4 Algorithms for Centrality Indices

Riko Jacob, Dirk Koschützki, Katharina Anna Lehmann, Leon Peeters,* and Dagmar Tenfelde-Podehl*

The usefulness of centrality indices stands or falls with the ability to compute them quickly. This is a problem at the heart of computer science, and much research is devoted to the design and analysis of efficient algorithms. For example, shortest-path computations are well understood, and these insights are easily applicable to all distance based centrality measures. This chapter is concerned with algorithms that efficiently compute the centrality indices of the previous chapters.

Most of the distance based centralities can be computed by directly evaluating their definition. Usually, this naïve approach is reasonably efficient once all shortest path distances are known. For example, the closeness centrality requires to sum over all distances from a certain vertex to all other vertices. Given a matrix containing all distances, this corresponds to summing the entries of one row or column. Computing all closeness values thus traverses the matrix once completely, taking n^2 steps. Computing the distance matrix using the fastest known algorithms will take between n^2 and n^3 steps, depending on the algorithm, and on the possibility to exploit the special structure of the network. Thus, computing the closeness centrality for all vertices can be done efficiently in polynomial time. Nevertheless, for large networks this can lead to significant computation times, in which case a specialized algorithm can be the crucial ingredient for analyzing the network at hand. However, even a specialized exact algorithm might still be too time consuming for really large networks, such as the Web graph. So, for such huge networks it is reasonable to approximate the outcome with very fast, preferably linear time, algorithms.

Another important aspect of real life networks is that they frequently change over time. The most prominent example of this behavior is the Web graph. Rather than recomputing all centrality values from scratch after some changes, we prefer to somehow reuse the previous computations. Such dynamic algorithms are not only valuable in a changing environment. They can also increase performance for vitality based centrality indices, where the definition requires to repeatedly remove an element from the network. For example, dynamic all-pairs shortest paths algorithms can be used in this setting.

This chapter not only lists the known results, but also provides the ideas that make such algorithms work. To that end, Section 4.1 recapitulates some basic shortest paths algorithms, to provide the background for the more special-

* Lead authors

ized centrality algorithms presented in Section 4.2. Next, Section 4.3 describes fast approximation algorithms for closeness centrality as well as for web centralities. Finally, algorithms for dynamically changing networks are considered in Section 4.4.

4.1 Basic Algorithms

Several good text books on basic graph algorithms are available, such as Ahuja, Magnanti, and Orlin [6], and Cormen, Leiserson, Rivest, and Stein [133]. This section recapitulates some basic and important algorithmic ideas, to provide a basis for the specific centrality algorithms in Section 4.2. Further, we briefly review the running times of some of the algorithms to indicate how computationally expensive different centrality measures are, especially for large networks.

4.1.1 Shortest Paths

The computation of the shortest-path distances between one specific vertex, called the source, and all other vertices is a classical algorithmic problem, known as the Single Source Shortest Path (SSSP) problem.

Dijkstra [146] provided the first polynomial-time algorithm for the SSSP for graphs with non-negative edge weights. The algorithm maintains a set of shortest-path labels $d(s,v)$ denoting the length of the shortest path found so-far between s and v. These labels are initialized to infinity, since no shortest paths are known when the algorithm starts. The algorithm further maintains a list P of permanently labeled vertices, and a list T of temporarily labeled vertices. For a vertex $v \in P$, the label $d(s,v)$ equals the shortest-path distance between s and v, whereas for vertices $v \in T$ the labels $d(s,v)$ are upper bounds (or estimates) on the shortest-path distances.

The algorithm starts by marking the source vertex s as permanent and inserting it into P, scanning all its neighbors $N(s)$, and setting the labels for the neighbors $v \in N(s)$ to the edge lengths: $d(s,v) = \omega(s,v)$. Next, the algorithm repeatedly removes a non-permanent vertex v with minimum label $d(s,v)$ from T, marks v as permanent, and scans all its neighbors $w \in N(v)$. If this scan discovers a new shortest path to w using the edge (v,w), then the label $d(s,w)$ is updated accordingly. The algorithm relies upon a priority queue for finding the next node to be marked as permanent. Implementing this priority queue as a Fibonacci heap, Dijkstra's algorithm runs in time $\mathcal{O}(m + n \log n)$. For unit edge weights, the priority queue can be replaced by a regular queue. Then, the algorithm boils down to Breadth-First Search (BFS), taking $\mathcal{O}(m+n)$ time. Algorithm 4 describes Dijkstra's algorithm more precisely.

Often, one is not only interested in the shortest-path distances, but also in the shortest paths themselves. These can be retraced using a function $\mathit{pred}(v) \in V$, which stores the predecessor of the vertex v on its shortest path from s. Starting at a vertex v, the shortest path from s is obtained by recursively applying $\mathit{pred}(v), \mathit{pred}(\mathit{pred}(v)), \ldots,$ until one of the $\mathit{pred}()$ functions returns s. Since

Algorithm 4: Dijkstra's SSSP algorithm

```
Input: Graph $G = (V, E)$, edge weights $\omega : E \to \mathbb{R}$, source vertex $s \in V$
Output: Shortest path distances $d(s, v)$ to all $v \in V$

$P = \emptyset, T = V$
$d(s, v) = \infty$ for all $v \in V, d(s, s) = 0, pred(s) = 0$
while $P \neq V$ do
    $v = \text{argmin}\{d(s, v) | v \in T\}$
    $P := P \cup v, T := T \setminus v$
    for $w \in N(v)$ do
        if $d(s, w) > d(s, v) + \omega(v, w)$ then
            $d(s, w) := d(s, v) + \omega(v, w)$
            $pred(w) = v$
```

the algorithm computes exactly one shortest path to each vertex, and no such shortest path can contain a cycle, the set of edges $\{(pred(v), v) \mid v \in V\}$, defines a spanning tree of G. Such a tree, which need not be unique, is called a shortest-paths tree.

Since Dijkstra's original work in 1954 [146], many improved algorithms for the SSSP have been developed. For an overview, we refer to Ahuja, Magnanti, and Orlin [6], and Cormen, Leiserson, Rivest, and Stein [133].

4.1.2 Shortest Paths Between All Vertex Pairs

The problem of computing the shortest path distances between all vertex pairs is called the All-Pairs Shortest Paths problem (APSP). All-pairs shortest paths can be straightforwardly computed by computing n shortest paths trees, one for each vertex $v \in V$, with v as the source vertex s. For sparse graphs, this approach may very well yield the best running time. In particular, it yields a running time of $\mathcal{O}(nm + n^2)$ for unweighted graphs.

For non-sparse graphs, however, this may induce more work than necessary. The following shortest path label optimality conditions form a crucial observation for improving the above straightforward APSP algorithm.

Lemma 4.1.1. *Let the distance labels $d(u, v), u, v \in V$, represent the length of some path from u to v. Then the labels d represent shortest path distances if and only if*

$$d(u, w) \leq d(u, v) + d(v, w) \text{ for all } u, v, w, \in V.$$

Thus, given some set of distance labels, it takes n^3 operations to check if these optimality conditions hold. Based on this observation and a theorem of Warshall [568], Floyd [217] developed an APSP algorithm that achieves an $\mathcal{O}(n^3)$ time bound, see Algorithm 5. The algorithm first initializes all distance labels to infinity, and then sets the distance labels $d(u, v)$, for $\{u, v\} \in E$, to the edge lengths $\omega(u, v)$. After this initialization, the algorithm basically checks whether there exists a vertex triple u, v, w for which the distance labels violate the condition in Lemma 4.1.1. If so, it decreases the involved distance label $d(u, w)$. This check is performed in a triple for-loop over the vertices. Since we are looking for all-pairs shortest paths, the algorithm maintains a set of predecessor indices $pred(u, v)$ that contain the predecessor vertex of v on some shortest path from u to v.

Algorithm 5: Floyd-Warshall's APSP algorithm

```
**Input**: Graph $G = (V, E)$, edge weights $\omega : E \to R$
**Output**: Shortest path distances $d(u, v)$ between all $u, v \in V$

$d(u, v) = \infty, pred(u, v) = 0$ for all $u, v \in V$
$d(v, v) = 0$ for all $v \in V$
$d(u, v) = \omega(u, v), pred(u, v) = u$ for all $\{u, v\} \in E$
for $v \in V$ do
    for $\{u, w\} \in V \times V$ do
        if $d(u, w) > d(u, v) + d(v, w)$ then
            $d(u, w) := d(u, v) + d(v, w)$
            $pred(u, w) := pred(v, w)$
```

4.1.3 Dynamic All-Pairs Shortest Paths

The dynamic variant of the APSP problem is particularly interesting in the context of network analysis. The dynamic APSP problem consists of maintaining an optimal set of shortest path distance labels $d(u, v), u, v \in V$, in a graph that changes by edge insertions and deletions. Typically, one also wants to simultaneously maintain the corresponding shortest paths themselves, rather than only the distances.

Thus, dynamic APSP's are of importance for vitality related questions, such as how shortest path distances change upon removing an edge. Since removing a vertex from a graph results in the removal of its incident edges, vertex vitality corresponds to sequences of edge removals in a dynamic APSP setting. Further, the dynamic APSP is clearly applicable in the setting of the changing Web graph.

The challenge for the dynamic APSP problem is to do better than recomputing a set of optimal distance labels from scratch after an update. Recently, Demetrescu and Italiano [142] described an algorithm for the dynamic APSP problem on directed graphs with non-negative real-valued edge weights. Per edge insertion, edge deletion, or edge weight change, their algorithm takes $\mathcal{O}(n^2 \log^3 n)$ amortized time to maintain the all-pairs shortest path distance labels. As the algorithm and its analysis are quite involved, their discussion falls outside the scope of this book. Instead, we refer to Demetrescu and Italiano [142] for details on the dynamic APSP.

Further, Thorup [549] provides an alternative description of the algorithm, as well as an improved amortized update time of $\mathcal{O}(n^2(\log n + \log^2(m + n/n)))$. Moreover, the improved algorithm allows for negative weights. Roditty and

Zwick [496] argue that the dynamic SSSP problem on weighted graphs is as difficult as the static APSP problem. Further, they present a randomized algorithm for the dynamic APSP, returning correct results with very high probability, with improved amortized update time for sparse graphs.

4.1.4 Maximum Flows and Minimum-Cost Flows

For flow betweenness (see Section 3.6.1), the maximum flow between a designated source node s and a designated sink node t needs to be computed. The maximum-flow problem has been studied extensively in the literature, and several algorithms are available. Some are generally applicable, some focus on restricted cases of the problem, such as unit edge capacities, and others provide improvements that may have more theoretical than practical impact. The same applies to minimum-cost flows, with the remark that minimum-cost flow algorithms are even more complex.

Again, we refer to the textbooks by Ahuja, Magnanti, and Orlin [6], and Cormen, Leiserson, Rivest, and Stein [133] for good in-depth descriptions of the algorithms. To give an idea of flow algorithms' worst-case running times, and of the resulting impact on centrality computations in large networks, we briefly mention the following algorithms. The preflow-push algorithm by Goldberg and Tarjan [252] runs in $\mathcal{O}(nm\log(n^2/m))$, and the capacity scaling algorithm by Ahuja and Orlin [8] runs in $\mathcal{O}(nm\log U)$, where U is the largest edge capacity. For minimum cost flows, the capacity scaling algorithm by Edmonds and Karp [172] runs in $\mathcal{O}((m\log U)(m+n\log n))$.

Alternatively, both maximum flow and minimum-cost flow problems can be solved using linear programming. The linear program for flow problems has a special structure which guarantees an integer optimal solution for any integer inputs (costs, capacities, and net inflows). Moreover, specialized network simplex algorithms for flow-based linear programs with polynomial running times are available.

4.1.5 Computing the Largest Eigenvector

Several centrality measures described in this part of the book are based on the computation of eigenvectors of a given matrix. This section provides a short introduction to the computation of eigenvectors and eigenvalues. In general, the problem of computing eigenvalues and eigenvectors is non-trivial, and complete books are dedicated to this topic. We focus on a single algorithm and sketch the main idea. All further information, such as optimized algorithms, or algorithms for special matrices, are available in textbooks like [256, 482]. Furthermore, Section 14.2 (chapter on spectral analysis) considers the computation of all eigenvalues of the matrix representing a graph.

The eigenvalue with largest absolute value and the corresponding eigenvector can be computed by the power method, which is described by Algorithm 6. As input the algorithm takes the matrix A and a start vector $\boldsymbol{q}^{(0)} \in \mathbb{R}^n$ with $||\boldsymbol{q}^{(0)}||_2 = 1$. After the k-th iteration, the current approximation of the largest eigenvalue in absolute value and the corresponding eigenvector are stored in the variables $\lambda^{(k)}$ and $\boldsymbol{q}^{(k)}$, respectively.

Algorithm 6: Power method for computating the largest eigenvalue

```
Input: Matrix $A \in \mathbb{R}^{n \times n}$ and vector $||\boldsymbol{q}^{(0)}||_2 = 1$
Output: Largest eigenvalue $\lambda^{(k)}$ in absolute value
and corresponding eigenvector $\boldsymbol{q}^{(k)}$
$k := 1$
repeat
    $\boldsymbol{z}^{(k)} := A\boldsymbol{q}^{(k-1)}$
    $\boldsymbol{q}^{(k)} := \boldsymbol{z}^{(k)} / ||\boldsymbol{z}^{(k)}||_2$
    $\lambda^{(k)} := (\boldsymbol{q}^{(k)})^T A \boldsymbol{q}^{(k)}$
    $k := k + 1$
until $\lambda^{(k)}$ and $\boldsymbol{q}^{(k)}$ are acceptable approximations
```

The power method is guaranteed to converge if the matrix $A \in \mathbb{C}^{n \times n}$ has a dominant eigenvalue, i.e., $|\lambda_1| > |\lambda_i|$ for $i \in \{2 \dots n\}$, or, alternatively, if the matrix $A \in \mathbb{R}^{n \times n}$ is symmetric. The ratio $\frac{|\lambda_2|}{|\lambda_1|}$ of the second largest and the largest eigenvalues determines the rate of convergence, as the approximation error decreases with $\mathcal{O}((\frac{|\lambda_2|}{|\lambda_1|})^k)$. Further details on the power method can be found in many textbooks on linear algebra, e.g., Wilkinson [587].

As the power method only requires matrix-vector multiplication, it is particularly suited for large matrices. For one iteration, it suffices to scan over the matrix once. So, the power method can be reasonably efficient, even without storing the complete matrix in main memory.

4.2 Centrality-Specific Algorithms

As already mentioned, most centrality indices can be computed reasonably fast by directly following their definition. Nevertheless, improvements over this straightforward approach are possible. This section elaborates on two algorithmic ideas for such an improvement.

4.2.1 Betweenness Centrality

Recall the definition of the betweenness centrality of a vertex $v \in V$:

$$c_B(v) = \sum_{s \neq v \neq t \in V} \frac{\sigma_{st}(v)}{\sigma_{st}},$$

with σ_{st} being the number of shortest paths between vertices s and t, and $\sigma_{st}(v)$ the number of those paths passing through vertex v. A straightforward idea for computing $c_B(v)$ for all $v \in V$ is the following. First compute tables with the length and number of shortest paths between all vertex pairs. Then, for each vertex v, consider all possible pairs s and t, use the tables to identify the fraction of shortest s-t-paths through v, and sum these fractions to obtain the betweenness centrality of v.

For computing the number of shortest paths in the first step, one can adjust Dijkstra's algorithm as follows. From Lemma 4.1.1, observe that a vertex v is on a shortest path between two vertices s and t if and only if $d(s,t) = d(s,v) + d(v,t)$. We replace the predecessor vertices by predecessor sets $pred(s,v)$, and each time a vertex $w \in N(v)$ is scanned for which $d(s,t) = d(s,v) + d(v,t)$, that vertex is added to the predecessor set $pred(s,v)$. Then, the following relation holds:

$$\sigma_{sv} = \sum_{u \in pred(s,v)} \sigma_{su}.$$

Setting $pred(s,v) = s$ for all $v \in N(s)$, we can thus compute the number of shortest paths between a source vertex s and all other vertices. This adjustment can easily be incorporated into Dijkstra's algorithm, as well as in the BFS for unweighted graphs.

As for the second step, vertex v is on a shortest s-t-path if $d(s,t) = d(s,v) + d(v,t)$. If this is the case, the number of shortest s-t-paths using v is computed as $\sigma_{st}(v) = \sigma_{sv} \cdot \sigma_{vt}$. Thus, computing $c_B(v)$ requires $\mathcal{O}(n^2)$ time per vertex v because of the summation over all vertices $s \neq v \neq t$, yielding $\mathcal{O}(n^3)$ time in total. This second step dominates the computation of the length and the number of shortest paths. Thus, the straightforward idea for computing betweenness centrality has an overall running time of $\mathcal{O}(n^3)$.

Brandes [92] describes a specific algorithm that computes the betweenness centrality of all vertices in a graph in $\mathcal{O}(nm + n^2 \log n)$ time for weighted graphs, and $\mathcal{O}(nm)$ time for unweighted graphs. Note that this basically corresponds to the time complexity for the n SSSP computations in the first step of the straightforward idea. We describe this betweenness algorithm below.

The pair-dependency of a vertex pair $s, t \in V$ on an intermediate vertex v is defined as $\delta_{st}(v) = \sigma_{st}(v)/\sigma_{st}$, and the dependency of a source vertex $s \in V$ on a vertex $v \in V$ as

$$\delta_{s\bullet}(v) = \sum_{t \in V} \delta_{st}(v).$$

So, the betweenness centrality of a vertex v can be computed as $c_B(v) = \sum_{s \neq v \in V} \delta_{s\bullet}(v)$.

The betweenness centrality algorithm exploits the following recursive relations for the dependencies $\delta_{s\bullet}(v)$.

Theorem 4.2.1 (Brandes [92]). *The dependency $\delta_{s\bullet}(v)$ of a source vertex $s \in V$ on any other vertex $v \in V$ satisfies*

$$\delta_{s\bullet}(v) = \sum_{w : v \in pred(s,w)} \frac{\sigma_{sv}}{\sigma_{sw}} (1 + \delta_{s\bullet}(w)).$$

Proof. First, extend the variables for the number of shortest paths and for the dependency as follows. Define $\sigma_{st}(v,e)$ as the number of shortest paths from s to t that contain both the vertex $v \in V$ and the edge $e \in E$. Further, define the pair-dependency of a vertex pair s,t on both a vertex v and an edge e as $\delta_{st}(v,e) = \sigma_{st}(v,e)/\sigma_{st}$. Using these, we write

$$\delta_{s\bullet}(v) = \sum_{t\in V} \delta_{st}(v) = \sum_{t\in V} \sum_{w:v\in pred(s,w)} \delta_{st}(v,\{v,w\}).$$

Consider a vertex w for which $v \in pred(s,w)$. There are σ_{sw} shortest paths from s to w, of which σ_{sv} go from s to v and then use the edge $\{v,w\}$. Thus, given a vertex t, a fraction σ_{sv}/σ_{sw} of the number of shortest paths $\sigma_{st}(w)$ from s to $t \neq w$ using w also uses the edge $\{v,w\}$. For the pair-dependency of s and t on v and $\{v,w\}$, this yields

$$\delta_{st}(v,\{v,w\}) = \begin{cases} \dfrac{\sigma_{sv}}{\sigma_{sw}} & \text{if } t = w, \\ \dfrac{\sigma_{sv}}{\sigma_{sw}} \cdot \dfrac{\sigma_{st}(w)}{\sigma_{st}} & \text{if } t \neq w. \end{cases}$$

Exchanging the sums in the above summation, and substituting this relation for $\delta_{st}(v,\{v,w\})$ gives

$$\begin{aligned} \sum_{w:v\in pred(s,w)} \sum_{t\in V} \delta_{st}(v,\{v,w\}) &= \sum_{w:v\in pred(s,w)} \left(\frac{\sigma_{sv}}{\sigma_{sw}} + \sum_{t\in V\setminus w} \frac{\sigma_{sv}}{\sigma_{sw}} \cdot \frac{\sigma_{st}(w)}{\sigma_{st}} \right) \\ &= \sum_{w:v\in pred(s,w)} \frac{\sigma_{sv}}{\sigma_{sw}} (1 + \delta_{s\bullet}(w)). \end{aligned}$$

□

The betweenness centrality algorithm is now stated as follows. First, compute n shortest-paths trees, one for each $s \in V$. During these computations, also maintain the predecessor sets $pred(s,v)$. Second, take some $s \in V$, its shortest-paths tree, and its predecessor sets, and compute the dependencies $\delta_{s\bullet}(v)$ for all other $v \in V$ using the dependency relations in Theorem 4.2.1. For vertex s, the dependencies can be computed by traversing the vertices in non-increasing order of their distance from s. In other words, start at the leaves of the shortest-paths tree, work backwardly towards s, and afterwards proceed with the next vertex s. To finally compute the centrality value of vertex v, we merely have to add all dependencies values computed during the n different SSSP computations. The resulting $\mathcal{O}(n^2)$ space usage can be avoided by immediately adding the dependency values to a 'running centrality score' for each vertex.

This algorithm computes the betweenness centrality for each vertex $v \in V$, and requires the computation of one shortest-paths tree for each $v \in V$. Moreover, it requires a storage linear in the number of vertices and edges.

Theorem 4.2.2 (Brandes [92]). *The betweenness centrality $c_B(v)$ for all $v \in V$ can be computed in $\mathcal{O}(nm + n^2 \log n)$ time for weighted graphs, and in $\mathcal{O}(nm)$ time for unweighted graphs. The required storage space is $\mathcal{O}(n + m)$.*

Other shortest-path based centrality indices, such as closeness centrality, graph centrality, and stress centrality can be computed with similar shortest-paths tree computations followed by iterative dependency computations. For further details on this, we refer to Brandes [92].

4.2.2 Shortcut Values

Another algorithmic task is to compute the shortcut value for all edges of a directed graph $G = (V, E)$, as introduced in Section 3.6.3. More precisely, the task is to compute the shortest path distance from vertex u to vertex v in $G_e = (V, E \setminus \{e\})$ for every directed edge $e = (u, v) \in E$. The shortcut value for edge e is a vitality based centrality measure for edges, defined as the maximum increase in shortest path length (absolute, or relative for non-negative distances) if e is removed from the graph.

The shortcut values for all edges can be naïvely computed by $m = |E|$ calls to a SSSP routine. This section describes an algorithm that computes the shortcut values for all edges with only $n = |V|$ calls to a routine that is asymptotically as efficient as a SSSP computation. To the best of our knowledge this is the first detailed exposition of this algorithm, which is based on an idea of Brandes.

We assume that the directed graph G contains no negative cycles, such that $d(i, j)$ is well defined for all vertices i and j. To simplify the description we assume that the graph contains no parallel edges, such that an edge is identified by its endpoints.

The main idea is to consider some vertex u, and to execute one computation to determine the shortcut values for all edges starting at u. These shortcut values are defined by shortest paths that start at vertex u and reach an adjacent vertex v, without using the edge (u, v). To compute this, define $\alpha_i = d(u, i)$ to be the length of a shortest path from u to i, the well known shortest path distance. Further, let the variable $\tau_i \in V$ denote the second vertex (identifying the first edge of the path) of all paths from u to i with length α_i, if this is unique, otherwise it is undefined, $\tau_i = \perp$. Thus, $\tau_i = \perp$ implies that there are at least two paths of length α_i from u to i that start with different edges. Finally, the value β_i is the length of the shortest path from u to i that does not have τ_i as the second vertex, ∞ if no such path exists, or $\beta_i = \alpha_i$ if $\tau_i = \perp$.

Assume that the values α_v, τ_v, and β_v are computed for a neighbor v of u. Then, the shortcut value for the edge (u, v) is α_v if $\tau_v \neq v$, i.e., the edge (u, v) is not the unique shortest path from u to v. Otherwise, if $\tau_v = v$, the value β_v is the shortcut value for (u, v). Hence, it remains to compute the values $\alpha_i, \tau_i, \beta_i$ for $i \in V$. The algorithm exploits that the values $\alpha_i, \tau_i, \beta_i$ obey some recursions. At the base of these recursions we have:

$$\alpha_u = 0, \quad \tau_u = \emptyset, \quad \beta_u = \infty$$

The values α_j obey the shortest paths recursion:

$$\alpha_j = \min_{i:(i,j) \in E} \big(\alpha_i + \omega(i, j)\big)$$

To define the recursion for τ_j, it is convenient to consider the set of incoming neighbors I_j of vertices from which a shortest path can reach j,

$$I_j = \{i \mid (i,j) \in E \text{ and } \alpha_j = \alpha_i + \omega(i,j)\}\,.$$

It holds that

$$\tau_j = \begin{cases} j & \text{if } I_j = \{u\}, \\ a & \text{if } a = \tau_i \text{ for all } i \in I_j (\text{all predecessors have first edge } (u,a)), \\ \perp & \text{otherwise.} \end{cases}$$

The value τ_j is only defined if all shortest paths to vertex j start with the same edge, which is the case only if all τ_i values agree on the vertices in I_j. For the case $\tau_j = \perp$ it holds that $\beta_j = \alpha_j$, otherwise

$$\beta_j = \min \left\{ \min_{\substack{i:(i,j)\in E, \\ \tau_i = \tau_j}} \beta_i + \omega(i,j) \quad , \quad \min_{\substack{i:(i,j)\in E, \\ \tau_i \neq \tau_j}} \alpha_i + \omega(i,j) \right\}\,.$$

To see this, consider the path p that achieves β_j, i.e., a shortest path p from u to j that does not start with τ_j. If the last vertex i of p before j has $\tau_i = \tau_j$, the path p up to i does not start with τ_j, and this path is considered in β_i and hence in β_j. If instead the path p has as the next to last vertex i, and $\tau_i \neq \tau_j$, then one of the shortest paths from u to i does not start with τ_j, and the length of p is $\alpha_i + \omega(i,j)$.

With the above recursions, we can efficiently compute the values $\alpha_i, \tau_i, \beta_i$. For the case of positive weights, any value α_i depends only on values α_j that are smaller than α_i, so these values can be computed in non-decreasing order (just as Dijkstra's algorithm does). If all edge weights are positive, the directed graph containing all shortest paths (another view on the sets I_j) is acyclic, and the values τ_i can be in topological order. Otherwise, we have to identify the strongly connected components of G, and contract them for the computation of τ. Observe that β_i only depends upon β_j if $\beta_j \leq \beta_i$. Hence, these values can be computed in non-decreasing order in a Dijkstra-like algorithm. In the unweighted case, this algorithm does not need a priority queue and its running time is only that of BFS.

If there are negative edge weights, but no negative cycles, the Dijkstra-like algorithm is replaced by a Bellman-Ford type algorithm to compute the α values. The computation of τ remains unchanged. Instead of computing β_i, we compute $\beta'_i = \beta_i - \alpha_i$, i.e., we apply the shortest-paths potential to avoid negative edge weights. This replaces all $\omega(i,j)$ terms with terms of the form $\omega(i,j) - \alpha_j + \alpha_i \geq 0$, and hence the β'_i values can be set in increasing order, and this computes the β_i values as well.

Note that the above method can be modified to also work in networks with parallel edges. There, the first edge of a path is no longer identified by the second vertex of the path, such that this edge should be used instead. We can even modify the method to compute the shortcut value of the vertex v, i.e., the two neighbors of v whose distance increases most if v is deleted from the network. To achieve this, negate the length and direction of the incoming edges, run the above algorithm, and subtract the length of the outgoing edges from the resulting β_i values on the neighbors of v. In this way, for all pairs of neighbors that can reach each other through v the difference between the direct connection and the shortest alternative are computed.

Summarizing, we showed that in the above mentioned types of graphs all shortcut values can be computed in the time of computing n times a SSSP.

4.3 Fast Approximation

Most of the centralities introduced in Chapter 3 can be computed in polynomial time. Although this is a general indication that such computations are feasible, it might still be practically impossible to analyze huge networks in reasonable time. As an example, it may be impossible to compute betweenness centrality for large networks, even when using the improved betweenness algorithm of Section 4.2.1. This phenomenon is particularly prominent when investigating the web graph. For such a huge graph, we typically do not want to invest more than a small number of scans over the complete input.

With this limited computational investment, it might not be possible to determine exact centrality values. Instead, the focus should be on approximate solutions and their quality. In this setting, approximation algorithms provide a guaranteed compromise between running time and accuracy.

Below, we describe an approximation algorithm for the calculation of closeness centrality, and then adapt this algorithm to an approximative calculation for betweenness centrality. Next, Section 4.3.2 discusses approximation methods for the computation of web centralities.

4.3.1 Approximation of Centralities Based on All Pairs Shortest Paths Computations

We have argued above that the calculation of centrality indices can require a lot of computing time. This also applies to the computation of all-pairs shortest paths, even when using the algorithms discussed in Section 4.1.2. In many applications, it is valuable to instead compute a good approximate value for the centrality index, if this is faster. With the random sampling technique introduced by Eppstein and Wang [179], the closeness centrality of all vertices in a weighted, undirected graph can be approximated in $\mathcal{O}(\frac{\log n}{\epsilon^2}(n \log n + m))$ time. The approximated value has an additive error of at most $\epsilon \Delta_G$ with high probability, where ϵ is any fixed constant, and Δ_G is the diameter of the graph. We adapt this technique for the approximative calculation of betweenness centrality, yielding an approximation of the betweenness centrality of all vertices in a weighted, directed graph with an additive error of $(n-2)\epsilon$, and with the same time bound as above.

The following randomized approximative algorithm estimates the closeness centrality of all vertices in a weighted graph by picking K sample vertices and computing single source shortest paths (SSSP) from each sample vertex to all other vertices. Recall the definition of closeness centrality of a vertex $v \in V$:

$$c_C(v) = \frac{\sum_{x \in V} d(v,x)}{n-1}. \tag{4.1}$$

The centrality $c_C(v)$ can be estimated by the calculation of the distance of v to K other vertices $v_1, \ldots, v_K$ as follows

$$\hat{c}_C(v) = \frac{n}{K \cdot (n-1)} \sum_{i=1}^{K} d(v, v_i). \tag{4.2}$$

For undirected graphs, this calculates the average distance from v to K other vertices, then scales this to the sum of distances to/from all other n vertices, and divides by $n-1$. As both c_C and $\hat{c}_C$ consider average distances in the graph, the expected value of $\hat{c}_C(v)$ is equal to $c_C(v)$ for any K and v. This leads to the following algorithm:

1. Pick a set of K vertices $\{v_1, v_2, \ldots, v_K\}$ uniformly at random from V.
2. For each vertex $v \in \{v_1, v_2, \ldots, v_K\}$, solve the SSSP problem with that vertex as source.
3. For each vertex $v \in V$, compute $\hat{c}_C(v) = \frac{n}{K \cdot (n-1)} \sum_{i=1}^{K} d(v, v_i)$

We now recapitulate the result from [179] to compute the required number of sample vertices K that suffices to achieve the desired approximation. The result uses Hoeffding's Bound [299]:

Lemma 4.3.1. *If $x_1, x_2, \ldots, x_K$ are independent with $a_i \le x_i \le b_i$, and $\mu = E[\sum x_i/K]$ is the expected mean, then for $\xi > 0$*

$$Pr\left\{\left|\frac{\sum_{i=1}^{K} x_i}{K} - \mu\right| \ge \xi\right\} \le 2 \cdot e^{-2K^2\xi^2/\sum_{i=1}^{K}(b_i - a_i)^2}. \tag{4.3}$$

By setting x_i to $\frac{n \cdot d(v_i,u)}{n-1}$, μ to $c_C(v)$, a_i to 0, and b_i to $\frac{n\Delta}{n-1}$, we can bound the probability that the error of estimating $c_C(v)$ by $\hat{c}_C(v)$, for any vertex, is more than ξ:

$$\begin{aligned}
\Pr\left\{\left|\frac{\sum_{i=1}^{K} x_i}{K} - \mu\right| \ge \xi\right\} &\le 2 \cdot e^{-2K^2\xi^2/\sum_{i=1}^{K}(b_i - a_i)^2} && (4.4)\\
&= 2 \cdot e^{-2K^2\xi^2/K(\frac{n\Delta}{n-1})^2} && (4.5)\\
&= 2 \cdot e^{-\Omega(K\xi^2/\Delta^2)} && (4.6)
\end{aligned}$$

If we set ξ to $\epsilon \cdot \Delta$ and use $\Theta(\frac{\log n}{\epsilon^2})$ samples, the probability of having an error greater than $\epsilon \cdot \Delta$ is at most $1/n$ for every estimated value.

The running time of an SSSP algorithm is $\mathcal{O}(n+m)$ in unweighted graphs, and $\mathcal{O}(m + n\log n)$ in weighted graphs, yielding a total running time of $\mathcal{O}(K \cdot (n+m))$ and $\mathcal{O}(K(m + n\log n))$ for this approach, respectively. With K set to $\Theta(\frac{\log n}{\epsilon^2})$, this results in running times of $\mathcal{O}(\frac{\log n}{\epsilon^2}(n+m))$ and $\mathcal{O}(\frac{\log n}{\epsilon^2}(m + n\log n))$.

We now adapt this technique to the estimation of betweenness centrality in weighted and directed graphs. As before, a set of K sample vertices is randomly picked from V. For every source vertex v_i, we calculate the total dependency $\delta_{v_i\bullet}(v)$ (see Section 3.4.2) for all other vertices v, and sum them up. The estimated betweenness centrality $\hat{c}_B(v)$ is then defined as

$$\hat{c}_B(v) = \sum_{i=1}^{K} \frac{n}{K}\delta_{v_i\bullet}(v). \tag{4.7}$$

Again, the expected value of $\hat{c}_B(v)$ is equal to $c_B(v)$ for all K and v. For this new problem, we set x_i to $n \cdot \delta_{v_i\bullet}$, μ to $c_B(v)$, and a_i to 0. The total dependency $\delta_{v_i\bullet}(v)$ can be at most $n-2$ if and only if v is the only responsible vertex for all shortest paths leaving v_i. Thus, we set b_i to $n(n-2)$. Using the bound (4.3.1), it follows that the probability that the difference between the estimated betweenness centrality $\hat{c}_B(v)$ and the betweenness centrality $c_B(v)$ is more than ξ is

$$\Pr\{|\hat{c}_B(v) - c_B(v)| \geq \xi\} \leq 2e^{-2K^2\xi^2/K\cdot(n(n-2))^2} \tag{4.8}$$

$$= 2 \cdot e^{-2K\xi^2/(n(n-2))^2} \tag{4.9}$$

Setting ξ to $\epsilon(n(n-2))$, and the number of sample vertices K to $\Theta(\log n/\epsilon^2)$, the difference between the estimated centrality value and the correct value is at most $\epsilon n(n-1)$ with probability $1/n$. As stated above, the total dependency $\delta_{v_i\bullet}(v)$ of a vertex v_i can be calculated in $\mathcal{O}(n+m)$ in unweighted graphs and in $\mathcal{O}(m + n\log n)$ in weighted graph. With K set as above, this yields running times of $\mathcal{O}(\frac{\log n}{\epsilon^2}(n+m))$ and $\mathcal{O}(\frac{\log n}{\epsilon^2}(m + n\log n))$, respectively. Hence, the improvement over the exact betweenness algorithm in Section 4.2.1 is the factor K which replaces a factor n, for the number of SSSP-like computations.

Note that this approach can be applied to many centrality indices, namely those that are based on summations over some primitive term defined for each vertex. As such, those indices can be understood as taking a normalized average, which makes them susceptible to random vertex sampling.

4.3.2 Approximation of Web Centralities

Most of the approximation and acceleration techniques for computing Web-centralities are designed for the PageRank method. Therefore, in the following we concentrate on this method. A good short overview of existing acceleration PageRank techniques can be found in [378]. We distinguish the following acceleration approaches:

- approximation by cheaper computations, usually by avoiding matrix multiplications,
- acceleration of convergence,
- solving a linear system of equations instead of solving an eigenvector problem,
- using decomposition of the Web-graph, and
- updating instead of recomputations.

We discuss these approaches separately below.

Approximation by Cheaper Computations. In [148] and [149], Ding et al. report on experimental results indicating that the rankings obtained by both PageRank and Hubs & Authorities are strongly correlated to the in-degree of the vertices. This especially applies if only the top-20 query results are taken into consideration. Within the unifying framework the authors propose, the ranking by in-degree can be viewed as an intermediate between the rankings produced by PageRank and Hubs & Authorities. This result is claimed to also theoretically show that the in-degree is a good approximation of both PageRank and Hubs & Authorities. This seems to be true for graphs in which the rankings of PageRank and Hubs & Authorities are strongly related. However, other authors performed computational experiments with parts of the Web graph, and detected only little correlation between in-degree and PageRank, see, e.g., [463]. A larger scale study confirming the latter result can be found in [380].

Acceleration of Convergence. The basis for this acceleration technique is the power method for determining the eigenvector corresponding to the largest eigenvalue, see Section 4.1.5.

Since each iteration of the power-method consists of matrix multiplication, and is hence very expensive for the Web graph, the goal is to reduce the number of iterations. One possibility was proposed by Kamvar et al. [340] and extended by Haveliwala et al. [292]. In the first paper the authors propose a quadratic extrapolation that is based on the so-called Aitken Δ^2 method. The Aitken extrapolation assumes that an iterate $\boldsymbol{x}^{(k-2)}$ can be written as a linear combination of the first two eigenvectors $\boldsymbol{u}$ and $\boldsymbol{v}$. With this assumption, the next two iterates are linear combinations of the first two eigenvectors as well:

$$\begin{aligned} \boldsymbol{x}^{(k-2)} &&= \boldsymbol{u} + \alpha \boldsymbol{v} \\ \boldsymbol{x}^{(k-1)} &= A\boldsymbol{x}^{(k-2)} &= \boldsymbol{u} + \alpha\lambda_2 \boldsymbol{v} \\ \boldsymbol{x}^{(k)} &= A\boldsymbol{x}^{(k-1)} &= \boldsymbol{u} + \alpha\lambda_2^2 \boldsymbol{v}. \end{aligned}$$

By defining

$$y_i = \frac{\left(x_i^{(k-1)} - x_i^{(k-2)}\right)^2}{x_i^{(k)} - 2x_i^{(k-1)} + x_i^{(k-2)}}$$

and some algebraic reductions (see [340]) we get $\boldsymbol{y} = \alpha \boldsymbol{v}$ and hence

$$\boldsymbol{u} = \boldsymbol{x}^{(k-2)} - \boldsymbol{y}. \tag{4.10}$$

Note that the assumption that $\boldsymbol{x}^{(k-2)}$ can be written as a linear combination of $\boldsymbol{u}$ and $\boldsymbol{v}$ is only an approximation, hence (4.10) is also only an approximation of the first eigenvector, which is then periodically computed during the ordinary power method.

For the quadratic extrapolation the authors assume that an iterate $\boldsymbol{x}^{(k-2)}$ is a linear combination of the first three eigenvectors $\boldsymbol{u}$, $\boldsymbol{v}$ and $\boldsymbol{w}$. Using the characteristic polynomial they arrive at an approximation of $\boldsymbol{u}$ only depending on the iterates:

$$\boldsymbol{u} = \beta_2\boldsymbol{x}^{(k-2)} + \beta_1\boldsymbol{x}^{(k-1)} + \beta_0\boldsymbol{x}^{(k)}.$$

As in the Aitken extrapolation, this approximation is periodically computed during the ordinary power method. The authors report on computational experiments indicating that the accelerated power method is much faster than the ordinary power method, especially for large values of the damping factor d, for which the power method converges very slowly. As we discuss in Section 5.5.2, this is due to the fact that d equals the second largest eigenvalue (see [290]), hence a large value for d implies a small eigengap.

The second paper [292] is based on the ideas described above. Instead of having a linear combination of only two or three eigenvector approximations, the authors assume that $\boldsymbol{x}^{(k-h)}$ is a linear combination of the first $h+1$ eigenvector approximations. Since the corresponding eigenvalues are assumed to be the h-th roots of unity, scaled by d, it is possible to find a simple closed form for the first eigenvector. This acceleration step is used as above.

Kamvar et al. [338] presented a further idea to accelerate the convergence, based on the observation that the speed of convergence in general varies considerably from vertex to vertex. As soon as a certain convergence criteria is reached for a certain vertex, this vertex is taken out of the computation. This reduces the size of the matrix from step to step and therefore accelerates the power method.

The Linear System Approach. Each eigenvalue problem

$$A\boldsymbol{x} = \lambda\boldsymbol{x}$$

can be written as homogeneous linear system of equations

$$(A - \lambda I)\,\boldsymbol{x} = \boldsymbol{0}_n.$$

Arasu et al. [33] applied this idea to the PageRank algorithm and conducted some experiments with the largest strongly connected component of a snapshot of the Web graph from 1998. The most simple linear system approach for the PageRank system

$$(I - dP)\,\boldsymbol{c}_{\mathrm{PR}} = (1-d)\,\boldsymbol{1}_n$$

is probably the Jacobi iteration. But, as was mentioned in the description of the PageRank algorithm, the Jacobi iteration is very similar to the power method, and hence does not yield any acceleration.

Arasu et al. applied the Gauss-Seidel iteration defined by

$$c_{\mathrm{PR}}^{(k+1)}(i) = (1-d) + d \sum_{j<i} p_{ij} c_{\mathrm{PR}}^{(k+1)}(j) + d \sum_{j>i} p_{ij} c_{\mathrm{PR}}^{(k)}(j).$$

For $d = 0.9$, their experiments on the above described graph are very promising: the Gauss-Seidel iteration converges much faster than the power iteration. Arasu et al. then combine this result with the fact that the Web graph has a so-called *bow tie* structure. The next paragraph describes how this structure and other decomposition approaches may be used to accelerate the computations.

Decomposition Techniques. Since the Web graph is very large, and grows larger every day, some researchers propose to decompose the graph. So, it is possible to determine centrality values in smaller components of the Web in a first step, and to adjust them to the complete Web graph in the second step, if necessary. As noted above, Arasu et al. [33] exploit the observation of Broder et al. [102] that the Web graph has a so-called *bow tie* structure, see Figure 4.1 and Section 15.3.2. Note that the Web crawl of Broder et al. was carried out in 1999, and it is not clear whether the web structure has changed since.

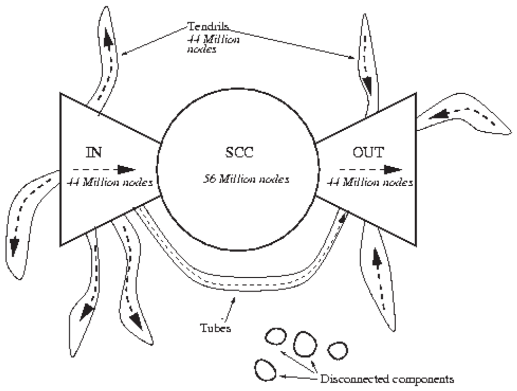

Fig. 4.1. Bow tie structure of the Web graph (from `http://www9.org/w9cdrom/160/160.html`)

This structure may be used for the power method, but the authors claim that it is especially well suited for the linear system approach, since the corresponding link-matrix has the *block upper triangular* form:

$$P = \begin{pmatrix} P_{11} & P_{12} & P_{13} & \dots & P_{1K} \\ 0 & P_{22} & P_{23} & \dots & P_{2K} \\ \vdots & \ddots & P_{33} & \dots & P_{3K} \\ \vdots & & \ddots & \ddots & \vdots \\ 0 & \dots & \dots & 0 & P_{KK} \end{pmatrix}.$$

By partitioning $\boldsymbol{c}_{\mathrm{PR}}$ in the same way, the large problem may be solved by the following sequence of smaller problems

$$(I - dP_{KK})\,\boldsymbol{c}_{\mathrm{PR},K} = (1-d)\mathbf{1}_{n_K}$$

$$(I - dP_{ii})\,\boldsymbol{c}_{\mathrm{PR},i} = (1-d)\mathbf{1}_{n_i} + d \sum_{j=i+1}^{K} P_{ij}\boldsymbol{c}_{\mathrm{PR},j}$$

A second approach was proposed by Kamvar et al. [339]. They investigated, besides a smaller partial Web graph, a Web crawl of 2001, and found the following interesting structure:

1. There is a block structure of the Web.
2. The individual blocks are much smaller than the entire Web.
3. There are nested blocks corresponding to domains, hosts and subdirectories within the path.

Algorithm 7: PageRank exploiting the block structure: BlockRank

```
1. For each block $I$,
   compute the local PageRank scores $c_{\mathrm{PR(I)}}(i)$ for each vertex $i \in I$
2. Weight the local PageRank scores
   according to the importance of the block the vertices belongs to
3. Apply the standard PageRank algorithm
   using the vector obtained in the first two steps
```

Based on this observation, the authors suggest the three-step-algorithm 7. In the first and third step the ordinary PageRank algorithm can be applied. The question is how to formalize the second step. This is done via a *block graph* B where each block I is represented by a vertex, and an edge (I, J) is part of the block graph if there exists an edge (i, j) in the original graph satisfying $i \in I$ and $j \in J$, where (i, j) may be a loop. The weight ω_{IJ} associated with an edge (I, J) is computed as the sum of edge weights from vertices $i \in I$ to $j \in J$ in the original graph, weighted by the local PageRank scores computed from Step 1:

$$\omega_{IJ} = \sum_{i \in I, j \in J} a_{ij} c_{\mathrm{PR(I)}}(i).$$

If the local PageRank vectors are normalized using the 1-norm, then the weight matrix $\Omega = (\omega_{IJ})$ is a stochastic matrix, and the ordinary PageRank algorithm can be applied to the block graph B to obtain the block weights b_I.

The starting vector for Step 3 is then determined by

$$c^{(0)}_{\mathrm{PR}}(i) = c_{\mathrm{PR(I)}}(i) b_I \ \forall\ I, \forall\ i \in I.$$

Another decomposition idea was proposed by Avrachenkov and Litvak [367] who showed that if a graph consists of several connected components (which is obviously true for the Web graph), then the final PageRank vector may be computed by determining the PageRank vectors in the connected components and combining them appropriately using the following theorem.

Theorem 4.3.2.

$$\boldsymbol{c}_{PR} = \left(\frac{|V_1|}{|V|} \boldsymbol{c}_{PR(1)}, \frac{|V_2|}{|V|} \boldsymbol{c}_{PR(2)}, \ldots, \frac{|V_K|}{|V|} \boldsymbol{c}_{PR(K)}, \right),$$

where $G_k = (V_k, E_k)$ are the connected components, $k = 1, \ldots, K$ and $\boldsymbol{c}_{PR(k)}$ is the PageRank vector computed for the kth connected component.

Finally, we briefly mention the 2-step-algorithm of Lee et al. [383] that is based on the observation that the Markov chain associated with the PageRank matrix is *lumpable.*

Definition 4.3.3. *If $\mathcal{L} = \{L_1, L_2, \ldots, L_K\}$ is a partition of the states of a Markov chain P then P is* lumpable *with respect to $\mathcal{L}$ if and only if for any pair of sets $L, L' \in \mathcal{L}$ and any state i in L the probability of going from i to L' doesn't depend on i, i.e. for all $i, i' \in L$*

$$\Pr[X_{t+1} \in L' | X_t = i] = \sum_{j \in L'} p_{ij} = \Pr[X_{t+1} \in L' | X_t = i'] = \sum_{j \in L'} p_{i'j} \ .$$

The common probabilities define a new Markov chain, the lumped chain P_L *with state space $\mathcal{L}$ and transition probabilities $p_{LL'} = \Pr[X_{t+1} \in L' | X_t \in L]$.*

The partition the authors use is to combine the dangling vertices (i.e., vertices without outgoing edges) into one block and to take all dangling vertices as singleton-blocks. This is useful since the number of dangling vertices is often much larger than the number of non-dangling vertices (a Web crawl from 2001 contained 290 million pages in total, but only 70 million non-dangling vertices, see [339]). In a second step, the Markov chain is transformed into a chain with all non-dangling vertices combined into one block using a state aggregation technique.

For the lumped chain of the first step, the PageRank algorithm is used for computing the corresponding centrality values. For the second Markov chain, having all non-dangling vertices combined, the authors prove that the algorithm to compute the limiting distribution consists of only three iterations (and one Aitken extrapolation step, if necessary, see Section 4.3.2). The vectors obtained in the two steps are finally concatenated to form the PageRank score vector of the original problem.

4.4 Dynamic Computation

In Section 4.3.2, several approaches for accelerating the calculation of page importance were described. In this section, we focus on the ‘on the fly’ computation of the same information, and on the problem of keeping the centrality values up-to-date in the dynamically changing Web.

4.4.1 Continuously Updated Approximations of PageRank

For the computation of page importance, e.g. via PageRank, the link matrix has to be known in advance. Usually, this matrix is created by a crawling process. As this process takes a considerable amount of time, approaches for the ‘on the fly’ computation of page importance are of interest. Abiteboul et al. [1] describe the ‘On-line Page Importance Computation’ (OPIC) algorithm, which computes an approximation of PageRank, and does not require to store the possibly huge link matrix.

The idea is based on the distribution of ‘cash.’ At initialization, every page receives an amount of cash and distributes this cash during the iterative computation. The estimated PageRank can then be computed directly from the current cash distribution, even while the approximation algorithm is still running.

Algorithm 8 describes the OPIC algorithm. The array c holds the actual distribution of cash for every page, and the array h holds the history of the cash for every page. The scalar g is just a shortcut for $\sum_{i=1}^{n} h[i]$.

An estimate of the PageRank of page i is given by $\boldsymbol{c}_{\text{PRapprox}}(i) = \frac{h[i]+c[i]}{g+1}$. To guarantee that the algorithm calculates a correct approximation of PageRank, the selection of the vertices is crucial. Abiteboul et al. discuss three strategies: random, greedy, and circular. The strategies ‘randomly select a page’ and ‘circularly select all pages’ are obvious. Greedy selects the page with the highest cash. For the convergence of the computation, the selection of the vertices has to be fair, and this has to be guaranteed in all selection strategies.

After several iterations the algorithm converges towards the page importance information defined by the eigenvector for the largest eigenvalue of the adjacency matrix of the graph. To guarantee the convergence of the calculation similar concepts as for the random surfer (see Section 3.9.3) have to be applied. These are, for example, the inclusion of a ‘virtual page’ that every page links upon. The original work contains an adaptive version that covers link additions and removals, and in some parts vertex additions and removals. This modified adaptive OPIC algorithm is not discussed here, and can be found in [1].

4.4.2 Dynamically Updating PageRank

An interesting approach to accelerate the calculation of page importance lies in the recomputation of the PageRank for the ‘changed’ part of the network only. In case of the Web these changes are page additions and removals and link additions and removals. For this idea, Chien et al. [124] described an approach for link additions.

Algorithm 8: OPIC: On-line Page Importance Computation

```
Input: The graph $G$
Output: $c$ and $h$: arrays for cash and history, $g$: sum of the history values

Initialization
for $i \leftarrow 1$ to $n$ do
    $c[i] \leftarrow 1/n$
    $h[i] \leftarrow 0$
$g \leftarrow 0$

repeat
    choose a vertex $i$ from $G$
    See text for vertex selection strategies

    Update the history of $i$
    $h[i] \leftarrow h[i] + c[i]$

    Distribute the cash from $i$ to children
    for each child $j$ of $i$ do
        $c[j] \leftarrow c[j] + c[i]/d^+[i]$

    Update the global history value
    $g \leftarrow g + c[i]$

    Reset cash for $i$
    $c[i] \leftarrow 0$
until hell freezes over
```

The idea is founded on an observation regarding the perturbation of the probability matrix P of the PageRank Markov chain for the Web graph W. This perturbation, stemming from link additions, can be modeled by the relation $P = \tilde{P} + E$, where E is an error matrix and $\tilde{P}$ is the perturbed matrix. For a single edge addition[1], E contains only changes in some row i. Therefore, the matrix $\tilde{P}$ differs from the original matrix P only in this row. Chien et al. observed that the recomputation of PageRank is required for a small area around the perturbation to achieve a good approximation for the modified Web graph W'. This small area is defined by the graph structure and can be extracted from the original Web graph W. The extraction yields a graph G that contains the new edge between i and j, and further every vertex and edge which are 'near' to the new edge. Additionally, the graph G contains a 'supervertex' that models all vertices from the graph W that are not in G. A transition matrix T for the graph G is constructed, and its stationary distribution τ is calculated.

For all vertices of the graph G (except for the supervertex), the stationary distribution $\tilde{\pi}$ of the perturbed matrix $\tilde{P}$ can, therefore, be approximated by the stationary distribution τ of the matrix T. For the vertices in W that are not covered by G, the stationary distribution $\tilde{\pi}$ of $\tilde{P}$ is simply approximated by the stationary distribution π of the matrix P. Several experiments showed that

[1] In the original work a description for the single edge case is given and extended towards multiple edge changes. We only cover the single edge case here.

this approach gives a good approximation for the modified Web graph W', and that the computation time decreases due to the computation of the stationary distribution of the smaller matrix T instead of $\tilde{P}$.