\Theta(n)$ nodes need be in the data structure, and so $\Theta(n)$ nodes that have not backtracked are there, and it cannot exceed the number of nodes, n.

We use the inequality $\log x \leq x - 1$ to show that $n = \Theta(1/\log(1/q))$. Using this inequality for q we find $\log q \leq -p \Rightarrow \log(1/q) \geq p$, and using it for $1/q$ we find $\log(1/q) \leq 1/q - 1 = p/q$. As $q = \Theta(1)$, we finally find that $\log(1/q) = \Theta(p)$ and that $1/\log(1/q) = \Theta(1/p) = \Theta(n)$.

Otherwise, we use v_i to represent the nodes visited up to depth i, so that $v_0 = 1$. When $p > \Theta(1/n)$, using any value for λ, according to Lemma 2.3, we find that the first node in the data structure inserts at least $p(1-\lambda)(n-1)$ and at most $p(1+\lambda)(n-1)$ other nodes, so that $1 + p(1-\lambda)(n-1) \leq v_1 \leq 1 + p(1+\lambda)(n-1)$. Similarly we find that

$$v_{i-1} + p(1-\lambda)(n - v_{i-1}) \leq v_i \leq v_{i-1} + p(1+\lambda)(n - v_{i-1})$$

for $i > 0$, which can be written as

$$(1 - p(1-\lambda))\, v_{i-1} + p(1-\lambda)n \leq v_i \leq (1 - p(1+\lambda))\, v_{i-1} + p(1+\lambda)n$$

and telescopes to

$$(1 - p(1-\lambda))^i + p(1-\lambda)n \sum_{0 \leq j < i} (1 - p(1-\lambda))^j \leq v_i \leq$$
$$(1 - p(1+\lambda))^i + p(1+\lambda)n \sum_{0 \leq j < i} (1 - p(1+\lambda))^j$$

or simply to

$$(1 - p(1-\lambda))^i + n\left(1 - (1 - p(1-\lambda))^i\right) \leq v_i \leq (1 - p(1+\lambda))^i + n\left(1 - (1 - p(1+\lambda))^i\right).$$

We know that $(1 - p(1-\lambda))^i < 1$ and $(1 - p(1+\lambda))^i < 1$, and these terms do not affect the following analysis. We examine the data structure at the moment that it has reached its maximum size, q_n. By Theorem 3.2, we know that q_n is linear, which implies that the number of visited nodes at that time is also linear, say αn and the number of steps k that have inserted all these nodes into the data structure, satisfy the relations:

$$(1 - p(1+\lambda))^k \leq 1 - \alpha \Rightarrow k \geq \frac{\log(1-\alpha)}{\log(1 - p(1+\lambda))}$$

and

$$(1 - p(1-\lambda))^k \geq 1 - \alpha \Rightarrow k \leq \frac{\log(1-\alpha)}{\log(1 - p(1-\lambda))}.$$

As α and λ are constants, the depth in this case is also $\Theta(\log(1/q))$. ■

When $p = \Theta(1/n)$, the depth is $\Theta(n)$ and in connected graphs the depth is smaller than its maximum possible value, $\Theta(\log n / \log(1/q))$.

3.4 Breadth-first search

A classical searching strategy is the breadth-first search strategy, that falls into the node-based searching model. Its characteristic is that it can be separated into phases, in each of which the nodes at a fixed distance from the origin of the search are visited. This fact gives it its name; unlike depth-first search, it proceeds along several paths originating from the first node scanned.

All nodes are marked as unvisited before starting the breadth-first search. We put the starting node into the data structure, and we mark it as visited. While the data structure is not empty, we take the top node, we append all of its yet unvisited neighbors to the data structure, and we mark them as visited. When the data structure becomes empty, all nodes reachable from the starting node have been visited. This is the node-based searching algorithm, with a first-in, first-out queue used as data structure.

Figure 3.2 shows a random graph with $n = 300$ nodes and $p = 3/n$. The nodes appear in the order visited by breadth-first search, and the nodes drawn in each level are visited in one phase. The number next to the rightmost node at each level is the number of this node when we number from top to bottom and from left to right on each level, so it is also the total number of visited nodes up to this level. It is easy to see the number of nodes on each levelgrows fast, and that finally 17 nodes remain unvisited (and do not appear in the graph, as indicated by the numbering), since there was no edge from any nodes of the breadth-first search tree to them.

The edges that breadth-first search used to visit the new nodes are drawn as solid lines, while the edges that led to already visited nodes appear dashed in the figure. In the graph, all of these edges are equivalent. Any edges that connect the remaining 17 nodes are not drawn, because they were not used by the current invocation of breadth-first search.

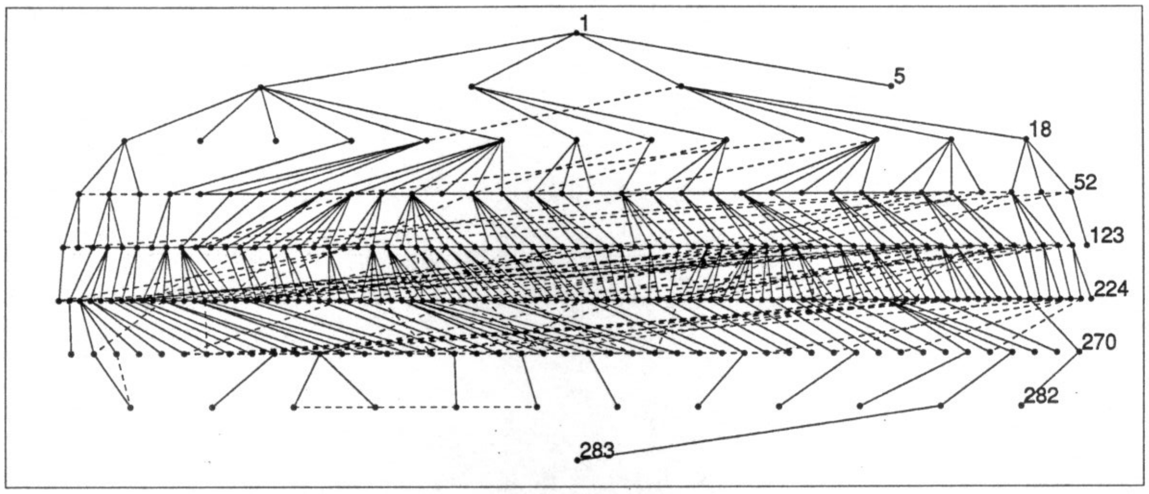

Figure 3.2: The edges scanned by breadth-first search in a graph.

We can view the same process in a different way. We group many of the previous iterations into one phase, and we describe the algorithm in terms of these phases. We only try to describe the status of the process between these phases, and not while a phase is in progress.

The first phase begins as soon as we insert the starting node into the queue. Each phase lasts until all nodes that were in the queue when the phase started have been removed from the queue and fully processed. The first phase ends when we delete the starting node from the queue and we insert all its neighbors into the queue. The second phase lasts until all neighbors of the starting node are processed, and all their previously unvisited neighbors have been inserted into the queue, and so on.

All nodes processed in a single phase have the same distance (number of edges separating them) from the starting node and in the example of Figure 3.2, they appear on the same level. Counting the number of phases needed is also useful, because it gives us the depth of the breadth-first search tree.

During the search we never stop the process at phase boundaries, and the algorithm is neither aware of phase boundaries nor takes any different action at these points; the search continues until the queue gets empty. We only consider these phases in order to count them.

At first, we insert the starting node to the queue and we mark it as visited. In the beginning of every phase the queue contains the nodes inserted during the previous phase. For each of these nodes we append all its unvisited neighbors to the queue, and we mark them as visited. If, during any phase, all nodes adjacent to the nodes already in the queue are visited, we append no nodes to the queue, the queue becomes empty at the end of the phase, and the search ends. All nodes reachable from the starting node have been visited, since we have used all edges on the adjacency lists of the visited nodes.

In the beginning of phase i the queue contains all nodes reachable from the starting node following exactly i edges on the shortest path. As we put nodes in the queue, we can immediately tell their shortest distance from the starting node.

Lemma 3.2. *When searching a random graph $G_p(n)$, with $p > (1+\epsilon)/(n-1)$ for any constant $\epsilon > 0$, using breadth-first search, the search either stops before visiting a linear number of nodes with no more than a constant probability, or visits at least a linear number of nodes, in $\Theta(\log(n)/\log(pn))$ phases.*

Proof: The search stops before a linear number of nodes, only if the search does not start in the giant component; this occurs with at most constant probability, the percentage of nodes in the small components.

Otherwise, as no more than a linear number of nodes, say $(1-\alpha)n$ for some constant $0 < \alpha < 1$, will be visited, in all phases we are considering (but maybe the last one) at any time, there will be at least a linear number of unvisited nodes, at least αn.

Let a_i the nodes at the end of phase i, and $a_0 = 1$ the nodes at the beginning of phase 1.

Using the value of λ as defined on Lemma 2.3 for the subgraph that contains the current node and all unvisited nodes, which is a random graph with at least (αn nodes), we can find that each of the a_i nodes contributes at least $(1-\lambda)p\alpha n$ unvisited nodes, so that $a_{i+1} \geq (1-\lambda)pna_i\alpha$.

We can see that a geometrical progress is formed, which grows fast. The solution of this formula, where $c = (1-\lambda)\alpha$ is a constant, is $a_i \geq (cpn)^i a_0 = (cpn)^i$ which indicates that we will have a linear number of visited nodes in

$$O(\log n/\log(cpn)) = O(\log n/(\log c + \log(pn))) = O(\log n/\log(pn))$$

phases, as $c = O(pn)$.

Similarly we can find that each of the a_i nodes contributes at most $(1+\lambda)pn$ edges for prospective nodes to be visited, for a final number of new visited nodes at most $(1+\lambda)pna_i$.

A new geometrical progress is formed, $a_{i+1} \leq (1+\lambda)pna_i$ with solution

$$a_i \leq ((1+\lambda)pn)^i a_0 = ((1+\lambda)pn)^i$$

which indicates that we will have a linear number of visited nodes in

$$\Omega\left(\log n/\log\left((1+\lambda)pn\right)\right) = \Omega\left(\log n/\left(\log(1+\lambda) + \log(pn)\right)\right) = \Omega(\log n/\log(pn))$$

phases, as $1+\lambda = O(pn)$.

So we finally need $\Theta(\log n/\log(pn))$ phases. ∎

It is easy to see that the number of nodes inserted in the last phase alone is linear to the number n.

Theorem 3.5. *The expected value of the depth of the breadth-first search algorithm is* $\Theta(\log(n)/\log(pn))$*, on a random graph* $G_p(n)$ *with* $p > (1+\epsilon)/(n-1)$ *for any constant* $\epsilon > 0$.

Proof: Using Lemma 3.2, we know that we need $\Theta(\log(n)/\log(pn))$ phases for reaching the state in our search where the next phase contains at least a linear number of nodes, say cn. After this stage, the search will continue until some or all the nodes, if possible, are visited.

In the example graph of Figure 3.2, during the first three levels the growth rate is really high, and almost non decreasing, and only a small portion of the nodes is already visited. This divides the two parts that correspond to the parts in our analysis. After that, only a constant number of phases is expected. On the intermediate levels, we have many nodes on all levels, the increase is significant, but the increase rate is not. The reverse effect can be observed on the last two or three levels. The number of nodes on these levels decreases drastically, and only a few nodes are left unvisited at that point.

We do not claim that the final graph will contain all nodes, and so that the graph will be connected, but just that the subset of these nodes that will be accessed by the initial node are really not far away in depth.

A node will be at distance at least k from the cn nodes, if there is no path of length at most $k-1$ from that node to any of the cn nodes, and we need at least k phases to reach this node. This implies that neither this node nor any of the next $k-2$ nodes on the path from this node to the cn nodes have any edge connecting them to any of the cn nodes. The probability that there is a node at distance k is $(q^{cn})^{k-1} = ((q^n)^c)^{k-1} < (\alpha^c)^{k-1} = (\beta^{k-1})$ for some constants $0 < \beta < \alpha < 1$. But if k is any increasing function of n, this converges to zero, which implies that at most a constant number of phases will be further needed, and the total bound is $\Theta(\log(n)/\log(pn))$. Since the probability that the search will terminate before we even reach a linear number of nodes is at most a constant, the expected value for the depth is not affected by that. ■

Note that $\Theta(\log n/\log(pn)) = O(\log n)$ for the graph densities we are considering.

Theorem 3.6. *The depth of breadth-first search on a connected graph G is at least half the diameter of the graph and at most the diameter of the graph.*

Proof: If the depth of breadth-first search was less than half the diameter of the graph, every node would be in distance less than half the diameter of the graph from the starting node, and the distance between any two nodes would be less than the diameter of the graph.

On the other hand, if the depth of breadth-first search is greater than the diameter of the graph, it means that there is a node in distance more than the diameter of the graph from the starting node. ■

Chapter 4

Depth-first search

Depth-first search, one of the oldest searching algorithms, finds numerous applications as part of other algorithms [Ta]. It is used in algorithms for biconnectivity, strong connectivity, planarity, isomorphism of planar graphs and others. The properties of depth-first searchhave been studied extensively. See Reference [Re], Reference [KO], Reference [CSW], Reference [AA] and Reference [AAK] for such examples.

Depth-first search tries to visit all nodes in a long, deep path. Whenever a new node is inserted into the data structure, it also becomes the new current node. Edges that lead to visited nodes are ignored. When no edge to an unvisited node can be found from the current node, the node on the maximum depth so far becomes the new current node.

Depth-first search falls under the new-node searching model, but not under the node-based searching model. This searching procedure, displayed in Program 4.1, can be obtained by modifying the generic searching procedure of Program 1.1:

In searching algorithms that do not fall into the node-based searching model (e.g. depth-first search), nodes whose adjacency list is partially scanned must be stored in the data structure. This can be done by keeping pointers to the adjacency lists of the nodes, in addition to the nodes themselves.

The depth-first search algorithm is most commonly represented by an elegant recursive routine. Each node contains a flag *visited* which is initially set to false, meaning that the node has not been encountered during the search yet. The flag is set the first time the node is visited. The purpose is to visit as many nodes as possible in the special order resulting from the following procedure, displayed in Program 4.2:

```
procedure depth-first search($s$ : node)
begin
    data structure:= empty
    insert $s$ into the data structure
    $visited[s]$ := true
    while the data structure is not empty
    begin
        select an element $v$ from the data structure
        while there exists $w$ such that there is an unused edge from $v$ to $w$
            if not $visited[w]$ then
            begin
                insert $w$ to the data structure
                $visited[w]$ := true
                if all nodes are visited then return
                $v := w$
            end
        remove $v$ from the data structure
    end
end
```

Program 4.1: The non-recursive depth-first search algorithm.

```
procedure dfs($v$ : node)
begin
    $visited[v]$ := true
    foreach $w$ such that there is an unused edge from $v$ to $w$
        if not $visited[w]$ then
        begin
            dfs($w$)
            if all nodes are visited then return
        end
end
```

Program 4.2: The recursive depth-first search algorithm.

We pick an arbitrary node as our starting node, and we mark it as visited. Then, we start examining all edges from this node to other nodes. Whenever we find an edge to an unvisited node, we recurse and initiate a depth-first search starting from the new node (we *follow* that edge); when the recursive step finishes, we continue examining the rest of the edges of our original node. When all edges of a node have been examined, we return (we *backtrack*).

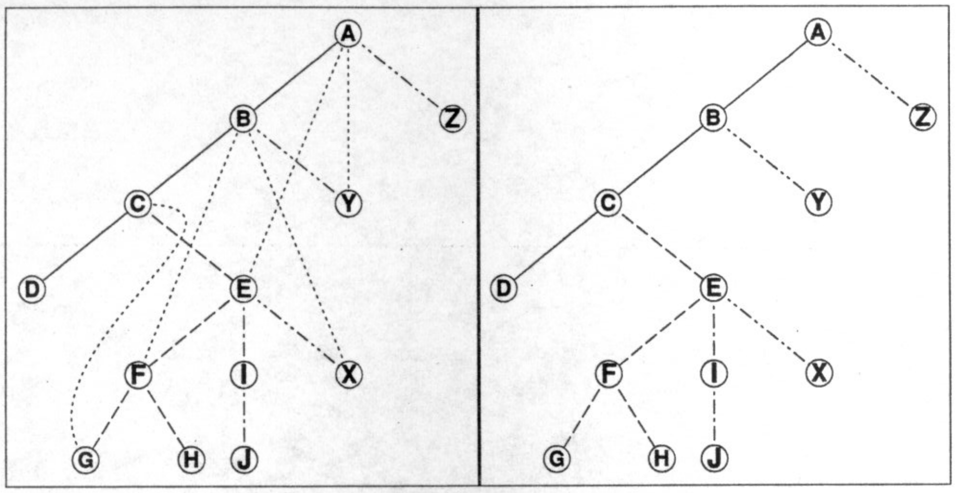

Figure 4.1: A graph and its depth-first search tree.

Figure 4.1 shows an example graph of 13 nodes (on the left) and its depth-first search tree (on the right), which is formed from the graph edges followed during the search. The node labels are derived from their searching order. When the search starts at node A, we visit nodes B, C, D (in this order). Then, as there is no other edge to follow, we backtrack to node C, and continue examining the nodes E, F, G, then we backtrack again, etc. Edges that lead to visited nodes are omitted from the resulting tree, since the algorithm ignores them anyway.

The different styles of the edge lines correspond to the edge grouping we use in the depth analysis, and will be explained later. They do not imply that the edges are not equivalent in any way. Depth-first search does not follow the dotted edges because they lead to visited nodes.

Depth-first search always tries to continue from the node visited last, and to go as far in depth as possible. Whenever a new node is visited, control is transferred into this node. If a visited node is found, it is ignored, and when there are no more edges to be scanned at a node, the search continues from the last node visited that still has unexamined edges.

In the literature, depth-first search has been confused with stack-based search sometimes, because both of them use a stack as data structure. The stack-based search algorithm may substitute for the real depth-first search in some cases. However, stack-based search is not equivalent to the real depth-first search. The difference becomes obvious in a complete graph: The stack-based search method, like all node-based searching methods, inserts all children of the starting node into the data structure; then it terminates, as all nodes have been visited. In contrast, depth-first search reaches depth n, since it does not backtrack until all nodes are visited, and there is always an edge from the current node to an unvisited node.

4.1 Time needed by depth-first search

The number of edges used by depth-first search is difficult to estimate. Backtracking, which can happen on nearly any node, makes the analysis very complex. It is not enough to concentrate our estimation on the current node, and compute some probability of backtracking, ignoring nodes searched before; if we backtrack, we resume the search from a previously visited node and we must remember all information associated with it.

We want to find the expected running time of depth-first search, so that we can compare this method with the others. We cannot consider the analysis complete unless the running time is estimated.

The case of the complete graph is easier; there is no backtracking, and the depth of the search is n. We only have to estimate the number of edges that will be scanned until we proceed to the next unvisited node.

We first examine extreme cases, when the edges are in a special order on the adjacency lists. We consider a complete graph with n nodes and we number the nodes from 1 to n. Let our starting node be node 1.

If the adjacency lists are sorted so that edges to higher numbered nodes always come after edges to lower numbered nodes, then we visit the nodes in their number order (from 1 to n). When we visit node i, we need to examine i edges in order to find node $i+1$, the first unvisited node on the adjacency list of node i. Thus we need to scan a total of $n(n-1)/2$ edges for the entire graph.

If the first edge on the adjacency list of node i is the edge leading to node $i+1$, for $1 \le i < n$, then we find the next unvisited node when we examine the first edge on the adjacency list of the current node, and we scan only $n-1$ edges for the whole graph.

The average case is somewhere in between the above examples.

Lemma 4.1. *In a complete graph with n nodes, the expected number of edges scanned by depth-first search is $nH_n - n$.*

Proof: When i of the other $n-1$ nodes (except the one we are processing) are visited, $0 \le i \le n-2$ we need to scan at most $i+1$ edges to visit a new node. If j (for $0 \le j \le i$) attempts have failed to find an unvisited node, the next edge scan fails with probability $(i-j)/(n-1-j)$. The probability of failing on at least the first k edges, $1 \le k \le i$ and have to scan more edges in order to visit a new node, is

$$\prod_{0 \le j < k} \frac{i-j}{n-1-j} = \frac{i!}{(i-k)!}\frac{(n-1-k)!}{(n-1)!} = \binom{n-1-k}{n-1-i} \Big/ \binom{n-1}{i}$$

which means that the number of edges scanned at this node is expected to be

$$1 + \sum_{1 \le k \le i} \binom{n-1-k}{n-1-i} \Big/ \binom{n-1}{i}$$

and for the entire search:

$$\begin{aligned}
& \sum_{0 \le i < n-1} \left(1 + \sum_{1 \le k \le i} \frac{\binom{n-1-k}{n-1-i}}{\binom{n-1}{i}} \right) \\
= \;& n-1 + \sum_{0 \le i < n-1} \sum_{1 \le k \le i} \frac{\binom{n-1-k}{n-1-i}}{\binom{n-1}{i}} \\
= \;& n-1 + \sum_{1 \le i < n-1} \frac{1}{\binom{n-1}{i}} \sum_{1 \le k \le i} \binom{n-1-k}{n-1-i} \\
= \;& n-1 + \sum_{1 \le i < n-1} \frac{1}{\binom{n-1}{i}} \sum_{0 \le j \le i-1} \binom{n-1-i+j}{n-1-i} \\
= \;& n-1 + \sum_{1 \le i < n-1} \frac{\binom{n-1}{i-1}}{\binom{n-1}{i}} \\
= \;& n-1 + \sum_{1 \le i < n-1} \frac{i}{n-i}
\end{aligned}$$

$$\begin{aligned}
&= n-1+\sum_{1\le i<n-1}\left(\frac{n}{n-i}-1\right)\\
&= 1+n\sum_{1\le i<n-1}\frac{1}{n-i}\\
&= 1+n(H_n-1-\frac{1}{n})\\
&= nH_n-n
\end{aligned}$$

using the identity

$$\sum_{0\le j\le r}\binom{k+j}{k}=\binom{k+r+1}{r}. \blacksquare$$

Notice that the result is exactly the same for directed and undirected graphs, where on undirected graphs we count each edge twice, if it was used in both directions. The time needed for depth-first search on a complete graph is $\Theta(n\log n)$ since $H_n \simeq \log n$.

Theorem 4.1. *In a disconnected random graph $G_p(n)$, depth-first search is expected to use $\Theta(pn^2)$ edges. For connected graphs (p is sufficiently large)*

$$T(G_p^U(n)) \le T(G_p^D(n)) + n - 1.$$

Proof: In a disconnected graph, the theorem is a direct consequence of Lemma 2.2; otherwise, of Lemma 2.5. ■

Searching an undirected graph can be cheaper than what we estimated above when we have backtracks; we know that no edge leading to the node we backtracked will be found on any unvisited node. As backtracks usually occur at the end of the search, such information is not be very helpful.

It is difficult to estimate the exact running time needed by depth-first search, because it is difficult to calculate the probability of finding a new node after backtracking. The probability of backtracking at a node increases for nodes with smaller degree, and the situation is difficult to analyze.

When the graph is disconnected, there are $O(n \log n)$ edges, and we scan all of them. In the complete graph, we only use $O(n \log n)$ of the edges. As we show in Chapter 6, the same holds for connected but not complete graphs.

Figure 4.2 depicts the time needed by depth-first search on a directed (left) and undirected (right) graph, for different values of n (on the x-axis) and p (on the y-axis). The two pictures are similar, and the same remarks apply to both of them. The value displayed is the number of edges used divided by $n \log n$. The values of p are large enough so that all graphs are normally connected.

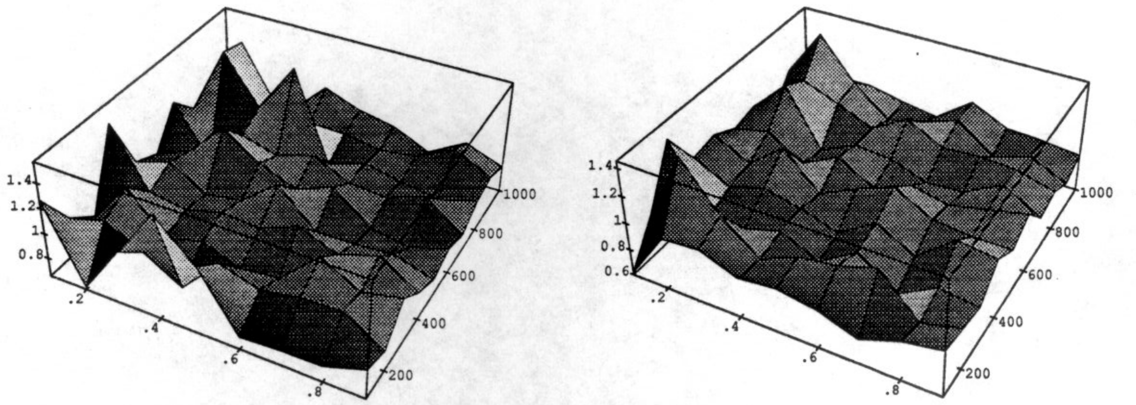

Figure 4.2: Edges used by depth-first search.

The number of edges used looks the same for directed and undirected graphs. Although there is a variation on the values displayed, the smallest and the largest of these values are within a factor of two, and the variation decreases as n gets larger.

4.2 Background

The quantity q^n appears in the following section; let us examine q^n when $n \to \infty$, where $0 < q = 1 - p < 1$ and $p = \Omega(1/n)$.

Lemma 4.2. *Let us denote* $0 < q = 1 - p < 1$. *If* $p = \Theta(1/n)$, *then* q^n *is bounded by constants, i.e.,* $q^n = \Theta(1)$, *when* $n \to \infty$. *But, if* $p > \Theta(1/n)$, *then* $q^n \to 0$, *when* $n \to \infty$.

Proof: When $p = c/n$, it is known that $q^n \to (1 - c/n)^n \to e^{-c}$; i.e., when $p = \Theta(1/n)$, there exist constants c_1, c_2 such that $c_1/n < p < c_2/n$ for large values of n so that q^n is bounded between e^{-c_1} and e^{-c_2} and $q^n = \Theta(1)$.

When $p > \Theta(1/n)$, we use the inequality $e^x \geq x + 1$, so that:

$$\begin{array}{rll}
p > \Theta(1/n) & \Rightarrow & \forall c \;\; \exists n_o \;\; \text{such that} \;\; \forall n \geq n_o \;\; p > c/n \\
& \Rightarrow & \forall c \;\; \exists n_o \;\; \text{such that} \;\; \forall n \geq n_o \;\; p > -\log(e^{-c})/n \\
& \Rightarrow & \forall \varepsilon \;\; \exists n_o \;\; \text{such that} \;\; \forall n \geq n_o \;\; p > -\log(\varepsilon)/n \\
& \Rightarrow & \forall \varepsilon \;\; \exists n_o \;\; \text{such that} \;\; \forall n \geq n_o \;\; p > 1 - e^{\log(\varepsilon)/n} \\
& \Rightarrow & \forall \varepsilon \;\; \exists n_o \;\; \text{such that} \;\; \forall n \geq n_o \;\; p > 1 - \varepsilon^{1/n} \\
& \Rightarrow & \forall \varepsilon \;\; \exists n_o \;\; \text{such that} \;\; \forall n \geq n_o \;\; |(1-p)| < \varepsilon^{1/n} \\
& \Rightarrow & \forall \varepsilon \;\; \exists n_o \;\; \text{such that} \;\; \forall n \geq n_o \;\; |q^n| < \varepsilon \\
& \Rightarrow & q^n \to 0. \;\blacksquare
\end{array}$$

4.3 The depth of the depth-first search

Estimating the average depth of the depth-first search is entirely different from estimating the time needed for a depth-first search traversal of a graph. We can stop monitoring the search earlier, when nearly all nodes are visited; visiting the last few nodes, albeit generally expensive in time (as it encounters many edges to visited nodes), does not add much to the final depth. Although this makes the problem easier, we do care about the structure of the depth-first search tree formed, which increases the difficulty of the calculation.

We estimate the depth of the depth-first search up to a certain point in the search. This means that the real depth is always greater or equal to the depth we estimate, up to the point where we stop.

The depth-first search procedure is recursive, so we describe it by recursive formulas. When we estimate the depth of the search, an important property is that searching a subgraph of the original graph is tantamount to searching a smaller graph, the graph with all unvisited nodes, since the algorithm never follows edges to visited nodes. The actual time spent by the search on the subgraph is longer than what would be needed for a graph of that size, but the depth that the search reaches is the same, since all extra edges (those to the visited nodes) are never followed.

The depth of the depth-first search is the maximum number of active nodes at any time, i.e., visited nodes whose search has not been completed at that time yet. This is the same as the size of the stack that will be needed.

The exact depth of the depth-first search in a particular graph is not a quantity of interest, since it depends on the graph, and even on the order of the edges on the adjacency list (this dependency can be viewed as another graph, with the edges rearranged) and the node we start from (this dependency can be viewed as another graph, with the nodes rearranged). We are interested in the quantity d_n, defined as follows:

Definition 4.1. *We define $D(G)$ as the depth of the depth-first search algorithm on a specific graph G. We also define $d_n = D(G_p(n))$ as the average value of $D(G)$ over all graphs $G_p(n)$.*

For example, suppose we have an undirected graph with $n = 3$ nodes, A, B and C, and each edge AB, AC and BC exists with probability $p = 1/2$. We have eight possible graphs, all with equal probability. Using depth-first search and starting from node A, the search stops at node A, reaching depth 1, in the graph with no edges and in the graph with only edge BC. The search reaches depth 2 in the graph with only edge AB, in the graph with only edge AC, and in the graph with edges AB and AC, and reaches depth 3 in the graph with edges AB and BC, in the graph with edges AC and BC, and in the complete graph. The average depth over all these graphs is $d_n = 1*2/8+2*3/8+3*3/8 = 17/8$. The depths are similar if we start from a different node, since all possible graphs are symmetrical with respect to the nodes, so that the average value is the same.

For a dense graph, it is intuitive that $D(G) = n$, since backtracking is unlikely. We are able to prove that d_n is $\Theta(n)$ even for sparse graphs, and asymptotically n when $p > \Theta(1/n)$. For connected graphs, we can apply Theorem 3.1 and find that we cannot backtrack on more than $\Theta(\log n/\log(1/q))$ nodes. Since the latter quantity is $\Theta(n)$ for $p = \Theta(\log n/n)$, the bound is inadequate for $p = O(\log n/n)$.

Figure 4.3 depicts the depth of the depth-first search on directed (left) and undirected (right) graphs, for different values of n (on the x-axis) and p (on the y-axis). As expected, the two pictures are similar. The value displayed is the depth reached divided by n. The values of p are large enough so that all graphs are normally connected.

The depth seems not to increase with p when the graph is sufficiently dense; this behavior is expected, because no backtracks occur. The diagrams for directed and undirected graphs have the same shape, as expected from Lemma 2.4.

Experiments for different values of n and p on directed graphs yield the values of $D(G)$ shown in the next table. Rows correspond to a particular value of n, columns correspond to

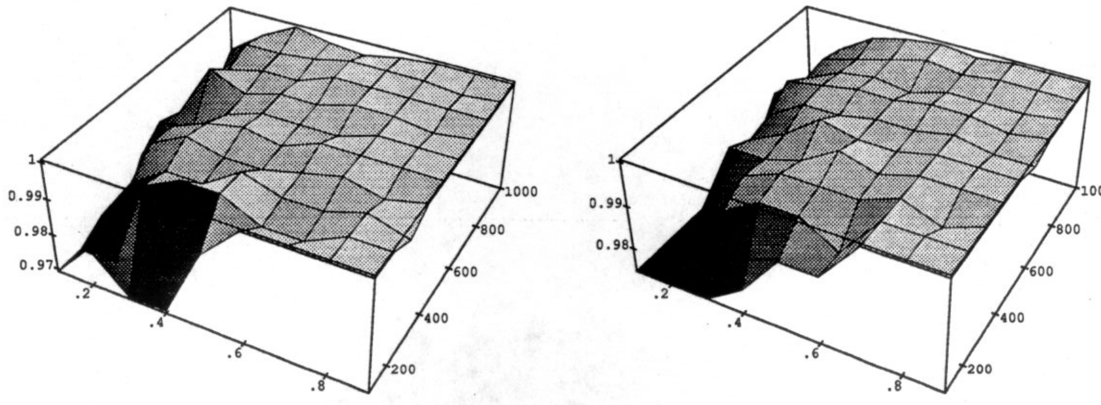

Figure 4.3: Depth reached by depth-first search.

a particular value of $r = pn$. The case where pn is constant is the most difficult to analyze, because the graph is not only sparse, but even disconnected for large values of n.

n\r	4	8	16
100	56	90	96
1000	602	794	904
10000	5968	7836	8983
100000	59387	79454	89819
1000000	594384	793946	897632

Note that $D(G)$ seems to be larger than $n/2$, i.e., linear, even for small values of p. It looks like it is converging to a linear function of n in all cases.

The total number of nodes we can access is a constant fraction of the nodes of the graph, given by Theorem 2.1(Bollobás). Thus we expect d_n to always be less than the number of nodes in the giant component, $(1 - t(c))n$, where $t(c)$ is given by Equation 2.1. This value is close to one and cannot exceed one, as no more than n nodes exist.

Lemma 4.3. *In a random graph $G_p(n)$ the probability p_{in} that depth-first search visits at least i nodes before backtracking for the first time, is equal to:*

$$p_{in} = \prod_{n-i<j<n} (1 - q^j). \tag{4.1}$$

Proof: For $i = 0$ and $i = 1$, p_{in} is 1, since we always visit at least 1 node before backtracking. The probability of backtracking at some node, after having visited exactly i nodes is q^{n-i}, and the probability that depth-first search visits at least i nodes before backtracking for the first time is:

$$p_{in} = p_{(i-1)n}(1 - q^{n-(i-1)}). \tag{4.2}$$

For $i > 1$, and as $p_{0n} = 1$, this telescopes to

$$p_{in} = \prod_{1 \leq j < i} (1 - q^{n-j}) = \prod_{n-i<j<n} (1 - q^j). \ \blacksquare$$

Notice that $p_{(n+1)n} = 0$. We use this fact below to keep the formulas simpler.

The probability of backtracking for the first time after visiting exactly i nodes is $p_{in} - p_{(i+1)n}$. Using Equation 4.1 and Equation 4.2, we derive two useful formulas relating probabilities p_{in} for different values of i and n:

$$p_{in} - p_{(i+1)n} = p_{in}q^{n-i} \tag{4.3}$$

and

$$p_{in} = (1 - q^{n-1})p_{(i-1)(n-1)}. \tag{4.4}$$

Definition 4.2. *We define $V(G)$ as the depth of the depth-first search algorithm on a specific graph G, until we backtrack for the first time. We also define $v_n = V(G_p(n))$ as the average value of $V(G)$ over all graphs $G_p(n)$.*

In the example of Figure 4.1, $V(G)$ is 4, because the search proceeds along the path ABCD, until it backtracks at D. The edges of this path are marked as solid in the picture. At node A, the depth of the depth-first search is 1, at node B it is 2, at node C it is 3, etc, until we backtrack for the first time, which happens at D in our example. Until the first backtrack, the depth of the search tree increases by 1 every time a new unvisited node is found, and this is very helpful in our calculations. After backtracking, we may not be at maximum depth, so that using a new node does not always increase the maximum depth encountered so far.

The values of $V(G)$ found experimentally for different values of n and p using directed graphs are displayed in the next table. Again, rows correspond to a particular value of n and columns correspond to a particular value of $r = pn$.

n\r	4	8	16
100	10	71	81
1000	9	463	669
10000	1	510	5807
100000	30	2892	40903
1000000	28	2727	336532
10000000	39	6080	2074453
100000000	3	415	2989693

Notice that for small values of p the search either stops soon (although this does not happen most of the time), or $V(G)$ seems to asymptotically converge to a certain constant value.

Lemma 4.4. *The value of v_n, as given by Definition 4.2, for $n \geq 1$ satisfies the relation:*

$$v_n = (1 - q^{n-1})v_{n-1} + 1 \qquad (4.5).$$

Proof: According to the Definition 4.2, $v_0 = 0$, although this is not used in the recursive formula, since the coefficient of v_0 is always $1 - q^0 = 0$.

Let us estimate v_n, using the values of n, p and q:

$$\begin{aligned} v_n &= \sum_{1\leq i\leq n} (p_{in} - p_{(i+1)n})i \\ &= \sum_{1\leq i\leq n} p_{in} \end{aligned}$$

$$
\begin{aligned}
&= \sum_{1\le i\le n} \prod_{1\le j<i} (1-q^{n-j}) \\
&= 1 + \sum_{2\le i\le n} \prod_{1\le j<i} (1-q^{n-j}) \\
&= 1 + \sum_{1\le i\le n-1} \prod_{1\le j\le i} (1-q^{n-j}) \\
&= 1 + (1-q^{n-1}) \sum_{1\le i\le n-1} \prod_{2\le j\le i} (1-q^{(n-1)+1-j}) \\
&= 1 + (1-q^{n-1}) \sum_{1\le i\le n-1} \prod_{1\le j<i} (1-q^{(n-1)-j}) \\
&= 1 + (1-q^{n-1})v_{n-1}. \ \blacksquare
\end{aligned}
$$

Equation 4.5 can also be derived by a different reasoning: when we search a graph with n nodes, we first visit the starting node. With probability q^{n-1} there is no edge to any other node, and the search stops. With probability $1-q^{n-1}$, we visit another node, where we start a new search in the rest of the graph, a graph of $n-1$ nodes. The recurrence implied is $v_n = q^{n-1} + (1-q^{n-1})(1+v_{n-1})$, which is equivalent to Equation 4.5.

Corollary 4.1. *The expected value of the depth of the depth-first search algorithm on a random graph $G_p(n)$ with $p = \Theta(1/n)$ is at least a constant.*

Proof: Using Lemma 4.2 the recurrence for v_n implies that $v_n \to e^c$ as $n \to \infty$ for $c = pn$ a constant. $\blacksquare$

As $e^4 = 54.6$, $e^8 = 2981$ and $e^{16} = 8886111$, these are the values to which the table columns should converge. We expect v_n to always be less than the number of nodes in the giant component, $(1-t(c))n$.

Lemma 4.5. *If $p > \Theta(1/n)$, then the depth of the depth-first search is asymptotically n.*

Proof: When $p > \Theta(1/n)$, we know from Lemma 4.2 that $q^n \to 0$ and the recurrence for v_n implies that $v_n - v_{n-1} \to 1$ and so $v_n/n \to 1$ or $v_n \to n$. $\blacksquare$

As the size of the giant component approaches n, because $t(c) \to 0$ as $c \to \infty$, we can access nearly all of the graph's nodes.

Since the depth of the depth-first search cannot be greater than n, the bound for $p > \Theta(1/n)$ is the best we can get.

Later we prove that even when $p = \Theta(1/n)$, the depth of the depth-first search is $\Theta(n)$. It is still an open problem to estimate the coefficient in the latter case. A possible approach is to generalize v_n to include paths after some backtracks; doing so for up to a particular number of consecutive backtracks resulted in a better constant, but not to a linear quantity.

Lemma 4.5 seems to say that, even though there is backtracking for sparse graphs, it is not likely to affect the length of the longest path.

In stack-based search and for $p > \Theta(1/n)$, the depth of the search is still linear, but the coefficient is not one, as in depth-first search.

Chapter 5

Walk-first search

Another searching method, walk-first search, is sometimes also called depth-first search. Depth-first search is not a good name for this algorithm, since the algorithm is not equivalent to depth-first search; walk-first search is not a stack based algorithm, although it is very similar. In the literature, the distinction between the two methods is not always clear.

Walk-first search is used for finding Eulerian circuits in graphs [PTW], because it forms long paths, not necessarily simple.

Each node contains a flag *visited*, which is initially set to false, meaning that the node has not been encountered during the search yet. The purpose is to visit as many nodes as possible in a special order and mark them as visited. Whenever we visit a node, visited or not, this node becomes the new current node, and its adjacency list is scanned next. This searching procedure, displayed in Program 5.1, can be obtained by modifying the generic searching procedure of Program 1.1.

This algorithm differs from depth-first search in that whenever we visit a visited node that is still in the data structure, we exchange that node with the current one (the last one in the data structure) before we continue searching. This becomes obvious by comparing the code for walk-first search and the non-recursive code for depth-first search.

Walk-first searchcreates long paths, usually not simple, and is appropriate for algorithms like those finding Eulerian circuits. Depth-first searchcreates long simple paths and is appropriate for algorithms like those detecting strongly connected components.

In searching algorithms not encompassed by the node-based searching model, like walk-first search, nodes whose adjacency list is partially scanned must be stored in the data

```
procedure walk-first search($s$ : node)
begin
    data structure:= empty
    insert $s$ into the data structure
    $visited[s]$ := true
    while the data structure is not empty
    begin
        select the last element $v$ from the data structure
        while there exists $w$ such that there is an unused edge from $v$ to $w$
            if not $visited[w]$ then
            begin
                insert $w$ to the data structure
                $visited[w]$ := true
                if all nodes are visited then return
                $v := w$
            end
            else if $w$ is still in the data structure then
            begin
                exchange $v$ and $w$ in the data structure
                $v := w$
            end
        remove $v$ from the data structure
    end
end
```

Program 5.1: The walk-first search algorithm.

structure. This can be done by keeping pointers to the adjacency lists of the nodes, in addition to the nodes themselves.

Walk-first search is the only algorithm we have seen so far that takes an action for the edges that lead to visited nodes, and thus is not encompassed by the new-node searching model: it exchanges the position in the data structure of the two nodes incident to the edge. This can be handled easily with inverse permutations, by keeping an array where each node can find its index in the data structure array.

Walk-first search also bears some similarity with a *random walk*, the graph traversal mechanism where we follow a random edge from the current node to reach a new current node; in walk-first search, after scanning a new edge, we usually continue the search from the target node. This similarity gives it its name. The difference is that in walk-first search we never use the same edge twice, and we never move to a node totally processed, i.e., a node which we have removed from the data structure after examining all its edges When a node is removed from the data structure, we use the data structure to find the last node there and use it as the current node to resume searching.

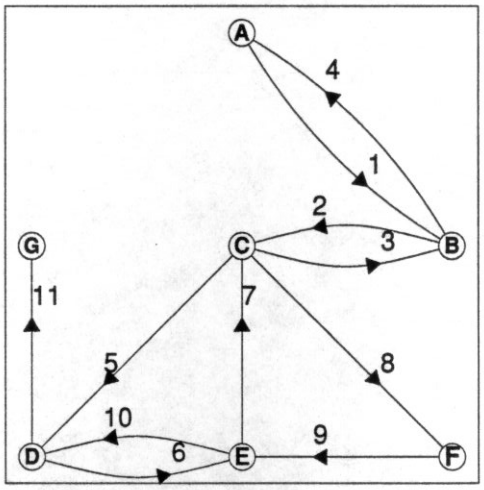

Figure 5.1: Walk-first search in a graph.

Figure 5.1 shows an example graph and its edges numbered in the order they are visited by walk-first search. The node labels are derived from the accessing order of the nodes. The search starts at node A, and the next nodes we visit are B and C. At this point the data structure contains the nodes A, B and C, and the current node is always the last node in the data structure. The first edge in the adjacency list of C leads to node B, which is already visited; depth-first search would ignore this edge and would look for the next edge of C. However, walk-first search does use the edge and visits B again. There is no need to mark it visited this time, it is already marked. At this point, the data structure contains the nodes A, C and B, as the new current node is exchanged with the previous one. Similarly, as the first edge on the adjacency list of B leads to A, walk-first search visits A again and the data structure contains the nodes B, C and A.

Node A does not have any other edge on its adjacency list, and is therefore deleted from the data structure; edge AB has been used in the past and is not used again. The last node in the data structure is C, and the search resumes from C. Edge CB will not be used again either, and the next node we visit is D, following the next edge on the adjacency list of C. From D, we find E next. The first edge on the adjacency list of E leads to C, which is visited again. The next edge of C leads to F; from F the first edge leads to E. The next edge on the adjacency list of E leads to D. From D, the next node is G. The search completes, since all nodes are visited now; any remaining edges will not be used (and are not illustrated in the picture).

5.1 Properties of walk-first search

The property of walk-first search to always use the next edge found on the adjacency list of the current node makes it quite different from the other searching algorithms. Nodes are treated in a more uniform way, since all possible edges have the same probability to be present in the graph, and next on the adjacency lists.

In node-based searching algorithms, we use the adjacency lists of some nodes as much as possible, up to exhaustion, and those of most other nodes not at all. In depth-first search, although we use the adjacency lists of all nodes in the data structure, we do not scan the same number of edges from each one of them. When we are processing the first few nodes, it is very easy to find an unvisited node, as most of them are unvisited; when we are processing the last nodes, however, it is very difficult to find an unvisited node, as most of them are visited by that time, and we have to use many entries of the adjacency list. Sometimes we even use all of the entries, and we backtrack to the previous node to continue using more entries from its adjacency list.

In walk-first search we need random access to the data structure. Adding and removing elements only at the two ends of the data structure is no longer enough; when we visit an already visited node, we have to exchange it with the current node in the data structure. Thus we also need to keep the position of each node in the data structure.

Another interesting consequence of the way that walk-first search works in undirected graphs is that for all (but one) nodes in the data structure, the number of edges used to enter the node is the same as the number of edges used to leave this node and enter another:

every time we enter a node, we use the first available edge and leave the node. Since the number of edges we can use to enter a node is the same as the number of edges we can use to leave the node (they are the same edges after all, used in the opposite directions), we can always find an edge to leave the node, and we do not remove the current node from the data structure as completely searched, even if we have used all edges entering and leaving the node.

The only node that can be removed from the data structure is the node from which we started the search, or, if this one is already removed, the node selected from the data structure to replace it, the last one in the data structure at the moment; this is the only node for which we have used an edge to leave the node, without having used an edge to enter the node previously.

We use edges from all nodes uniformly, so the probability of exhausting the adjacency list of the starting node, and remove it from the data structure, is very small.

A similar situation occurs in directed graphs. When the adjacency list of a node in the data structure is exhausted, we remove this node from the data structure and continue searching from its successor in the data structure. There may still be nodes with edges leading to the deleted node, because there can be more edges entering a node than leaving it. When any of these edges is encountered, we ignore it, since the target node is not in the data structure and should not be inserted. Equivalently, we can insert the target node into the data structure again, try to continue the search from that node, delete it from the data structure (because it has no unused entries on its adjacency list), and continue the search from the previous node, which is at the top of the data structure.

For all nodes in the data structure, the number of edges used to enter the node (the in-degree used so far) is less or equal to the number of edges used to leave this node and enter another (the out-degree used so far), depending on the existence of edges leading to nodes already out of the data structure.

Until the first node is deleted from the data structure, all nodes (except the starting one) have the same number of edges used to enter and leave this node, as in undirected graphs. Thus, the probability of node removal from the data structure is very small.

If the graph is disconnected, although nodes are not removed from the data structure very early, all nodes will be removed eventually, since all their edges will be used in the effort to find a path to the remaining nodes.

In walk-first search, the notion of depth is not well defined. Is it the size of the data structure needed ? Although this quantity has an obvious practical usefulness, it really represents nothing during the search, since the elements of the data structure are shuffled extensively. Is it the longest path from the starting node to any other node ? The longest simple path from the starting node to any other node may make more sense, since we omit cycles, but both of them have no direct correspondence with any search parameters or the data structure contents.

The only quantities that seem to be of interest are the running time and the size of the data structure needed to complete a walk-first search. We note here that the cases of directed and undirected graphs must be considered separately, because edges to already visited nodes are not ignored, but are used to change the current node.

5.2 Time needed by walk-first search

The exact time spent on walk-first search is not estimated easily, because it is difficult to keep track of the state of each node. Fortunately, we can get an upper bound for it. Note that this is an upper bound on the average time needed, not a worst case time!

We start with a remark on the number of edges needed by walk-first search, which is used in the subsequent proofs. The expected time to search a random graph $G_p(n)$ decreases whenever we forbid the existence of an edge between two visited nodes, or, similarly, whenever we have already used an edge and we know that we will not use this edge between these two nodes again. The reason is that the probability to find an edge to an unvisited node from this particular node increases.

As usual, it is easier to analyze the complete graph. The complete graph analysis happens to be the same for directed and undirected graphs; in both cases we can move from any node to any other node exactly once.

Lemma 5.1. *In a complete graph with n nodes, the expected number of edges scanned by walk-first search is less than* $(n-1)H_{n-1}$.

Proof: Let us assume that we have already visited i other nodes (apart from the current node), for some $0 \leq i \leq n-2$. We want to estimate how many edges we need to traverse until we visit the next unvisited node.

If we have never visited the current node before, all edges are equally likely to be found first, and we find an unvisited node with probability $1-i/(n-1)$. Otherwise, the edges that have been used in the past will not be reused. Since these edges all lead to visited nodes, the probability of finding an unvisited node on the next edge is more than $1-i/(n-1)$.

In both cases, the probability of finding an unvisited node is at least $1-i/(n-1)$, so that the expected number of edges needed is at most $1/(1-i/(n-1))$.

Adding these edges for all $n-1$ steps, we get that the number of edges to be scanned is at most:

$$\sum_{0\leq i<n-1} \frac{1}{1-i/(n-1)} = \sum_{0\leq i<n-1} \frac{n-1}{(n-1)-i} = \sum_{0<i\leq n-1} \frac{n-1}{i} = (n-1)H_{n-1}. \blacksquare$$

We compare walk-first search with *covering* a random graph, a random walk until all nodes are visited. The difference between the two is that covering may use the same path more than once, while walk-first search does not spend time on paths already explored. They have the similarity that after an edge is used, the target node of the edge becomes the new current node and the search continues on the new node.

In a directed graph, we can use a proof very similar to the one for the complete graph: any edge leads to any other node with the same probability, and we do not care that not all possible edges appear on every adjacency list; after using one edge, we continue our search at another node.

Lemma 5.2. *In a connected directed random graph $G_p^D(n)$, the number of edges used by walk-first search is at most $(n-1)H_{n-1}$.*

Proof: Let us assume that we have already visited i other nodes (apart from the current node), for some $0 \le i \le n-2$. the probability of finding an unvisited node on the next edge used is at least $1 - i/(n-1)$ until we find the node, so that the expected number of edges needed is at most $1/(1 - i/(n-1))$.

Adding these edges for all $n-1$ steps, we get that the number of edges to be scanned is at most:

$$\sum_{0 \le i < n-1} \frac{1}{1 - i/(n-1)} = \sum_{0 \le i < n-1} \frac{n-1}{(n-1)-i} = \sum_{0 < i \le n-1} \frac{n-1}{i} = (n-1)H_{n-1}. \blacksquare$$

The above proof cannot be applied to undirected graphs, because the probability to find an unvisited node decreases; we know of one edge that leads to a visited node and has not been used yet, as we have entered the current node once more than the number of times we have left it.

Lemma 5.3. *In a connected undirected random graph $G_p^U(n)$, the number of edges used by walk-first search is at most $(n-2+1/p)H_{n-1}$.*

Proof: Let us assume that we have already visited i other nodes (apart from the current node), for some $0 \leq i \leq n-2$. There are $(n-1)-i$ possible edges that lead to unvisited nodes and at most one edge that leads to a visited node (the reverse of the edge we used to enter the current node) and $n-2$ other possible nodes (if some of these edges have already been used, they are not possible any more). the probability of finding an unvisited node on the next edge used is at least $((n-1)-i)\,p/\,(1+(n-2)p)$ so that the expected number of edges needed is at most $(1+(n-2)p)\,/\,((n-1-i)p)$.

Adding these edges for all $n-1$ steps, we get that the number of edges to be scanned is at most:

$$\sum_{0 \leq i < n-1} \frac{1+(n-2)p}{(n-1-i)p} = (n-2+\frac{1}{p}) \sum_{0 \leq i < n-1} \frac{1}{n-1-i} = (n-2+\frac{1}{p})H_{n-1}. \quad \blacksquare$$

Figure 5.2 depicts the time needed by walk-first search to search a directed (left) and undirected (right) graph, for different values of n (on the x-axis) and p (on the y-axis). The value displayed is the number of edges used divided by $n \log n$. The values of p are large enough so that all graphs are normally connected.

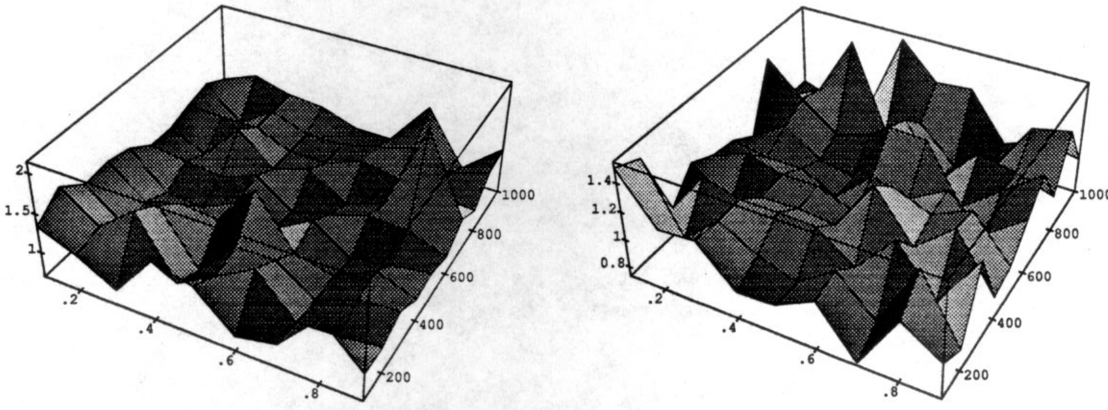

Figure 5.2: Edges used by walk-first search.

The number of edges needed in the undirected graphs seems smaller, but the variation of the values displayed is the same in both cases; the smallest and the largest of these values are within a factor of two. As n gets larger, the variation becomes smaller in the directed case.

Theorem 5.1. *Walk-first search is expected to use $\Theta(pn^2)$ edges in a disconnected random graph $G_p(n)$, and $O(n \log n)$ edges in a connected random graph $G_p(n)$.*

Proof: In a disconnected graph, Lemma 2.2 applies. For connected graphs, Lemma 5.2 and Lemma 5.3 conclude our proof. ∎

Covering random graphs needs $\Theta(n \log n)$ edges to run to completion [Al]. For connected graphs, we observe that walk-first search is expected to complete the search sooner than covering.

Let us assume that we have already visited i other nodes (apart from the current node), for some $0 \leq i \leq n-2$. If we have never visited the current node before, all edges are equally likely to be found first, and we find an unvisited node with probability $1 - i/(n-1)$.

If we have visited the current node j times before, we know that j of its edges have been used and have lead to visited nodes. Since in walk-first search we do not use these edges again, the probability of finding a new edge leading to an unvisited node is $1 - (i-j)/\left((n-1)-j\right)$, which is higher than $1 - i/(n-1)$, the corresponding probability in covering. On the other hand, covering will use any edge with the same probability. Any edge we have not used, has probability $1 - i/(n-1)$ of discovering an unvisited node, while the edges we have used, all lead to visited nodes. Thus the probability of finding an unvisited node is even less, at most $1 - (i-j)/\left((n-1)-j\right)(d-j)/d$, where d is the out-degree of the node, which is less than $1 - i/(n-1)$ even for $d = n-1$.

5.3 Size of the data structure needed by walk-first search

If we estimate the size of the data structure at the end of the search, we derive a lower bound on the size of the data structure needed, since at some point during the search we may have had more nodes in the data structure.

The upper bound on the size of the data structure is n, as in all searching methods. In this particular method, the size is very close to n, and in the complete graph it is always n.

Lemma 5.4. *In a complete graph with n nodes, the size of the data structure used by walk-first search is n.*

Proof: When the search is over, all nodes have been in the data structure at some point. In the complete graph there are no deletions from the data structure before we visit all nodes; a deletion occurs when all edges on the adjacency list of the deleted node have been used, which in a complete graph implies that all nodes have been visited and the search has terminated, a contradiction. Thus, at the end of the search all n nodes are in the data structure. ■

A small modification to Program 5.1 changes the algorithm to to check for search termination before inserting the last node into the data structure; as the last node visited is never needed in the data structure. The difference is always this one node only and does not affect the asymptotic behavior of the algorithm.

Theorem 5.2. *In a random graph $G_p(n)$ with $p > (1+\epsilon)/(n-1)$ for any constant $\epsilon > 0$, the expected size of the data structure needed by walk-first search is asymptotically n.*

Proof: The size of the data structure needed is expected to be an increasing function of p; for higher values of p, the expected degree of the nodes increases, and the probability that the adjacency list of a node is exhausted decreases.

We examine the cases of directed and undirected graphs separately and we start with directed graphs. It suffices to prove the theorem for $p = \Theta(1/n)$. The probability that a node is deleted from the data structure is q^{n-1-j}, where j is the number of edges used from the adjacency list. Since j is less than the maximum degree, this probability is at most a constant $c < 1$. The probability that more than a constant number of nodes, $f(n)$, for $f(n)$ any increasing function of n, are deleted from the data structure is $c^{f(n)}$, which becomes asymptotically 0 with probability one.

In undirected graphs, as mentioned before, in the properties of the walk-first search, only one node can be deleted from the data structure at any time. We expect the size to converge to n faster compared to the directed graph case, since only one node can be removed from the data structure, even if the probability that this node is the current node is higher than the average node. We can only perform a deletion when the adjacency list of the node is exhausted and the node is the starting node. ■

Intuitively, is that the size of the data structure in walk-first search is expected to be at least as big as the size of the data structure in depth-first search, which is the depth of the depth-first search. A deletion in the walk-first search data structure is less likely than a backtrack in depth-first search, because in walk-first search we use all adjacency lists uniformly; in depth-first search we use some of the adjacency lists for long periods of time, and so it is easier to exhaust these adjacency lists. Thus, we expect the size of the data structure to be n for a range of values for p larger than depth-first search.

Experiments running walk-first search on random directed and undirected graphs for values of n from 100 to 1000 in steps of 100 nodes and for values of p from 0.1 to 0.9 in steps of 0.1, used a data structure of size n in all cases, except in the case of the directed graph with $n = 100$ nodes and probability $p = 0.1$, where a data structure of size 99 was used!

Chapter 6

Searching time

When we talk about analysis of an algorithm, the first thing that comes to mind is the running time of the algorithm. There are other quantities of interest, though, like the size of the data structures and the memory requirements, how many files we need to use, how many times we are going to read a file, how random our accesses to the cache, the memory or the disks are, or even how many of a set of expensive functions are performed (e.g. calls to other programs, communication with other computers, etc).

We are mostly interested in the running time of an algorithm because this affects more the interactive usage of a computer. While this is generally true, when a program uses big quantities of some resources, it may not find enough resources to be able to run, or it may have to use them in an inefficient way, and become much slower: it may thrash, temporary files may have to be used, etc. When other resources are in adequate supply, time becomes the primary concern.

In the searching algorithms examined so far, we have not always been able to find their running time on connected graphs. In some cases, a comparison of the time needed by two different searching methods would be helpful. We examine such issues in the present chapter.

Searching on dense connected graphs should not be very expensive, although the search can use all edges sometimes. Cover time, the time of a random walk until all nodes are visited, on any graph, should be longer than searching time, since searching does not reuse edges scanned. Starting from this intuitive argument, we derive an upper bound on the time required for searching a random graph, using any searching algorithm.

We also need a lower bound on the time required for searching a random graph. Intuitively, node-based search algorithms should be among the fastest ones, since they only use the adjacency lists of as few nodes as possible. The advantage of this approach is that when some edges of the adjacency list are already scanned, they lead to visited nodes, and since no duplicate edges are allowed, the rest of the edges can only lead to the rest of the nodes, where the unvisited nodes are in higher percentage. This means that the probability of finding a new node is increased.

6.1 The node degree lemma

In any searching method, using an edge from an adjacency list where some of the edges have already been used gives higher probability of discovering an unvisited node than using an edge from a previously untouched adjacency list, because we know that some edges that lead to already visited nodes can not possibly be followed in the first case.

For the same reason, the more edges used from an adjacency list the higher the probability of discovering an unvisited node.

We have to use the first entries of the adjacency lists before using the next, the better ones. Algorithms like walk-first search, use all adjacency lists and scan only the beginning of them. Such algorithms would benefit if, instead of scanning the first edges on a new list they scanned the next elements of an already partially scanned list. In contrast, algorithms like the node-based searching ones, try to make the most benefit from this remark. They do not leave unused edges at the last positions of an adjacency list and they always scan such edges before proceeding to the adjacency list of another node.

Lemma 6.1. *When searching a directed random graph $G_p^D(n)$, we have higher probability of finding a new unvisited node if we use an edge from the adjacency list of a node with more edges already used.*

Proof: Let us assume that we have already visited i other nodes for some i between 0 and $n-2$. If we have already used j edges from the adjacency list of the current node, the probability of finding a new unvisited node by following the next edge on the adjacency list is $1-(i-j)/\left((n-1)-j\right)=\left((n-1)-i\right)/\left((n-1)-j\right)$, which increases with j.

We can find the same result in another way, by estimating the expected number of visited nodes at any time. This number gives us a better indication of the progress made at each algorithmic step. In the beginning of the search, we have one visited node, the starting node, and no edges have been used from any adjacency list.

At any snapshot of the searching algorithm, we have used d_j edges from the adjacency list of node j, $0<j\leq n$, where $0\leq d_j<n$, and d_j can be greater than 0 only for visited nodes. The number of visited nodes does not depend on the order in which we used all these edges, only on the particular edges used.

Let us call a_j the number of nodes visited using edges from nodes 1 to j only. Then $a_1=1+d_1$, and $a_2=1+d_1+d_2-d_1d_2/(n-1)$, since each of the d_2 edges will find a visited node with probability $(a_1-1)/(n-1)$, and the general term is $a_j=a_{j-1}+d_j-(a_{j-1}-1)/(n-1)$.

Using all these edges, we find that the total number of visited nodes a_n telescopes to

$$1+\sum_{1\leq j\leq n}d_j-\sum_{1\leq i<j\leq n}\frac{d_id_j}{(n-1)}+\sum_{1\leq i<j<k\leq n}\frac{d_id_jd_k}{(n-1)^2}-\cdots+(-1)^{n-1}\prod_{1\leq j\leq n}d_j/(n-1)^{n-1}.$$

As expected, the formula is symmetrical for all d_j. Also, no d_j is included more than once in each term. Let us try to increase d_i and decrease d_j by 1, so that the same number of edges is still used in the graph. The sum of all terms that only include d_i or d_j will still be the same, because increases and decreases will cancel each other, and only the terms that contain the product d_id_j will affect the value of the sum. The new value of the product will be $(d_i+1)(d_j-1)=d_id_j-(d_i-d_j+1)$ which will be less than before if $d_i\geq d_j$. As the first term containing this product has a negative sign, and all other successive terms containing the same product are in decreasing order (because $d_k<n-1$ for all k) and of alternating signs, the sum will finally increase. ∎

Lemma 6.2. *The number of edges needed to search an undirected connected random graph $G_p^U(n)$ minus the number of edges needed to search a directed connected random graph $G_p^D(n)$, $T(G_p^U(n)) - T(G_p^D(n))$, is minimum for stack-based search.*

Proof: All node-based searching algorithms have the least number of edges of known presence, $\Theta(\log n/\log(1/q))$, according to Theorem 3.1; no searching algorithm can process fewer adjacency lists, so that no searching algorithm can have a lower overhead from known edges.

In stack-based search, we have the maximum number of edges of known absence, since no searching algorithm scans more than $\Theta(\log n/\log(1/q))$ nodes completely. Also, the current node does not have an edge to any of the nodes that were completely processed before this node was inserted into the data structure; thus, selecting the last node from the data structure (or any node visited from the same node), will certainly give us the node with the most known absent edges to visited nodes. ∎

Other node-based searching algorithms, only lack the knowledge of a few edges that stack-based search knows about, so they should usually be nearly as efficient as stack-based search. They have the same complexity, the difference is a few more edges.

Corollary 6.1. *In a connected random graph $G_p(n)$, no searching algorithm is faster on the average-case than a node-based searching algorithm. In other words, the searching time of the node-based searching algorithms in a connected random graph $G_p(n)$ is optimal.*

Proof: Using Lemma 6.1 we see that the number of edges that a node-based searching algorithm is expected to scan in directed graphs is optimal; if a non-node-based searching algorithm was the fastest searching method, scanning the unused edges at the end of the adjacency lists of the first nodes of the search instead of the last edges scanned, we would yield an even faster algorithm, a contradiction.

According to Lemma 6.2, this is even more obvious for undirected graphs. ∎

Other searching methods can be as fast as the node-based searching methods, if, for example, they use the same set of edges, but in a different order.

If we know from the beginning how many edges each node has, we can come up with a more efficient algorithm, by selecting the nodes in predefined order, sorted by their degree. In our model such information is not available, and we need effort proportional to the total number of edges in the graph to retrieve it.

Corollary 6.2. *Among all searching algorithms, the fastest algorithm on the average-case is the stack-based search for connected undirected random graphs $G_p^U(n)$ while all node-based searching algorithms are the fastest for connected directed random graphs $G_p^D(n)$.*

Proof: The result follows from Lemma 6.2, Corollary 6.1 and Corollary 3.1. ■

Corollary 6.3. *The expected time to search a connected, but not complete, random graph $G_p(n)$ is $\Omega(n \log n)$.*

Proof: The result follows from Corollary 6.3, and the time bound of $\Theta(n \log n)$ for node-based searching algorithms applied on random graphs by Theorem 3.3. This result does not apply in the case of $p = 1$, the complete graph, as the bound for the time needed by node-based search for that case is lower. Complete graphs have to be examined separately for each searching method, in order to find a tighter lower bound for them. ■

Corollary 6.4. *Among all searching algorithms, no algorithm is slower on the average than walk-first search, when searching a connected directed random graph $G_p^D(n)$.*

Proof: When we use walk-first search on a directed graph, all adjacency lists get consumed uniformly. By Lemma 6.1, if we increase the number of edges used on some adjacency lists and decrease the number of edges used on other lists by the same amount, we find a faster algorithm. All other searching methods do not use all adjacency lists uniformly, so they are faster than walk-first search. ■

6.2 Comparisons of searching algorithms

Covering a random graph, i.e., a random walk until all nodes are visited [BK], is a good place to start from. It is memoryless, and needs to maintain neither the edges used nor the order of the edges on adjacency lists. At each step it picks a random edge incident on the current node.

Lemma 6.3. *The expected cover time on a connected random graph $G_p(n)$ is $\Theta(n \log n)$.*

Proof: In random walk, we choose a random edge from the current node, and select the target node as our new current node. The advantage in the analysis is that this model is memoryless; whenever we reach a node we perform exactly the same action no matter which other nodes have been visited. All nodes are identical in that sense. When all nodes have been visited (covered), we stop.

When i of the other $n-1$ nodes are visited (not counting the current node), $0 \leq i \leq n-2$ then the probability that an unvisited node will be found is $p_i = (n-1-i)/(n-1)$. We have an infinite number of tries to reach a new node, and all of them will succeed with the same probability p_i mentioned above. This means that the expected time for reaching a new unvisited node is $1 + (1-p_i) + (1-p_i)^2 + (1-p_i)^3 + ...$, or $1/p_i = (n-1)/(n-1-i)$. Adding over all nodes, we get that the total time is

$$\sum_{0 \leq i \leq n-2} \frac{n-1}{n-1-i} = (n-1) \sum_{0 \leq i \leq n-2} \frac{1}{n-1-i} = (n-1) \sum_{1 \leq i \leq n-1} \frac{1}{i} = (n-1)H_{n-1}.$$

Similarly for undirected random graphs, we find

$$\sum_{0 \leq i \leq n-2} \frac{(n-2)p+1}{(n-1-i)p} = (n-2+\frac{1}{p})H_{n-1}.$$

These bounds asymptotically become $\Theta(n \log n)$. ∎

Lemma 6.3 is a known result [Al]. We provided an alternate proof, similar to the other ones given before.

Lemma 6.4. *Covering a connected random graph $G_p(n)$, is slower on the average-case than searching the same graph using any searching algorithm.*

Proof: When i of the other $n-1$ nodes are visited (not counting the current node), $0 \leq i \leq n-2$ the probability that an unvisited node is found is $(n-1-i)/(n-1)$ in a random walk, and $(n-1-i)/(n-1-j)$ in searching, where j is the number of edges of the current node that have already been used. As $(n-1-i)/(n-1-j) \geq (n-1-i)/(n-1)$, searching is always expected to complete sooner than covering. ∎

We conclude with a very important result.

Theorem 6.1. *The expected time to search a connected, but not complete, random graph $G_p(n)$ is $\Theta(n \log n)$.*

Proof: Using Corollary 6.3 and Lemma 6.4 and the corresponding bound, the result follows directly. ∎

Lemma 6.5. *The size of the data structure is minimum for the node-based searching algorithms when searching a random graph $G_p(n)$.*

Proof: In node-based search methods the edge used next is always the one that has the highest probability of causing a deletion from the data structure: the edge is taken from the adjacency list with the most edges in it so far (the only one that may have more than zero edges used). The edges that have already been used cannot reappear, so there are fewer possible edges, and the deletion probability is maximized. ∎

Theorem 6.2. *The expected size of the data structure when searching a random graph $G_p(n)$ is $\Theta(n)$.*

Proof: The lower limit is a consequence of Theorem 3.2 and Lemma 6.5, and the upper limit comes from the fact that we have at most one entry per node in the data structure at any instance. ∎

For each node we may keep more than one piece of information in the data structure, but of at most constant size, so that we consider it as one entry. We expect the size of the data structure to always be linear, since we have to keep information about all nodes visited but not yet fully processed, and each node processed visits more nodes. During the first steps, more nodes are visited than completely processed.

Theorem 6.3. *The expected value of the depth of the depth-first search on a random graph $G_p(n)$ is $\Theta(n)$ when $p > (1+\epsilon)/(n-1)$ for any constant $\epsilon > 0$, and asymptotically n when $p > \Theta(1/n)$.*

Proof: The depth of the depth-first search is the maximum size of its data structure needed at any time during the search, which is $\Theta(n)$ when $p > (1+\epsilon)/(n-1)$, by Theorem 6.5. We do not get information about the coefficient of the term n from Theorem 6.5.

When $p > \Theta(1/n)$, the depth is asymptotically n by Lemma 4.5.

Since we estimated the lower bound of the real depth of the depth-first search, we have proved the theorem. ■

Theorem 6.4. *The expected time to search a connected random graph $G_p(n)$ using any node-based searching algorithm decreases as the graph gets denser.*

Proof: A proof can be given following Theorem 3.3. In connected graphs, we have $\Omega(n \log n)$ edges. The number of edges scanned by node-based searching algorithms is between $(1-\lambda)p(n-1)$ and $(1+\lambda)p(n-1)$ times the nodes scanned, since, on the average, each node is incident upon between $(1-\lambda)p(n-1)$ and $(1+\lambda)p(n-1)$ other nodes, for any constant λ, according to Lemma 2.3.

Each node taken out of the data structure tests the existence of, on the average, at least $(1-\lambda)p(n-1)$ edges and we continue scanning nodes from the data structure until all nodes are visited. Every time a new node is scanned, a fraction less than $1-(1-\lambda)p$, of the previously unvisited nodes remain unvisited. After k steps, the unvisited nodes are at most $(n-1)(1-(1-\lambda)p)^k$, and we want to have at most a constant number of unvisited nodes, so we can require that $(n-1)(1-(1-\lambda)p)^k = c$ for c a constant, like $1/2$, or even a decreasing function of n, like $1/\log n$.

We can derive that $k = (\log c - \log n)/\log(1-(1-\lambda)p)$, whenever $(1-(1-\lambda)p)$, and therefore q, is not zero (the complete graph case) and is not one (the graph with no edges). Then we will finally scan between $k(1-\lambda)p(n-1)$ and $k(1+\lambda)p(n-1)$ edges.

Taking derivatives we see that the number of edges used decreases when p increases. ■

A different proof can be constructed using Lemma 6.1. As the graph gets denser, new edges are added at the end of the adjacency lists. If we still use the previous edges, we are going to visit the same nodes. If, instead, we replace the last of these edges with the newly added edges at the end of the adjacency lists scanned before, according to Lemma 6.1, we get a faster searching. The last construction is a node-based search in the new graph!

Conjecture 6.1. *The expected time to search a connected random graph $G_p(n)$ using depth-first search decreases as the graph gets denser.*

When the graph is "almost" complete, we can estimate the increase in the cost due to the addition of edges, and the decrease in the cost due to no longer needed backtrackings. But for sparser graphs, the win from eliminated backtrackings is not estimated easily. In disconnected graphs, the cost is proportional to the number of edges in the graph, and does not depend on the particular searching algorithm.

Corollary 6.5. *The expected time to search a random graph $G_p(n)$ is smaller when using any node-based searching algorithm than using depth-first search, which in turn is smaller than using walk-first search.*

Proof: We use Corollary 6.1 and Corollary 6.4. ■

Chapter 7

Special cases

7.1 Adjacency list randomization and extreme input cases

A random graph $G_p(n)$ represents any n-node graph where all edges exist with the same probability p. However, the graphs that appear in applications are never really random; they usually have some specific properties that make the study of the graph interesting. Consider, for example, the graph constructed from the map of an area, where nodes represent geographic locations, and edges represent roads connecting them. Roads do not all exist with the same probability. There are some crossings, where more roads lead to, and the graph is mostly planar. A graph of the airports as nodes and the airline connections as edges also has accumulation points, maybe more than what a real random graph would have. The airline graph is "less" planar than the road graph. A graph with nodes the telephones in a town and edges connecting all pairs of nodes that had a connection during the last day, seems more random, and not at all planar.

Random graph results can rarely be applied to solve a problem exactly, since in most cases the input is not modeled precisely by a random graph. However, averaging the inputs over all individual problems yields an input distribution that is closer to the random graph. Thus, we can use results on random graphs as indicative of general bounds.

When the input is not random, some algorithms may behave differently. A classical example is sorting, where quicksort on random data takes $\Theta(n \log n)$ time, sorted data it takes $\Theta(n^2)$ time. If our data are not random, and in many cases they may be sorted or close to sorted, we should revise our method.

One way to avoid such pitfalls is to examine special cases before running the algorithm, and see if they apply to our data; this takes time proportional to the number of cases, which can be quite large. Another possibility, which avoids modifying the initial algorithm, is to randomize the data, either before the algorithm is run, or during its execution.

The randomization technique can be applied to graphs. We are not allowed to change the input graph, but we can rearrange the edges on the adjacency lists, so that their order will be random, if it is not random at the beginning of the search.

For graph-searching, this affects some of the parameters we study, but not necessarily all: It does not really matter which unvisited node we find next; there is nothing special about any of the nodes, on the average, and we get the same behavior. When we estimate the size of the data structure, or the depth reached, in a new-node searching algorithm, the order of the edges is irrelevant, since edges to visited nodes are ignored and edges to unvisited nodes are all alike. But even in these algorithms, the running time of the search depends on the number of edges we examine and ignore before finding the desired edge.

If the edges have a special arrangement on the adjacency list, the performance can be much worse or better than expected. These cases do not contradict the average value calculated, since the probability of using such a graph can be so small that this case does not affect the final result significantly.

For most graph-searching methods we can construct a connected graph with $m > n$ edges where we need to search $O(m)$ edges to visit all nodes. We can also construct a graph where we only need $n - 1$ edges to visit all nodes.

There is no global "unlucky" or "lucky" graph, where all algorithms need to search many or few edges. Each algorithm has its own worst and best input graph, which is constructed by following the decisions of the algorithm on which node to select.

We can construct a bad graph (even if this graph represents a random graph with very small probability) by "hiding" a node from the searching algorithm. If many edges are missing from a graph, we can find a graph where most edges to a specific node are missing, and the existing ones are at a place that will be processed at a late step.

The construction of a bad complete graph is more interesting, since all edges do exist and searching can be very expensive. We try to hide all edges to some node in such places on the adjacency lists where they will be processed last.

For walk-first search, the adjacency list of node i in the bad complete graph contains the edges to nodes $i+1, i+2, \ldots, n-1, 1, \ldots, i-1, n$, so that the searching will go $n-2$ times through nodes 1 to $n-1$, and then reach node n.

For depth-first search, the adjacency list of node i in the bad complete graph contains the edges to nodes $1, 2, \ldots, i-1, i+1, \ldots, n$, so that searching node i will first visit all other visited nodes, $1, 2, \ldots, i-1$, before reaching the next unvisited node $i+1$.

For node-based search, all complete graphs can be searched using $n-1$ edges. Bad input with hidden nodes can only be constructed for incomplete graphs.

Similarly, the good graphs can be constructed by always placing an edge to an unvisited node in the next position examined by the algorithm.

If, for depth-first search and walk-first search, the first edge on the adjacency list of node i is the edge leading to node $i+1$, then we never need to search any but the first edge on the adjacency list of any node to find the next unvisited node, and we finally scan only $n-1$ edges in the entire graph, just as in the node-based search algorithms on any complete graph.

Following these guidelines, we can construct good and bad cases of incomplete graphs by rearranging the edges on their adjacency lists.

7.2 Multigraphs

In *multigraphs*, any two nodes can be connected with more than one edge. Some searching algorithms, like walk-first search, use duplicate edges to change the current node. All new-node searching algorithms examine and discard an extra edge and are not affected otherwise.

In order to perform calculations, we extend the random graph model to include multigraphs:

Definition 7.1. *A random multigraph $MG_p(n)$ with n nodes and probability p for each edge, is a multigraph where with probability p, there exists at least one edge connecting two nodes, and it is independent of the existence of all other edges in the multigraph. In a random multigraph $MG_p(n)$, q denotes the probability of the absence of any more edges between two nodes, so that $q = 1 - p$.*

Random multigraphs may be directed or undirected, just like graphs. When $p \neq 1$, an undirected graph with n nodes and m (independent) edges is in $MG_p^U(n)$ with probability $p^m q^{\binom{n}{2}}$, and a directed graph with n nodes and m edges is in $MG_p^D(n)$ with probability $p^m q^{n(n-1)}$. There are $\binom{\binom{n}{2}+m-1}{m}$ possible undirected multigraphs with m edges and $\binom{n(n-1)+m-1}{m}$ directed. The expected number of edges connecting two nodes is $\sum_{i>0} ip^i q = p/q$. When $p = 1$, there is an unlimited number of edges connecting any two nodes.

We assume the *random adjacency list representation of random multigraphs*, simply referred to as *random multigraphs* in this chapter, where the adjacency list of each node contains all incident edges in a random order, independent of the order at any other node. The adjacency matrix model is inadequate for representing a multigraph because it is not capable of keeping duplicate edge information.

In multigraphs, an edge may appear more than once on the adjacency list of each node, and any of these instances can be used to visit the node. During searching, we use the first of multiple edges connecting two nodes to visit a new node.

Under this model, with probability q there is no edge between two nodes, exactly as in random graphs, and with probability p we have at least one edge; in random graphs we had *exactly* one edge. Every edge is independent of any other edges, even edges connecting the same nodes. Given that we have at least one edge, the probability that there is no other edge connecting the same nodes is q, and the probability that we have at least one more edge, is p. Telescoping, we see that the probability that we have at least i edges between two nodes is p^i and the probability that we have exactly i edges between two nodes is qp^i. Notice that $\sum_{i>0} qp^i = p$.

These probabilities do not make this model complex. On the contrary, this model is usually easier to analyze than the random graph model, because the random multigraph model is memoryless; given that an edge exists, and that we are not going to use this edge again, the probability that at least one new edge exists connecting the same nodes is also p. We no longer have to remember the edges used in the past, because they will not be used again and do not affect the edges we may find subsequently.

We now study how the searching algorithms are affected by the multiple edges. Duplicated edges always lead to visited nodes, so all new-node searching algorithms are only affected in their running time. Since the quantities p and q have the same meaning in the two models (probability of two nodes being or not being connected) the results have the exact same formula.

For each multigraph the corresponding graph is the graph where all but the first occurrence of any edge have been removed. The nodes accessible on a multigraph and its corresponding graph are the same.

Under the multigraph model we estimate the time needed for searching using node-based search, depth-first search and walk-first search. Unlike graphs, in multigraphs, the number of edges in a graph can exceed $O(n^2)$, so it is even more important to bound the running time!

The complete multigraph, the multigraph with $p = 1$, has infinitely many edges on each adjacency list. Using node-based searching methods on such a graph is not meaningful, because all node-based searching methods first put all edges in the data structure, and then try to process them. This process will take for ever! Other algorithms, though, like depth-first search, that only access the next edge on the adjacency list and process it, will behave reasonably.

Theorem 7.1. *The expected number of edges examined in searching a random multigraph $MG_p(n)$ with $p < 1$ using any node-based searching procedure is $1/q$ times the number of edges examined by the same algorithm in the corresponding graph.*

Proof: Searching on the multigraph visits the same nodes and in the same order as on the corresponding graph. The extra time spent in searching the multigraph is due to the multiple edges.

For each edge scanned in the graph corresponding to the multigraph, we find exactly i more edges with probability qp^i, so the expected number of extra copies of the edge is $\sum_{i\geq 0} qip^i = p/q$ and the number of edges searched is $1 + p/q = 1/q$ times the number of edges searched in the corresponding graph.

We used the equality [GKP]

$$\sum_{1\leq j\leq n} x^{j-1}j = \frac{1-(n+1)x^n + nx^{n+1}}{(1-x)^2}. \quad \blacksquare$$

The node-based search algorithms may not be the most efficient algorithms on multigraphs, because they may process possibly a lot of duplicate edges at each node; other algorithms that use the first edges on the adjacency lists only, may find unvisited nodes faster, and in many cases will not need to use the rest of the edges, including the duplicated ones.

The number of edges used in a disconnected graph is still the same for all searching methods, because this is a property of the graphs, not of the random graphs. As implied by Theorem 7.1, this is the number of edges expected to be present in that graph, i.e., $1/q$ times what it would be before, or pn^2/q.

In depth-first search, we do not use adjacency lists up to exhaustion and we may never need to access most of the multiple edges, so we have to use a different technique. The analysis of the complete graph case illustrates our method:

Lemma 7.1. *In a complete multigraph depth-first search is expected to use* $(n-1)H_{n-1}$ *edges.*

Proof: Let us assume that we have already visited i other nodes, $0 \le i \le n-2$. The probability of finding an unvisited node on the next edge used is exactly $1 - i/(n-1)$, no matter how many times we try; thus, the expected number of edges needed until visiting the next node is $1/(1 - i/(n-1))$.

Adding these edges for all $n-1$ steps, we get that the expected total number of edges needed is:

$$\sum_{0 \le i < n-1} \frac{1}{1 - i/(n-1)} = \sum_{0 \le i < n-1} \frac{n-1}{(n-1) - i} = \sum_{0 < i \le n-1} \frac{n-1}{i} = (n-1)H_{n-1}. \quad \blacksquare$$

We observe that depth-first search in a complete multigraph is like covering a complete graph, without changing the current node; at any time we have equal probability to move to any other node. As the graph is symmetrical with respect to all nodes, it does not really matter that the current node may not change.

Theorem 7.2. *In a connected random multigraph* $MG_p(n)$ *depth-first search is expected to use* $(n-1)H_{n-1}$ *edges for directed multigraphs and at most* $(n-1)H_{n-1} + n - 1$ *edges for undirected multigraphs.*

Proof: We start with the directed multigraph case. Let us assume that we have already visited i other nodes, $0 \le i \le n-2$. The probability of finding an unvisited node on the next edge is exactly $1 - i/(n-1)$. No matter how many times we try, or even if we have to backtrack and continue the search for the next unvisited node from another node, this probability does not change; therefore, the expected number of edges needed until visiting the next node is $1/(1 - i/(n-1))$.

Adding these edges for all $n-1$ steps, we get that the expected total number of edges scanned is:

$$\sum_{0 \le i < n-1} \frac{1}{1 - i/(n-1)} = \sum_{0 \le i < n-1} \frac{n-1}{(n-1) - i} = \sum_{0 < i \le n-1} \frac{n-1}{i} = (n-1)H_{n-1}.$$

In the undirected case, Lemma 2.5 applies. which holds for multigraphs too. This time the extra cost can really be $n-1$ (while it was $q(n-1)$ for graphs) because the known existence of an edge between two nodes does not prohibit the existence of a second edge. ∎

A previous remark was that walk-first search on graphs was like a random walk; depth-first search in complete multigraphs also bears a similarity to the random walk. We observe that walk-first search in a complete multigraph is the same problem as covering: at every step we follow an edge to any node, with the same probability, until all nodes are visited!

The following proof for walk-first search is similar to the proof for depth-first search for the case of the directed multigraph, since the only difference between the two methods is that walk-first search changes the current node on each step; however, continuing the search from another node does makes no difference in a memoryless graph.

Theorem 7.3. *In a connected random multigraph $MG_p(n)$ walk-first search is expected to use $(n-1)H_{n-1}$ edges for directed multigraphs and $(n-2+1/p)H_{n-1}$ edges for undirected multigraphs.*

Proof: We start with the directed multigraph case. Let us assume that we have already visited i other nodes (apart from the current node), for some $0 \leq i \leq n-2$. The probability of finding an unvisited node on the next edge used is exactly $1 - i/(n-1)$, no matter how many times we try, and even if we have to delete the current node from the data structure. Therefore, the expected number of edges needed until visiting the next node is $1/(1-i/(n-1))$.

Adding these edges for all $n-1$ steps, we get that the expected total number of edges scanned is:

$$\sum_{0 \leq i < n-1} \frac{1}{1-i/(n-1)} = \sum_{0 \leq i < n-1} \frac{n-1}{(n-1)-i} = \sum_{0 < i \leq n-1} \frac{n-1}{i} = (n-1)H_{n-1}.$$

In the undirected case, let us assume that we have already visited i other nodes (apart from the current node), $0 \leq i \leq n-2$. There are $(n-1)-i$ possible edges that lead to unvisited nodes, one edge that leads to a visited node (the reverse of the edge we used to enter the current node) and $n-2$ other possible edges to visited nodes. The probability of finding an unvisited node on the next edge used is $((n-1)-i)\,p/\,(1+(n-2)p)$, so that the expected number of edges needed is $(1+(n-2)p)\,/\,((n-1-1)p)$.

Adding these edges for all $n-1$ steps, we get that the total number of edges to be scanned is:

$$\sum_{0\leq i<n-1} \frac{1+(n-2)p}{(n-1-i)p} = (n-2+\frac{1}{p}) \sum_{0\leq i<n-1} \frac{1}{n-1-i} = (n-2+\frac{1}{p})H_{n-1}. \ \blacksquare$$

7.3 Self loops

We can extend both the random graph and the random multigraph model to include self loops, which are edges from one node to itself.

Self loops do not change any property of the searching algorithms, except the running time, because any algorithm that finds such an edge ignores it. In multigraphs, the increase in the number of edges used can be more than n, while in graphs it cannot, since we have at most n such edges.

A self loop edge has the same probability p of existence as any other edge in the graph. In every n edges used, asymptotically one of them will be a self loop edge and $n-1$ of them will not. This means that we now need n edges whenever we needed $n-1$ edges to visit the same nodes before, and all formulas for the searching time with self loops allowed can be derived from the formulas for the searching time without self loops by substituting $n+1$ for all occurrences of n.

7.4 Forests

When we are searching a disconnected graph, we stop the search when all accessible edges are exhausted, without accessing all nodes.

Sometimes it is not desirable to stop the search without accessing all nodes, even if some nodes are not reachable from the starting node. Instead, whenever the search stops, we pick one of the unvisited nodes as a new starting node, and continue the search from that node. When no unvisited nodes are left, the search terminates.

The extra process of locating an unvisited node each time we have used all edges in the current graph component requires a total of just n operations; we have to scan through the nodes once, and stop at the next unvisited ones only.

Since the graph has a giant component and some much smaller ones, of size $O(logn)$ each, the giant component dominates the searching process. We start from depth 0 each time, we empty the data structure, etc, so that the expected values for the depth and the size of the data structure are the same for the whole graph as for the giant component.

Running time is different. We are interested in the total running time spent on all components. In disconnected graphs, this is the total number of edges in the graph, which is $\Theta(pn^2)$.

7.5 Premature termination

When we are searching a graph, we sometimes want to stop the search when a certain condition is met, possibly before all nodes are visited. In this case we have premature termination of the search. We cannot make any general analysis, since the time at which the condition is met depends on the condition itself.

Usually the condition is to reach a specific target node or a node from a given set of target nodes. In the former case, as the target node can be any of the $n-1$ other nodes with the same probability, the expected time needed to reach it is the average value of the expected times needed to reach any of the $n-1$ other nodes.

Theorem 7.4. *The expected time to search a random graph $G_p(n)$ until a particular node is found is $\Theta(n)$.*

Proof: Let us number the nodes in the order they are visited. The starting node is node 1. The target node j can be any of the nodes 2 to n, with the same probability. In order to access node i, we need to use at least $i-1$ edges, to visit all previous nodes. Thus the expected number of edges needed for visiting node j is at least

$$\frac{1}{n-1}\sum_{2\leq j\leq n}(j-1) = \frac{1}{n-1}\frac{(n-1)(n-2)}{2} = \frac{n-2}{2}$$

which is $\Omega(n)$.

Since any searching algorithm is faster than random walk, we get an upper bound on the searching time by using a proof similar to that in random walk. We assume that the target node is accessible from the starting node. When i of the other n nodes are visited (counting the current node), $1 \leq i \leq n-1$, then the probability that an unvisited node is found is at least $(n-i)/(n-1)$ until we finally find one. Thus, the expected number of edges needed until finding the next unvisited node is at most $(n-1)/(n-i)$, and the total number of edges until node j is found is $\sum_{1\leq i<j}(n-1)/(n-i)$. Averaging over all j,

$$\frac{1}{n-1}\sum_{2\leq j\leq n}\sum_{1\leq i<j}\frac{n-1}{n-i} = \sum_{2\leq j\leq n}\sum_{1\leq i<j}\frac{1}{n-i} = \sum_{1\leq i<n}\sum_{i<j\leq n}\frac{1}{n-i} = \sum_{1\leq i<n}(n-i)\frac{1}{n-i} = n-1$$

which is $O(n)$.

If the graph is disconnected, we may not be able to reach the target node. The lower bound is still the same, since the search is held in the giant component with probability at least a constant. The worst case for the upper bound is when we start searching in the giant component and we have to examine all of its edges because the target node is inaccessible from the starting node. This happens with small probability $(1 - t(pn))t(pn)$, and the total number of edges examined is at most pn^2, where $t(c)$ is given by Equation 2.1. The contribution of this case to the expected number of edges examined is $(1 - t(pn))t(pn)pn^2$; the expression $t(pn)pn$ is a decreasing function of pn and its maximum value when $p = \Theta(1/n)$ is a constant. A smaller number of edges examined is expected when the search starts at a smaller component and the target node is still inaccessible.

Thus, the number of edges scanned is $O(n)$ no matter whether the starting and target nodes are on the same or on different graph components. ■

We expect the size of the data structure to be linear, since we expect to visit a linear number of nodes (half on the average), and most deletions from the data structure are performed during the last steps, when the data structure contains more nodes and the unvisited nodes are fewer.

Theorem 7.5. *The expected size of the data structure needed to search a random graph $G_p(n)$ with $p > (1+\epsilon)/(n-1)$ for any constant $\epsilon > 0$, until a particular node is found, is $\Theta(n)$.*

Proof: Let us number the nodes in the order they are visited. The starting node is node 1. The target node j can be any of the nodes 2 to n, with the same probability. We study the search until a constant fraction α of the nodes remains unvisited, i.e., until node $(1-\alpha)(n-1)+1$ is found, for some constant α. We use the value of λ as defined in Lemma 2.3, on the subgraph with the current node and the (at least $\alpha(n-1)$) unvisited nodes, which is a random graph. Every node inserts into the data structure at least $p(1-\lambda)\alpha(n-1)$ nodes, on the average. This means that the $j-1 \leq (1-\alpha)(n-1)$ nodes visited by this process, need at most $(j-1)/\left(p(1-\lambda)\alpha(n-1)\right)$ steps which is also the number of nodes that will be out of the data structure at that time, because their adjacency lists will have been exhausted and all their neighbors will have already been in the data structure. The data structure at that time has at least $(j-1) - (j-1)/\left(p(1-\lambda)\alpha(n-1)\right)$ nodes. Averaging over all $j \leq (1-\alpha)(n-1)+1$ we find that the expected size of the data structure is at least:

$$\begin{aligned}
&\frac{1}{(1-\alpha)(n-1)} \sum_{2\leq j\leq(1-\alpha)(n-1)+1} \left((j-1) - \frac{j-1}{p(1-\lambda)\alpha(n-1)}\right) \\
=&\frac{(1-\alpha)(n-1)+1}{2}(1 - \frac{1}{p(1-\lambda)\alpha(n-1)}).
\end{aligned}$$

This is linear whenever $p(1-\lambda)\alpha(n-1)$ is at least a constant greater than 1. When $p > \Theta(1/n)$, any value for α satisfies this condition, while when $p = \Theta(1/n)$, we can always find an appropriate value for α because Lemma 2.3 is satisfied for any λ such that $(p(n-1))^{-1/2} < \lambda < 1$.

The upper bound on the size of the data structure is always n, and when the graph is not connected, with probability at least a constant we start searching on the giant component, thus the expected size of the data structure is $\Theta(n)$. ∎

Reaching a node from a given set of target nodes is expected to be faster than reaching a specific target node. It can be proven that it is still linear, by modifying the above proofs.

A similar analysis can be made for different kinds of conditions and for tighter bounds when using specific searching algorithms, but further analysis depends on the internals of the searching algorithm used and is out of the scope of this thesis.

7.6 End of search

In the programs implementing the algorithms mentioned above, one change is very common: the relocation of the termination condition checking.

In the original version, the termination condition is checked after a new node is visited. If this check is relocated, the algorithm still visits the same nodes and in the same order, but it may continue and use some more edges, even when no unvisited node remains.

Algorithms usually check for termination after many edges are examined. For example, in node-based searching methods this can be done after each adjacency list is fully examined, and in walk-first search and depth-first search when we delete an element from the data structure and backtrack.

The advantage is that the algorithm may seem simpler. The disadvantage is that we may have to examine more edges now, before realizing that the search is over. For node-based search and depth-first search, the extra edges scanned are at most n, since we only continue scanning the adjacency list of the current node, with no hope to find unvisited nodes any more, and the asymptotic behavior of the algorithm is the same.

In disconnected graphs, the check is done once for each accessible node anyway, no matter if it is performed when we first visit the node or after we process it entirely, because we are going to process all accessible nodes.

For node-based search on connected graphs, and nodes are marked in groups and the relocation seems reasonable. On the average $pn/2$ extra edges are examined, while the check is performed $\Theta(\log n/\log(1/q))$ times, instead of n, according to Theorem 3.1. When $p = \Theta(\log n/n)$, we check the condition a linear number of times, but as this cannot exceed n, the extra cost is at most $\log n/2$. Otherwise, the number of checks is always less than linear, and the extra cost is negative: The node-based searching algorithm becomes more efficient.

In some other cases the termination condition is completely omitted! For disconnected graphs we may get a slight improvement in the running time, as we avoid checking a condition that can never be met. In dense connected graphs the performance may be much worse than before, since the algorithm will now examine all edges in the graph, ignoring the fact that it has visited all nodes and the rest of the edges may be omitted.

Chapter 8

Experimental results

8.1 Purpose of empirical results

What is the purpose of experimental results ? Let us reverse the question. What is the purpose of theoretical results ? Just to estimate the experimental results, when we really need them. But then, why not use the experimental results directly when needed, instead of the theoretical results?

Theoretical results give a better answer. They predict what will happen, even for instances of greater size and range than the ones in the experiments. On the other hand, the experimental results may not be accurate. We are usually unable to run an algorithm with all its possible inputs, so that we could get exact results; we just run it with some input cases, and they may happen to be all good or bad, giving a misleading intuition.

In some cases, the input distribution can not be created easily, or we may only have inaccurate computational models for it. In cases of complex algorithms with user and other computer interaction, it may even be impossible to set up an experiment that will give us results close to reality.

So here we do the opposite, we use the experimental results to estimate the theoretical quantity we want to evaluate. In the ideal case both results agree, i.e., we have a tight theoretical estimate, which fully describes our model without any approximations, and the experimental results are always very close to that, although their variation, even for similar inputs, may be large, depending on the algorithm (and this complicates both our intuition and the analysis).

Even if this deal is not achieved, spending some computer cycles to understand an algorithm may save us many more computing cycles on more complex applications, where the algorithm is used repeatedly as part of other algorithms.

We run a lot of simulations, most of them before trying to estimate the desired quantities, so that we can get intuition on what we should look for, and also on invariants that we suspect may exist when running the algorithm. We mention some results in this chapter that are related to the theoretical estimates of previous chapters, and also the principles and methods used for conducting the experiments, since they can be used for better understanding of the results, as well as for conducting similar experiments. The results themselves can also be used as a starting point for similar experiments, or other theoretical studies.

8.2 Principles

The edges are stored in adjacency lists, one for each node. This way, both the storage and the total access time of all nodes is proportional to the number of edges in the graph. For searching, this format is the most convenient, since we only need to access once all edges leaving any node and can always find an unused edge leaving a specific node, and we never test for the existence of a specific edge, between two known nodes.

In order to trust an experiment, we have to perform it many times and for different inputs and input sizes. Although big input sizes are of interest, since they are the ones that take a lot of time when used, smaller input sizes are useful too. They indicate increases in the running time as a function of the input size, and since they are faster to run anyway, they do not add too much time to the total time of the experiments.

Another important guideline when measuring running time, is to never measure the time itself, even when comparing the running time of two instances. The computer load is the factor that most often affects the running time of a program, so that it can be different in different executions. Many other factors also affect the running time, such as the compiler ability to optimize certain pieces of code, the size of cache on the computer, the allocation of computing resources to programs, input/output time, etc. Furthermore, running time is not comparable across computers of different types or sizes, so all experiments should be run on the same computer.

On the other hand, inserting counting statements at specific positions in the algorithm code, can count any quantity needed, not only running time, with just a small constant factor overhead, which we believe it is worth.

Finally, the code has to be free of bugs!

8.3 Graph creation

In fast algorithms like the ones examined here, the most time consuming part in the simulations is the creation of the data, the graph. Let us see the techniques used for creating directed graphs first.

According to our model, each edge may exist with probability p, independently of the existence of any other edge. We can find the adjacency list of each node by using a random number for each possible edge. This means we need to check each edge separately, and this requires $\Theta(n^2)$ time, which is too much for big sparse graphs. The time of the creation of the graph cannot be less than the number of edges created, anyway, but we can try to find a better algorithm than the straightforward quadratic one.

The memory requirements are smaller for sparse graphs, and we are able to create graphs with more nodes than denser graphs. Since the straightforward approach of checking for the existence of any possible edge would need a lot of time to execute, we have to change our graph creation model. We tried to use another model for random graphs, which is equivalent to our model for big graphs. Under this model, we find the expected number of edges in the graph and then pick that many edges at random. If we have duplicate edges, we delete all but one copy and select some more random edges, until all edges are unique.

Although this process is not linear in the number of edges (in limited memory), the running time is acceptable for sparse graphs. A small modification can make it even faster: Using some extra memory, we can pick a few more edges than the ones we need, to compensate for the duplicate ones. We repeat the previous process of eliminating the duplicate edges. If we have more edges than desired at the end, we delete some of the selected edges at random.

This model takes running time close to the number of edges, but has the drawback that as it needs twice as much memory as the quadratic time algorithm during the edge creation phase, for storing both ends of each edge; we cannot put the edges directly inside the adjacency lists, because we do not know how much space should be allocated, before the edges have been created. As a result, an efficient implementation of our initial model will be able to run on graphs twice as big in size. Also, the graph model it represents is not exactly the one we use for our theoretical estimates, although it is asymptotically equivalent.

A third approach with running time not much higher than the number of edges, is to first find how many edges will be allocated to each node, by picking a random number with binomial distribution, and then select that many random distinct destination nodes for these edges. The random distinct node selection is easy, since we can always afford memory proportional to the number of nodes, and can be done in linear time. But the random number with binomial distribution is not trivial [De]; it needs a lot of time to iterate and uses slow mathematical functions [Kn80]; This time is the limiting factor for the size of the problem instances we can solve.

The bottleneck of the previous approach, the binomial distribution random number generator, can be circumvented by substituting a sum of many geometric distribution random numbers for it [Pre86]. This makes the execution time proportional to the random numbers that we produce as result, which is the number of edges in the resulting graph.

Furthermore, the random node selection can be eliminated; we use the geometric distribution for finding the next edge on the adjacency list of each node. Since the geometric distribution is calculated efficiently, the resulting algorithm is quite fast and the available memory becomes the limiting factor.

The above methods describe graph creation techniques for directed graphs. Undirected graphs are created with similar methods: We only decide about the existence of every edge once, generating edges from higher number nodes to lower number nodes (or the reverse).

Then, we add the existing edges to the adjacency list of both connected nodes. This way all edges still have the same probability of existence.

8.4 Algorithm implementation

The implementation of any of the above algorithms is the easiest part of the whole implementation. After all, if the algorithm is difficult to implement, nobody is really going to use it in one's application; one will replace it with an alternative algorithm (where available). Especially in a complex application, one wants to concentrate on the main algorithm and does not want to spend a lot of time on all of the algorithms that are part of it.

The only modification to the algorithms, was the inclusion of statements for counting the quantities needed, inside the algorithm steps.

8.5 Running equivalent algorithm variants

The size of the problems we are able to solve is limited by the available resources, like memory and running time. Different computers impose different limits on these resources. But even if a problem is too big to be solved by a computer today, we would like to estimate its running time, because the resource restrictions become weaker every day. And the bigger the program instance we get experimental results from, the better knowledge of the process we get. This is why we try to use some techniques to solve big instances, which we would not be able to solve otherwise because of the current limitations on our resources, and we mention the most important of these techniques below.

Normally, when we search a graph, the graph is given from the beginning, and the graph description has to be read in the order given. We usually put the entire graph in memory, and we can access the adjacency lists of the nodes, in any order, whenever we need them.

In our case, the graph is generated by the same program that is using it, with the property that it is a random graph. The program can create the adjacency lists in any order, and the most convenient order may be the order used! In most cases we can create the graph on demand, i.e., we create the adjacency list for each node only when we want to use it. In directed graphs, adjacency lists of inaccessible nodes will never be created, and the space occupied by adjacency lists that will not be used again, can be reused by the program.

In many cases we can avoid calculating the whole graph, by omitting information that does not affect the properties of the algorithm we are studying, or by using a variant of the original algorithm that will yield the same results. Unfortunately, different variants (if any exist) are usually necessary to estimate different properties of the same algorithm.

As an example, in depth-first search, when we try to estimate the depth we reach, we are not interested in edges that lead to visited nodes, and we do not even check for their existence. When we scan the adjacency list of a node, we only use the edges that lead to unvisited nodes from it, and we create these. However, the same model cannot be used to estimate the time needed for the search, since there we also spend time accessing edges that lead to visited nodes.

A special case arises when we examine the adjacency list of only one node at a time and we never refer to that list again; in this case, the storage for keeping edges is proportional to the number of nodes, and our simulations do not encounter any memory problems at all.

The directed node-based searching algorithms fall into this category, since for each node we add all of its unvisited neighbors to the data structure, and proceed to other nodes. A similar case arises in the estimation of the depth of the depth-first search until the first time we backtrack; we do not need the incompletely scanned adjacency lists of visited nodes, except that of the current node, since we stop when we backtrack for the first time and never try to use these adjacency lists again.

Furthermore, in the latter case, all visited nodes are viewed by our algorithm identically, as we do not keep any information to distinguish them. We could number the nodes in the order we visit them and only use a number, the counter of the visited nodes, instead of the *visited* array; any node with number less than the counter is visited, and all other nodes are not. For example, when we visit node i, according to our numbering scheme, we have visited nodes 1 through $i-1$, and edges to these nodes are ignored. Whenever we find an edge to an unvisited node, this node has a number $j > i$. But we do not really care which node we visit, since all of them are random, and in this experiment we do not even need to search the rest of the adjacency list after backtracking, so we can number this node $i+1$ and continue with our search.

8.6 Removing recursion from depth-first search

According to its definition, depth-first search is a recursive algorithm. We consider functionally equivalent searching algorithms that do not use recursion [Se]. By functionally equivalent we mean that they visit the same nodes and in the same order, but do not necessarily need the same number of operations.

The naive approach would be to implement recursion manually by maintaining the recursion stack explicitly: We could use arrays for storing the procedure arguments and local variables that we want to preserve, and use an index to indicate how far in "recursion" depth we currently are. Then we could replace every recursive call with instructions to save these variables into the arrays and increase the counter, and every return statement with instructions to decrease the counter and restore the variables, or to really return if the counter has reached zero. A slight modification of this approach uses storage on a per node rather than a per depth level basis, because each node can be the current node of only one recursion level. Program 4.1 illustrates this approach. This solution removes the recursion from the code, but not from the program logic.

A different approach results in an entirely different implementation. Whenever we visit a node, we first add all of its edges that lead to unvisited nodes to the data structure, in reverse order. The data structure is used as a stack, so we pop the last edge on the stack (which was the first edge from the current node to an unvisited node) and we follow this edge. When we backtrack, we pop a new edge from the stack. This time, we do not necessarily follow this edge. The node that this edge leads to, may have been visited by now, in which case we discard the edge and pop the next one. We produce an algorithm that is like the other searching algorithms.

The above approach is not practical, since it requires a data structure that can grow as big as the number of edges. All other searching methods do not need a data structure bigger than the number of nodes. We have to modify this algorithm to use a smaller data structure. We observe that whenever we use an edge to visit an unvisited node, all other edges on the stack that lead to the same node become obsolete and will be ignored when accessed. Furthermore, whenever we push an edge on the stack, any other edges already on the stack that lead to the same node become obsolete, since the current edge will always be popped before them; if the node it leads to is still unvisited, we will visit it. By the time we pop the other edges, they will lead to a visited node and we will ignore them.

We conclude that from all edges that lead to the same node we only need the one that was pushed most recently. Since all useful edges are at most as many as the nodes, the size of the stack is acceptable. In order to implement these deletions in constant time, so that the complexity of the algorithm will not increase, we need to be able to delete elements in arbitrary positions in our data structure. The easiest way to achieve this is to keep the data structure as a doubly linked list. We also need an array, indexed by the nodes, that gives us the position in the data structure of the edge that leads into this node; this can be updated in constant time, for each edge found, and provides us with constant time location of the element to be deleted in the data structure.

This algorithm requires the same amount of memory as depth-first search, multiplied by a constant factor. Its running time is proportional to the number of nodes and edges in the graph, since all edges are eventually inserted into the data structure, and we only do a constant amount of work for each node found. The recursive depth-first search is expected to be faster than this variant, since it does not always exhaust the adjacency lists of all nodes.

8.7 Marking nodes in depth-first search order

We want to use a node-based searching algorithm to mark the nodes in depth-first search order; the reason for that can be that node-based searching algorithms are generally easier to implement, especially in a programming language that does not permit recursion. The desirable algorithm cannot visit the nodes in depth-first search order, as the node-based searching algorithms do not permit partially scanned adjacency lists, but can mark the nodes in the same order as depth-first search.

Since this algorithm simulates depth-first search, it has to scan all adjacency lists, except the one of the last node. As a node-based searching algorithm, it has to scan fully all adjacency lists it ever uses. This means that the resulting algorithm will not be very efficient, because it is expected to use almost all edges.

In addition to the storage needed for a node-based search algorithm, we need two extra arrays, with one entry for each node. The first one is the *marked* array; $mark[v]$ is **true** if depth-first search would have visited node v. In this case, v is deleted from the data structure.

The second array is the *parent*. The value of $parent[v]$ is the node that would have visited node v during depth-first search. The algorithm does not use the value of this array, it is only set for user's reference.

The following program, displayed in Program 8.1, is what we are looking for, when the **foreach** statement produces the edges in reverse adjacency list order:

```
procedure newdfs($s$ : node)
begin
    data structure:= empty
    insert $s$ into the data structure
    $visited[s]$ := true
    while the data structure is not empty
    begin
        select and remove the top element $v$ from the data structure
        $mark[v]$ := true
        foreach $w$ such that there is an unused edge from $v$ to $w$
            if not $visited[w]$ then
            begin
                insert $w$ to the data structure
                $visited[w]$ := true
                $parent[w]$ := $v$
            end
            else if not $marked[w]$ then
                $parent[w]$ := $v$
    end
end
```

Program 8.1: The node-based searching depth-first search algorithm.

8.8 Other results

In this section we mention some of the empirical results derived from the simulations. Other results derived from the simulations were presented in previous chapters, and are not repeated here.

This list tries to be indicative rather than complete, showing what kind of results may be useful either directly in estimating quantities, or indirectly, helping the reader to understand better how the algorithm works and make more realistic predictions.

More results, but for different ranges of the graph parameters, appear in the appendix.

Every experiment used a random graph different from any other graph created in another experiment; a random seed was used for this purpose. Even when we use the same measured quantity for the same graph size, we use independent runs to create these data. We may use two different results from one experiment, however.

Unless it is mentioned explicitly, otherwise each point plotted or number written is based on one run only, and is not the average of many runs. Although the average value over a few independent runs is a better measure, independent runs were not usually necessary in the present case, as the variation of the results found was small enough and single values also illustrate the variation easily. Furthermore, since many experiments of slightly different sizes were run, a diversion on a specific value can be noticed easily.

Figure 8.1 depicts the number of accessible nodes in a graph. The x-axis indicates the value of $c = pn$.

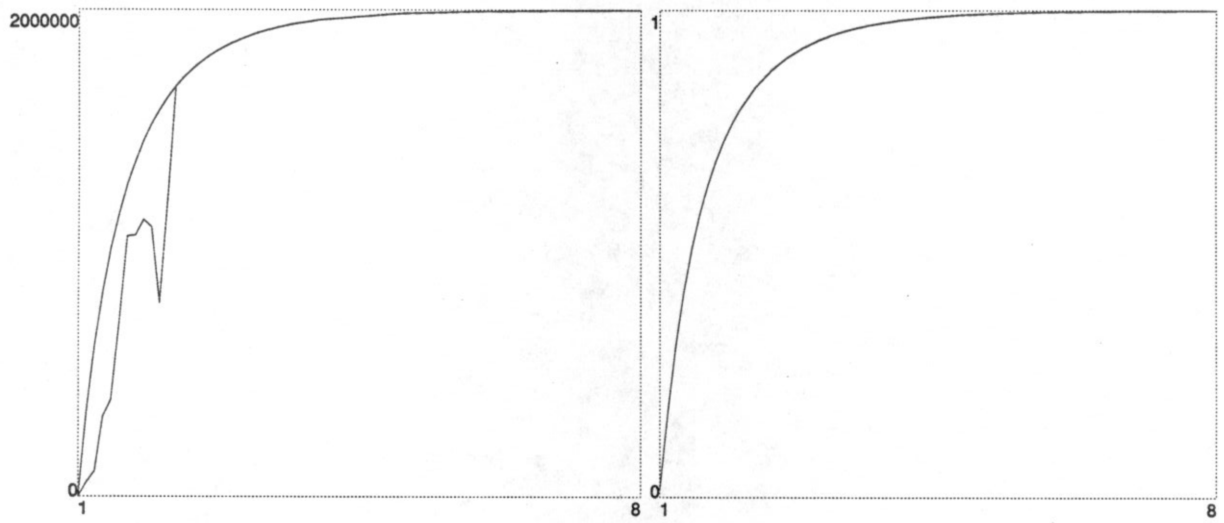

Figure 8.1 The number of accessible nodes and function $1 - t(c)$.

On the left, we see experimental values for the accessible number of nodes, from directed graphs of size $n = 2,000,000$ and different densities. Two curves are displayed on the left diagram. For each point plotted into the first curve, we used one execution on the giant component by restarting all executions that yield a very small (less than 10) number of accessible nodes. This way we compensate for the probability that the search starts in a smaller component.

On the second curve of the left diagram we averaged the number of accessible nodes found on all executions, no matter on which component the execution started. Thus, we found the *effective number of accessible nodes*, which includes the probability of searching small components. For the lower densities, $1 \leq pn < 2$, we used 18 executions, while for

$2 \leq pn < 3$, we used 4 executions, and 1 execution otherwise. As two of the executions with $pn = 2$ started in a small component, the corresponding value is unusually low.

On the right, we see the expected fraction of the nodes in the giant component, $1 - t(c)$, where the function $t(c)$ is given by Equation 2.1. This curve is identical to the first curve on the right diagram.

Figure 8.2 depicts the value of $t(1)$, as derived from Equation 2.1:

$$t(1) = \sum_{k \geq 1} \frac{k^{k-1}}{k!} e^{-k}.$$

The value of $t(1)$ is very close to 1, but $t(c)$ approaches 0 fast when $c > 1$. The x-axis indicates the number of iterations involved, k.

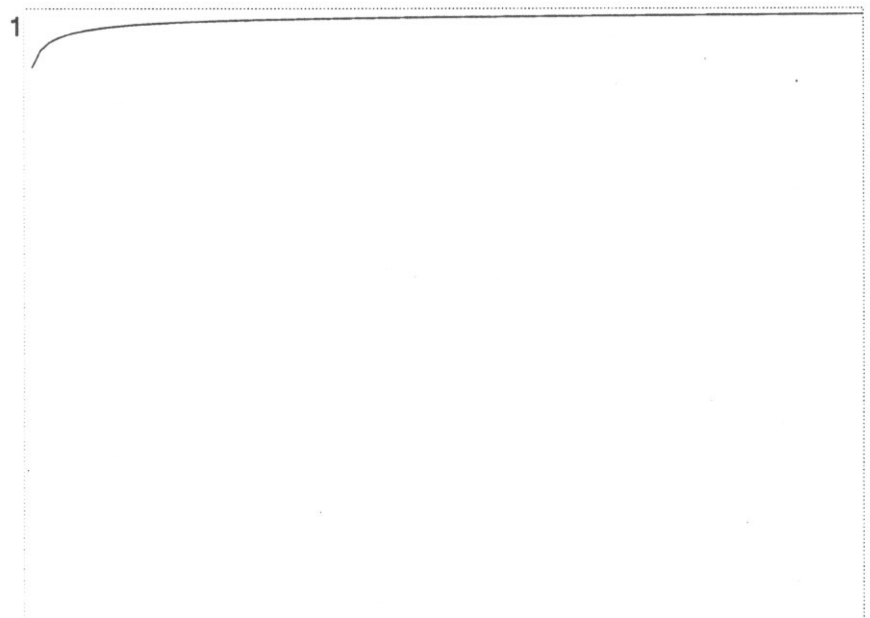

Figure 8.2 The value of $t(1)$.

We see that $t(1)$ seems to converge to 1.

Figure 8.3 depicts the number of nodes in each phase of breadth-first search on a graph. The depth from the starting node, i.e., the index number of the current phase, is associated with the x-axis. Directed graphs with $n = 100,000,000$ nodes are used and the three curves from left to right in each diagram represent different graph-searching experiments with densities of $p = 16/n$, $p = 8/n$, and $p = 4/n$ respectively.

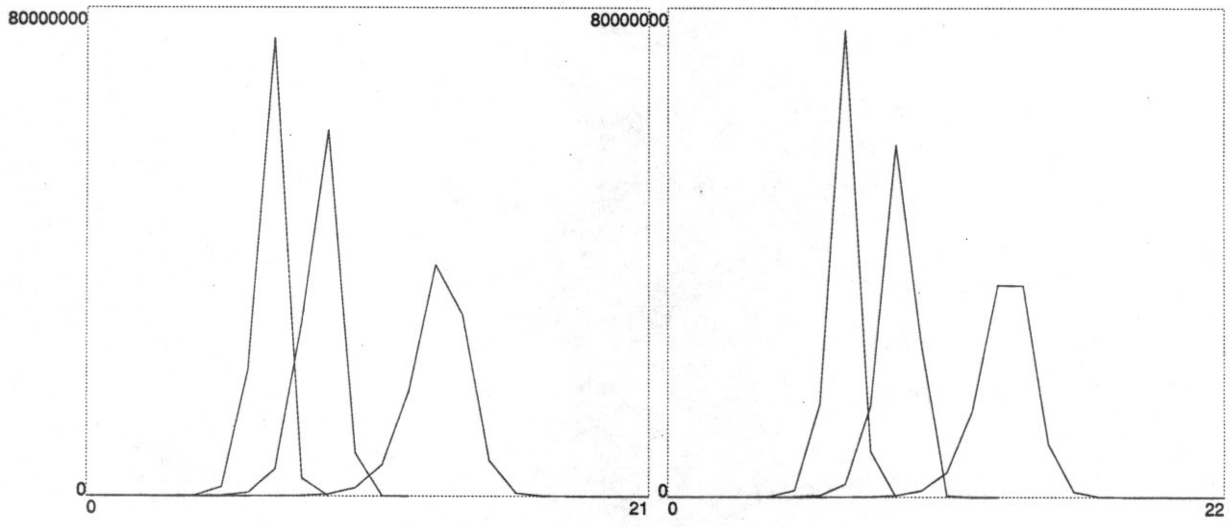

Figure 8.3 The phase sizes of breadth-first search.

We can see the fast growth of the number of nodes in each phase in disconnected large graphs, and that the three curves have a similar shape, with a different rate of ascent.

On the left, the simulation results are shown. On the right, the expected results using the recursive formula $x = u(1 - q^x)$ are displayed, where x is the number of nodes in a phase and u is the number of the remaining unvisited nodes; the probability of visiting a new node is $1 - q^x$.

Figure 8.4 depicts the number of nodes in each phase (on the left) and the total number of nodes in all phases (on the right) of breadth-first search on a graph. The depth from the starting node, i.e., the index number of the current phase, is associated with the x-axis. A directed graph with $n = 1,000,000$ nodes and probability $p = 4/n$ was used.

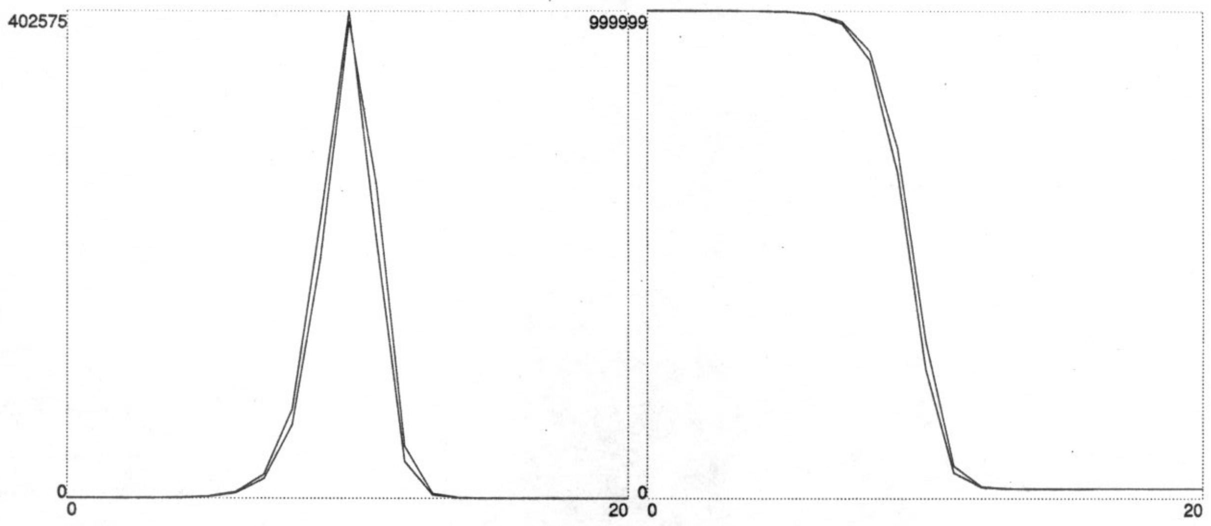

Figure 8.4 The phase size and accessible nodes in breadth-first search.

The two curves in each diagram represent the values found experimentally and the ones calculated using the recursive formula $x_i = u_{i-1}(1 - q^{x_{i-1}})$ where x_i is the number of nodes in the ith phase and u_i is the number of the remaining unvisited nodes in the ith phase; the probability that a particular new node will be visited in phase i is $1 - q^{x_{i-1}}$.

We can see how close these two curves are, for both the number of nodes in each phase (left) and all nodes in all phases (right).

The next figures explore the tree formed by depth-first search; we concentrate on the backtracks that occur when searching sparse graphs.

Figure 8.5 depicts for each node degree (on the x-axis) the number the nodes (on the y-axis) that backtracked without visiting any new node, using depth-first search on directed (left) and undirected (right) graphs. In the experiments, graphs with $n = 10,000$ nodes and probability $p = 6/n$ were used, and the sum over 10 searches is displayed.

The shape of the two curves is similar, although the scaling is different. Most of the nodes that backtrack without following any of their edges have degree close to 6, since most of the nodes in the graph have degree $pn = 6$. The curves are not symmetrical around the value 6, because a node with higher degree backtracks with smaller probability.

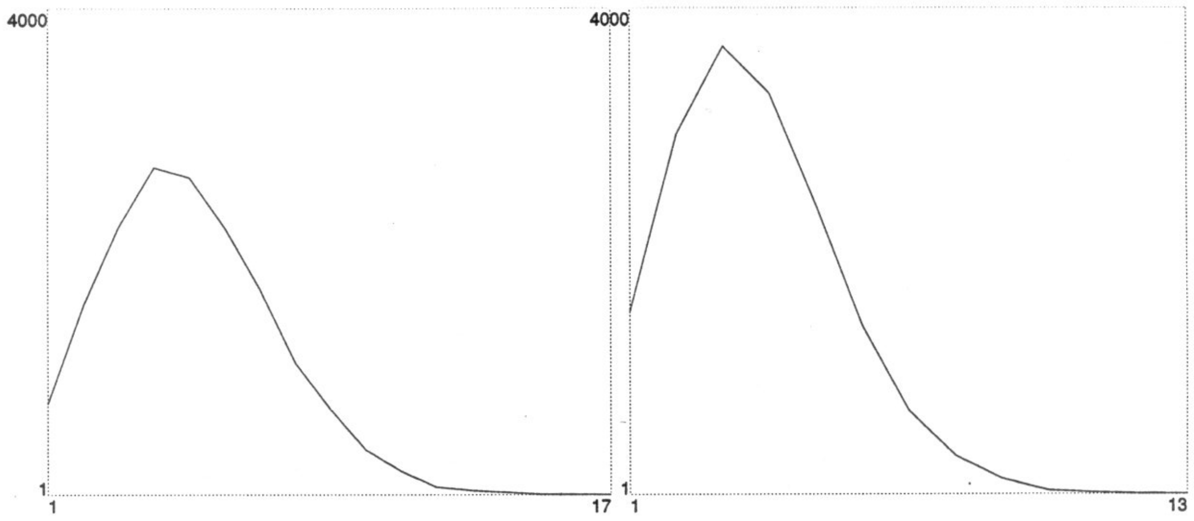

Figure 8.5 Nodes backtracked over node degree.

Figure 8.6 depicts the number of nodes that backtrack without following any of their edges (y-axis) as a function of the depth of the search at that moment (x-axis), using depth-first search on directed (left) and undirected (right) graphs. The values are the sum of these nodes over 10 graphs having $n = 10,000$ nodes and probability for each edge $p = 3/n$.

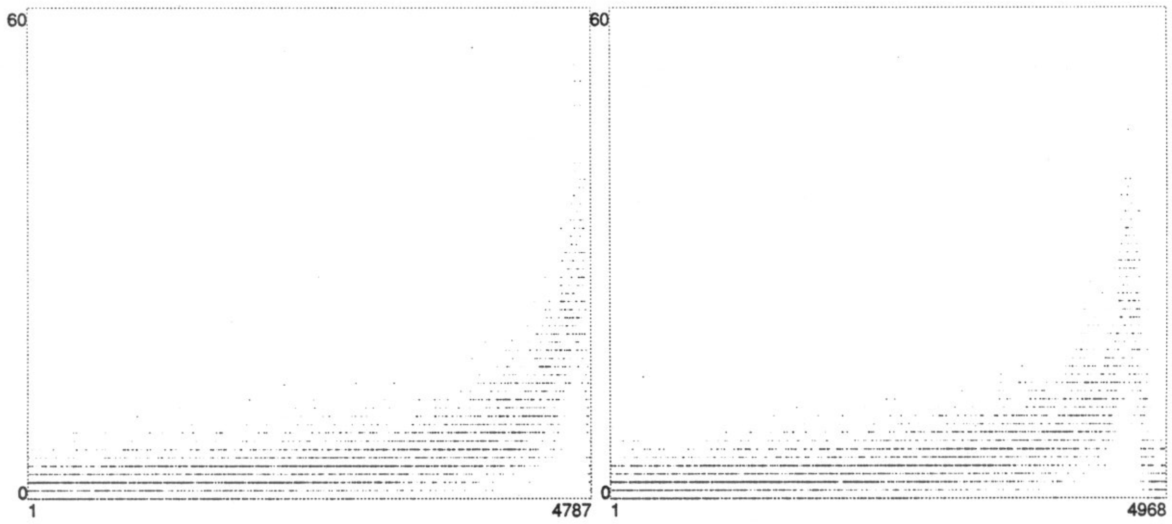

Figure 8.6 Nodes backtracking as a function of the depth.

One dot is drawn for each depth; the depth takes integer values only and thus the dots appear on distinct horizontal levels. The pictures for the directed and undirected case look indistinguishable.

We conclude that nodes backtrack on any depth in sparse graphs, while the backtracking frequency slightly increases with the depth. At about the maximum depth, the number of backtracks increases sharply.

The 10 different runs that gave us the above data did not reach the same depth, although they should have reached similar depth. Thus, the sharp parts of all curves added do not coincide and the displayed curve does not fall very sharply.

Figure 8.7 depicts the number of edges examined (on the y-axis) using depth-first search on directed (left) and undirected (right) graphs. Graphs of $n = 4,000$ nodes are used with graph densities ranging from $p = 1/n$ up to the complete graph. The x-axis contains the value of pn; one experiment was used for each integer value of pn.

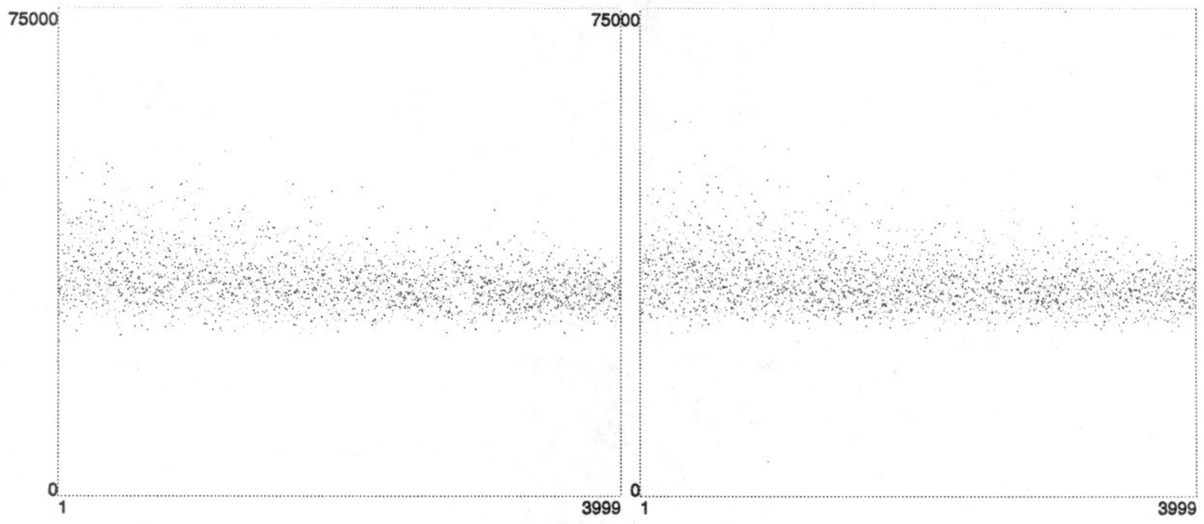

Figure 8.7 Edges scanned by depth-first search.

One dot is drawn for each execution; the variance of the number of edges examined seems to be big. Although all values have the same order of magnitude, they do not converge to a certain line. The number of edges examined seems to be dropping for higher graph densities, and the variance decreases.

Figure 8.8 depicts the depth reached (on the y-axis) using depth-first search on directed (left) and undirected (right) graphs. Graphs of $n = 4,000$ nodes are used with graph densities ranging from $p = 1/n$ up to the complete graph. The x-axis contains the value of pn; one experiment was used for each integer value of pn.

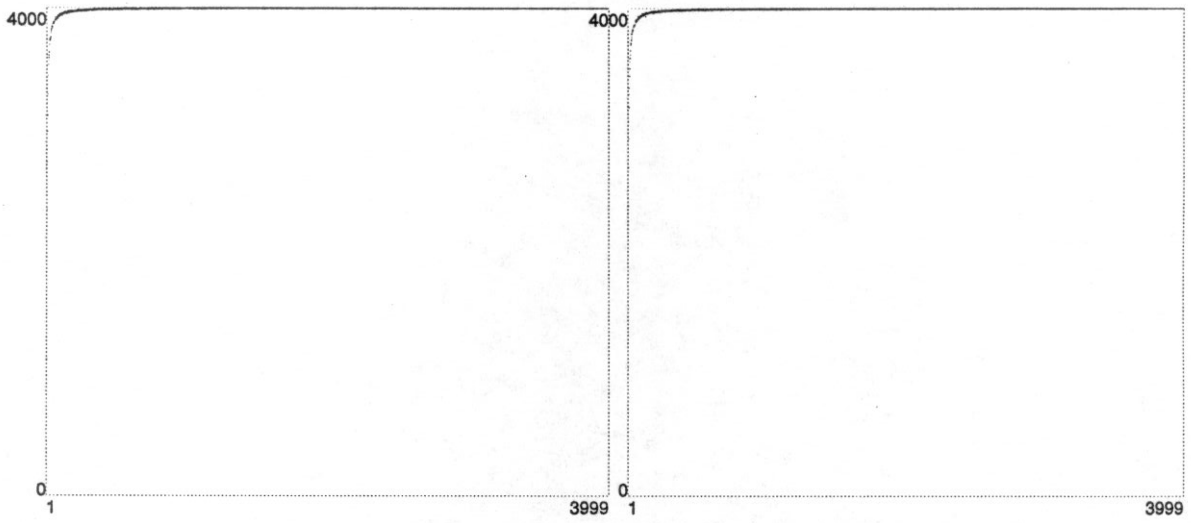

Figure 8.8 Depth reached in depth-first search.

One dot is drawn for each execution; the variance of the depth seems small and the depth seems to converge to n soon. The depth is expected to increase for higher graph densities, so it stays n as soon as it reaches this value. The increase rate can be seen better on pictures with only sparse graphs.

Figure 8.9 depicts the depth of the depth-first search on directed (left) and undirected (right) graphs, for different values of n (on the x-axis) and $r = pn$ (on the y-axis). The value displayed is the depth reached divided by n. The values of p are small enough so that all graphs are normally disconnected.

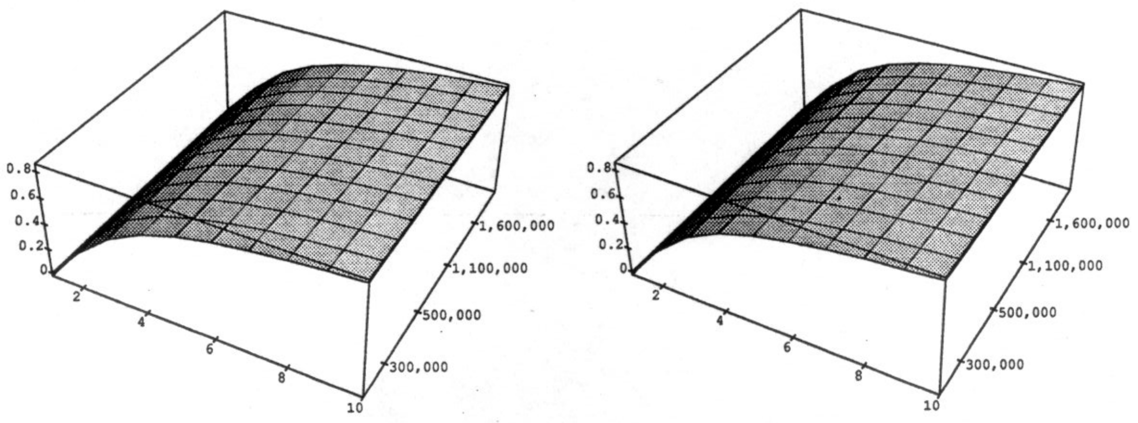

Figure 8.9 Depth reached by depth-first search.

As expected, the two pictures are similar in shape. Also the curves seem to be independent of n, close to the curve of the accessible nodes, and to have very small variance. Compare with Figure 4.3 and Figure 8.8, for denser ranges of graphs.

Figure 8.10 depicts the depth of the depth-first search on directed (left) and undirected (right) graphs, for different values of n (on the x-axis) and $r = pn$ (on the y-axis). The value displayed is the depth reached divided by n. The ranges of graphs from disconnected to connected are covered.

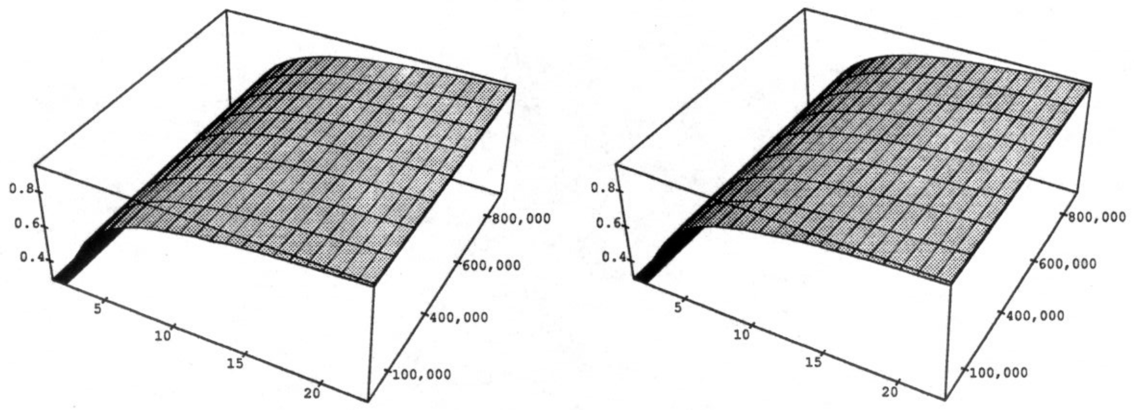

Figure 8.10 Depth reached by depth-first search.

Compare with Figure 4.3, Figure 8.8 and Figure 8.9. The remarks for those figures apply here, too.

Figure 8.11 depicts the number of edges examined by depth-first search on directed (left) and undirected (right) graphs, for different values of n (on the x-axis) and $r = pn$ (on the y-axis). The ranges of graphs from disconnected to connected are covered.

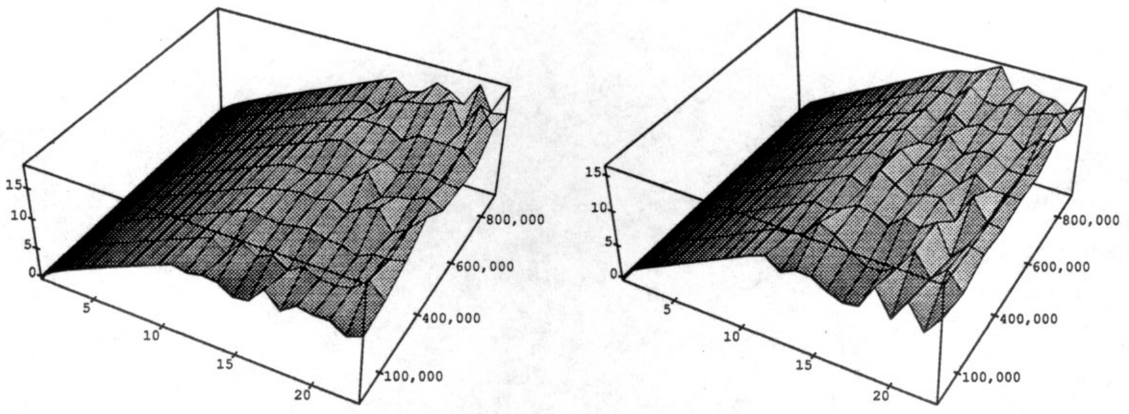

Figure 8.11 Edges examined by depth-first search.

Compare with Figure 2.4 for sparser graphs, and Figure 8.7, for denser graphs. The remarks for those figures apply here, too.

Figure 8.12 depicts the number of nodes visited by depth-first search on directed (left) and undirected (right) graphs, for different values of n (on the x-axis) and $r = pn$ (on the y-axis). The value displayed is the number of nodes visited divided by n. The ranges of graphs from disconnected to connected are covered.

Compare with Figure 2.3, for sparser graphs; the same remarks apply here.

Figure 8.13 depicts the depth of the depth-first search on directed (left) and undirected (right) graphs, for different values of $r = pn$ (on the x-axis). Three curves are displayed in each diagram, for $n = 50,000$, $n = 100,000$, and $n = 200,000$ nodes respectively. The ranges of graphs from disconnected to connected are covered.

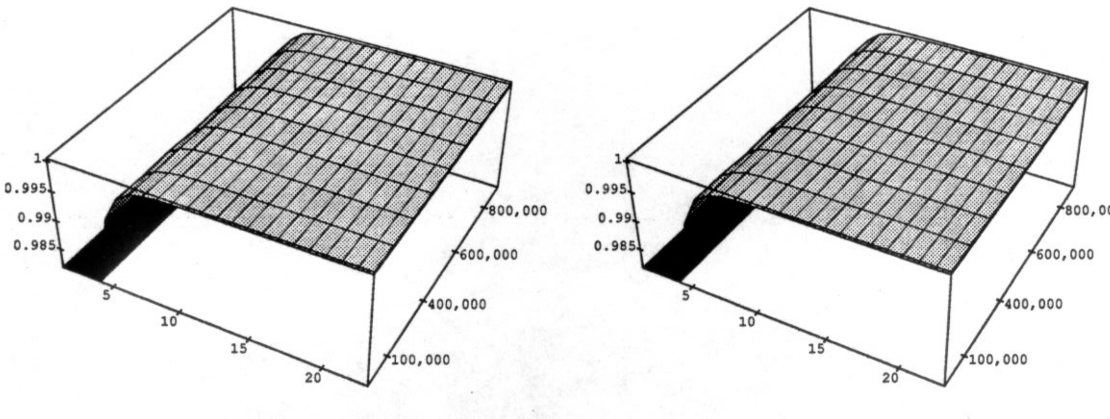

Figure 8.12 Nodes visited by depth-first search.

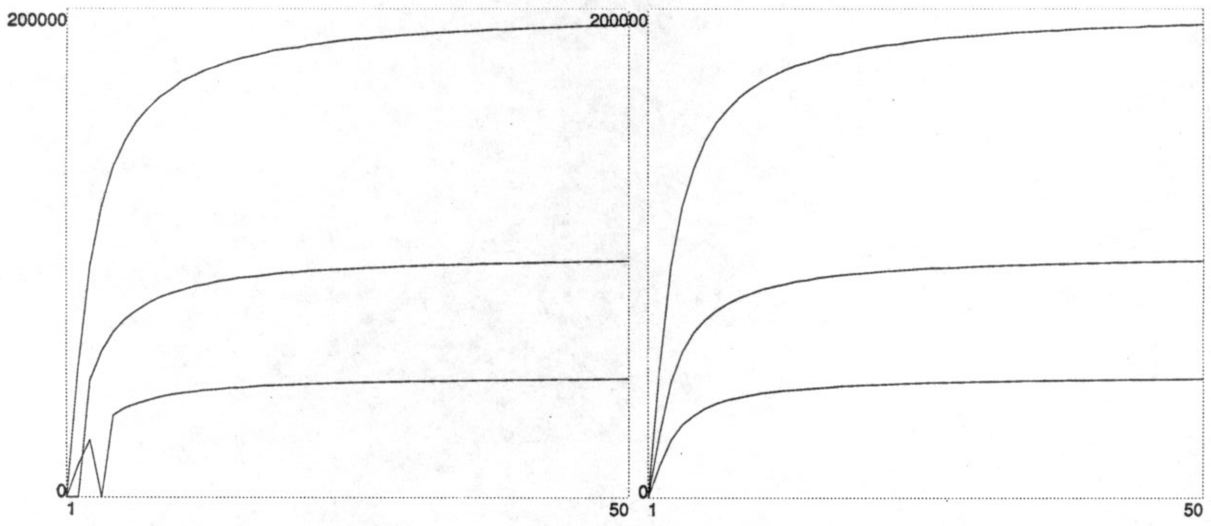

Figure 8.13 Depth reached by depth-first search.

Compare with Figure 4.3, Figure 8.8, Figure 8.9 and Figure 8.10. The remarks for those figures apply here, too.

Figure 8.14 depicts the number of edges examined by depth-first search on directed (left) and undirected (right) graphs, for different values of $r = pn$ (on the x-axis). Three curves are displayed in each diagram, for $n = 50,000$, $n = 100,000$, and $n = 200,000$ nodes respectively. The value displayed is the depth reached divided by n. The ranges of graphs from disconnected to connected are covered.

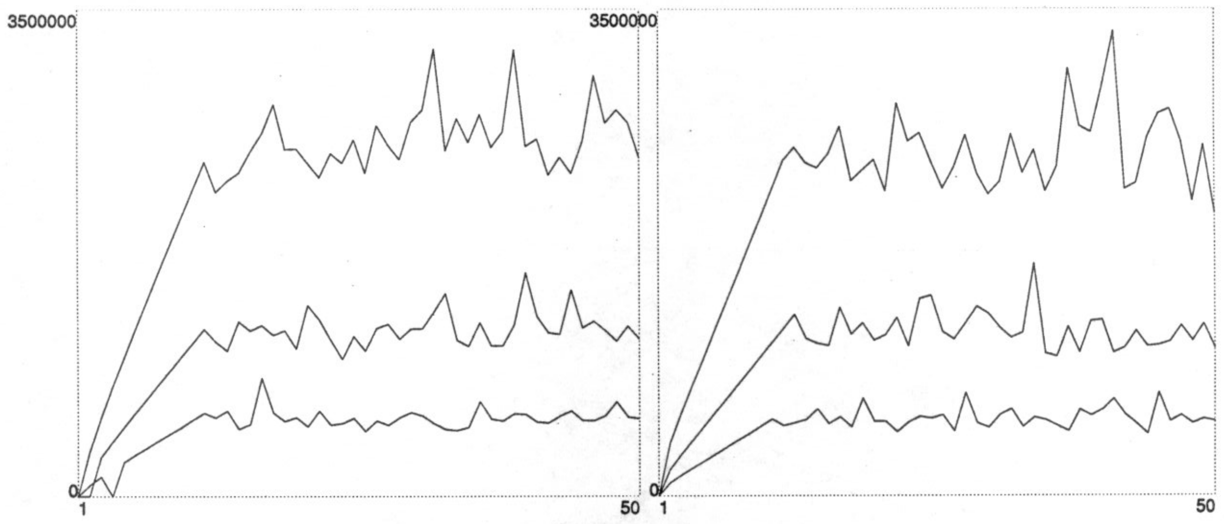

Figure 8.14 Edges examined by depth-first search.

Compare with Figure 2.4 for sparser graphs, and Figure 8.7, for denser graphs. The remarks for those figures apply here, too.

Figure 8.15 depicts the number of nodes visited by depth-first search on directed (left) and undirected (right) graphs, for different values of $r = pn$ (on the x-axis). Three curves are displayed in each diagram, for $n = 50,000$, $n = 100,000$, and $n = 200,000$ nodes respectively. The ranges of graphs from disconnected to connected are covered.

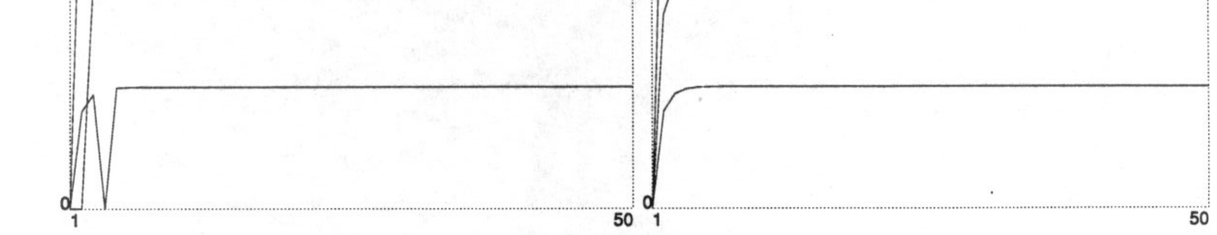

Figure 8.15 Nodes visited by depth-first search.

Compare with Figure 2.3, for sparser graphs; the same remarks apply here.

Figure 8.16 depicts the number of edges examined (on the y-axis) using walk-first search on directed (left) and undirected (right) graphs. Graphs of $n = 4,000$ nodes are used with graph densities ranging from $p = 1/n$ up to the complete graph. The x-axis contains the value of pn; one experiment was used for each integer value of pn.

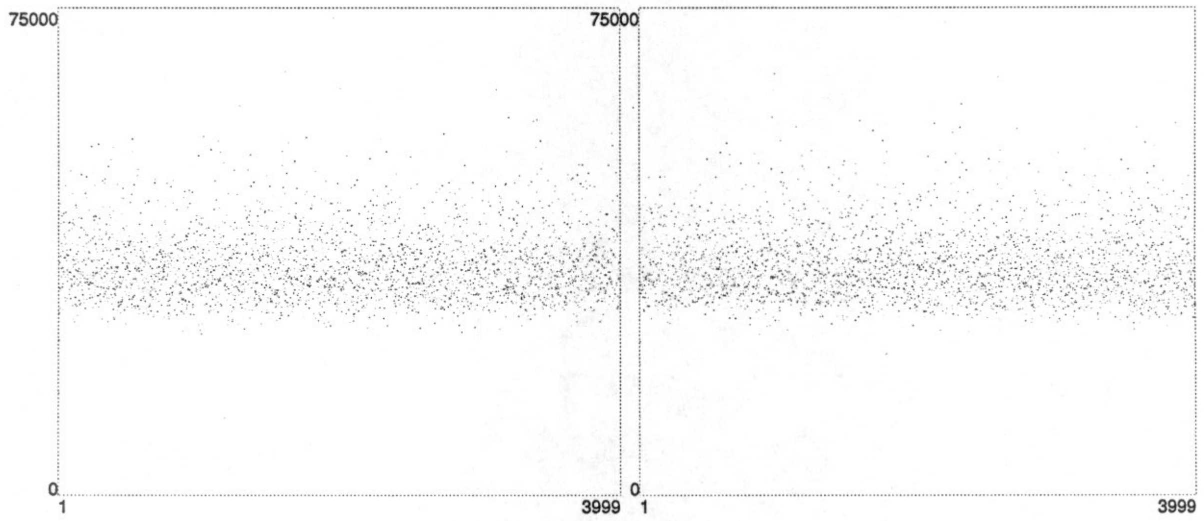

Figure 8.16 Edges scanned by walk-first search.

One dot is drawn for each execution; the variance of the number of edges examined seems to be big; although all values have the same order of magnitude, they do not converge to a certain line. The number of edges examined seems to be independent of the graph density.

Figure 8.17 depicts the size of the data structure needed (on the y-axis) using walk-first search on directed (left) and undirected (right) graphs. Graphs of $n = 4,000$ nodes are used with graph densities ranging from $p = 1/n$ up to the complete graph. The x-axis contains the value of pn; one experiment was used for each integer value of pn.

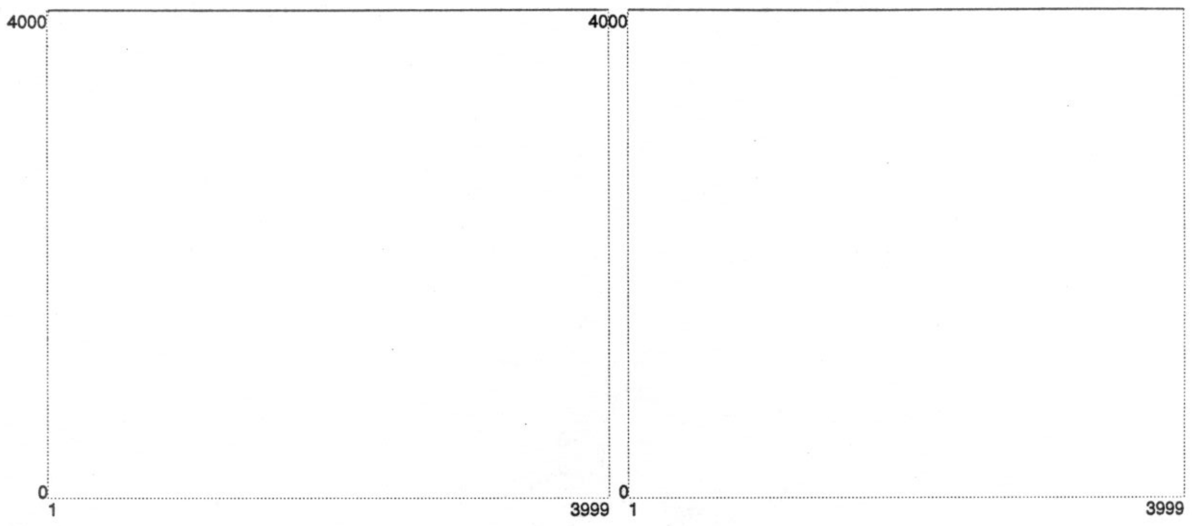

Figure 8.17 Size of the data structure in walk-first search.

One dot is drawn for each execution. The size of the data structure needed seems to always be n. Figures for sparse graphs can illustrate better the increase of the size up to its final value, n.

Figure 8.18 depicts the size of the data structure used by walk-first search on directed (left) and undirected (right) graphs, for different values of n (on the x-axis) and $r = pn$ (on the y-axis). The value displayed is the size needed divided by n. The values of p are small enough so that all graphs are normally disconnected.

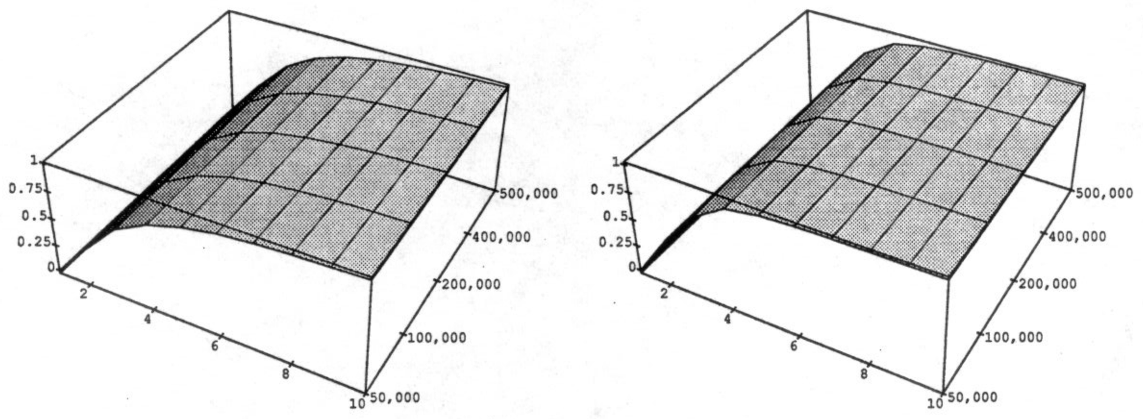

Figure 8.18 Size of the data structure needed by walk-first search.

As expected, the two pictures are similar in shape. Also, the curves seem to be independent of n, close to the curve of the accessible nodes, and to have very small variance. Compare with Figure 8.17, for denser ranges of graphs.

Figure 8.19 depicts the size of the data structure used by walk-first search on directed (left) and undirected (right) graphs, for different values of n (on the x-axis) and $r = pn$ (on the y-axis). As expected, the two pictures are similar. The value displayed is the size needed divided by n. The ranges of graphs from disconnected to connected are covered.

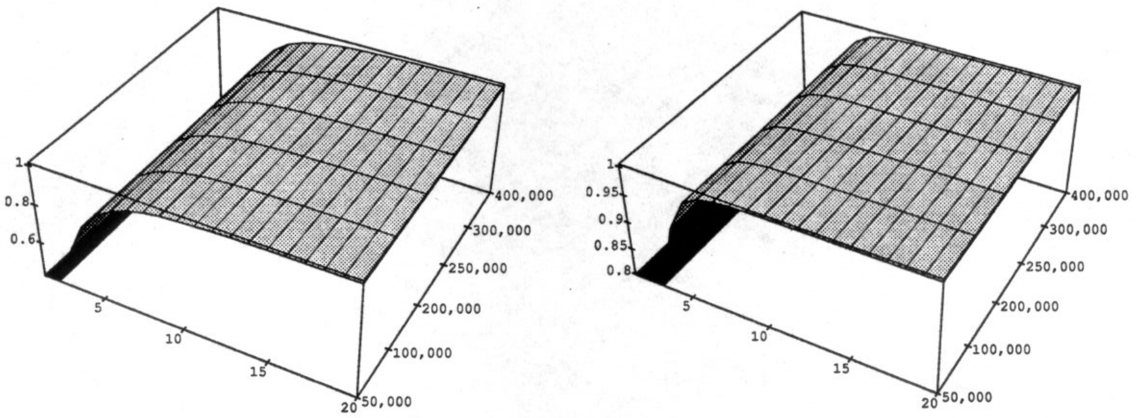

Figure 8.19 Size of the data structure needed by walk-first search.

Compare with Figure 8.17 and Figure 8.18. The remarks for those figures apply here, too.

Figure 8.20 depicts the number of edges examined by walk-first search on directed (left) and undirected (right) graphs, for different values of n (on the x-axis) and $r = pn$ (on the y-axis). As expected, the two pictures are similar. The ranges of graphs from disconnected to connected are covered.

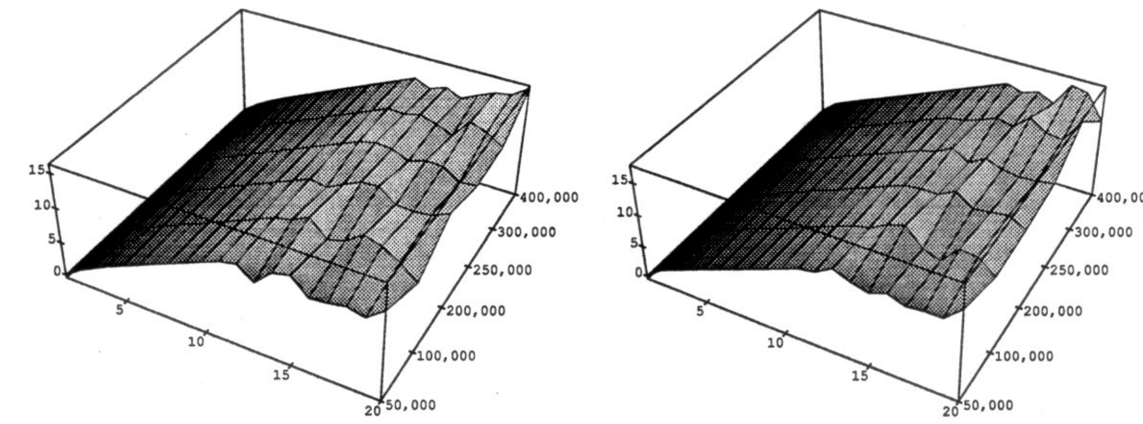

Figure 8.20 Edges examined by walk-first search.

Compare with Figure 2.4 for sparser graphs, and Figure 8.16, for denser graphs. The remarks for those figures apply here, too.

Figure 8.21 depicts the number of nodes visited by walk-first search on directed (left) and undirected (right) graphs, for different values of n (on the x-axis) and $r = pn$ (on the y-axis). As expected, the two pictures are similar. The value displayed is the number of nodes visited divided by n. The ranges of graphs from disconnected to connected are covered.

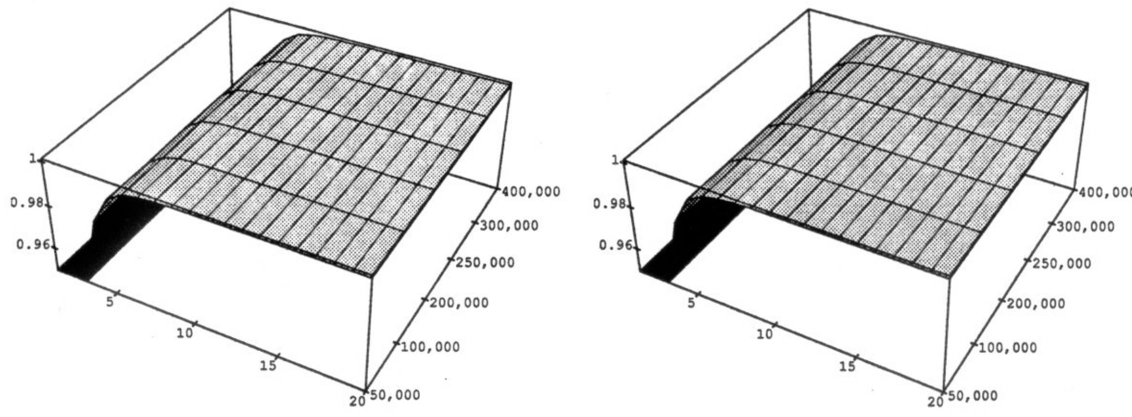

Figure 8.21 Nodes visited by walk-first search.

Compare with Figure 2.3, for sparser graphs; the same remarks apply here.

Figure 8.22 depicts the size of the data structure used by walk-first search on directed (left) and undirected (right) graphs, for different values of $r = pn$ (on the x-axis). Three curves are displayed in each diagram, with $n = 50,000$, $n = 100,000$, and $n = 200,000$ nodes respectively. The ranges of graphs from disconnected to connected are covered.

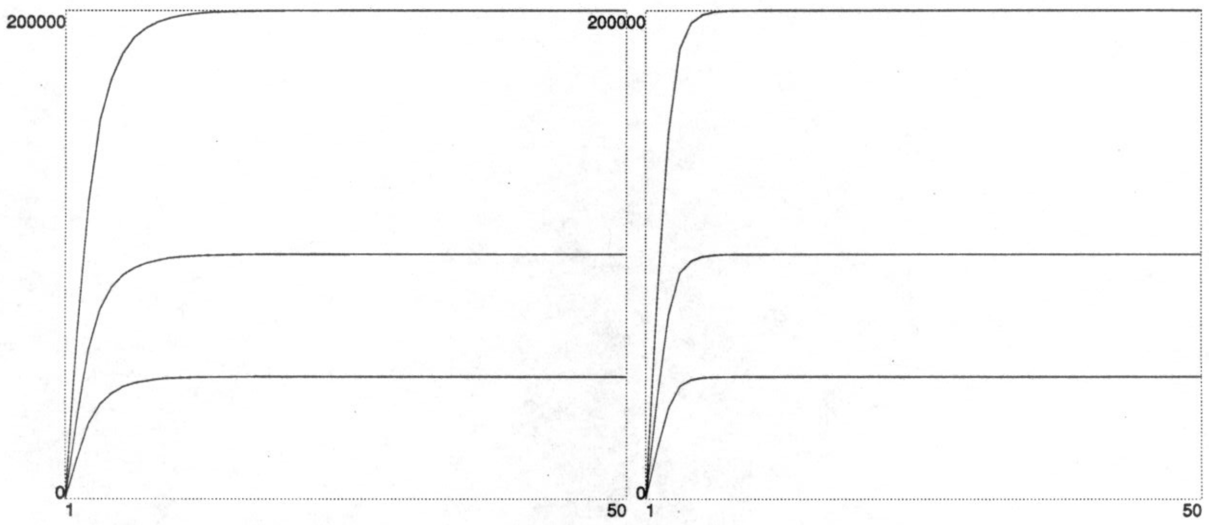

Figure 8.22 Size of the data structure needed by walk-first search.

Compare with Figure 8.17 and Figure 8.18. The remarks for those figures apply here, too.

Figure 8.23 depicts the number of edges examined by walk-first search on directed (left) and undirected (right) graphs, for different values of $r = pn$ (on the x-axis). Three curves are displayed in each diagram, with $n = 50,000$, $n = 100,000$, and $n = 200,000$ nodes respectively. The ranges of graphs from disconnected to connected are covered.

Compare with Figure 2.4 for sparser graphs, and Figure 8.16, for denser graphs. The remarks for those figures apply here, too.

Figure 8.24 depicts the number of nodes visited by walk-first search on directed (left) and undirected (right) graphs, for different values of $r = pn$ (on the x-axis). Three curves are displayed in each diagram, with $n = 50,000$, $n = 100,000$, and $n = 200,000$ nodes respectively. The ranges of graphs from disconnected to connected are covered.

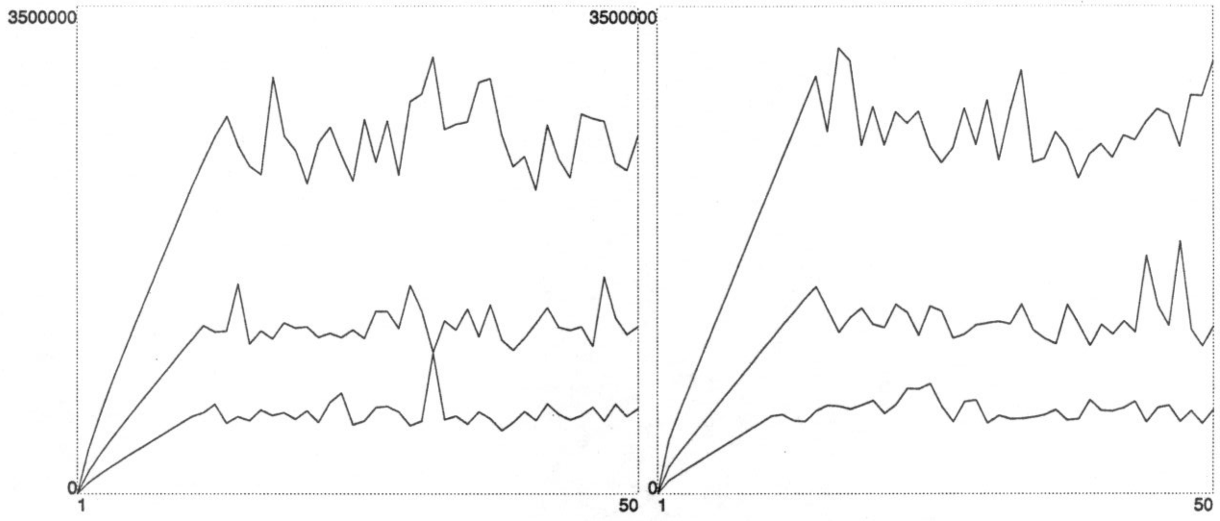

Figure 8.23 Edges examined by walk-first search.

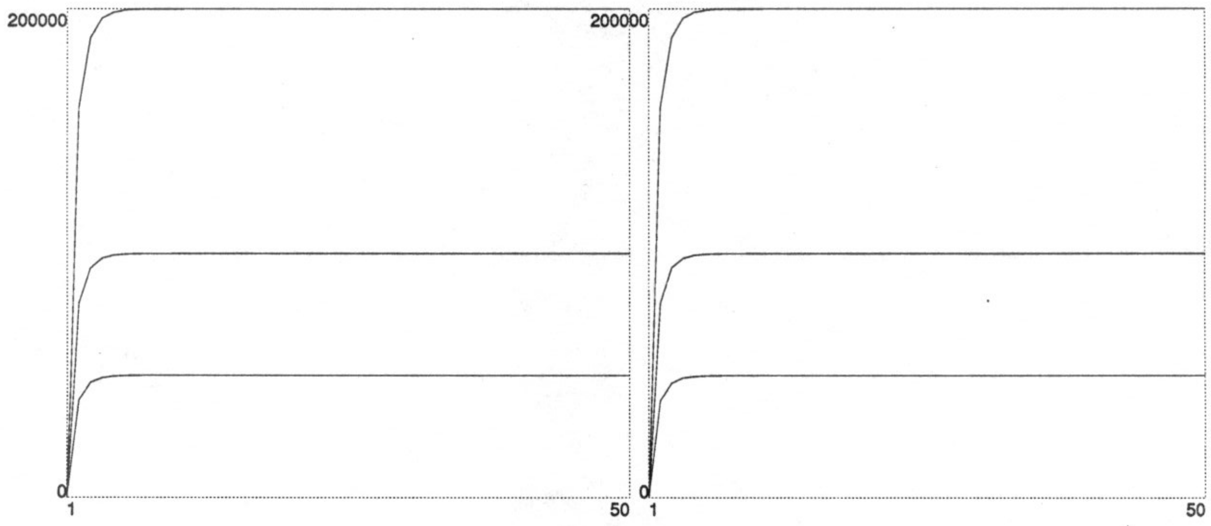

Figure 8.24 Nodes visited by walk-first search.

Compare with Figure 2.3, for sparser graphs; the same remarks apply here.

Figure 8.25 depicts the size of the data structure used by walk-first search and depth-first search on directed (left) and undirected (right) graphs, for different values of $r = pn$ (on the x-axis) for graphs with $n = 500,000$ nodes. The ranges of graphs from disconnected to connected are covered. In depth-first search, the size of the data structure used is also its depth. A different graph was used for each searching method. The number of accessible nodes in the two graphs is also displayed, for better comparison.

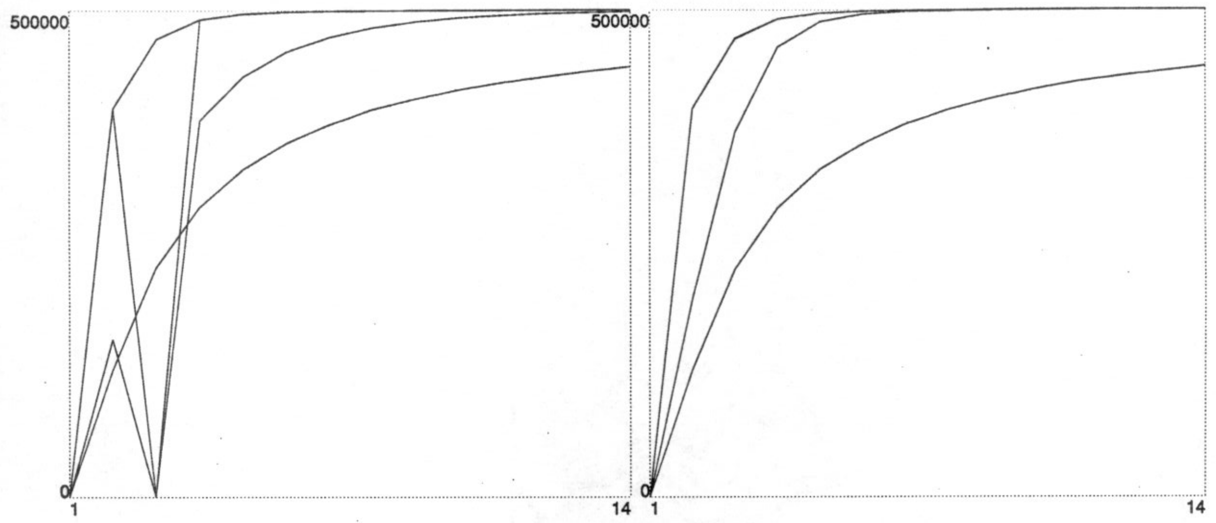

Figure 8.25 Accessible nodes and size of the data structure.

The size of the data structure used by walk-first search is always larger than that needed by depth-first search, and much closer to the number of accessible nodes. Also, in the undirected graph, it seems to converge faster to n.

Chapter 9

Conclusions

Searching a graph takes $O(n \log n)$ time (even when the graph is very dense), and linear space. In connected graphs the running time of any searching algorithm is expected to be $\Theta(n \log n)$, except for node-based searching in a complete graph.

Node-based searching algorithms have a lot of interesting properties. They are more efficient than the other algorithms and can even search a complete graph in linear time; they process $\Theta(logn/\log(1/q))$ nodes. They benefit even more on undirected graphs, and scan the smallest number of edges that do not lead to visited nodes.

Node-based searching algorithms require the minimum possible size of the data structure. Breadth-first search finds the shortest distance between the starting node and all accessible nodes and reaches depth $\Theta(logn/\log(pn))$. Stack-based search is the most efficient algorithm on undirected random graphs and reaches depth $\Theta(1/\log(1/q))$ on all graphs.

Depth-first search creates the longest searching simple path and the new current node is always a neighbor of the previous one. Walk-first search is the slowest searching algorithm, needs the maximum size of the data structure, n, and forms long paths, not necessarily simple.

Searching for a particular node is expected to require linear number of edges to be examined and a data structure of linear size. Although searching takes less time than searching the entire graph, it needs data structure of the same order of size.

Searching in multigraphs, is not expected to be significantly more expensive than searching ordinary random graphs. Searching forests is of the same complexity as searching the giant component. Searching graphs with self loops is only slightly more expensive than without the self loops.

9.1 Applications

The results can be used to estimate the expected size of related data structures and analyze the average-case of other algorithms that use searching algorithms, when applied to this type of input.

A lot of graph and network communications algorithms that use searching can benefit from this analysis.

Since we also address sparse graphs, the results can be used in Databases and Artificial Intelligence, where we have a huge amount of items(nodes) connected by relations(edges) whose number is of the same order of magnitude.

The techniques used in the programs to make simulations in big graphs (few millions of nodes) possible are of general interest, especially to those performing similar experiments.

9.2 Open Problems

- Average-case analysis has not been done for other algorithms, mainly graph algorithms. So there are plenty of open problems in that area: minimum spanning tree algorithms, maximum flow algorithms, matching algorithms, etc.
- A tighter upper bound on the depth of the depth-first search, when $p = \Theta(1/n)$.
- A tighter bound on the size of the data structure needed by any searching algorithm, when $p = \Theta(1/n)$.
- Under certain, common conditions for premature termination of searching, we need average-case analysis for all possible quantities of interest: time, size of the data structure, and depth.

9.3 References

[AA] A. Aggarwal and J. Anderson. A random NC algorithm for depth-first search, *Proceedings of the 19th Annual ACM Symposium on the Theory of Computing*, 1987, 325-334.

[AAK] A. Aggarwal, R.J. Anderson and Ming-Yang Kao. Parallel depth-first search in general directed graphs, *Proceedings of the 21st Annual ACM Symposium on the Theory of Computing*, 1989, 297-308.

[AF] M. Ajtai and R. Fagin. Reachability is Harder for Directed than for Undirected finite graphs, *Proceedings of the 29th IEEE Symposium on Foundations of Computer Science*, 1988, 358-367.

[Al] David J. Aldous. Lower bounds for Covering Times for Reversible Markov Chains and Random Walks on Graphs, *Journal of Theoretical Probability*, Vol. 2 (1989), 91-100.

[AKLLR] R. Alelimunas, R.M. Karp, R.J. Lipton, L. Lovasz and C. Rackoff. Random walks, universal traversal sequences, and the complexity of maze problems, *Proceedings of the 20th IEEE Symposium on Foundations of Computer Science*, 1979, 218-233.

[AV] D. Angluin and L.G. Valiant. Fast probabilistic algorithms for Hamilton circuits and matchings, *Journal of Computer and System Sciences*, Vol. 18 (1979), 155-193.

[AS] Cecilia R. Aragon and Raimund G. Seidel. Randomized Search Trees, *Proceedings of the 30th IEEE Symposium on Foundations of Computer Science*, 1989, 540-545.

[BES] L. Babai, P. Erdös and S.M. Selkow. Random graph isomorphisms, *SIAM Journal on Computing*, Vol. 9 (1980), 628-635.

[Bog] Kenneth P. Bogard. *Introductory Combinatorics*, Harcourt Brace Jovanovich Inc, 1990.

[Bo82] Bela Bollobás. Vertices of a given degree in a random graph, *Journal of Graph Theory*, Vol. 6 (1982), 147-155.

[BT] Bela Bollobás and A. Thomason. Random graphs of smaller order, *Random Graphs '83*, Amsterdam: North Holland, 1985, 47-97.

[BFF] B. Bollobás, T.I. Fenner and A.M. Frieze. An algorithm for finding Hamilton Cycles in a Random Graph, *Proceedings of the 17th Annual ACM Symposium on the Theory of Computing*, 1985, 430-439.

[Bo85] Bela Bollobás. *Random Graphs*, Academic Press, 1985.

[Bo86] Bela Bollobás. *Extremal graph theory with emphasis on probabilistic methods*, Providence, R.I. : Published for the Conference Board of the Mathematical Sciences by the American Mathematical Society, 1986.

[Br] Andrei Broder. Generating random spanning trees, *Proceedings of the 30th IEEE Symposium on Foundations of Computer Science*, 1989, 442-447.

[BK] A.Z. Broder and A.R. Karlin. Bounds on Cover Times, *Journal of Theoretical Probability*, Vol. 2 (1989), 101-120.

[BKRU] A.Z. Broder, A.R. Karlin Prabhakar Raghavan, and Eli Upfal. Trading space for time in undirected s-t connectivity, *Proceedings of the 21st Annual ACM Symposium on the Theory of Computing*, 1989, 543-549.

[CSW] L. Carter, L. Stockmeyer and M. Wegman. The Complexity of Backtrack Searches, *Proceedings of the 17th Annual ACM Symposium on the Theory of Computing*, 1985, 449-457.

[CGM] Vinod Chachra, Prabhakar M. Ghare and James M. Moore. *Applications of Graph Theory Algorithms*, New York: North Holland, 1979.

[CH] Joseph Cherigan and Torben Hagerup. A randomized maximum flow algorithm, *Proceedings of the 30th IEEE Symposium on Foundations of Computer Science*, 1989, 118-123.

[De] Luc Devroye. *Non-uniform random variate generation*, Springer-Verlag, 1986.

[DF] M.E. Dyer and A.M. Frieze. Fast Solution of some Random NP-Hard Problems, *Proceedings of the 27th IEEE Symposium on Foundations of Computer Science*, 1986, 331-336.

[ER59] P. Erdös and A. Rényi. On random graphs I, *Publications Mathematical Debrecen*, Vol. 6 (1959), 290-297.

[ER60] P. Erdös and A. Rényi. On the evolution of random graphs, *Publications of the Mathematical Institute of the Hungarian Academy of Sciences*, Vol. 5 (1960), 17-61.

[EP] P. Erdös and Z. Palka. Trees in random graphs, *Discrete Mathematics*, Vol. 46 (1983), 145-150.

[Ev] Shimon Even. *Graph algorithms*, Potomac, Md. : Computer Science Press, c1979.

[Fe] W. Fernandez de la Vega. Random graphs almost optimally colorable in polynomial time, *Random Graphs '83*, Amsterdam: North Holland, 1985, 311-318.

[GJ] F. Göbel and A.A. Jagers. Random Walks on Graphs, *Stochastic Processes and their Applications*, Vol. 2 (1974), 311-336.

[GM] Michel Gondran and Michel Minoux. *Graphs and Algorithms*, John Wiley and Sons Ltd 1984.

[GKP] R.L. Graham, D.E. Knuth and O. Patashnik. *Concrete Mathematics: A Foundation for Computer Science*, Addison-Wesley, 1989.

[GS] Yuri Gurevich and Saharon Shelah. Expected Computation Time for Hamiltonian Path Problem, *SIAM Journal on Computing*, Vol. 16 (1987), 468-502.

[Ha] Carl Hanser. *Graphs, Data Structures, Algorithms*, Verlag Munchen Wien 1979.

[HL] D.S. Hirschberg and L.L. Larmore. Average Case Analysis of Marking Algorithms, *SIAM Journal on Computing*, Vol. 15 (1986), 1069-1074.

[Iv] G. Ivchenko. On the asymptotic behaviour of degree of vertices in a random graph, *Theory of Probability and its Applications*, Vol. 18 (1973), 188-195.

[JS] J. Jaworski and I.H. Smit. On a random digraph, *Random Graphs '85*, Amsterdam; New York: North Holland, 1987, 111-127.

[Karó] M. Karónski. A review of random graphs, *Journal of Graph Theory*, Vol. 6 (1982), 349-389.

[Karp] Richard M. Karp. The probabilistic analysis of some combinatorial search algorithms, *Algorithms and Complexity*, Academic Press, New York, 1976, 1-19.

[KS] R.M. Karp and M. Sipser. Maximum matchings in sparse random graphs, *Proceedings of the 22th IEEE Symposium on Foundations of Computer Science*, 1981, 364-375.

[KL] Richard M. Karp and Michael Luby. Monte-Carlo Algorithms for Enumeration and Reliability Problems, *Proceedings of the 24th IEEE Symposium on Foundations of Computer Science*, 1983, 56-64.

[KP] Richard M. Karp and Judea Pearl. Searching for an Optimal Path in a tree with Random Costs, *Artificial Intelligence*, Vol. 21 (1983), 99-116.

[KUW] Richard M. Karp, Eli Upfal and Avi Widgerson. The complexity of parallel search, *Journal of Computer and System Sciences*, Vol. 36 (1988), 225-253.

[KT] D.J. Kelly and J.W. Tolle. Expected simplex algorithm behaviour for random linear programs, *Methoths of Operation Research*, Vol 31 (1978) Proceedings of the Third Symposium on Operations Research, 361-368.

[Ke] R. Kemp. The expected number of nodes and leaves at level k in ordered trees, *6th GI-Conference on Theoretical Conference Science, Springer Lecture Notes in Computer Science 145*, Springer, New York, 1983, 153-163.

[Kn72] Donald E. Knuth. Mathematical Analysis of Algorithms, *Information Processing 71*, (Proceedings of the 1971 IFIP congress), Amsterdam: North Holland, 1972, 19-27.

[Kn80] Donald Knuth. *The Art of Computer Programming: Vol. 2, Seminumerical Algorithms*, Addison-Wesley, 1980.

[KO] E. Korsch and Z. Ostfeld. DFS tree constructions: Algorithms and characterizations, *Graph-theoretic Concepts in Computer Science*, Springer-Verlag 1989, 87-106.

[Ms] A. Marchetti-Spaccamela. On the estimate of the size of a directed graph, *Graph-theoretic Concepts in Computer Science*, Springer-Verlag 1989, 317-326.

[Ma] W. H. Matthews. *Mazes and Labyrinths: Their history and development*, Dover Publications Inc, 1970.

[Mc] Colin McDiarmid. Average-Case Lower Bounds for Searching, *SIAM Journal on Computing*, Vol. 17 (1988), 1044-1060.

[MM] A. Meir and J.W.Moor. On the altitude of nodes in random trees, *Canadian Journal of Mathematics*, Vol 30 (1978), 997-1015.

[MT] Alistair Moffat and Tadao Takaoka. An all pairs shortest path algorithm with expected time $O(n^2 \log n)$, *SIAM Journal on Computing*, Vol. 16 (1987), 1023-1031.

[Mot] Rajeev Motwani. Expanding graphs and the average-case analysis of algorithms for matchings and related problems, *Proceedings of the 21st Annual ACM Symposium on the Theory of Computing*, 1989, 550-561.

[OSW] Th. Ottmann, H.W. Six and D. Wood. The analysis of search trees, A survey, *Graph-theoretic Concepts in Computer Science*, Springer-Verlag 1981, 234-249.

[Palk] Z. Palka. Extreme Degrees in Random Graphs, *Journal of Graph Theory*, Vol. 11 (1987), 121-134.

[PTW] George Pólya, Robert E. Tarjan and Donald R. Woods. *Notes on introductory combinatorics*, Birkhüser Boston, Inc., 1986.

[Po] L. Posa. Hamiltonian Circuits in Random Graphs, *Discrete Math*, Vol. 14 (1976), 359-364.

[Pre86] William H. Press ... [et al.]. *Numerical recipes : the art of scientific computing*, Cambridge University Press, 1986.

[Pro] H. Prodinger. On the Altitude of Specified Nodes in Random Trees, *Journal of Graph Theory*, Vol. 8 (1984), 481-485.

[Re] J.H. Reif. Depth-first search is inherently sequential, *Information Processing Letters*, Vol. 20 (1985), 229-234.

[Se] Robert Sedgewick. *Algorithms in C*, Addison-Wesley, 1990.

[SS] S. Shelah and J. Spencer. Threshold Spectra for Random Graphs, *Proceedings of the 19th Annual ACM Symposium on the Theory of Computing*, 1987, 421-424.

[Sp] Paul Spirakis. The diameter of Connected Components of Random Graphs, *Graph-theoretic Concepts in Computer Science*, Springer-Verlag 1987, 264-276.

[Ta] R.E. Tarjan. Depth-first search and linear algorithms, *SIAM Journal on Computing*, Vol. 1(1972), 146-160.

[Ti] G. Tinhofer. On the use of some almost sure graph properties, *Graph-theoretic Concepts in Computer Science*, Springer-Verlag 1981, 113-126.

[Tu] W.T. Tutte. *Connectivity in Graphs*, University of Toronto Press, 1966.

[Ya78] A.C. Yao. 2–3 Trees, *Acta Informatica*, Vol. 9 (1978), 159-170.

[Ya87] A.C. Yao. Lower Bounds to Randomized Algorithms for Graph Properties, *Proceedings of the 28th IEEE Symposium on Foundations of Computer Science*, 1987, 393-400.

Appendix

More experimental results

We used more than one hundred trillion machine instructions in our computer simulations. We include here results for different ranges of the graph parameters than these found on the preceded chapters.

Figure A.1 depicts the time needed to search directed graphs using a node-based searching algorithm for different values of n (on the x-axis) and p (on the y-axis). The value displayed is the number of edges examined divided by $n \log n$. The values of p are large enough so that all graphs are normally connected, and the method used for these experiments was breadth-first search.

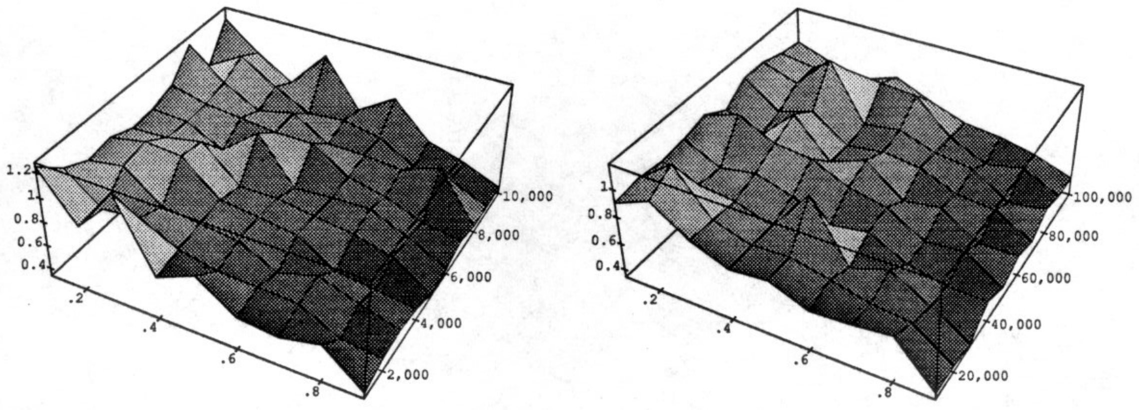

Figure A.1 Number of edges used by node-based search.

The two pictures represent experiments in graphs of different sizes. Compare these pictures with Figure 3.1. The remarks for Figure 3.1 apply here, too.

Experiments for different values of n and p on directed graphs yield the values for the depth of the depth-first search shown in the next table. Rows correspond to a particular value of n, columns correspond to a particular value of $r = pn$.

n\r	4	8	16
100	66.1	84.1	91.4
1000	551	805	902.2
10000	5957.6	8003	8985.3
100000	59160.9	79464.7	89690.2
1000000	594023	794632	897253

This table is similar to the one in Chapter 4, but in this one each value is derived from the average of then experiments. The remarks for that table apply here, too.

The values of $V(G)$, as defined in Chapter 4, found experimentally for different values of n and p using directed graphs are displayed in the next table. Again, rows correspond to a particular value of n and columns correspond to a particular value of $r = pn$.

n\r	4	8	16
100	23.3	64.5	84.6
1000	46.9	356.9	745.5
10000	78.8	1404.1	4681.1
100000	33.1	2353	41190.5
1000000	60.3	3962	231984
10000000	46.5	1473.3	1428106
100000000	56.1	2402.4	3432754

This table is similar with the one in Chapter 4, but in this one each value is derived from the average of then experiments. The remarks for tat table apply here, too.

Figure A.2 depicts the number of edges examined (on the y-axis) using depth-first search on directed (left) and undirected (right) graphs. Graphs of $n = 1,000$ nodes are used with graph densities ranging from $p = 1/n$ up to the complete graph. The x-axis contains the value of pn; one experiment was used for each integer value of pn.

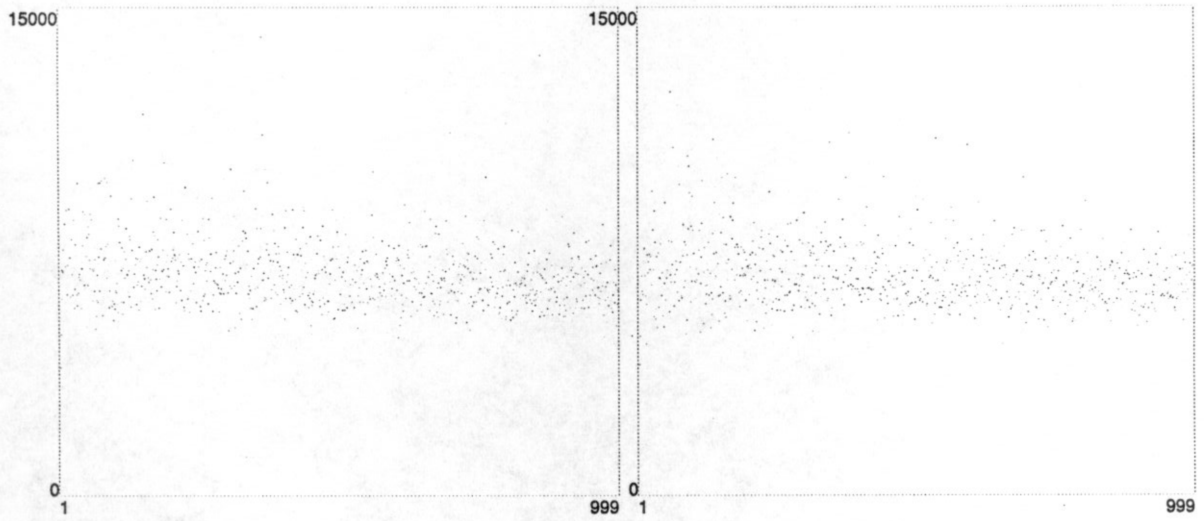

Figure A.2 Edges scanned in 1,000 node graphs by depth-first search.

One dot is drawn for each execution; compare with Figure 8.7. The remarks for Figure 8.7 apply here, too.

Figure A.3 depicts the number of edges examined (on the y-axis) using depth-first search on directed (left) and undirected (right) graphs. Graphs of $n = 2,000$ nodes are used with graph densities ranging from $p = 1/n$ up to the complete graph. The x-axis contains the value of pn; one experiment was used for each integer value of pn.

One dot is drawn for each execution; compare with Figure 8.7 and Figure A.2. The remarks for Figure 8.7 apply here, too.

Figure A.4 depicts the number of edges examined (on the y-axis) using depth-first search on directed (left) and undirected (right) graphs. Graphs of $n = 3,000$ nodes are used with graph densities ranging from $p = 1/n$ up to the complete graph. The x-axis contains the value of pn; one experiment was used for each integer value of pn.

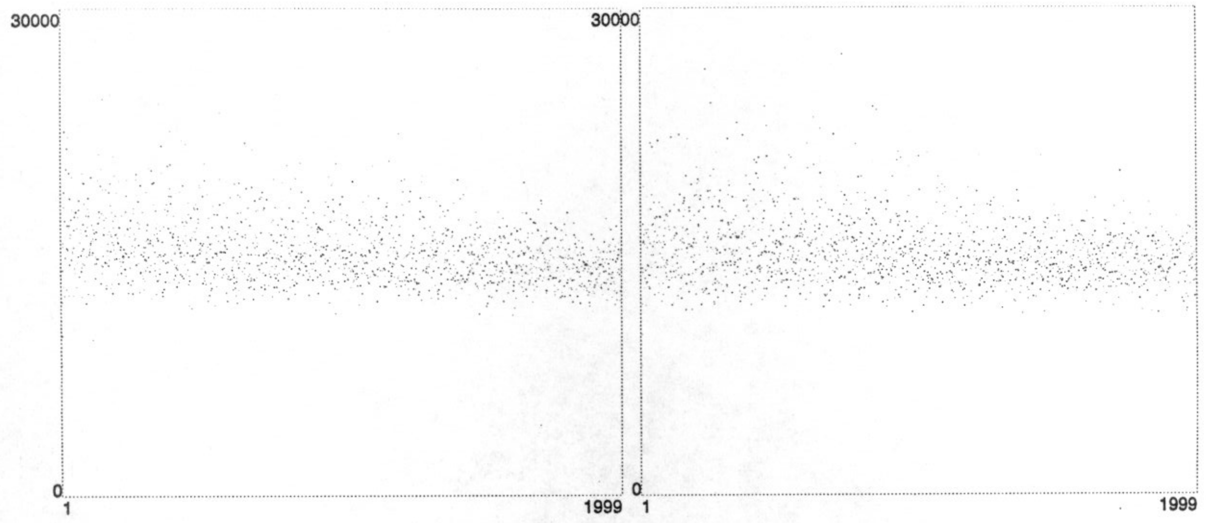

Figure A.3 Edges scanned in 2,000 node graphs by depth-first search.

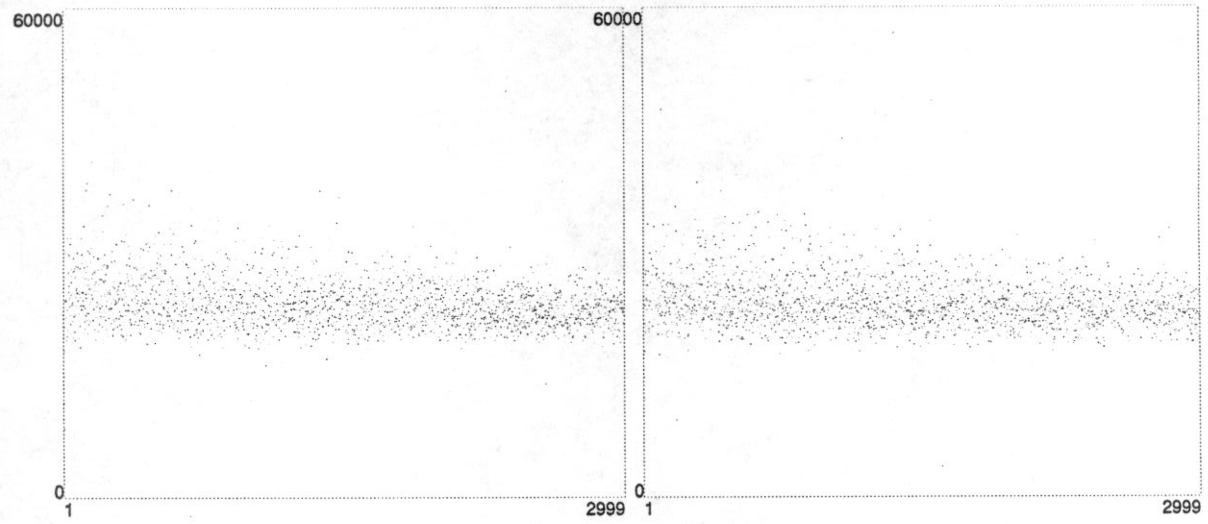

Figure A.4 Edges scanned in 3,000 node graphs by depth-first search.

One dot is drawn for each execution; compare with Figure 8.7, Figure A.2 and Figure A.3. The remarks for Figure 8.7 apply here, too.

Figure A.5 depicts the depth reached (on the y-axis) using depth-first search on directed (left) and undirected (right) graphs. Graphs of $n = 1,000$ nodes are used with graph densities ranging from $p = 1/n$ up to the complete graph. The x-axis contains the value of pn; one experiment was used for each integer value of pn.

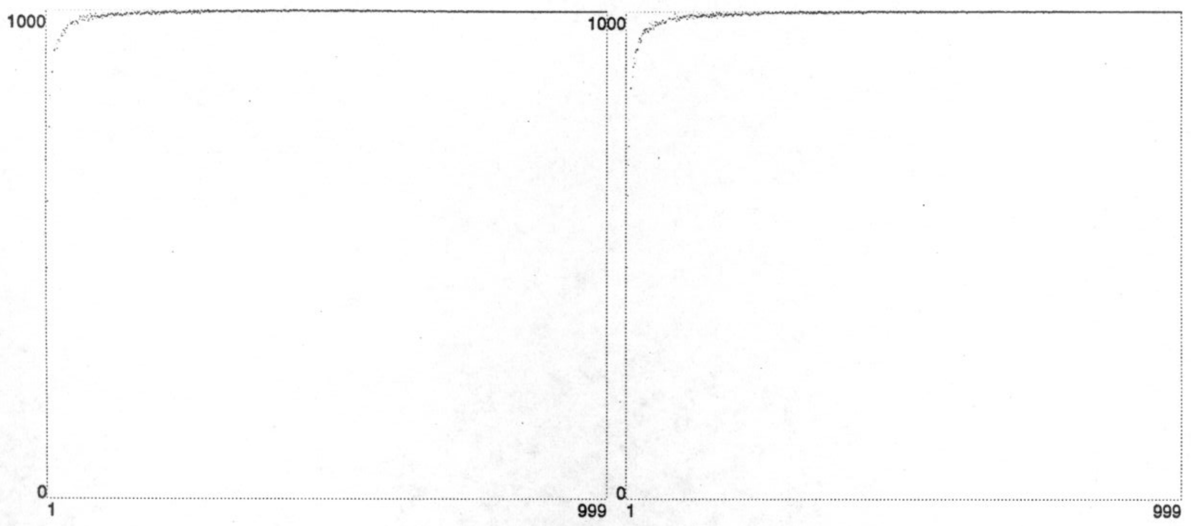

Figure A.5 Depth reached in 1,000 node graphs by depth-first search.

One dot is drawn for each execution; compare with Figure 8.8. The remarks for Figure 8.8 apply here, too.

Figure A.6 depicts the depth reached (on the y-axis) using depth-first search on directed (left) and undirected (right) graphs. Graphs of $n = 2,000$ nodes are used with graph densities ranging from $p = 1/n$ up to the complete graph. The x-axis contains the value of pn; one experiment was used for each integer value of pn.

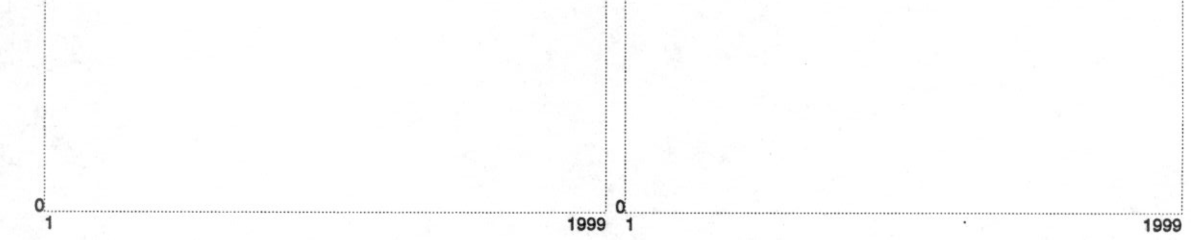

Figure A.6 Depth reached in 2,000 node graphs by depth-first search.

One dot is drawn for each execution; compare with Figure 8.8 and Figure A.5. The remarks for Figure 8.8 apply here, too.

Figure A.7 depicts the depth reached (on the y-axis) using depth-first search on directed (left) and undirected (right) graphs. Graphs of $n = 3,000$ nodes are used with graph densities ranging from $p = 1/n$ up to the complete graph. The x-axis contains the value of pn; one experiment was used for each integer value of pn.

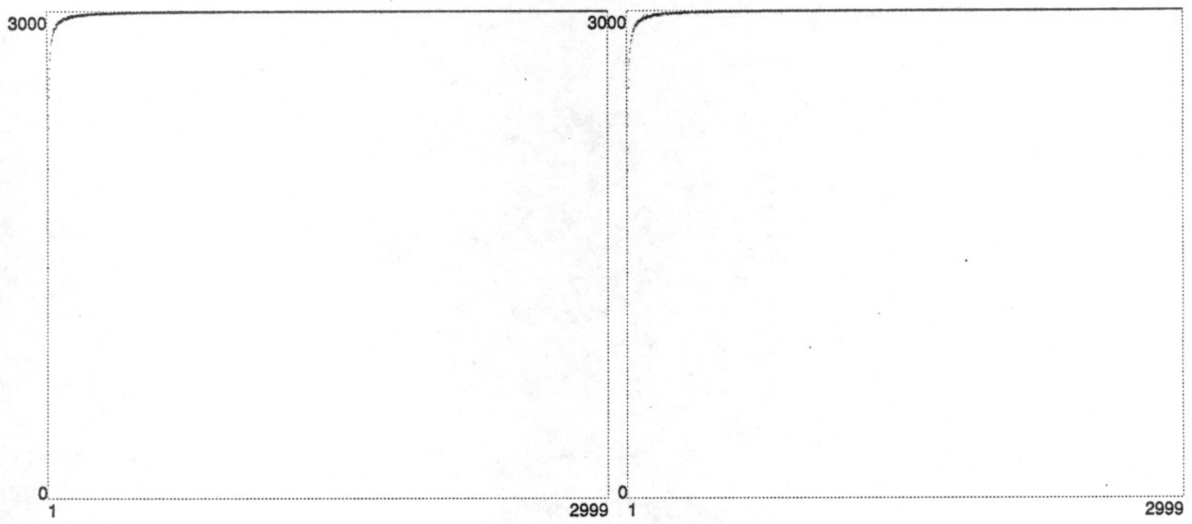

Figure A.7 Depth reached in 3,000 node graphs by depth-first search.

One dot is drawn for each execution; compare with Figure 8.8, Figure A.5 and Figure A.6. The remarks for Figure 8.8 apply here, too.

Figure A.8 depicts the number of edges examined (on the y-axis) using walk-first search on directed (left) and undirected (right) graphs. Graphs of $n = 1,000$ nodes are used with graph densities ranging from $p = 1/n$ up to the complete graph. The x-axis contains the value of pn; one experiment was used for each integer value of pn.

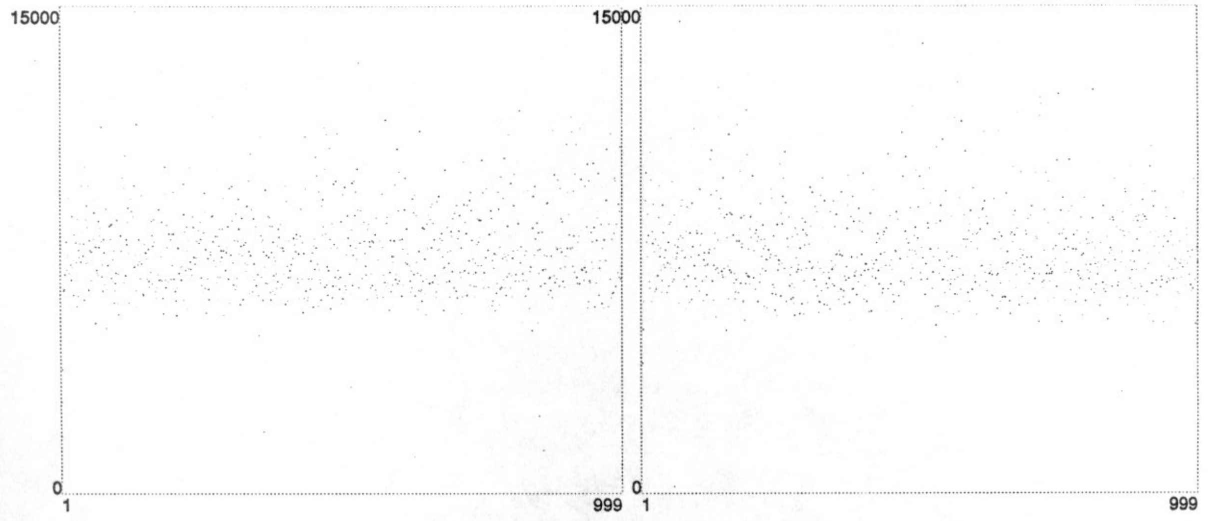

Figure A.8 Edges scanned in 1,000 node graphs by walk-first search.

One dot is drawn for each execution; compare with Figure 8.16. The remarks for Figure 8.16 apply here, too.

Figure A.9 depicts the number of edges examined (on the y-axis) using walk-first search on directed (left) and undirected (right) graphs. Graphs of $n = 2,000$ nodes are used with graph densities ranging from $p = 1/n$ up to the complete graph. The x-axis contains the value of pn; one experiment was used for each integer value of pn.

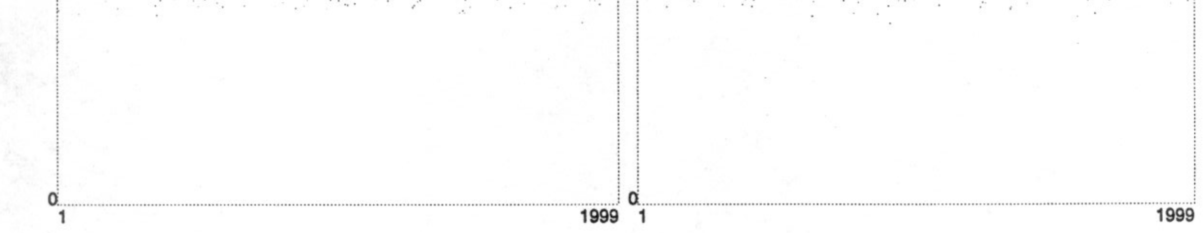

Figure A.9 Edges scanned in 2,000 node graphs by walk-first search.

One dot is drawn for each execution; compare with Figure 8.16 and Figure A.8. The remarks for Figure 8.16 apply here, too.

Figure A.10 depicts the number of edges examined (on the y-axis) using walk-first search on directed (left) and undirected (right) graphs. Graphs of $n = 3,000$ nodes are used with graph densities ranging from $p = 1/n$ up to the complete graph. The x-axis contains the value of pn; one experiment was used for each integer value of pn.

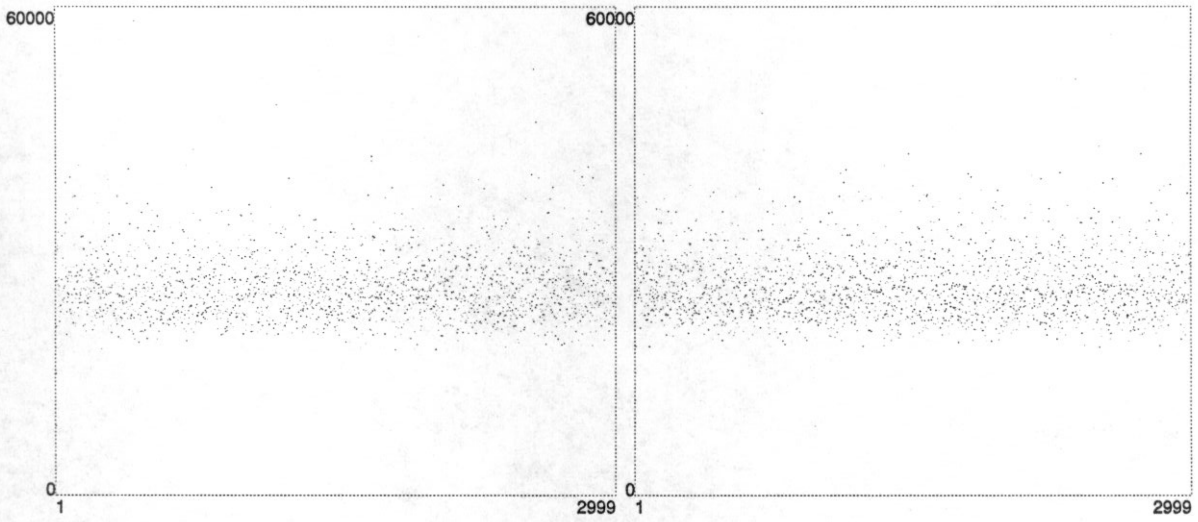

Figure A.10 Edges scanned in 3,000 node graphs by walk-first search.

One dot is drawn for each execution; compare with Figure 8.16, Figure A.8 and Figure A.9. The remarks for Figure 8.16 apply here, too.

Figure A.11 depicts the size of the data structure needed (on the y-axis) using walk-first search on directed (left) and undirected (right) graphs. Graphs of $n = 1,000$ nodes are used with graph densities ranging from $p = 1/n$ up to the complete graph. The x-axis contains the value of pn; one experiment was used for each integer value of pn.

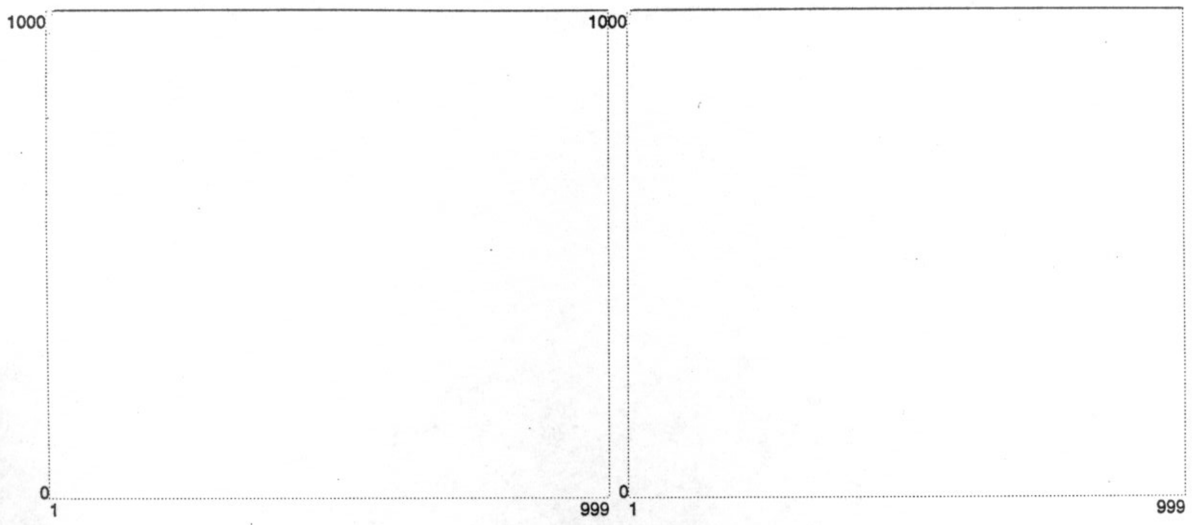

Figure A.11 Size of the data structure in 1,000 node graphs by walk-first search.

One dot is drawn for each execution; compare with Figure 8.17. The remarks for Figure 8.17 apply here, too: the size of the data structure converges to n extremely fast.

Figure A.12 depicts the size of the data structure needed (on the y-axis) using walk-first search on directed (left) and undirected (right) graphs. Graphs of $n = 2,000$ nodes are used with graph densities ranging from $p = 1/n$ up to the complete graph. The x-axis contains the value of pn; one experiment was used for each integer value of pn.

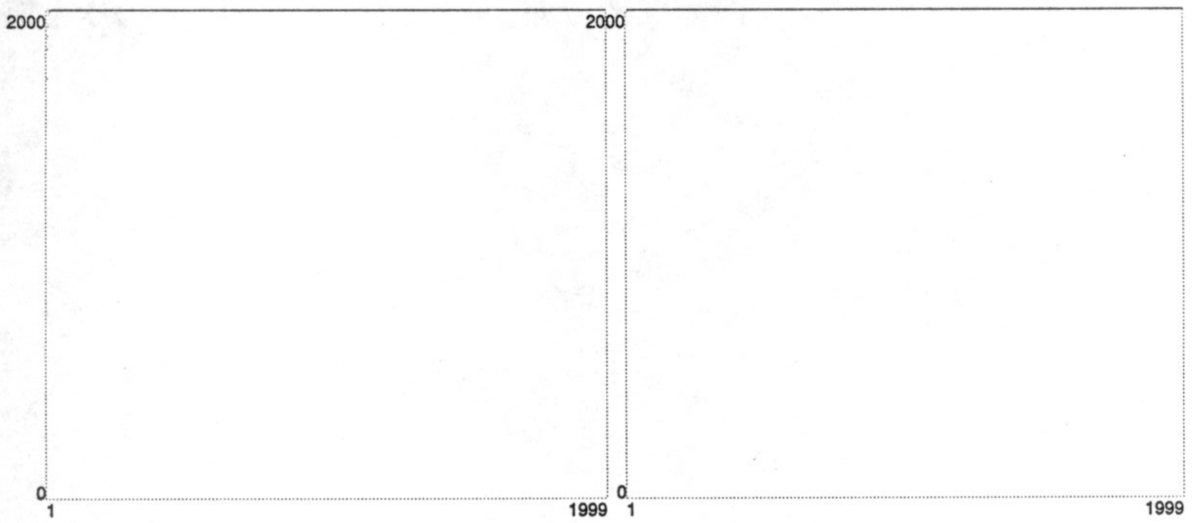

Figure A.12 Size of the data structure in 2,000 node graphs by walk-first search.

One dot is drawn for each execution; compare with Figure 8.17 and Figure A.11. The remarks for Figure 8.17 apply here, too.

Figure A.13 depicts the size of the data structure needed (on the y-axis) using walk-first search on directed (left) and undirected (right) graphs. Graphs of $n = 3,000$ nodes are used with graph densities ranging from $p = 1/n$ up to the complete graph. The x-axis contains the value of pn; one experiment was used for each integer value of pn.

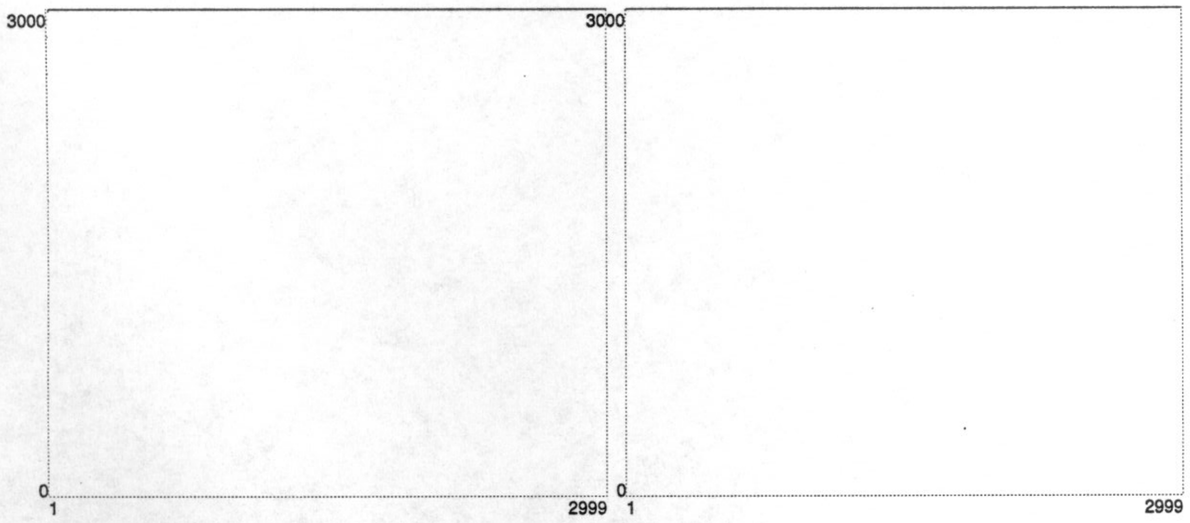

Figure A.13 Size of the data structure in 3,000 node graphs by walk-first search.

One dot is drawn for each execution; compare with last figure and Figure 8.17. The remarks for Figure 8.17 apply here, too.

Figure A.14 depicts for each node degree (on the x-axis) the number the nodes (on the y-axis) that backtracked without visiting any new node, using depth-first search on directed (left) and undirected (right) graphs. In the experiments, graphs with $n = 10,000$ nodes and probability $p = 3/n$ were used, and the sum over 10 searches is displayed.

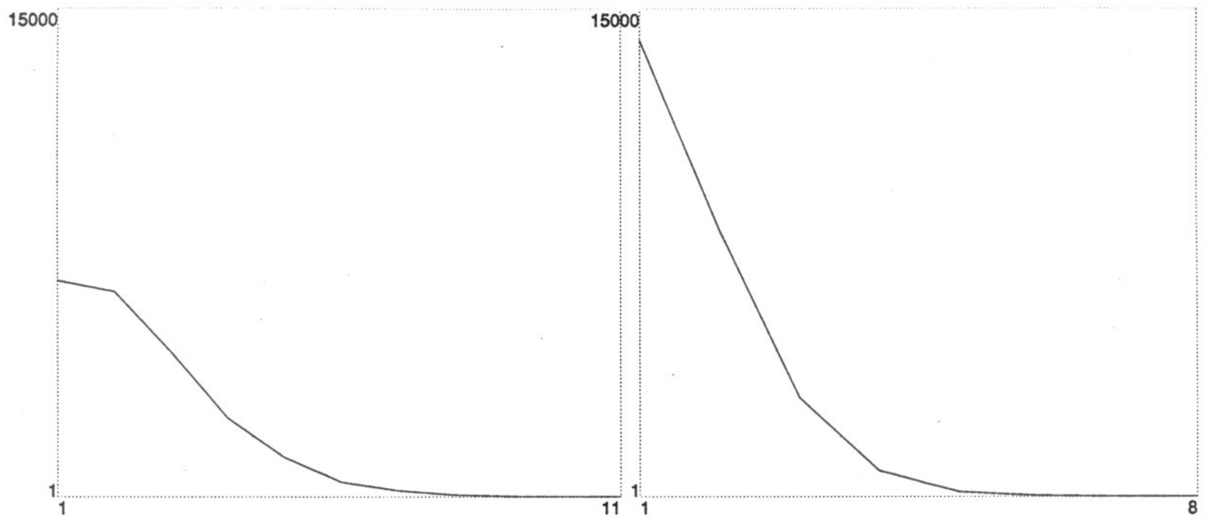

Figure A.14 Nodes backtracked over node degree, $p = 3/n$.

One dot is drawn for each execution; compare with Figure 8.5. The remarks for Figure 8.5 apply here, too.

Figure A.15 depicts the number of nodes that backtrack without following any of their edges (y-axis) as a function of the depth of the search at that moment (x-axis), using depth-first search on directed (left) and undirected (right) graphs. The values are the sum of these nodes over 10 graphs having $n = 10,000$ nodes and probability for each edge $p = 6/n$.

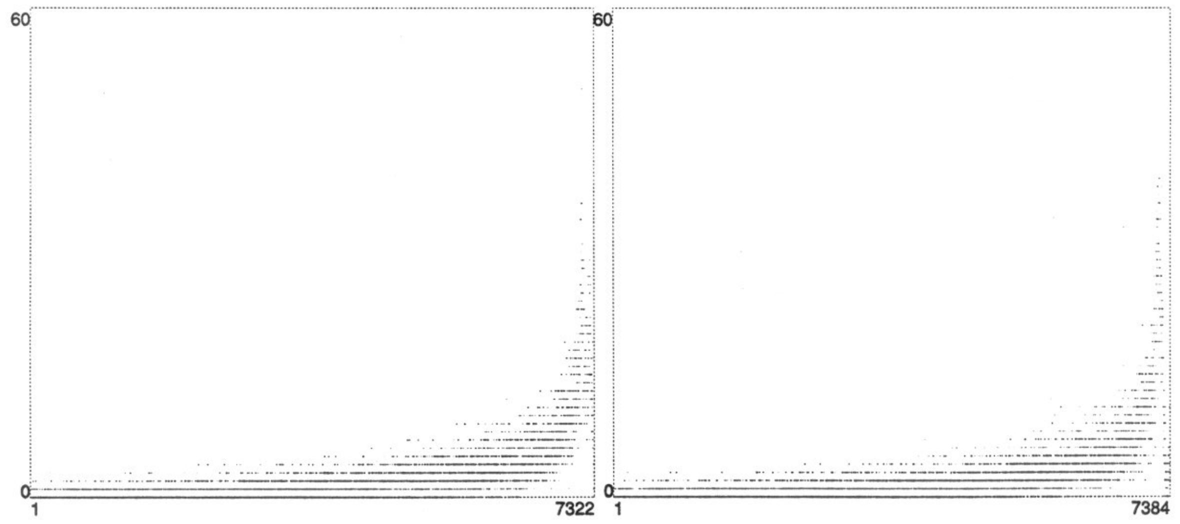

Figure A.15 Nodes backtracking as a function of the depth, $p = 6/n$.

One dot is drawn for each depth; the depth takes integer values only and thus the dots appear on distinct horizontal levels. Compare with Figure 8.6. The remarks for Figure 8.6 apply here, too.

Figure A.16 depicts the number of nodes visited by walk-first search on disconnected directed graphs. The number of nodes visited divided by n is displayed. The dependence of the number of visited nodes on both n (on the x-axis) and the product $r = pn$ (on the y-axis) can be observed; the shape of the curve seems totally independent of n. The picture for undirected graphs is identical.

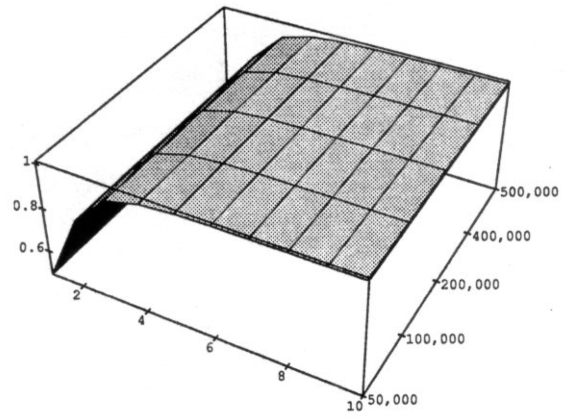

Figure A.16 The number of accessible nodes on graphs.

Compare with Figure 2.3, the same results found with depth-first search. The remarks for Figure 2.3 apply here, too.

Figure A.17 depicts the number of edges used by walk-first search on disconnected directed graphs. The dependence of the number of visited nodes on both n (on the x-axis) and the product $r = pn$ (on the y-axis) can be observed; the shape of the curve seems totally independent of n. The number of edges displayed has been divided by n, so that for all graphs the value drawn does not exceed r.

Compare with Figure 2.4, the same results found with depth-first search. The remarks for Figure 2.4 apply here, too.

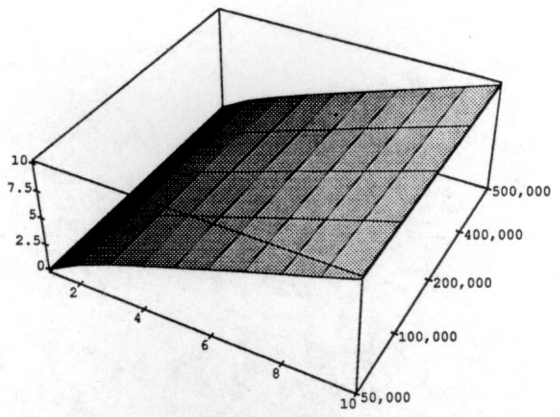

Figure A.17 The number of edges used on disconnected graphs.