Clique Partitions, Graph Compression and Speeding-up Algorithms

Tomás Feder [*]

Rajeev Motwani [†]

IBM Almaden Research Center
650 Harry Road
San Jose, CA 95120

Computer Science Department
Stanford University
Stanford, CA 94305-2140

Abstract

We first consider the problem of partitioning the edges of a graph $\mathcal{G}$ into bipartite cliques such that the total order of the cliques is minimized, where the order of a clique is the number of vertices in it. It is shown that the problem is NP-complete. We then prove the existence of a partition of small total order in a sufficiently dense graph and devise an efficient algorithm to compute such a partition. It turns out that our algorithm exhibits a trade-off between the total order of the partition and the running time. Next, we define the notion of a compression of a graph $\mathcal{G}$ and use the result on graph partitioning to efficiently compute an optimal compression for graphs of a given size. An interesting application of the graph compression result arises from the fact that several graph algorithms can be adapted to work with the compressed representation of the input graph, thereby improving the bound on their running times, particularly on dense graphs. This makes use of the trade-off result we obtain from our partitioning algorithm. The algorithms analyzed include those for matchings, vertex connectivity, edge connectivity and shortest paths. In each case, we improve upon the running times of the best-known algorithms for these problems.

[*]Supported by a Bell Scholarship. Part of this work was done while the author was at Stanford University.

[†]Supported by NSF Grant CCR-9010517 and grants from the Mitsubishi Corporation and OTL.

1 Introduction

We introduce the notion of a *compression* $\mathcal{G}^*$ of a graph $\mathcal{G}$. The compressed representation of a graph encodes some aspects of the structure of the original graph and has the following properties:

(a) $\mathcal{G}^*$ is a graph with m^* edges, where m^* is smaller than the number of edges in $\mathcal{G}$.

(b) it is computationally easy to convert $\mathcal{G}$ into $\mathcal{G}^*$, and vice versa.

The compression may be viewed as a data structure for representing the graph $\mathcal{G}$. We show that several graph algorithms run faster when adapted to work with the compressed representation of the input graph. Thus, we achieve a speed-up over the running times of the best-known algorithms for graph problems such as matchings, vertex connectivity, edge connectivity and all-pairs shortest paths. To efficiently construct such compressions we study a hard optimization problem, viz. *partition into bipartite cliques*, which is interesting in its own right. The goal is to minimize the sum of the orders of the bipartite cliques, where the order of a graph is the number of vertices in it. We show that computing the partition of minimum total order is NP-complete. However, it is established that there exists a reasonably good partition for sufficiently dense graphs, and that it can be computed efficiently. Using such partitions, we show how to compute a compression of a graph which is optimal in size.

Turan [Tur] had previously studied the problem of succinctly representing an unlabeled graph. His results were confined to the case of *planar graphs* and were mainly of theoretical interest. The representation obtained was optimal in size but could not be used to speed-up algorithms. His work left open the problem of finding a succinct representation of general graphs, and this was recently solved by Naor [Nao]. Our notion of a succinct representation of a graph is stronger. Our results are not only concerned with representing a graph using the fewest possible number of bits, but also with finding representations that do not obscure the structure present in the graph so as to enable an efficient implementation of a large class of algorithms. Our representation uses fewer bits in the case of sparse graphs unlike the earlier results and our results are derived for the case of *labeled* graphs.

We first describe the results obtained for the graph partition problem. A *bipartite clique* is a complete bipartite graph, and its *order* is the number of vertices in it. The order of a collection of bipartite cliques is the sum of the orders of the individual cliques. We establish that the problem of computing a minimum order partition is NP-complete. However, we can show that every sufficiently dense graph has a large bipartite clique as a subgraph and that this clique can be computed efficiently. This allows us to show that a graph with n vertices and m edges can be partitioned into bipartite cliques of total order $p(\mathcal{G}) = \mathrm{O}(\frac{m \log \frac{n^2}{m}}{\log n})$. Notice that if $m = \Omega(n^{2-\epsilon})$ then $p(\mathcal{G}) = \mathrm{O}(\epsilon m)$; further, when $m = \Omega(n^2)$ then $p(\mathcal{G}) = \mathrm{O}(\frac{m}{\log n})$. Compare this with the trivial upper bound of $\mathrm{O}(m)$ on $p(\mathcal{G})$. We present an efficient algorithm to compute such a partition. Our algorithm demonstrates an interesting trade-off between the order of the partition computed and the running time required.

The compression $\mathcal{G}^*$ of a graph $\mathcal{G}$ is constructed via the partition into bipartite cliques. Each clique in the partition is replaced by a tree, thereby reducing the number of edges used in the new representation. However, the path structure of the graph $\mathcal{G}$ is totally maintained in $\mathcal{G}^*$. This is precisely what enables us to use the new representation as an input to algorithms which are concerned only with the path structure of $\mathcal{G}$.

The compressed representation of a graph can be looked upon as a data structure for storing the graph. This data structure has the property that it can efficiently support reachability queries with respect to the original graph. The cost of these queries, when amortized over a long sequence, turns out to be significantly less than the cost of answering queries in the original graph directly. (Of course, the cost of computing the compression also needs to be amortized over a sequence of such queries.) This helps in improving the performance of algorithms which make a large number of reachability tests in the input graph, notably algorithms for special cases of flow problems where the queries correspond to a search for an augmenting path.

Our algorithmic results are as follows. Let k denote the ratio $\frac{m}{m^*}$ which is the compression factor achieved by our algorithm. Then, we speed-up the running times of the bipartite matching algorithm (Hopcroft-Karp [HK] or Dinic [Din]) by a factor of k. This is achieved by modifying these algorithms to take advantage of the new representation. Similar speed-ups are attained for the vertex connectivity algorithms of Even-Tarjan [ET] and Galil [Gal], the edge connectivity algorithm of Matula [Mat] and Mansour-Schieber [MS], and the all-pairs shortest paths algorithm. Also, there has been some recent work on vertex-connectivity [CT, NI] which involves the computation of so-called sparse certificates for k-connectivity. Our results can be combined with these to obtain further speed-ups in the case of dense graphs. As for the value of k, it turns out to be roughly $\log n$ provided the original graph is dense enough.

The speed-ups in the running times are obviously fairly small. However, our approach involves a fairly simple and elegant notion of compression. It is therefore surprising that these ideas are sufficient to improve upon the long-standing bound of $\mathrm{O}(\sqrt{n}m)$ on the running time of the matching algorithm. We believe that the underlying idea may lead to further and more significant improvements in the running times. The compression that we compute retains all the information present in the original graph. This seems wasteful and should be modified to a representation which only retains some essential information that allows certain types of queries to be efficiently supported. This approach of massaging the input itself into an efficient data structure is quite different from the usual manner of using data structures to improve the performance of graph algorithms. On the other hand, better solutions to the graph partition problem might in themselves provide an improvement in the running time of the algorithms considered here.

The remaining sections are organized as follows. In Section 2 we study the clique partition problem and present the results mentioned above. Section 3 is devoted to the development of the compression algorithm based on the clique partition results. Finally, in Section 4 we apply all these results to the graph algorithms mentioned earlier and demonstrate how they can be modified to improve their running times.

2 Clique Partitions of Bipartite Graphs

Let $\mathcal{B}(U, V, E)$ be a bipartite graph with the vertex sets $U = \{u_1, \ldots, u_n\}$ and $V = \{v_1, \ldots, v_n\}$, and the edge set E with $|E| = m$. A *bipartite clique* in $\mathcal{B}$ is a complete bipartite subgraph. A *clique partition* for $\mathcal{B}$ is a collection of bipartite cliques $\mathcal{C} = \{C_1, C_2, \ldots C_p\}$ such that edge sets $E(C_1), \cdots$, $E(C_p)$ form a partition of the edge set E. Holyer [Hol] has shown that the problem of partitioning the edges of any graph into a minimum number of *non-bipartite* cliques is NP-complete. We adapt his proof to obtain the following theorem.

Theorem 2.1 *The problem of verifying that the minimum order partition into bipartite cliques for any given graph $\mathcal{G}(V, E)$ has total order k or less is NP-complete.*

Proof: We will use the notation and ideas from [Hol]; the reader should refer to that paper for details. Consider the special case of Holyer's result which shows that the problem EP_3 is NP-complete. This is the problem of deciding whether the edges of a (non-bipartite) graph can be partitioned into K_3's. We first modify that proof to show that deciding whether a bipartite graph can be partitioned into $K_{2,2}$'s (length four cycles) is NP-complete.

For handling EP_3, Holyer defined a class of graphs called $H_{3,p}$ which can be thought of as a tiling of the torus by triangles. He then reduced 3-SAT to EP_3 by using one copy of $H_{3,p}$ for each variable and clause literal in the formula F. These were then patched together so as to ensure that F is satisfiable if and only if the resulting graph F^* can be partitioned into K_3's.

Let G_p be the graph corresponding to a $p \times p$ grid embedded on the torus. This is a 4-regular bipartite graph. We replace the use of $H_{3,p}$ by G_p in the construction due to Holyer. It is routine to verify that the same proof gives us the NP-completeness of checking whether the graph F^* obtained from the 3-SAT formula F can be partitioned into $K_{2,2}$'s.

A key property of F^* is that any two vertices have at most two neighbors in common. This implies that the maximal cliques in the graph are either $K_{2,2}$ or $K_{1,q}$, for some value q. Thus, a minimum order partition into bipartite cliques must have total order at least m, where m is the number of edges in F^*. In fact, this minimum value will only be achieved there is a partition into $K_{2,2}$'s. Thus, if we could test whether a bipartite clique has a partition of total order m or less, then we would be able to check whether F^* has a partition into $K_{2,2}$'s and hence decide the satisfiability of the boolean formula F. This implies the desired result

□

In the remaining sections we devise an efficient algorithm for finding the promised clique partition of $\mathcal{B}$. In Section 2.1 we demonstrate that $\mathcal{B}$ must have a reasonably large clique if m is large enough, and in Section 2.2 we provide an efficient algorithm to find such a clique. Finally, in Section 2.3 we give an algorithm to compute the partition $\mathcal{C}$. From here on the cliques will be understood to refer to bipartite cliques.

The problem of demonstrating the existence of a large bipartite clique in a sufficiently dense bipartite graph has received some attention in the literature on Ramsey Theory [GRS, ES]. The problem was first posed by Zarankiewicz [Zar] and subsequently several existential results were obtained, particularly for the case where the bipartite clique has the same number of vertices on each side of the bipartition. Similarly, the size of a *covering* of the edges of a graph by bipartite cliques has been considered before [Tuz]. Our result on the existence of a large clique (Theorem 2.2) in a bipartite graph could have been obtained by similar techniques. However, our main goal is to obtain an efficient *construction* of a bipartite clique and thence of a clique partition. Towards this end, we use a non-standard approach in our proof and this helps considerably in obtaining a constructive result.

2.1 Large Cliques in Dense Graphs

An (α, β)-clique in $\mathcal{B}$ is a subgraph of $\mathcal{B}$ which is a complete bipartite graph on vertex sets $A \subseteq U$ and $B \subseteq V$, such that $|A| = \alpha$ and $|B| = \beta$.

Definition 2.1 *Let δ be any constant such that $0 \leq \delta \leq 1$, and let $k(n, m, \delta) = \left\lfloor \frac{\delta \log n}{\log \frac{2n^2}{m}} \right\rfloor$. A δ-clique in $\mathcal{B}$ is defined to be an (α, β)-clique with $\alpha = \lceil n^{1-\delta} \rceil$ and $\beta = k$.*

Note that the value of k depends upon the density of the graph $\mathcal{B}$ but we will not mention this explicitly in what follows. Also, all logarithms in this paper will be with base 2.

The following theorem proves that there exists a δ-clique in every graph. First, let us build up some notation. For any vertex x and vertex set X, we will denote by $\Gamma(x)$ and $\Gamma(X)$ their neighborhood sets. The degrees of the vertices $u_i \in U$ are denoted by $d_i = |\Gamma(u_i)|$. An *ordered subset* of a set S will be written as $\vec{X} \subseteq S$. Given an ordered set $\vec{X}$, we will sometimes write X if the ordering of its elements is irrelevant. The notation $s^{\underline{k}}$ will be used to denote the product $s(s-1)(s-2)\ldots(s-k+1)$. Thus, if $|S| = s$, then the number of distinct ordered subsets of S of size k is exactly $s^{\underline{k}}$. Note that when $s < k$, then the product $s^{\underline{k}}$ equals 0, as it should.

Theorem 2.2 *Every bipartite graph $\mathcal{B}(U, V, E)$ contains a δ-clique.*

Proof: We will demonstrate the existence of an ordered set $\vec{K} \subset V$ such that $|\vec{K}| = k$ and there are at least $n^{1-\delta}$ vertices $u \in U$ with $K \subset \Gamma(u)$. This proof remains essentially the same if we work with an *unordered* set K. However, as we will see later, the constructive version of the proof is much simplified while working with ordered sets.

Given a fixed ordered set $\vec{K} \subset V$, let $\mathcal{N}_1(K)$ denote the number of vertices $u \in U$ such that $K \subseteq \Gamma(u)$. Similarly, given a fixed vertex $u \in U$, let $\mathcal{N}_2(u)$ denote the number of distinct ordered subsets $\vec{K} \subset V$ such that $|K| = k$ and $K \subseteq \Gamma(u)$. Define C_k as follows,

$$C_k = \sum_{\substack{\vec{K} \subset V \\ |K| = k}} \mathcal{N}_1(K) = \sum_{u_i \in U} \mathcal{N}_2(u_i) = \sum_{u_i \in U} d_i^{\underline{k}} \tag{1}$$

In effect, C_k is the number of *ordered* $(1, k)$-cliques in $\mathcal{B}$, i.e. bipartite cliques where we impose an ordering on the vertices from V.

By the Pigeon-hole Principle, there must exist at least one ordered set $\vec{K} \subset V$ of size k such that $\mathcal{N}_1(K) \geq C_k / n^{\underline{k}}$. We claim that $C_k = \sum_{u_i \in U} d_i^{\underline{k}}$ is minimized when the degrees d_i are nearly-equal, i.e. when each $d_i = \lfloor \frac{m}{n} \rfloor$ or $\lceil \frac{m}{n} \rceil$. Given the claim, we have that

$$\begin{aligned} \mathcal{N}_1(K) &\geq \frac{C_k}{n^{\underline{k}}} = \frac{\sum_{u_i \in U} d_i^{\underline{k}}}{n^{\underline{k}}} \\ &\geq \frac{n \cdot (\lfloor \frac{m}{n} \rfloor)^{\underline{k}}}{n^{\underline{k}}} \geq \frac{n \cdot (\frac{m}{n} - k)^k}{n^k} \end{aligned}$$

Since $k(n, m, \delta) \leq \frac{\delta \log n}{\log \frac{2n^2}{m}} \leq \frac{m}{2n}$, it follows that $\mathcal{N}_1(K) \geq n \cdot (\frac{m}{2n^2})^k \geq n^{1-\delta}$. This proves that there exists an (ordered) δ-clique.

It remains to prove our claim that C_k is minimized when the d_i's are nearly-equal. We first show that for $b > a$,

$$a^{\underline{k}} + b^{\underline{k}} \geq (a+1)^{\underline{k}} + (b-1)^{\underline{k}}.$$

Note that this is trivially true for $a \leq k - 1$, since then $a^{\underline{k}} = 0$. Assume now that $b > a \geq k$, then

$$\begin{aligned} \left(a^{\underline{k}} + b^{\underline{k}}\right) - \left((a+1)^{\underline{k}} + (b-1)^{\underline{k}}\right) &= \\ \left((a-k+1)a^{\underline{k-1}} + b(b-1)^{\underline{k-1}}\right) - & \\ \left((a+1) \cdot a^{\underline{k-1}} + (b-k) \cdot (b-1)^{\underline{k-1}}\right) &= \\ k \cdot \left((b-1)^{\underline{k-1}} - a^{\underline{k-1}}\right) &\geq 0 \end{aligned}$$

where the last inequality follow from the fact that $b - 1 \geq a$.

It is now clear that the sum $C_k = \sum_{u_i \in U} d_i^{\underline{k}}$ will only decrease if we reduce the largest d_i by 1 and increase the smallest d_i by 1, provided they differ by 2 or more. Repeated application of this argument yields the desired claim. □

2.2 Clique Stripping

We now present an algorithm for *clique stripping* - removing a δ-clique from the bipartite graph $\mathcal{B}$. We may look upon the proof of Theorem 2.2 as a *probabilistic proof of existence* of a large clique in a dense graph. From this view-point, the following construction uses a technique which is reminiscent of the *method of conditional probabilities* [Spe, Rag, MNN] that has been used earlier to convert probabilistic existence proofs into polynomial time constructions. For our applications, it is not sufficient to have a polynomial-time construction; we require the polynomial to be of low degree since this algorithm will be used as a preprocessor for graph algorithms of relatively low running time.

The idea behind the algorithm is to perform a binary search for each element, in order, of the ordered set $\vec{K}$ whose existence is demonstrated in the proof of Theorem 2.2. To better understand this search procedure, let us generalize the counting argument embodied in (1). Let $\mathcal{C}$ denote some collection of ordered subsets of V, where each subset is of size k. Define $\mathcal{N}_1(K, U')$ as the number of vertices $u \in U' \subseteq U$ such that $K \subseteq \Gamma(u)$; similarly, define $\mathcal{N}_2(u, \mathcal{C})$ as the number of ordered sets $\vec{K} \in \mathcal{C}$ such that $K \subseteq \Gamma(u)$. Clearly,

$$C(U', \mathcal{C}) \triangleq \sum_{\vec{K} \in \mathcal{C}} \mathcal{N}_1(K, U') = \sum_{u \in U'} \mathcal{N}_2(u, \mathcal{C}). \tag{2}$$

In the definition of C_k, equation (1), we had $U' = U$ and $\mathcal{C}$ included every ordered subset of V of size k.

The aim of the binary search procedure is to find a particular ordered set $\vec{K}$ such that $\mathcal{N}_1(K, U) \geq C(U, \mathcal{C})/|\mathcal{C}|$, i.e. the average value of $\mathcal{N}_1(K, U)$ over $\mathcal{C}$. This would yield a δ-clique since Theorem 2.2 guarantees that the average value is at least $n^{1-\delta}$. The search proceeds by partitioning $\mathcal{C}$ into $\mathcal{C}_0$ and $\mathcal{C}_1$, and then computing the two average values $C(U, \mathcal{C}_0)/|\mathcal{C}_0|$ and $C(U, \mathcal{C}_1)/|\mathcal{C}_1|$. The search then recurses on the sub-collection which has a higher average value of $\mathcal{N}_1$, this must be at least $n^{1-\delta}$. The only problem with this search procedure is that it is not obvious that we can efficiently compute $C(U, \mathcal{C})$. However, an appropriate choice of the partition at each stage and a use of the equation (2) will allow us to overcome this obstacle.

Assume that $n = 2^r$. This will be convenient for the description of our binary search over V, though we can deal with arbitrary n easily by finding near-equal partitions or adding enough dummy vertices to make n a power of 2. We will associate any s-bit string w, such that $0 \leq s \leq r$, with a subset $V_w \subseteq V$ of size $n/2^s$, as follows. The empty string ϵ will index the set $V_\epsilon = V$. Given a set V_w of size $n/2^s$, the set $V_{w \cdot 0}$ ($V_{w \cdot 1}$) will contain the $n/2^{s+1}$ smallest-numbered (resp. largest-numbered) vertices in V_w. Thus, for a fixed value s, the strings of length s will index a collection of sets which are each of size $n/2^s$ and form a partition of V. For each vertex $u_i \in U$, define $d_{i,w} = |\Gamma(u_i) \cap V_w|$, i.e. the number of neighbors of u_i in V_w. An efficient implementation of the *clique stripping* algorithm will require the notion of *neighborhood trees*.

Definition 2.2 *The* neighborhood tree T_i *for a vertex* $u_i \in U$ *is a labeled binary tree of depth* r. *The nodes of this tree are labeled with bit strings* w, *where* $0 \leq |w| \leq r$. *The labeling of a node encodes the path to it from the root in the natural fashion (with a 0 representing a left turn and a 1 representing a right turn). A node labeled* w *is associated with the set* V_w *and we store* $d_{i,w}$ *at that* *node.*

Note that it is not necessary that a neighborhood tree be a complete binary tree. In our application, a node w may not be present unless $d_{i,w} > 0$. We will describe the construction of these trees in the next section.

The search procedure will construct an ordered set $\vec{K} = (x_1, \ldots x_k) \subseteq V$ in stages, determining x_i in stage i. At stage t of the search, the first $t-1$ elements of $\vec{K}$ will have been determined to be some $y_1, \ldots y_{t-1} \in V$. The remaining elements must now belong to the set $V_t = V - \{y_1, \ldots y_{t-1}\}$. Let $U_t = \{u \in U \mid y_1, \ldots y_{t-1} \in \Gamma(u)\}$, we are omitting vertices from U which are not adjacent to each of the first $t-1$ choices of the elements in $\vec{K}$. Denote by $\mathcal{B}_t$ the subgraph of $\mathcal{B}$ induced by the vertex sets U_t and V_t. The search procedure will work with new graph $\mathcal{B}_t$ and will update the neighborhood trees for each $u_i \in U_t$ to correspond to the structure of $\mathcal{B}_t$.

The determination of x_t, during stage t, will proceed by successively restricting the choice of x_t to a set V_w, where initially $w = \epsilon$ and its length is increased by 1 at each step. At a general step of this stage, the choice of x_t would have been restricted to some V_w with $|w| = s$. Denote by $\mathcal{C}_{t,w}$ the collection of all ordered sets $\vec{K} \subset V$ of size k such that $x_t \in V_w$ and, for $1 \le j \le t-1$, $x_j = y_j$. The binary search will further restrict x_t by computing $c_0 = C(U_t, \mathcal{C}_{t,w\cdot 0})$ and $c_1 = C(U_t, \mathcal{C}_{t,w\cdot 1})$, choosing the next bit of w depending on whether c_0 or c_1 is larger. Note that we do not need to worry about taking the average since

$$|\mathcal{C}_{t,w\cdot 0}| = |\mathcal{C}_{t,w\cdot 1}| = \frac{n}{2^{s+1}} \cdot (n-t)^{\underline{k-t}}$$

The next lemma shows that c_0 and c_1 can be computed efficiently.

Lemma 2.1 *For any w, with $0 \le |w| \le r$,*

$$C(U_t, \mathcal{C}_{t,w}) = \sum_{u_i \in U_t} d_{i,w} \cdot (d_i - 1)^{\underline{k-t}}$$

and this expression can be computed in $\mathrm{O}(nk)$ *time.*

Proof: Let us first validate the expression given above. From (2) we have that,

$$C(U_t, \mathcal{C}_{t,w}) = \sum_{\vec{K} \in \mathcal{C}_{t,w}} \mathcal{N}_1(K, U_t) = \sum_{u_i \in U_t} \mathcal{N}_2(u_i, \mathcal{C}_{t,w})$$

We may compute each term of the rightmost expression as

$$\mathcal{N}_2(u_i, \mathcal{C}_{t,w}) = d_{i,w} \cdot (d_i - 1)^{\underline{k-t}}$$

This is because $\mathcal{N}_2$ counts the number of ordered sets in $\mathcal{C}_{t,w}$ all of whose elements are adjacent to u_i. The first $t-1$ elements of all the sets in $\mathcal{C}_{t,w}$ are fixed and are adjacent to u_i by the definition of U_t. The t^{th} element must belong to V_w and therefore lies in $\Gamma(u_i) \cap V_w$ which is of cardinality is $d_{i,w}$ (recall that the neighborhood tree T_i has been updated to reflect the fixing of $y_1, \ldots y_{t-1}$).

The remaining $k - t$ elements of an ordered set adjacent to u_i may be chosen arbitrarily from the remaining $d_i - 1$ neighbors of u_i.

Assume that the search procedure keeps a pointer for each neighborhood tree, pointing to the node labeled by the current value of w. Thus, for each $u_i \in U_t$, the expression for $\mathcal{N}_2$ can be computed in O(k) time.

□

The *Clique Stripping Algorithm* is presented below. We will assume that the algorithm is initially provided with the neighborhood trees T_i, at the end the neighborhood trees will be modified to reflect the removal of the δ-clique.

Clique Stripping Algorithm:

Step 1. Initialize a pointer to the root of each neighborhood tree T_i.

Step 2. $t \leftarrow 1$; $U_t \leftarrow U$; $\mathcal{B}_t \leftarrow \mathcal{B}$.

Step 3. Perform stage t of the binary search.

Step 3.1. $w \leftarrow \epsilon$;

Step 3.2. Compute c_0 and c_1 as described in Lemma 2.1 using the pointers to node w in each T_i.

$$c_0 \leftarrow C(U_t, \mathcal{C}_{t,w\cdot 0}) = \sum_{u_i \in U_t} d_{i,w\cdot 0} \cdot (d_i - 1)^{\underline{k-t}}$$

$$c_1 \leftarrow C(U_t, \mathcal{C}_{t,w\cdot 1}) = \sum_{u_i \in U_t} d_{i,w\cdot 1} \cdot (d_i - 1)^{\underline{k-t}}$$

Step 3.3. **IF** $c_0 \geq c_1$ **THEN** $w \leftarrow w \cdot 0$
ELSE $w \leftarrow w \cdot 1$.

Step 3.4. Update each T_i's pointer to the node labeled by the new value of w.

Step 3.5. **IF** $|w| < r$ **THEN** GOTO Step 3.2 **ELSE** $y_t \leftarrow v$, where $V_w = \{v\}$.

Step 4. $V_{t+1} \leftarrow V_t - \{y_t\}$;
$U_{t+1} \leftarrow \{u \in U_t \mid y_t \in \Gamma(u)\}$;
$\mathcal{B}_{t+1}$ is the graph induced by the vertex sets U_{t+1} and V_{t+1}.

Step 5. Update the neighborhood trees T_i to correspond to $\mathcal{B}_{t+1}$. This is done by subtracting 1 from each d-value stored at nodes on the path from the root to the leaf labeled w. Also, initialize the tree pointers to point to the root of each tree.

Step 6. **IF** $t < k$ **THEN** $t \leftarrow t + 1$; GOTO Step 3.

A δ-clique may be obtained from the set $\vec{K} = \{y_1, \ldots y_k\}$ computed by this algorithm, as follows. Let $U_K = \{u_i \in U \mid K \subseteq \Gamma(u_i)\}$, the bipartite set of vertices U_K and K induce a bipartite clique in $\mathcal{B}$.

Theorem 2.3 *The Clique Stripping Algorithm computes a δ-clique in time* $\mathrm{O}(nk^2 \log n)$*, not including the time required to construct the initial neighborhood trees.*

Proof: We first prove the correctness of this algorithm by showing that $|U_K| \geq n^{1-\delta}$. To show this, we establish by induction on t that at the start of the t^{th} stage of the algorithm

$$C(U_t, \mathcal{C}_{t,\epsilon})/|\mathcal{C}_{t,\epsilon}| \;\geq\; n^{1-\delta} \tag{3}$$

Observe that from the proof of Theorem 2.2 it follows that

$$C(U_1, \mathcal{C}_{1,\epsilon}) \;=\; C(U, \mathcal{C}_{1,\epsilon}) \;\geq\; n^{1-\delta}$$

establishing the basis of our induction.

Suppose now that the inductive statement (3) is true at the start of some stage t. During stage t, in Step 3 of the algorithm, we successively partition $\mathcal{C}_{t,w}$ into two *equal* parts, viz. $\mathcal{C}_{t,w\cdot 0}$ and $\mathcal{C}_{t,w\cdot 1}$. We then choose to extend w to $w \cdot 0$ or $w \cdot 1$, depending upon which maximizes the expression $C(U_t, \mathcal{C}_{t,w})$. Thus, at the end of stage t, we have that

$$\frac{C(U_t, \mathcal{C}_{t,w})}{|\mathcal{C}_{t,w}|} \geq n^{1-\delta}$$

This implies that the inductive statement (3) is true at the start of stage $t+1$ since $C(U_{t+1}, \mathcal{C}_{t+1,\epsilon})$ is exactly the same as $C(U_t, \mathcal{C}_{t,w})$, for the final value of w in stage t.

At the end of the last stage, $\mathcal{C}_{t,w}$ contains only the ordered set $\vec{K}$ which is the output of the algorithm. This in turn implies that U_K has at least $n^{1-\delta}$ vertices.

The running time of the algorithm is determined as follows. The time required by Step 1 is $\mathrm{O}(n)$, not accounting for the time required to construct the neighborhood trees initially. Steps 2 and 6 are executed only once at each stage and require constant time for each execution. Each execution of Step 4 requires $\mathrm{O}(n)$ time, and it is executed once at each stage, for a total time requirement of $\mathrm{O}(nk)$. Similarly, each execution of Step 5 requires $\mathrm{O}(n \log n)$ time, for a total time requirement of $\mathrm{O}(nk \log n)$. Finally, in Step 3 the running-time is dominated by Step 3.2 which requires $\mathrm{O}(nk)$ time. This step is executed r times in each stage, for a total running time of $\mathrm{O}(nrk^2)$. Since $r = \log n$, this implies the desired result.
$\square$

2.3 The Clique Partition Algorithm

In this section we present the algorithm for constructing the clique partition $\mathcal{C}$ of a graph $\mathcal{B}$. This algorithm will repeatedly invoke the *Clique Stripping Algorithm* to strip off cliques from $\mathcal{B}$. The algorithm terminates when the density of the graph $\mathcal{B}$ falls below the threshold required to find a non-trivial clique, viz. when $m < 2n^{2-\delta}$, or $k(n, m, \delta) = 0$.

The *Clique Stripping Algorithm* needs to be provided with the neighborhood trees to be able to compute the δ-clique. If the neighborhood trees were to be recomputed after each stripping, the total cost of tree computation would dominate the cost of the entire partition algorithm. Therefore, the partition algorithm computes the neighborhood trees only once initially, and the *Clique Stripping Algorithm* ensures that they are updated to reflect the removal of each clique. As the following lemma shows, the initial computation of the neighborhood trees can be performed efficiently.

Lemma 2.2 *The n neighborhood trees for the vertices in U can be constructed in* $\mathrm{O}(m \log n)$ *time.*

Proof: It suffices to show that the i^{th} tree, T_i, can be constructed in $\mathrm{O}(d_i \log n)$ time. The following algorithm will traverse one root-leaf path for each edge incident on u_i, implying the desired result.

Initially, the tree T_i will consist of only the root, labeled ϵ, with $d_{i,\epsilon} = 0$ stored there. For each edge (u_i, v) in turn, the algorithm will start at the root and traverse a path to a leaf node (at level r), always taking a branch to a node labeled w such that $v \in V_w$. If the particular node labeled w does not exist, then it is created and the value $d_{i,w} = 1$ is stored there; otherwise, the value of $d_{i,w}$ is incremented by 1. It is easy to verify that our definitions imply that the set of nodes labeled w such that $v \in V_w$ form a root-leaf path. It follows that after all the d_i edges incident at u_i have been processed, the values stored at the nodes of the tree T_i will be correct.

□

The input to the partition algorithm is a bipartite graph $\mathcal{B}(U, V, E)$ with $|U| = |V| = n$ and $|E| = m$. It outputs the clique partition $\mathcal{C}$.

Algorithm Partition:

Step 1. Initialize: $i \leftarrow 0$; $n \leftarrow |U|$; $\widehat{m} \leftarrow |E|$.

Step 2. Compute the neighborhood trees, as described in Lemma 2.2.

Step 3. **WHILE** $\widehat{m} \geq n^{2-\delta}$ **DO**

- **Step 3.1.** Start i^{th} stage: $i \leftarrow i + 1$; $\widehat{k} \leftarrow \left\lfloor \frac{\delta \log n}{\log \frac{2n^2}{\widehat{m}}} \right\rfloor$.
- **Step 3.2.** Using the *Clique Stripping Algorithm* and the neighborhood trees, determine sets of vertices $K \subseteq V$ and $U_K \subseteq U$ which form a δ-clique in $\mathcal{B}$. The i^{th} δ-clique C_i has the vertex sets $U_i \leftarrow U_K$ and $V_i \leftarrow K$. The neighborhood trees are modified by the *Clique Stripping Algorithm* to reflect the removal of all edges in $U_K \times K$.
- **Step 3.3.** Update $\mathcal{B}$ as follows: $E \leftarrow E - (U_i \times V_i)$; $\widehat{m} \leftarrow |E|$.

Step 4. The remaining edges in E are partitioned into cliques consisting of a single edge each.

Note that $\widehat{m}$ is used to refer to the number of edges remaining in $\mathcal{B}$ at any point during the execution of the algorithm; similarly, $\widehat{k}$ refers to $k(n, \widehat{m}, \delta)$. Thus, m and k refer to the original values of these two quantities as computed from the input graph. The following theorem results.

Theorem 2.4 *Let δ be any constant such that $0 < \delta < 1$, and let $\mathcal{B}(U, V, E)$ be any bipartite graph with $|U| = |V| = n$ and $|E| = m \geq n^{2-\delta}$. Then,* Algorithm Partition *partitions the edges of $\mathcal{B}$ into edge-disjoint cliques of total order $p(\mathcal{B}) = \mathrm{O}\left(\frac{m}{k(n,m,\delta)}\right)$ in time $\mathrm{O}(mn^{\delta}\log^2 n)$.*

Proof: We first analyze the running time of the algorithm. Observe that besides Steps 2 and 3.2, the remaining steps in the algorithm take time $\mathrm{O}(m)$ over all the iterations. By Lemma 2.2, Step 2 can be performed in time $\mathrm{O}(m \log n)$. Consider now an execution of Step 3.2 with some fixed value of $\widehat{m}$. By Theorem 2.3, a clique stripping takes time $\mathrm{O}(n\widehat{k}^2 \log n)$, where $\widehat{k} = k(n, \widehat{m}, \delta)$, and it removes at least $\widehat{k}n^{1-\delta}$ edges from $\mathcal{B}$. This implies that, for the fixed value of $\widehat{m}$, the removal of each edge takes on the average $\mathrm{O}(n^{\delta}\widehat{k} \log n)$ time. Since $\widehat{k} = \mathrm{O}(\log n)$, each edge's removal takes $\mathrm{O}(n^{\delta}\log^2 n)$ time on the average. Since all the invocations of Step 3.2 can together remove at most m edges, we have that the total running time of this step over all iterations is $\mathrm{O}(mn^{\delta}\log^2 n)$. Thus, the entire algorithm terminates within the stated bound.

Let $p(\mathcal{B})$ denote the order of the partition computed by this algorithm. We will assume that initially $p(\mathcal{B})$ is 0, and as the algorithm progresses its value is incremented by the order of the cliques being stripped off. To estimate $p(\mathcal{B})$, we will divide the iterations of the algorithm into stages. The t^{th} stage will include the iterations which occur after the first time when the number of edges remaining in $\mathcal{B}$ falls below $\frac{m}{2^{t-1}}$, and before the first time when the number of remaining edges first falls below $\frac{m}{2^t}$. The cliques whose removal causes the number of edges to fall from $\frac{m}{2^{t-1}}$ to $\frac{m}{2^t}$ are the cliques of stage t. During stage t, the value of $\widehat{k}$ will always be at least

$$k_t = \frac{\delta \log n}{2 \log \frac{2^t \cdot 2n^2}{m}}$$

Consider a clique C_i belonging to this stage, with the vertex sets $K \subseteq V$ and $U_K \subseteq U$. The stripping of this clique will cause the removal of $|K| \cdot |U_K|$ edges from $\mathcal{B}$, and it will cause an increment of $|K| + |U_K|$ to the value of $p(\mathcal{B})$. Thus, the average value added to $p(\mathcal{B})$, for each edge removed from $\mathcal{B}$ during C_i's stripping is

$$\frac{|K| + |U_K|}{|K| \cdot |U_K|} = \left(\frac{1}{|K|} + \frac{1}{|U_K|}\right) \leq \left(\frac{1}{k_t} + \frac{1}{n^{1-\delta}}\right)$$

where the last inequality follows from the fact that $|K| = \widehat{k} \geq k_t$ during stage t, and $|U_K| \geq n^{1-\delta}$ by the definition of a δ-clique. Since the total number of edges removed from $\mathcal{B}$ during stage t cannot exceed $\frac{2m}{2^t}$, it follows that the total increment in $p(\mathcal{B})$ during stage t cannot exceed

$$\left(\frac{1}{k_t} + \frac{1}{n^{1-\delta}}\right) \frac{2m}{2^t}$$

Let M be the total increment in $\mathcal{B}^*$ before the number of edges in $\mathcal{B}$ falls below $n^{2-\delta}$ and the algorithm terminates. Then,

$$\begin{aligned} M &\leq \sum_{t=1}^{\lceil \log \frac{m}{n^{2-\delta}} \rceil} \left(\frac{1}{k_t} + \frac{1}{n^{1-\delta}} \right) \frac{2m}{2^t} \\ &\leq \sum_{t=1}^{\infty} \left(\frac{1}{k_t} + \frac{1}{n^{1-\delta}} \right) \frac{2m}{2^t} \end{aligned}$$

Noting that $\frac{1}{2k_t} \leq \frac{1}{k} + \frac{t}{\delta \log n}$ we have that

$$\begin{aligned} M &\leq \frac{4m}{k} \sum_{t=1}^{\infty} \frac{1}{2^t} + \frac{4m}{\delta \log n} \sum_{t=1}^{\infty} \frac{t}{2^t} + \frac{2m}{n^{1-\delta}} \sum_{t=1}^{\infty} \frac{1}{2^t} \\ &\leq 2m \left(\frac{2}{k} + \frac{4}{\delta \log n} + \frac{1}{n^{1-\delta}} \right) \\ &= \mathrm{O}\left(\frac{m}{k} \right) \end{aligned}$$

Finally, we have to deal with the $2n^{2-\delta}$ edges remaining in $\mathcal{B}$ at the end of the algorithm that are also added to $\mathcal{B}^*$. However, for all m, it is clear that $n^{2-\delta} = \mathrm{O}\left(\frac{m}{k}\right)$. Partitioning these into the trivial cliques (singleton edges) establishes the desired bound on $p(\mathcal{B})$.
□

Observe the trade-off between the running time of the partition algorithm and the order of the partition into cliques.

3 Graph Compressions

We now formally define a graph compression and show how the clique partitions can be used to compute such a compression. Let $\mathcal{G}(V, E)$ be a labeled graph with $|V| = n$ and $|E| = m$. We define a *compression* of $\mathcal{G}$ as some labeled graph $\mathcal{G}^*(V^*, E^*)$ such that

1. $n^* = |V^*|$ is polynomial in n.
2. $m^* = |E^*|$ is significantly smaller than m, i.e. $m^* = \mathrm{o}(m)$.
3. The mapping $* : \mathcal{G} \to \mathcal{G}^*$ is 1-1.

The compression $*$ is called efficient if there exists an efficient (polynomial-time) algorithm to convert from $\mathcal{G}$ to $\mathcal{G}^*$, and *vice versa*.

For the moment, we are only concerned with providing an efficient compression for a bipartite graph $\mathcal{B}$. Our compression of $\mathcal{B}$ is a tripartite graph $\mathcal{B}^*(U, V, W, E^*)$, where U and V are as before while W contains some additional vertices. To compress a graph $\mathcal{B}$, we first partition its edges into a collection of edge-disjoint cliques. Let the i^{th} clique in the partition, C_i, be a clique on the vertices in $U_i \subseteq U$ and $V_i \subseteq V$. We may now obtain $\mathcal{B}^*$ from $\mathcal{B}$ as follows. For each clique C_i in the partition, introduce a vertex $w_i \in W$. The only edges in E^* are the edges which connect each w_i to the vertices in the sets U_i and V_i. Observe that we have replaced $|U_i| \times |V_i|$ edges by only $|U_i| + |V_i|$ edges. If the clique C_i contains only one edge (u_i, v_i) then we need not introduce the vertex w_i and can place the the edge (u_i, v_i) directly into E^*. It is easy to see that there is an efficient (linear-time) algorithm for reconstructing $\mathcal{B}$ given its compression $\mathcal{B}^*$. Similarly, given the partition into cliques, the process for obtaining $\mathcal{B}^*$ takes only linear time.

The following theorem details the quality of the compression that can be achieved using the results from the previous section. Here *Algorithm Compress* refers to a modified version of *Algorithm Partition* which constructs a new tripartite graph $\mathcal{B}^*(U, V, W, E^*)$ as described above, using the cliques computed by the partition algorithm.

Theorem 3.1 *Let δ be any constant such that $0 < \delta < 1$, and let $\mathcal{B}(U, V, E)$ be any bipartite graph with $|U| = |V| = n$ and $|E| = m \geq n^{2-\delta}$. Then,* Algorithm Compress *computes a compression $\mathcal{B}^*(U, V, W, E^*)$ of $\mathcal{B}$ with $|E^*| = m^* = \mathrm{O}(\frac{m}{k(n,m,\delta)})$ in time $\mathrm{O}(mn^\delta \log^2 n)$. The number of additional vertices introduced in W is at most $\mathrm{O}(\frac{m}{n^{1-\delta}})$.*

Observe the trade-off between the running time of the compression algorithm and the size of the compressed graph. Since m^* is linear in $\frac{1}{\delta}$ and the running time is exponential in δ, it makes sense to pick δ to be some small positive constant. The next theorem shows that the compression obtained above is optimal within constant factors. Note that it does not assume anything about the structure of the compressed graph or the time required to compute it.

Theorem 3.2 *There does not exist any compression $\mathcal{G}(V^*, E^*)$ of bipartite graphs $\mathcal{B}(U, V, E)$ on given sets U, V with $|U| = |V| = n$ and $|E| = m$, such that $|V^*|$ is polynomial in n and $|E^*| = \mathrm{o}\left(\frac{m}{k(n,m,\delta)}\right)$.*

Proof: The proof is by a simple information-theoretic argument. Consider any compression with $n^* \leq n^c$ for a constant $c > 0$. Suppose that this compression scheme uses at most m^* edges when compressing bipartite graphs on m edges. Using the fact that the number of image graphs must be larger than the number of graphs with m vertices, we have the following inequality.

$$\binom{n^2}{m} \leq \sum_{M=0}^{m^*} \binom{n^{2k}}{M}$$

Straight-forward algebraic manipulation now gives the desired lower bound on the value of m^*.
□

We now show that the compression algorithm can be extended to the case of non-bipartite and undirected graphs. Let $\mathcal{G}(V, E)$ be a directed graph with n vertices and m edges. We first represent $\mathcal{G}$ as a bipartite graph $\mathcal{B}(L, R, E')$ such that each vertex $u \in V$ has two counterparts $u_L \in L$ and

$u_R \in R$. For each directed edge (u,v) in E we introduce the edges (u_L, v_R) into E', directed from L to R. Additionally, we have directed edges going from u_R to u_L, for each $u \in V$. The bipartite graph containing only the edges directed from L to R is compressed exactly as described above, and is then augmented by the n additional edges. The fact that the path structure of the compressed graph is essentially faithful to the original graph is sufficient to allow certain algorithms to work successfully on the new graph. To handle undirected graphs is easy, we just replace each edge between vertices u and v by directed edges from u to v and vice versa.

Theorem 3.3 *Let δ be any constant such that $0 < \delta < 1$, and let $\mathcal{G}(V,E)$ be any (undirected or directed) graph with $|V| = n$ and $|E| = m \geq n^{2-\delta}$. Then,* Algorithm Compress *computes a compression $\mathcal{G}^*(L,R,W,E^*)$ of $\mathcal{G}$ with $|E^*| = m^* = \mathrm{O}(\frac{m}{k(n,m,\delta)})$ in time $\mathrm{O}(mn^\delta \log^2 n)$. The number of additional vertices introduced in W is $\mathrm{O}(\frac{m}{n^{1-\delta}})$.*

4 Speeding-Up Algorithms

In this section we apply the above results to improving the running times of various graph algorithms. The basic idea in each analysis is to use the compression of a graph as an input to the algorithm. Thus, we may think of the compression algorithm as a preprocessor for these graph algorithms. In some cases the original algorithm can be used without any modifications, although their analysis has to be adapted to the new class of inputs. In some other cases we actually have to modify the algorithms in a non-trivial fashion to fully exploit the compressed representation and to ensure the algorithm's correctness.

The results stated below assume the deterministic unit-cost RAM model. In fact, since all the problems are concerned with unweighted graphs, these results can actually be obtained in a logarithmic-cost RAM model too.

Throughout, we will assume that $\mathcal{G}(V,E)$ is a graph with $|V| = n$ and $|E| = m$, and $\mathcal{B}(U,V,E)$ is a bipartite graph with $|U| = |V| = n$ and $|E| = m$. Letting $\kappa(n,m) = \frac{\log n}{\log \frac{n^2}{m}}$, it follows that $k(n,m,\delta) = \mathrm{O}(\kappa(n,m))$ for any fixed δ such that $0 < \delta < 1$.

4.1 All-pairs Shortest Paths

Our compressed representation maintains all the path information in the original graph. Thus, it is not surprising that the problem of computing all-pairs shortest paths in an unweighted graph is an easy application of our technique. The best known bound on finding all-pairs shortest paths is $\mathrm{O}(nm)$ and the algorithm is simply to find a bfs (breadth-first search) tree rooted at each vertex. Using fast matrix multiplication [Fre, Rom] it is possible to improve this bound if only the lengths of the shortest paths are desired. We are concerned with finding the paths themselves and an efficient representation is in terms of the bfs-trees.

Theorem 4.1 *The unweighted all-pairs shortest paths problem can be solved in time* $\mathrm{O}\left(\frac{nm}{\kappa(n,m)}\right)$.

Proof: We describe the result for directed graphs, and this includes as a special case the application to undirected graphs. Assume that a directed graph G has been compressed in the manner described earlier, and also that the compressed graph contains reverse edges (directed from u_R to u_L) for the two copies of any vertex u in the original graph.

Consider an edge from x to y in the original graph. In the compressed graph, this edge will be represented by a path of length 3 with the vertices x_L, z, y_R and y_L, where z is a vertex from the set W. Consider a bfs-tree rooted at any vertex u_L in the compressed representation. We claim that the path from u_L to any vertex v_L is exactly three times longer than the corresponding path from u to v in the original graph. This is because any edge of the shortest path from u to v in the original graph is represented by the length three path described above. It is not very hard to see that this gives a representation of the bfs-tree which can be converted into a bfs-tree of the original graph in linear time.
$\square$

4.2 Matching Algorithms

Consider first the bipartite matching algorithm. We will work with the adaptation of Dinic's algorithm to the instance of the 0-1 flow problem which models the problem of finding a maximum matching in a bipartite graph. In the following theorem we show that for sufficiently dense graphs our techniques lead to a speed-up of this algorithm's running time by a factor of $\log n$. Recently, Cheriyan-Hagerup-Mehlhorn [CHM] have devised a bipartite matching algorithm which runs in time $\mathrm{O}(n^{2.5}/\log n)$. While this matches the running time of our algorithm in the worst case, note that their time bound is independent of the number of edges in the input graph. Our algorithm will have a significantly smaller running time on graphs with $o(n^2)$ edges. Moreover, their results are obtained by an exploitation of the unit-cost RAM model to perform bit-packing and table look-ups using the ability to process logarithmic size words in unit time. In contrast, our results are purely combinatorial and relatively independent of the model of computation. Our approach is to perform graph compression and then run the standard algorithms on the compressed graphs without any exploitation of the quirks in the model of computation.

Theorem 4.2 *Dinic's algorithm finds a maximum matching in the compression of* $\mathcal{B}$ *in* $\mathrm{O}\left(\frac{\sqrt{n}m}{\kappa(n,m)}\right)$ *time.*

Proof: We first compute a compression of $\mathcal{B}$ with $\delta < \frac{1}{2}$. Note that this can be done in time $o(\sqrt{n}m)$ time and the cost of compression is then negligible compared to the cost of computing the matching itself. The compressed graph $\mathcal{B}^*$ is then converted into an instance of a 0-1 flow problem, such that a maximum flow in the network corresponds to a maximum matching in the original graph $\mathcal{B}$. Moreover, given such a flow, the matching can be computed in $\mathrm{O}(n)$ time. The flow problem is obtained by introducing a source s with an out-going edge to every vertex on the left (U), and a sink t with an in-coming edge from each vertex on the right (V); further, all edge capacities are set to 1. We claim that Dinic's algorithm finds a maximum flow in this network within the given time bound, or in time $O(\sqrt{n}m^*)$. The claim is based on a minor modification of the analysis for bipartite matchings as described by Even [Ev2, Section 6.1]; we briefly review the main ideas from there.

The original analyis observes that the instance of 0-1 flow problem arising out of the application to bipartite matchings has the *type 2* property that, besides the source and the sink, every vertex either has in-degree 1 or out-degree 1. Consider any flow of value M in this network, and observe that it can be decomposed into vertex-disjoint paths from s to t. Since there are M such vertex-disjoint paths, it is easy to see that the layered network obtained from this flow has length $O(n/M)$ [Ev2, Lemma 6.4]. Moreover, it is the case that the residual network with respect to any feasible flow inherits the type 2 property [Ev2, Lemma 6.5]. A simple calculation [Ev2, Theorem 6.3] now yields the desired bound on the running time of this algorithm.

We now describe how this analysis can be adapted to the compressed representation of the input graph. The flow instance obtained from the compressed graph does not have the type 2 property, but a slight weakening of this property is still applicable. Observe that any flow in the resulting network can be decomposed into edge-disjoint paths from s to t such that every vertex in $U \cup V$ occurs on at most one path. Moreover, along these paths no two consecutive vertices can be from W. Thus the length of the path is twice the number of vertices from $U \cup V$ on it, and these paths do not share the vertices from $U \cup V$. There are $O(n)$ vertices in $U \cup V$ and each of the M paths alternates between these vertices and those from W. Thus, we still obtain a bound of $O(n/M)$ on the length of these paths when the flow value is M. Finally, the residual graph with respect to any feasible flow still has the above structure since we will never introduce an edge between two vertices from W. This allows us to conclude that the number of stages of flow augmentation is still bounded by $O(\sqrt{n})$, even though the number of vertices in the new graph may be significantly more than n. Since each stage of flow augmentation can be implemented in $O(m^*)$ time, the result follows.

□

The above result does not require any modification to the algorithm. Another way of achieving the same result is to work with the algorithm of Hopcroft & Karp [HK] which is essentially equivalent to Dinic's algorithm. However, in this algorithm we avoid the flow formulation and work directly with matchings. This algorithm consists of a number of stages of augmentation, where each stage computes a maximal collection of disjoint augmenting paths with respect to the current matching. To adapt this algorithm to our compressed representation, we can easily modify the search for these augmenting paths in such a way that we can still guarantee that each stage runs in time linear in the number of edges (now only m^*).

The $O(m^*\sqrt{n})$ time bound can be extended (by means of appropriate data structures) to the case where $\mathcal{G}^*$ is any representation for $\mathcal{G}$ with the property that adjacency in $\mathcal{G}$ corresponds to reachability in $\mathcal{G}^*$ (in essence, $\mathcal{G}$ is the transitive closure of $\mathcal{G}^*$). (See [Fed].) It is in fact this result that prompted the usefulness of recognizing bipartite cliques, as the simplest graph that can be represented by means of reachability in a smaller graph.

4.3 Edge Connectivity

We now consider the problem of computing the edge connectivity of a graph. The fastest algorithm known for this problem is due to Matula [Mat] and it has running time $\mathrm{O}(nm)$. A recent result of Gabow [Gab] shows that the edge connectivity of can be determined in time $O(km\log\frac{n^2}{m})$ for directed graphs, and $O(m+k^2 n\log\frac{n}{k})$ for undirected graphs, where k is connectivity. We show that using our compressed representation the running time of Matula's algorithm may be improved by a factor of $\log n$ for dense graphs. (This yields an improvement over Gabow's algorithm only in the case where k is very close to n, essentially $k=\Omega(n/\log n)$.) Our results also apply to the case of directed graphs, speeding up the algorithm of Mansour and Schieber [MS].

We only describe the application of our ideas to undirected graph connectivity, in particular to speeding up Matula's algorithm. In this case, the algorithm cannot be used directly on our compressed representation since it computes arbitrary 0-1 flows in stages, and the residual graph is different at each stage. We present a modification to the compression strategy to take this into account.

Theorem 4.3 *The modified version of Matula's algorithm can compute the edge connectivity of $\mathcal{G}$ in $\mathrm{O}\left(\frac{nm}{\kappa(n,m)}\right)$ time.*

Proof: Matula's edge connectivity algorithm for a graph $\mathcal{G}$ operates in phases. The running time of each phase is dominated by the time required for a maximum flow computation on a network with unit capacities, obtained from the graph $\mathcal{G}$ by orienting each edge of $\mathcal{G}$ in both directions, and by selecting a source and multiple sinks. If the flow value for a phase is K, then an $O(Km)$ time bound for the flow computation can be obtained by running an algorithm that sends one unit of flow at a time from the source to a sink, taking $O(m)$ time per unit of flow. This bound is then used to prove an $O(nm)$ time bound for the entire algorithm.

In order to obtain an $O(nm^*)$ bound, we must show that the flow computation at each phase can be performed in $O(Km^*)$ time, or $O(m^*)$ time per unit of flow. To send the first unit of flow, we just determine reachability in $\mathcal{G}$ from the source to a sink. This can be done by traversing the compressed representation of $\mathcal{G}$, in $O(m^*)$ time. More generally, the ith unit of flow is sent by determining reachability in the residual graph $\mathcal{G}'_i$ obtained from $\mathcal{G}$ after the first $i-1$ units of flow have been sent. Unfortunately, we cannot afford the computational cost of finding a compressed representation for each such $\mathcal{G}'_i$. Instead, we only compute this compressed representation for certain

values of i, those of the form $i_r = \lfloor rn^\epsilon \rfloor$ with $r \geq 1$ integer. For the remaining values of i, the flow computation is performed by (explicitly) keeping track of the (small) discrepancy between $\mathcal{G}'_i$ and the latest $\mathcal{G}'_{i_r}$ whose compressed representation is known; the number of edges in this discrepancy is at most n per unit of flow, hence at most $n \cdot n^\epsilon < m^*$. This makes it possible to send each unit of flow in $O(m^*)$ time, and to amortize the $O(m^* n^\delta \log^2 n)$ cost of finding compressed representations for the $\mathcal{G}_{i_r}$ over the n^ϵ values of i that separate two consecutive values i_r and i_{r+1}, provided that $\epsilon > \delta$. This gives an amortized $O(m^*)$ bound per unit of flow, an $O(Km^*)$ bound per phase, and an $O(nm^*)$ bound for the entire edge connectivity algorithm.
$\square$

4.4 Vertex Connectivity

Consider now the problem of computing the vertex connectivity of a graph. Let $c_\mathcal{G}$ denote the connectivity of the graph $\mathcal{G}$. The algorithm of Even-Tarjan [ET] computes vertex connectivity in time $\mathrm{O}(c_\mathcal{G} n^{1.5} m)$. We improve this result as follows. (Note that there is a significant improvement in the running time only for very large value of connectivity, at least $n^{1-o(1)}$.)

Theorem 4.4 *The modified version of the Even-Tarjan algorithm can compute the vertex connectivity of $\mathcal{G}$ in $\mathrm{O}\left(\frac{c_\mathcal{G} n^{1.5} m}{\kappa(n,m)}\right)$ time.*

Note that $c_\mathcal{G} = \mathrm{O}(\frac{m}{n})$ and the running time can be bounded independent of the connectivity. The algorithm repeatedly solves a 0-1 flow problem in a network derived from the original graph by selecting different source-sink pairs. The flow problem has the additional constraint that the vertex capacities are 1, implying that at most one unit of flow can be routed through each vertex. This implies that the network can be viewed as a type 2 network without edge capacities, and our analysis for the case of bipartite matchings can now be used directly. We can obtain a compressed representation of this network using our compression of $\mathcal{G}$. Thus, we speed up the solution of the flow problem exactly as in the case of the bipartite matching problem. There is a slightly better algorithm for vertex connectivity due to Galil [Gal] which is based on an idea due to Even [Ev1]. In a like manner, we can speed up this algorithm to obtain the following theorem.

Theorem 4.5 *The modified version of Galil's algorithm can compute the vertex connectivity of $\mathcal{G}$ in $\mathrm{O}\left(\frac{\max\{c_\mathcal{G}, \sqrt{n}\} c_\mathcal{G} \sqrt{n} m}{\kappa(n,m)}\right)$ time.*

Recently, Cheriyan-Thurimella [CT] and Nagamochi-Ibaraki [NI] have obtained results on vertex connectivity which appear to be in a similar spirit. They show that every graph has a subgraph with $\mathrm{O}(kn)$ edges such that the original graph is k-connected if and only if the subgraph is k-connected. Such a subgraph is called a *sparse k-certificate*, and Nagamochi-Ibaraki provide an $\mathrm{O}(m)$ time algorithm to compute a sparse certificate. Clearly, checking k-connectivity in the certificate is more efficient when $m \geq kn$. Running Galil's algorithm on the sparse certificate yields a k-connectivity algorithm which runs in time $\mathrm{O}\left(\max\{c_\mathcal{G}, \sqrt{n}\} {c_\mathcal{G}}^2 n^{1.5}\right)$. It is possible to speed-up even this certificate-based algorithm using our compressed representation, as specified in the following theorem. Once again, the speed-up is significant only for large values of k.

Theorem 4.6 *The modified version of Galil's algorithm (which first picks a sparse k-certificate, and then compresses it) can check for k-vertex connectivity of $\mathcal{G}$ in* $\mathrm{O}\left(\frac{\max\{c_{\mathcal{G}}, \sqrt{n}\} {c_{\mathcal{G}}}^2 n^{1.5}}{\kappa(n, nk)}\right)$ *time.*

5 Further Work

A natural direction for further work is to obtain a linear time algorithm for the compression problem. This would require a more efficient version of our clique partition algorithm. One way of obtaining a better speed-up in the algorithmic applications would be to perform a many-one compression, unlike our result which gives a one-one compression. While this will result in the loss of information about the structure of the original graph, the remaining information may be sufficient to solve certain problems on the compressed representation. A good example of this is the recent work on vertex k-connectivity [CT, NI]. There a many-one compression was obtained and it retained enough information to verify the k-connectivity of the original graph. Subsequent to our work, Agarwal, Alon, Aronov and Suri [AAAS] have used a similar approach to construct compressions of graphs arising in geometric situations (visibility graphs). These graphs have a special structure which enables substantially improved compressions, thereby providing a greater speed-up for certain geometric algorithms.

A very interesting question which arises out of our work is whether the graph matching problem can be solved in substantially less time than $\mathrm{O}(\sqrt{n}m)$. A possible approach towards this goal would be to find an efficient many-one compression of a bipartite graph which does not reduce the size of the maximum matching. A less ambitious goal would be find a small subgraph of a bipartite graph which contains a matching of size within a factor c of the size of the maximum matching in the original graph. Note that $c = \frac{1}{2}$ can be achieved in linear time by just picking out a maximal matching, and in fact any constant c can be achieved in linear time by running a constant number $\lceil c/(1-c) \rceil$ of augmenting phases of a matching algorithm.

Acknowledgements

We would like to thank Dan Greene for his valuable suggestions during this work. We would also like to express our gratitude to the STOC-91 committee members for providing references to related work.

References

[AAAS] Pankaj K. Agarwal, Noga Alon, Boris Aronov and Subash Suri, "Can Visibility Graphs Be Represented Compactly?" *Proceedings of the Ninth Annual Symposium on Computational Geometry*, 1993, pp. 338-347.

[CHM] J. Cheriyan, T. Hagerup and K. Mehlhorn, "Fast and Simple Network Flow Algorithms," Preprint, 1990.

[CT] J. Cheriyan and R. Thurimella, "Algorithms for Parallel k-Vertex Connectivity and Sparse Certificates," *Proceedings of the 23rd ACM Symposium on Theory of Computing*, 1991.

[Din] E. A. Dinic, "Algorithm for solution of a problem of maximum flow in a network with power estimation," *Soviet Math. Doklady*, vol. 11 (1970), pp. 1277-1280.

[ES] P. Erdos and J. H. Spencer, *Probabilistic Methods in Combinatorics.* Academic Press, 1974.

[Ev1] S. Even, "An algorithm for determining whether the connectivity of a graph is at least k," *SIAM Journal on Computing*, vol. 4 (1975), pp. 393-396.

[Ev2] S. Even, *Graph Algorithms*, Computer Science Press (1979).

[ET] S. Even and R. E. Tarjan, "Network flow and testing graph connectivity," *SIAM Journal on Computing*, vol. 4 (1975), pp. 507-518.

[Fed] T. Feder, "2-Satisfiability and Network Flow," to appear in *Algorithmica.*

[Fre] M. L. Fredman, "New bounds on the complexity of the shortest path problems," *SIAM Journal on Computing*, vol. 5 (1976), pp. 83-89.

[Gab] H. Gabow, "A matroid approach to finding edge connectivity and packing arborescences," *Proceedings of the 23rd ACM Symposium on Theory of Computing*, 1991, pp. 112-124.

[Gal] Z. Galil, "Finding the Vertex Connectivity of Graphs," *SIAM Journal on Computing*, vol. 9 (1980), pp. 197-199.

[GRS] R. L. Graham, B. L. Rothschild and J. H. Spencer, *Ramsey Theory.* John Wiley & Sons, 1980.

[Hol] I. Holyer, "The NP-completeness of some edge partitioning problems," *SIAM Journal on Computing*, vol. 10 (1981), pp. 713-717.

[HK] J. E. Hopcroft and R. M. Karp, "An O($n^{5/2}$) algorithm for maximum matchings in bipartite graphs," *SIAM Journal on Computing*, vol. 2 (1973), pp. 225-231.

[Mat] D. W. Matula, "Determining Edge Connectivity in O(nm)," *Proceedings of the 28th IEEE Symposium on Foundations of Computer Science*, (1987), pp. 249-251.

[MNN] R. Motwani, J. Naor and M. Naor, "The probabilistic method yields deterministic parallel algorithms," *30th Annual Symposium on the Foundations of Computer Science*, 1989.

[MS] Y. Mansour and B. Schieber, "Finding the edge connectivity of directed graphs," *Journal of Algorithms*, vol. 10 (1989), pp. 76-85.

[MV] S. Micali and V. Vazirani, "An O($|V|^{0.5} \cdot |E|$) Algorithm for Finding Maximum Matchings in General Graphs," *Proceedings of the 21st IEEE Symposium on Foundations of Computer Science*, (1980), pp. 17-27.

[Nao] M. Naor, "Succinct representations of general unlabeled graphs," *Discrete Applied Mathematics*, vol. 28 (1990), pp. 303-308.

[NI] H. Nagamochi and T. Ibaraki, "Linear Time Algorithms for Finding a Sparse k-Connected Spanning Subgraph of a k-Connected Graph," *Algorithmica*, to appear.

[Rag] P. Raghavan, "Probabilistic construction of deterministic algorithms: approximating packing integer programs," *JCSS*, vol. 37 (1988), pp. 130-143.

[Rom] F. Romani, "Shortest-path problem is not harder than matrix multiplication," *Information Processing Letters*, vol. 11 (1980), pp. 134-136.

[Spe] J. Spencer, *Ten lectures on the probabilistic method.* SIAM (Philadelphia), 1987.

[Tur] G. Turan, "On the Succinct Representation of Graphs," *Discrete Applied Mathematics*, vol. 8 (1984), pp. 289-294.

[Tuz] Z. Tuza, "Covering of graphs by complete bipartite subgraphs: complexity of 0-1 matrices," *Combinatorica*, vol. 4 (1984), pp. 111-116.

[Zar] K. Zarankiewicz, Problem P. 101, *Colloq. Math.*, vol. 2 (1951), p. 301.