Chapter 8

Constant-Ratio Approximations

This chapter concentrates on the study of optimization problems that have polynomial time approximation algorithms with approximation ratio bounded by a constant. These problems will be called *approximable optimization problems*, and the class of approximable optimization problems has been named APX.

For a given approximable optimization problem Q, development of an approximation algorithm in general involves four steps:

1. design a polynomial time approximation algorithm A for Q;

2. analyze the algorithm A and derive an upper bound c_A on the approximation ratio for A;

3. study the optimility of the value c_A, i.e. is there another $c'_A < c_A$ such that c'_A is also an upper bound for the approximation ratio for the algorithm A?

4. study the optimality of the algorithm A, i.e., is there another approximation algorithm A' for the problem Q such that the approximation ratio of A' is smaller than that of A?

Step 1 may involve a wide range of techniques in general algorithm design. Many approximation algorithms are based intuition, experience, or deeper insight on the given problems. Popular techniques include the greedy method, branch and bound, and other combinatorical methods. Probabilistic method has also turned out to be very powerful. Step 2 is special in particular for the study of approximation algorithms. One challenging task in this step is the estimation of the value of an optimal solution, which is necessary in comparison with the value of the approximation solution given by the algorithm A to derive the ratio c_A. To prove that the value c_A is the best possible for the algorithm A in Step 3, it suffices to construct a single instance α and show that the algorithm A on the instance α gives a solution whose value is equal to $c_A \cdot Opt(\alpha)$ if Q is a minimization problem and $Opt(\alpha)/c_A$ is Q is a maximization problem. In some cases, the algorithm designer through his development of the algorithm may have got some ideas about what are the "obstacles" for his algorithm. In this case, Step 3 may become pretty easy. However, there are also other examples of approximation algorithms, for which Step 3 have turned out to be extremely difficult. In most cases, Step 4 is the most challenging task in the study of approximation algorithms for optimization problems, which involves the study of "precise" and "intrinsic" polynomial time approximability for optimization problems.

Also note that if an optimization problem has polynomial time approximation scheme, then the answers to the questions in Step 3 and Step 4 become trivial. Therefore, powerful techniques for identifying optimization problems that have no polynomial time approximation scheme is also central in the study of approximable optimization problems.

The discussion in this chapter will be centered around the above issues. We first present constant-ratio approximation algorithms for a number of well-known approximable optimization problems, based on popular combinatorical methods in approximation algorithms including the greedy method, dynamic programming, branch and bound, local search, and combinatorical transformations. The problems we study include the metric traveling salesman problem, the maximum satisfiability problem, the maximum 3-dimensional matching problem, and the minimum vertex cover problem. Note that these four problems are, respectively, the optimization versions of four of the six "basic" NP-complete problems according to Garey and Johnson [52]: the Hamiltonian circuit problem, the satisfiability problem, the 3-dimensional matching problem, and the vertex cover problem. For each of the problems, we start with a simple approximation algorithm and analyze its approximation ratio. We then discuss how to achieve improved approximation ratio using more sophisticated techniques or more thorough analysis, or both.

We then introduce a more recently developed probabilistic method that turns out to be very powerful in developing approximation algorithms for optimization problems. We will illustrate how efficient approximation algorithms for optimization problems can be developed based on the probabilistic methods. We start with a few basic concepts and useful principles in probability theory that are directly related to our discussion. We then describe a general derandomization technique. Randomized approximation algorithms for a variety of NP-hard optimization problems are then presented. These randomized algorithms can be derandomized based on the derandomization techniques.

8.1 Metric TSP

In the previous chapter, we have discussed in detail the Traveling Salesman problem in Euclidean space, and shown that the problem has a polynomial time approximation scheme. Euclidean spaces are special cases of a *metric space*, in which a non-negative function ω (the *metric*) is defined on pairs of points such that for any points p_1, p_2, p_3 in the space:

(1) $\omega(p_1, p_2) = 0$ if and only if $p_1 = p_2$,
(2) $\omega(p_1, p_2) = \omega(p_2, p_1)$, and
(3) $\omega(p_1, p_2) \leq \omega(p_1, p_3) + \omega(p_3, p_2)$.

The third condition $\omega(p_1, p_2) \leq \omega(p_1, p_3) + \omega(p_3, p_2)$ is called the *triangle inequality*. In an Euclidean space, the metric between two points is the distance between the two points. Many non-Euclidean spaces are also metric spaces. An example is a traveling cost map in which points are cities while the metric between two cities is the cost for traveling between the two cities.

In this section, we consider the Traveling Salesman problem on a general metric space. Since the metric between two points p_1 and p_2 in a metric space can be represented by an edge of weight $\omega(p_1, p_2)$ between the two points, we can formulate the problem in terms of weighted graphs.

Definition 8.1.1 A graph G is a *metric graph* if G is a weighted, undirected, and complete graph, in which edge weights are all positive and satisfy the triangle inequality.

A *salesman tour* π in a metric graph G is a simple cycle in G that contains all vertices of G. The *weight* $wt(\pi)$ of the salesman tour π is the sum of weights of the edges in the tour. The Traveling Salesman problem on metric graphs is formally defined as follows.

Metric TSP
I_Q: the set of all metric graphs,

S_Q: $S_Q(G)$ is the set of all salesman tours in G,
f_Q: $f_Q(G, \pi)$ is the weight of the salesman tour π in G,
opt_Q: min.

Since EUCLIDEAN TSP is NP-hard in the strong sense [50, 104], and EUCLIDEAN TSP is a subproblem of METRIC TSP, we derive that METRIC TSP is also NP-hard in the strong sense and, by Theorem 6.4.8, METRIC TSP has no fully polynomial time approximation scheme unless P = NP.

We will show later that METRIC TSP is actually "harder" than EUCLIDEAN TSP in the sense that METRIC TSP has no polynomial-time approximation schemes unless P = NP. In this section, we present approximation algorithms with approximation ratio bounded by a constant for the problem METRIC TSP.

8.1.1 Approximation based on a minimum spanning tree

Our first approximation algorithm for METRIC TSP is based on minimum spanning trees. See the algorithm in Figure 8.1, here the constructed salesman tour is given in the array $V[1..n]$ as a (cyclic) sequence of the vertices in G, in the order the vertices appear in the tour.

```
Algorithm. MTSP-Apx-I
INPUT:  a metric graph $G$
OUTPUT:  a salesman tour $\pi$ in $G$, given in an array $V[1..n]$
1.  construct a minimum spanning tree $T$ for $G$;
2.  let $r$ be the root of $T$;   $i = 0$;
3.  Seq($r$).

Seq($v$)
1.  $i = i + 1$;
2.  $V[i] = v$;
3.  for (each child $w$ of $v$) do Seq($w$).
```

Figure 8.1: Approximating METRIC TSP.

The minimum spanning tree T can be constructed in time $O(n^2)$. Therefore, step 1 of the algorithm **MTSP-Apx-I** takes time $O(n^2)$. Step 3 calls a recursive subroutine **Seq**(r), which is essentially a depth-first-search traversing on the minimum spanning tree T to order the vertices of T in terms of their depth-first-search numbers. Since the depth-first-search process takes time $O(m+n)$ on a graph of n vertices and m edges, step 3 of the algorithm **MTSP-Apx-I** takes time $O(n)$. In conclusion, the time complexity of the algorithm **MTSP-Apx-I** is $O(n^2)$.

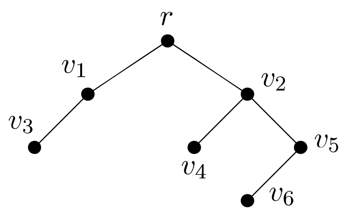

Figure 8.2: The minimum spanning tree T

The depth-first-search process **Seq**(r) on the tree T can be regarded as a closed walk π_0 in the tree (a *closed walk* is a cycle in T in which vertices may repeat). Each edge $[u, v]$, where u is the father of v in T, is traversed exactly twice in the walk π_0: the first time when **Seq**(u) calls **Seq**(v) we traverse the edge from u to v, and the second time when **Seq**(v) is finished and returns back to **Seq**(u) we traverse the edge from v to u. Therefore, the walk π_0 has weight exactly twice the weight of the tree T. It is also easy to see that the list $V[1..n]$ produced by the algorithm **MTSP-Apx-I** can be obtained from the walk π_0 by deleting for each vertex v all but the first occurrence of v in the list π_0. Since each vertex appears exactly once in the list $V[1..n]$, $V[1..n]$ corresponds to a salesman tour π in the metric graph G.

Example. Consider the tree T in Figure 8.2, where r is the root of the tree T. The depth-first-search process (i.e., the subroutine **Seq**) traverses the tree T in the order

$$\pi_0 : \quad r, v_1, v_3, v_1, r, v_2, v_4, v_2, v_5, v_6, v_5, v_2, r.$$

By deleting for each vertex v all but the first vertex occurrence for v, we obtain the list of vertices of the tree T ordered by their depth-first-search numbers

$$\pi : \quad r, v_1, v_3, v_2, v_4, v_5, v_6.$$

Deleting a vertex occurrence of v in the list $\{\cdots uvw \cdots\}$ is equivalent to replacing the path $\{u, v, w\}$ of two edges by a single edge $[u, w]$. Since the metric graph G satisfies the triangle inequality, deleting vertex occurrences from the walk π_0 does not increase the weight of the walk. Consequently, the weight of the salesman tour π given in the array $V[1..n]$ is not larger than the weight of the closed walk π_0, which is bounded by 2 times the weight of the minimum spanning tree T.

Since removing any edge (of non-negative weight) from a minimum weighted salesman tour results in a spanning tree of the metric graph G, the weight of a minimum weighted salesman tour in G is at least as large as the weight of the minimum spanning tree T. In conclusion, the salesman tour π given in the array $V[1..n]$ by the algorithm **MTSP-Apx-I** has its weight bounded by 2 times the weight of a minimum weighted salesman tour. This gives the following theorem.

Theorem 8.1.1 *The approximation ratio of the algorithm* **MTSP-Apx-I** *is bounded by* 2.

Two natural questions follow from Theorem 8.1.1. First, we have shown that the ratio of the weight $wt(\pi)$ of the salesman tour π constructed by the algorithm **MTSP-Apx-I** and the weight $wt(\pi_o)$ of a minimum weighted salesman tour π_o is bounded by 2. Is it possible, by more careful analysis, to show that $wt(\pi)/wt(\pi_o) \leq c$ for a smaller constant $c < 2$? Second, is there a polynomial-time approximation algorithm for METRIC TSP whose approximation ratio is better than that of the approximation algorithm **MTSP-Apx-I**?

These two questions constitute two important and in general highly non-trivial topics in the study of approximation algorithms. Essentially, the first question asks whether our analysis is the best possible *for the algorithm*, while the second question asks whether our algorithm is the best possible *for the problem.*

The answer to the first question some times is easy if we can find an instance for the given problem on which the solution constructed by the algorithm reaches the specified approximation ratio. In some cases, such instances can be realized during our analysis on the algorithm: these instances are the obstacles preventing us from further lowering down the approximation ratio in our analysis. However, there are also situations in which finding such instances is highly non-trivial.

The algorithm **MTSP-Apx-I** for the METRIC TSP problem belongs to the first category. We give below simple instances for METRIC TSP to show that the ratio 2 is tight for the algorithm in the sense that there are instances for METRIC TSP for which the algorithm **MTSP-Apx-I** produces solutions with approximation ratio arbitrarily close to 2.

Consider the figures in Figure 8.3, where our metric space is the Euclidean plane and the metric between two points is the Euclidean distance between the two points.

Suppose we are given $2n$ points on the Euclidean plane with polar coordinates $x_k = (b, 360k/n)$ and $y_k = (b + d, 360k/n)$, $k = 1, \dots, n$, where d is much smaller than b. See Figure 8.3(a). It is not hard (for example, by Kruskal's algorithm for minimum spanning trees [30]) to see that the edges

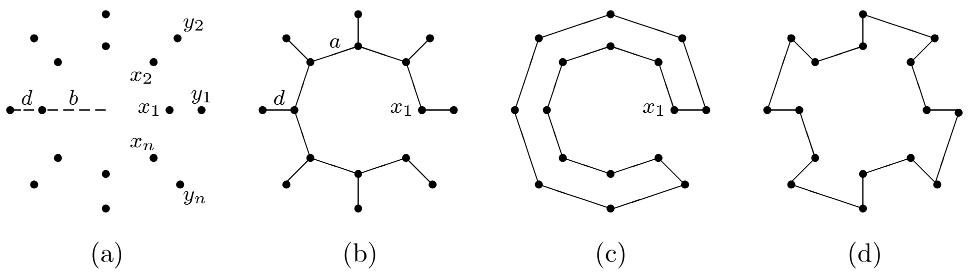

Figure 8.3: Metric TSP instances for **MTSP-Apx-I**.

$[x_k, x_{k+1}]$, $k = 1, \ldots, n-1$ and $[x_j, y_j]$, $j = 1, \ldots, n$ form a minimum spanning tree T for the set of points. See Figure 8.3(b). Now if we perform a depth first search on T starting from the vertex x_1 and construct a salesman tour, we will get a salesman tour π_c that is shown in Figure 8.3(c), while an optimal salesman tour π_d is shown in Figure 8.3(d).

The weight of the salesman tour π_c is about $2a(n-1) + 2d$, where a is the distance between two adjacent points x_k and x_{k+1} (note that when d is sufficiently small compared with a, the distance between two adjacent points y_k and y_{k+1} is roughly equal to the distance between the two corresponding points x_k and x_{k+1}), while the optimal salesman tour π_d has weight roughly $nd + na$. When d is sufficiently small compared with a and when n is sufficiently large, the ratio of the weight of the tour π_c and the weight of the tour π_d can be arbitrarily close to 2.

8.1.2 Christofides' algorithm

Now we turn our attention to the section question. Is the approximation algorithm **MTSP-Apx-I** the best possible for the problem Metric TSP? In other words, are there approximation algorithms for Metric TSP that have a better approximation ratio?

Let us look at the algorithm **MTSP-Apx-I** in Figure 8.1 in detail. After the minimum spanning tree T is constructed, we traverse the tree T by a depth first search process (the subroutine **Seq**) in which each edge of T is traversed exactly twice. This process can be re-interpreted as follows:

1. construct a minimum spanning tree;
2. double each edge of T into two edges, each of which has the same weight as the original edge. Let the resulting graph be D;
3. make a closed walk W in the graph D such that each edge of D is traversed exactly once in W;

4. use "shortcuts", i.e., delete all but the first occurrence for each vertex in the walk W to make a salesman tour π.

There are three crucial facts that make the above algorithm correctly produce a salesman tour with approximation ratio 2: (1) the graph D gives a closed walk W in the graph G that contains all vertices of G; (2) the total weight of the closed walk W is bounded by 2 times the weight of an optimal salesman tour; and (3) the shortcuts do not increase the weight of the closed walk W so that we can derive a salesman tour π from W without increasing the weight of the walk.

If we can construct a graph D' that gives a closed walk W' with weight smaller than that of W constructed by the algorithm **MTSP-Apx-I** such that D' contains all vertices of G, then using the shortcuts on W' should derive a better approximation to the optimal salesman tour.

Graphs whose edges constitute a single closed walk have been studied based on the following concept.

Definition 8.1.2 An *Eulerian tour* in a graph G is a closed walk in G that traverses each edge of G exactly once. An undirected connected graph G is an *Eulerian graph* if it contains an Eulerian tour.

Note that the above definition and the results described below apply to graphs that have multi-edges (i.e., two vertices in the graphs may be connected by more than one edge). Eulerian graphs have been extensively studied in graph theory literature (see for example, [62]). More recent research has shown that Eulerian graphs play an important role in designing efficient parallel graph algorithms [83]. A proof of the following theorem can be found in Appendix A (see Theorems A.1 and A.2).

Theorem 8.1.2 *An undirected connected graph G is an Eulerian graph if and only if every vertex of G has an even degree. Moreover, an Eulerian tour in an Eulerian graph can be constructed in linear time.*

Thus, the graph D described above for the algorithm **MTSP-Apx-I** is actually an Eulerian graph and the closed walk W is an Eulerian tour in D.

Now we consider how a better Eulerian graph D' can be constructed based on the minimum spanning tree T, which leads to a better approximation to the minimum salesman tour.

Let G be a metric graph, that is an instance of the METRIC TSP problem. Let T be a minimum spanning tree in G. We have

Lemma 8.1.3 *The number of vertices of the tree T that has an odd degree in T is even.*

Proof. Let v_1, …, v_n be the vertices of the tree T. Since each edge $e = [v_i, v_j]$ of T contributes one degree to v_i and one degree to v_j, and T has exactly $n-1$ edges, we must have

$$\sum_{i=1}^{n} deg_T(v_i) = 2(n-1)$$

where $deg_T(v_i)$ is the degree of the vertex v_i in the tree T. We partition the set of vertices of T into odd-degree vertices and even-degree vertices. Then

$$\sum_{v_i:\text{ even-degree}} deg_T(v_i) + \sum_{v_j:\text{ odd-degree}} deg_T(v_j) = 2(n-1)$$

Since both $\sum_{v_i:\text{ even-degree}} deg_T(v_i)$ and $2(n-1)$ are even numbers, the value $\sum_{v_j:\text{ odd-degree}} deg_T(v_j)$ is also an even number. Consequently, the number of vertices that have odd degree in T must be even. $\square$

By Lemma 8.1.3, we can assume, without loss of generality, that v_1, v_2, …, v_{2h} are the odd-degree vertices in the tree T. The vertices v_1, v_2, …, v_{2h} induce a complete subgraph H in the original metric graph G (recall that a metric graph is a complete graph). Now construct a minimum weighted perfect matching M_h in H (a *perfect matching* in a complete graph of $2h$ vertices is a matching of h edges. See Section 2.4.2 for more detailed discussions). Since each of the vertices v_1, v_2, …, v_{2h} has degree 1 in the graph M_h, adding the edges in M_h to the tree T results in a graph $D' = T + M_h$ in which all vertices have an even degree. By Theorem 8.1.2, the graph D' is an Eulerian graph. Moreover, the graph D' contains all vertices of the original metric graph G. We are now able to derive a salesman tour π' from D' by using shortcuts.

We formally present this in the algorithm given in Figure 8.4. The algorithm is due to Christofides [26].

According to Theorem 2.4.5, the minimum weighted perfect matching M_h in the complete graph H induced by the vertices $v_1, \ldots, v_{2h}$ can be constructed in time $O(h^3) = O(n^3)$.[1] By Theorem 8.1.2, step 4 of the algorithm

[1]In fact, because the weighted graph H induced by the vertices $v_1, \ldots, v_{2h}$ is a complete graph, a minimum weighted perfect matching M_h in H can be constructed in a simpler way without using Theorem 2.4.5. The following algorithm was suggested by Mr. Mykyta

```
Algorithm. Christofides
INPUT: a metric graph $G$
OUTPUT: a salesman tour $\pi'$ in $G$
1.  construct a minimum spanning tree $T$ for $G$;
2.  let $v_1, \ldots, v_{2h}$ be the odd degree vertices in $T$, construct the completed graph $H$
    induced by the vertices $v_1, \ldots, v_{2h}$;
3.  construct a minimum weighted perfect matching $M_h$ in $H$;
4.  construct an Eulerian tour $W'$ in the Eulerian graph $D' = T + M_h$;
5.  use shortcuts to derive a salesman tour $\pi'$ from $W'$;
6.  return $\pi'$.
```

Figure 8.4: Christofides' Algorithm for METRIC TSP.

Christofides takes linear time. Thus, the algorithm **Christofides** runs in time $O(n^3)$.

Now we study the approximation ratio for the algorithm **Christofides**.

Lemma 8.1.4 *The weight of the minimum weighted perfect matching M_h in the complete graph H induced by the vertices $v_1, \ldots, v_{2h}$, $\sum_{e \in M_h} wt(e)$, is at most $1/2$ of the weight of a minimum salesman tour in the graph G.*

PROOF. Let π_o be an optimal salesman tour in the metric graph G. By using shortcuts, i.e., by removing the vertices that are not in $\{v_1, v_2, \ldots, v_{2h}\}$ from the tour π_o, we obtain a simple cycle π that contains exactly the vertices $v_1, \ldots, v_{2h}$. Since the metric graph G satisfies the triangle inequality, the weight of π is not larger than the weight of π_o.

The simple cycle π can be decomposed into two disjoint perfect matchings in the complete graph H induced by the vertices $v_1, \ldots, v_{2h}$: one matching is obtained by taking every other edge in the cycle π, and the

Makovenko when he was taking my course *Computational Optimization* in Fall 2021. On the complete graph H, construct another complete graph H_r in which the edge weight $wt_r(e)$ for each edge in H_r is defined as $wt_r(e) = w_{\max} - wt(e)$, where $wt(e)$ is the weight of the edge e in the graph H and $w_{\max}$ is equal to 1 plus the largest edge weight in H. Thus, H_r is a complete graph in which all edges have positive weights. It is easy to see that a maximum weighted matching M in the graph H_r must be a perfect matching in H_r: otherwise we would be able to add an edge (of positive weight) to M to make a matching of larger weight in H_r (note that H_r is a complete graph and has an even number of vertices). Moreover, since $w_{\max}$ is a constant and a perfect matching in the graph H consists of exactly h edges, a maximum weighted (thus perfect) matching in the graph H_r must correspond to a minimum weighted perfect matching in the graph H, which thus can be constructed in time $O(h^3 + h^2 \log h) = O(n^3)$ using the maximum weighted matching algorithm given in Theorem 2.4.3.

other matching is formed by the rest of the edges. Of course, both of these two perfect matchings in H have weight at least as large as the minimum weighted perfect matching M_h in H. This gives

$$wt(\pi_o) \geq wt(\pi) \geq 2 \cdot wt(M_h)$$

This completes the proof. $\square$

Now the analysis is clear. We have $D' = T + M_h$. Thus

$$wt(D') = wt(T) + wt(M_h)$$

As we discussed in the analysis for the algorithm **MTSP-Apr-I**, the weight of the minimum spanning tree T of the metric graph G is not larger than that of a minimum salesman tour for G. Combining this with Lemma 8.1.4, we conclude that the weight of the Eulerian graph D' is bounded by 1.5 times that of a minimum salesman tour in G. Thus, the Eulerian tour W' constructed in step 3 of the algorithm **Christofides** has weight bounded by 1.5 times that of a minimum salesman tour in G. Finally, the salesman tour π' constructed by the algorithm **Christofides** is obtained by using shortcuts on the Eulerian tour W' and the metric graph G satisfies the triangle inequality. Thus, the weight of the salesman tour π' constructed by the algorithm **Christofides** is bounded by 1.5 times that of a minimum salesman tour in G. We summarize these discussions in the following theorem.

Theorem 8.1.5 *The algorithm* **Christofides** *for the* Metric TSP *problem runs in time $O(n^3)$ and has an approximation ratio* 1.5.

As for the algorithm **MTSP-Apx-I**, one can show that the ratio 1.5 is tight for the algorithm **Christofides**, in the sense that there are instances of Metric TSP for which the algorithm **Christofides** produces salesman tours whose weights are arbitrarily close to 1.5 times the weight of a minimum salesman tour. The readers are encouraged to construct these instances for a deeper understanding of the algorithm.

It has been a well-known open problem whether the ratio 1.5 can be further improved for approximation algorithms for the Metric TSP problem. In Chapter 11, we will show that the Metric TSP problem has no polynomial time approximation schemes unless $\mathsf{P} = \mathsf{NP}$. This implies that there is a constant $c > 1$ such that no polynomial time approximation algorithm for Metric TSP can have approximation ratio smaller than c (under the assumption $\mathsf{P} \neq \mathsf{NP}$). However, little has been known for this constant c. Very recently, an approximation algorithm with an approximation ratio $1.5 - \epsilon$, where $\epsilon > 10^{-36}$, for the Metric TSP problem has been announced [82], which slightly improves the ratio of Christofides' algorithm after 45 years.

8.2 Minimum vertex cover

Let G be an undirected graph. A *vertex cover* of G is a set C of vertices in G such that every edge in G has at least one end in C (thus, the set C "covers" the edges of G). The VERTEX COVER problem is for a given graph G to construct a minimum vertex cover (i.e., a vertex cover of the fewest vertices). Formally, the problem is defined as follows:

VERTEX COVER = $\langle I_Q, S_Q, f_Q, opt_Q \rangle$, where

I_Q**:** the set of all undirected graphs

S_Q**:** $S_Q(G)$ is the set of all vertex covers of the graph G

f_Q**:** $f_Q(G, C)$ is the size of the vertex cover C of G

opt_Q**:** min

The decision version of the VERTEX COVER problem is one of the six "basic" NP-complete problems [52]. Thus, the optimization version of the problem, i.e., the VERTEX COVER problem, is NP-hard, which has been a central problem in the study of approximation algorithms.

Vertex covers of a graph are related to independent sets of the graph by the following lemma.

Lemma 8.2.1 *A set C of vertices in a graph $G = (V, E)$ is a vertex cover of G if and only if the set $V - C$ is an independent set in G.*

PROOF. Suppose C is a vertex cover. Since every edge in G has at least one end in C, no two vertices in $V - C$ are adjacent. That is, $V - C$ is an independent set.

Conversely, if $V - C$ is an independent set, then every edge in G has at least one end not in $V - C$. Therefore, every edge in G has at least one end in C and C forms a vertex cover. $\square$

8.2.1 Vertex cover and matching

Recall that a *matching* in a graph G is a set M of edges such that no two edges in M share a common end. A vertex is *matched* if it is an end of an edge in M and *unmatched* otherwise.

The problems GRAPH MATCHING and VERTEX COVER are closely related. We first present a simple approximation algorithm for VERTEX COVER based on matching.

Lemma 8.2.2 *Let M be a matching in a graph G and let C be a vertex cover of G, then $|M| \leq |C|$. In particular, the size of a minimum vertex cover of G is at least as large as the size of a maximum matching in G.*

PROOF. Since the vertex cover C covers all edges in G, each edge in the matching M has at least one end in C. Since no two edges in M share a common end, the number $|C|$ of vertices in the vertex cover C is at least as large as the number $|M|$ of edges in the matching M. □

A matching M in a graph G is *maximal* if there is no edge e in G such that $e \notin M$ and $M \cup \{e\}$ still forms a matching. An approximation algorithm for VERTEX COVER based on maximal matchings is given in Figure 8.5.

```
Algorithm. VC-Apx-I
INPUT:  a graph $G$
OUTPUT:  a vertex cover $C$ of $G$
1.  $C = \emptyset$;
2.  for (each edge $e$ in $G$) do
        if (no end of $e$ is in $C$)  add both ends of $e$ to $C$;
3.  return $C$.
```

Figure 8.5: Approximating vertex cover I.

Theorem 8.2.3 *The algorithm* **VC-Apx-I** *is a linear time approximation algorithm with approximation ratio* 2 *for the* VERTEX COVER *problem.*

PROOF. The algorithm obviously runs in linear time.

Because of the **for** loop in step 2 of the algorithm, every edge in G has at least one end in the set C. Therefore, C is a vertex cover of G.

Actually, step 2 of the algorithm implicitly constructs a maximal matching M in G, as follows. Suppose we initialize $M = \emptyset$ in step 1, and in step 2 whenever we encounter an edge e with no end in C, we, in addition to adding both ends of e to C, also add the edge e to M. It is straightforward to see that the set M constructed this way is a maximal matching and C is the set of ends of the edges in M. Thus, $2|M| = |C|$. By Lemma 8.2.2, we have (where $Opt(G)$ is the size of a minimum vertex cover of G)

$$\frac{|C|}{Opt(G)} = \frac{2|M|}{Opt(G)} \leq \frac{2 \cdot Opt(G)}{Opt(G)} = 2.$$

Thus, the approximation ratio of the algorithm is bounded by 2. □

GRAPH MATCHING and VERTEX COVER are actually dual problems in their formulations by integer linear programming. To see this, let G be a graph of n vertices v_1, v_2, …, v_n and m edges e_1, e_2, …, e_m. Introduce n integral variables x_1, x_2, …, x_n to record the membership of the vertices of a vertex cover in G such that $x_i > 0$ if and only if the vertex v_i is in the vertex cover. Then the instance G of VERTEX COVER can be formulated as an instance Q_G of the INTEGER LP problem as follows:

Primal Instance Q_G

$$\begin{array}{ll} \text{minimize} & x_1 + \cdots + x_n, \\ \text{subject to} & x_{i_1} + x_{i_2} \geq 1, \quad \text{for } i = 1, 2, \ldots, m, \\ & \{\text{suppose the two endpoints of the edge } e_i \text{ are } v_{i_1} \text{ and } v_{i_2}\} \\ & x_j \text{ are integers and } x_j \geq 0, \quad \text{for } j = 1, 2, \ldots, n. \end{array}$$

The formal dual problem of the instance Q_G for INTEGER LP is (the reader is referred to Section 4.3 of the current book or to Chapter 3 of [106] for more detailed and formal discussions on primal-dual instances and their relationships in linear programming):

Dual Instance Q'_G

$$\begin{array}{ll} \text{maximize} & y_1 + \cdots + y_m, \\ \text{subject to} & y_{j_1} + y_{j_2} + \cdots + y_{j_{h_j}} \leq 1, \quad \text{for } j = 1, 2, \ldots, n, \\ & \{\text{suppose vertex } v_j \text{ is incident to edges } e_{j_1}, e_{j_2}, \ldots, e_{j_{h_j}}\} \\ & y_j \text{ are integers and } y_i \geq 0, \quad \text{for } i = 1, 2, \ldots, m. \end{array}$$

If we define a set M of edges in G based on the dual instance Q'_G such that $y_i > 0$ if and only if the edge e_i in the graph G is in M, then the condition $y_{j_1} + \cdots + y_{j_{h_j}} \leq 1$ for $j = 1, \ldots, n$ requires that each vertex v_j in G be incident to at most one edge in M, or equivalently, that the set M forms a matching. Therefore, the dual instance Q'_G in the INTEGER LP problem exactly characterizes the instance G for the GRAPH MATCHING problem.

8.2.2 Vertex cover in bipartite graphs

Lemma 8.2.2 indicates that the size of a maximum matching of a graph G is not larger than the size of a minimum vertex cover of the graph. This provides an effective lower bound for the minimum vertex cover of a graph. Since GRAPH MATCHING can be solved in polynomial time while VERTEX COVER is NP-hard, one should not expect that in general these two values are equal. However, for certain important graph classes, the equality does hold, which induces polynomial time (precise) algorithms for VERTEX COVER on the graph classes. In this subsection, we use this idea to develop a polynomial time (precise) algorithm for VERTEX COVER on bipartite graphs. The algorithm will turn out to be very useful in the study of approximation algorithms for VERTEX COVER on general graphs.

Let M be a matching in a graph G. Recall that an *alternating path* w.r.t. M is a path that traverses alternatively between edges in M and edges not in M. In particular, if an alternating path starts and ends at unmatched vertices, then it is an *augmenting path*. By Theorem 2.1.2, the matching M is maximum if and only if there is no augmenting path w.r.t. M.

We say a vertex w is *M-reachable* from a vertex v if there is an alternating path starting at v and ending at w. For a set U vertices, we say that a vertex w is M-reachable from U if w is M-reachable from a vertex in U.

Let $G = (V_1 \cup V_2, E)$ be a bipartite graph, where every edge in G has one end in V_1 and the other end in V_2. Let M be a maximum matching in G. Consider the algorithm given in Figure 8.6. The algorithm **VC-BGraph** produces a set of vertices for the bipartite graph G, which we will prove is a minimum vertex cover of the graph G.

```
Algorithm. VC-BGraph($G, M$)
INPUT:  bipartite graph $G = (V_1 \cup V_2, E)$ and maximum matching $M$ in $G$
OUTPUT:  a minimum vertex cover $C$ of $G$
1.  let $U_1$ be the set of unmatched vertices in $V_1$;
2.  let $N_1$ be the set of vertices in $V_1$ that are not $M$-reachable from $U_1$;
3.  let $R_2$ be the set of vertices in $V_2$ that are $M$-reachable from $U_1$;
4.  output $C = N_1 \cup R_2$.
```

Figure 8.6: Constructing a minimum vertex cover in bipartite graphs.

Lemma 8.2.4 *The algorithm* **VC-BGraph** *runs in linear time and constructs a minimum vertex cover C for the bipartite graph G. In particular, we have $|C| = |M|$.*

PROOF. The set R of all vertices in G that are M-reachable from the set U_1 can be constructed in linear time using the algorithm **Bip-Augment** given in Figure 2.3. Basically, we perform a searching process similar to breadth first search, starting from the vertices in the set U_1. Note that in this situation, the algorithm **Bip-Augment** never stops at step 4.1.2 and step 4.2.2 since according to Theorem 2.1.2, there is no augmenting path with respect to the maximum matching M. Once the set R is constructed, the set $C = N_1 \cup R_2$ is easily obtained.

Consider the set N_1 of vertices in V_1 that are not M-reachable from U_1. Every vertex in N_1 is matched because every unmatched vertex in V_1 is in the set U_1, which is obviously M-reachable from U_1.

Now consider the set R_2 of vertices in V_2 that are M-reachable from U_1. We claim that all vertices in R_2 are also matched. In fact, if $v_2 \in R_2$ is unmatched, then let P be an alternating path starting from an unmatched vertex v_1 in U_1 and ending at v_2. Then, the path P would be an augmenting path w.r.t. the maximum matching M, contradicting Theorem 2.1.2.

Let $v_1 \in N_1$ and let $[v_1, v_2]$ be the edge in the matching M. We claim $v_2 \notin R_2$. In fact, if v_2 is in R_2 then the alternating path from a vertex u_1 in U_1 to v_2 plus the edge $[v_2, v_1]$ in M would form an alternating path from u_1 to v_1. This would imply that v_1 is M-reachable from U_1, contradicting the assumption that $v_1 \in N_1$. As a result, each edge in the matching M has at most one end in the set $C = N_1 \cup R_2$. Moreover, by the above analysis, all vertices in C are matched. Thus, $|C| \le |M|$.

Now we prove that C is a vertex cover of G. According to the above discussion, the vertex set V_1 can be partitioned into three disjoint parts: the set U_1 of unmatched vertices, the set R_1 of matched vertices that are M-reachable from U_1, and the set N_1 of matched vertices that are not M-reachable from U_1. Let $e = [v_1, v_2]$ be any edge in G, where $v_1 \in V_1$ and $v_2 \in V_2$.

If $v_1 \notin N_1$, then $v_1 \in U_1$ or $v_1 \in R_1$. In case $v_1 \in U_1$ then the edge e is not in M. Thus, $[v_1, v_2]$ is an alternating path and $v_2 \in R_2$. On the other hand, suppose $v_1 \in R_1$. Let $P = \{u_0, \ldots, v_1\}$ be an alternating path from $u_0 \in U_1$ to v_1. Since v_1 is in the set V_1, by the bipartiteness of the graph G, P is of even length. Therefore, either the vertex v_2 is contained in the path P, or the path P plus the edge $[v_1, v_2]$ forms an alternating path from u_0 to v_2 (note that if v_2 is not in the path P, then $[v_1, v_2]$ cannot be an edge in M because v_1 has matched with the vertex before it on the path P). In either case, $v_2 \in R_2$. This proves that for any edge $e = [v_1, v_2]$ in G, either $v_1 \in N_1$ or $v_2 \in R_2$, i.e., $C = N_1 \cup R_2$ is a vertex cover of G.

Combining the fact that C is a vertex cover of G with the inequality $|C| \le |M|$ and Lemma 8.2.2, we conclude that $|C| = |M|$ and C is a minimum vertex cover of G. $\square$

Theorem 8.2.5 *The* VERTEX COVER *problem on bipartite graphs can be solved in time $O(m\sqrt{n})$.*

PROOF. By Theorem 2.3.6, a maximum matching of a (general) graph G can be constructed in time $O(m\sqrt{n})$. Combining this with Lemma 8.2.4, we complete the proof of the theorem. □

8.2.3 Local approximation and local optimization

We now get back to the VERTEX COVER problem on general graphs, which is NP-hard. By Theorem 8.2.3, the simple approximation algorithm **VC-Apx-I** given in Figure 8.5 for VERTEX COVER on general graphs has an approximation ratio 2. One may expect that the ratio can be further improved using more sophisticated techniques. However, despite long time efforts, no significant progress has been made and asymptotically, the ratio 2 still stands as the best approximation ratio for polynomial time approximation algorithms for the problem. In this subsection, we introduce several techniques that lead to slight improvements on the approximation ratio for VERTEX COVER. The techniques can also be extended to approximation algorithms with the same ratio for the weighted version of VERTEX COVER, in which each vertex has an assigned weight and we are looking for a vertex cover of the minimum weight.

The first technique has been called the "local optimization" in the literature, developed by Nemhauser and Trotter [103], which turns out to be very useful in the study of approximation algorithms for VERTEX COVER, for both weighted and unweighted versions.

For a subset V' of vertices in a graph G, denote by $G(V')$ the subgraph of G induced by the vertex set V', that is, $G(V')$ has V' as its vertex set and contains all edges in G that have their both ends in V'.

Theorem 8.2.6 *There is an $O(m\sqrt{n})$-time algorithm that, given a graph G, constructs two disjoint subsets C_0 and V_0 of the vertices in G such that*

(1) *the set C_0 plus any vertex cover of $G(V_0)$ forms a vertex cover of G;*
(2) *there is a minimum vertex cover $C'_{\min}$ of G such that $C_0 \subseteq C'_{\min}$;*
(3) *$Opt(G(V_0)) \geq |V_0|/2$.*

PROOF. Let $\{v_1, v_2, \ldots, v_n\}$ be the set of vertices in the graph G. Construct a bipartite graph B of $2n$ vertices: v_1^L, v_1^R, v_2^L, v_2^R, …, v_n^L, v_n^R such that there is an edge $[v_i^L, v_j^R]$ in B if and only if $[v_i, v_j]$ is an edge in G.

Let C_B be a minimum vertex cover of the bipartite graph B. Define two disjoint subsets of vertices in the graph G:

$$C_0 = \{v_i \mid \text{both } v_i^L \text{ and } v_i^R \text{ are in } C_B\}$$
$$V_0 = \{v_j \mid \text{exactly one of } v_j^L \text{ and } v_j^R \text{ is in } C_B\}$$

According to Theorem 8.2.5, the minimum vertex cover C_B of the bipartite graph B can be constructed in time $O(m\sqrt{n})$. Therefore, in order to prove the theorem, it suffices to prove that the constructed subsets C_0 and V_0 satisfy the conclusions in the theorem.

Let $I_0 = \{v_1, \ldots, v_n\} - (C_0 \cup V_0)$, then I_0 is the set of vertices v_i in G such that both v_i^L and v_i^R are not in C_B. For each edge $[v_i, v_j]$ in G, by the definition, $[v_i^L, v_j^R]$ and $[v_j^L, v_i^R]$ are edges in the bipartite graph B. Therefore, we have the following facts that will be used heavily in the discussion:

If $[v_i, v_j]$ is an edge in G, then

Fact 1. $v_i \in I_0$ implies $v_j \in C_0$, and
Fact 2. $v_i \in V_0$ implies $v_j \notin I_0$.

Proof for (1). Let C_{V_0} be a vertex cover of the induced subgraph $G(V_0)$. For any edge $[v_i, v_j]$ in G, if $[v_i, v_j]$ is not covered by C_{V_0}, i.e., if neither of v_i and v_j is in C_{V_0}, then one of v_i and v_j must be in $C_0 \cup I_0$ – otherwise, $[v_i, v_j]$ is an edge in $G(V_0)$ that should be covered by C_{V_0}. Without loss of generality, let $v_i \in C_0 \cup I_0$. Thus, if $v_i \notin C_0$, then $v_i \in I_0$, which, by Fact 1 above, will imply $v_j \in C_0$. Therefore, if the edge $[v_i, v_j]$ is not covered by C_{V_0}, then it must be covered by C_0. This proves statement (1) of the theorem that for any vertex cover of the induced subgraph $G(V_0)$, the set $C_0 \cup C_{V_0}$ is a vertex cover of the graph G.

Proof for (2). Let $C_{\min}$ be a minimum vertex cover of the graph G. We show that the vertex set $C'_{\min} = C_0 \cup (C_{\min} \cap V_0)$ is also a minimum vertex cover of the graph G.

For any edge $[v_i, v_j]$ in the graph G, if $v_i \notin C'_{\min}$, then $v_i \in I_0$ or $v_i \in V_0 - C_{\min}$. If $v_i \in I_0$, then by Fact 1 above $v_j \in C_0 \subseteq C'_{\min}$. If $v_i \in V_0 - C_{\min}$, then by Fact 2 above $v_j \notin I_0$, i.e., either $v_j \in C_0$ or $v_j \in V_0$. Moreover, $v_i \notin C_{\min}$ implies $v_j \in C_{\min}$. Thus, v_j must be in the set $C_0 \cup (V_0 \cap C_{min})$. Combining all these, we conclude that the set $C'_{\min} = C_0 \cup (C_{\min} \cap V_0)$ covers the edge $[v_i, v_j]$. Since $[v_i, v_j]$ is an arbitrary edge in G, this proves that $C'_{\min}$ is a vertex cover of the graph G.

Now we prove $|C'_{\min}| = |C_{\min}|$. For this we first construct a vertex cover for the bipartite graph B. Let

$$T = C_0 \cup V_0 \cup (C_{\min} \cap I_0) \quad \text{and} \quad W = C_{\min} \cap C_0.$$

Define two subsets of vertices in the bipartite graph B:

$$L_T = \{v_i^L \mid v_i \in T\} \quad \text{and} \quad R_W = \{v_j^R \mid v_j \in W\}.$$

We prove that $C'_B = L_T \cup R_W$ is a vertex cover of the bipartite grpah B.

Let $[v_i^L, v_j^R]$ be an edge in B. By the definition, $[v_i, v_j]$ is an edge in G. If $v_i^L \notin L_T$, then $v_i \notin T = C_0 \cup V_0 \cup (C_{\min} \cap I_0)$, so $v_i \in I_0 - C_{\min}$, that is, $v_i \in I_0$ and $v_i \notin C_{\min}$. Since $C_{\min}$ must cover the edge $[v_i, v_j]$, we have $v_j \in C_{\min}$. From $v_i \in I_0$, by Fact 1 we have $v_j \in C_0$. Therefore, in case $v_i^L \notin L_T$, we have $v_j \in C_{\min} \cap C_0 = W$, which implies $v_j^R \in R_W$. Thus, $C'_B = L_T \cup R_W$ is a vertex cover of the bipartite graph B. By the definition of the minimum vertex cover C_B for the bipartite graph B, and that of the vertex sets C_0 and V_0 in G (see the second paragraph of this proof), we have $|C_B| = |V_0| + 2|C_0|$. Thus,

$$\begin{aligned} |V_0| + 2|C_0| &= |C_B| \le |C'_B| = |L_T| + |R_W| \\ &= |C_0| + |V_0| + |C_{\min} \cap I_0| + |C_{\min} \cap C_0|. \end{aligned}$$

The inequality above is because C'_B is a vertex cover while C_B is a minimum vertex cover of the bipartite graph B. From this we get immediately

$$|C_0| \le |C_{\min} \cap I_0| + |C_{\min} \cap C_0| = |C_{\min} \cap (I_0 \cup C_0)|. \tag{8.1}$$

Therefore,

$$\begin{aligned} |C'_{\min}| &= |C_0 \cup (C_{\min} \cap V_0)| \\ &= |C_0| + |C_{\min} \cap V_0| \\ &\le |C_{\min} \cap (I_0 \cup C_0)| + |C_{\min} \cap V_0| \\ &= |C_{\min} \cap (I_0 \cup C_0 \cup V_0)| = |C_{\min}|, \end{aligned}$$

where the inequality is from (8.1). Since $C'_{\min}$ is a vertex cover and $C_{\min}$ is a minimum vertex cover of the graph G, we must have $|C'_{\min}| = |C_{\min}|$ and $C'_{\min}$ is a also a minimum vertex cover of the graph G. Since $C_0 \subseteq C'_{\min}$, the statement (2) of the theorem is proved.

Proof for (3). Let C_1 be a minimum vertex cover of the induced subgraph $G(V_0)$. Then by statement (1) of the theorem, $C_2 = C_0 \cup C_1$ is a vertex cover of the graph G. Now if we let $L_2 = \{v_i^L | v_i \in C_2\}$ and $R_2 = \{v_i^R | v_i \in C_2\}$, then clearly $L_2 \cup R_2$ is a vertex cover of the bipartite graph B. Therefore

$$|V_0| + 2|C_0| = |C_B| \le |L_2 \cup R_2| = 2|C_2| = 2|C_0| + 2|C_1|$$

The inequality is because C_B is a minimum vertex cover while $L_2 \cup R_2$ is a vertex cover of the bipartite graph B. This derivation gives immediately, $|V_0| \le 2|C_1| = 2 \cdot Opt(G(V_0))$. The statement (3) of the theorem follows. $\square$

Corollary 8.2.7 *Let G be a graph, and let C_0 and V_0 be the subsets given in Theorem 8.2.6. For any vertex cover C_V of the induced subgraph $G(V_0)$, $C_0 \cup C_V$ is a vertex cover of G and*

$$\frac{|C_0 \cup C_V|}{Opt(G)} \leq \frac{|C_V|}{Opt(G(V_0))}$$

PROOF. The claim that $C_0 \cup C_V$ is a vertex cover of the graph G is given by the statement (1) in Theorem 8.2.6.

By the statement (2) of Theorem 8.2.6, there is a minimum vertex cover $C_{\min}$ of G such that $C_0 \subseteq C_{\min}$. Let $C^-_{\min} = C_{\min} - C_0$. Then $C^-_{\min}$ covers all edges in the induced subgraph $G(V_0)$. In fact, $C^-_{\min}$ is a minimum vertex cover of the induced graph $G(V_0)$. This can be seen as follows. First, $C^-_{\min}$ is a subset of V_0: if $C^-_{\min}$ is not a subset of V_0, then the smaller set $C^-_{\min} \cap V_0$ is a vertex cover of $G(V_0)$. By the statement (1) of Theorem 8.2.6, $(C^-_{\min} \cap V_0) \cup C_0$ is a vertex cover of G. Now $|(C^-_{\min} \cap V_0) \cup C_0| < |C^-_{\min} \cup C_0| = |C_{\min}|$ contradicts the definition of $C_{\min}$. This shows that $C^-_{\min}$ is a subset of V_0 thus $C^-_{\min}$ is a vertex cover of $G(V_0)$. $C^-_{\min}$ is also a minimum vertex cover of $G(V_0)$ since any smaller vertex cover of $G(V_0)$ plus C_0 would form a vertex cover of G smaller than the minimum vertex cover $C_{\min} = C^-_{\min} \cup C_0$ of the graph G. Therefore

$$\frac{|C_0 \cup C_V|}{Opt(G)} = \frac{|C_0| + |C_V|}{|C_{\min}|} = \frac{|C_0| + |C_V|}{|C_0| + |C^-_{\min}|} \leq \frac{|C_V|}{|C^-_{\min}|} = \frac{|C_V|}{Opt(G(V_0))}$$

The inequality has used the fact that $C^-_{\min}$ is a minimum vertex cover of $G(V_0)$ so $|C^-_{\min}| \leq |C_V|$. □

Corollary 8.2.7 indicates that in order to improve the approximation ratio for the VERTEX COVER problem on the graph G, we only need to concentrate on the induced subgraph $G(V_0)$. Note that an approximation ratio 2 is trivial for the induced subgraph $G(V_0)$: by the statement (3) in Theorem 8.2.6, $|V_0|/Opt(G(V_0)) \leq 2$. Therefore, simply including all vertices in the graph $G(V_0)$ gives a vertex cover of size at most twice of $Opt(G(V_0))$.

By Lemma 8.2.1, the complement of a vertex cover is an independent set, the above observation suggests that in order to improve the approximation ratio for VERTEX COVER, we can try to identify a large independent set in $G(V_0)$. Our first improvement is given in Figure 8.7.

```
Algorithm. VC-Apx-II
Input: a graph $G$
Output: a vertex cover $C$ of $G$
1.  apply Theorem 8.2.6 to construct the subsets $C_0$ and $V_0$;
2.  $G_1 = G(V_0)$;   $I = \emptyset$;
3.  while $G_1$ is not empty do
        pick any vertex $v$ in $G_1$;
        $I = I \cup \{v\}$;
        delete $v$ and all its neighbors from the graph $G_1$;
4.  return $C = (V_0 - I) \cup C_0$.
```

Figure 8.7: Approximating vertex cover II.

Theorem 8.2.8 *The algorithm* **VC-Apx-II** *for* Vertex Cover *runs in time* $O(m\sqrt{n})$ *and has an approximation ratio bounded by* $2 - 2/(\Delta + 1)$*, where* Δ *is the largest vertex degree in the given graph.*

Proof. The running time of the algorithm **VC-Apx-II** is dominated by step 1, which by Theorem 8.2.6 takes time $O(m\sqrt{n})$.

Consider the loop in step 3. The constructed set I is obviously an independent set in the graph $G(V_0)$. According to the algorithm, for each group of at most $\Delta + 1$ vertices in $G(V_0)$, we conclude a new vertex in I. Thus, the number of vertices in I is at least $|V_0|/(\Delta + 1)$. Therefore, $V_0 - I$ is a vertex cover of $G(V_0)$ and $|V_0 - I| \leq (\Delta \cdot |V_0|)/(\Delta + 1)$. Now

$$\frac{|V_0 - I|}{Opt(G(V_0))} \leq \frac{(\Delta \cdot |V_0|)/(\Delta + 1)}{|V_0|/2} = 2 - \frac{2}{\Delta + 1},$$

where we have used the fact $Opt(G(V_0)) \geq |V_0|/2$ proved in Theorem 8.2.6. Now the theorem follows directly from Corollary 8.2.7. □

For graphs of low degrees, the approximation ratio of the algorithm **VC-Apx-II** is significantly better than 2. However, the value Δ can be as large as $n - 1$. Therefore, in the worst case, what we can conclude is only that the algorithm **VC-Apx-II** has an approximation ratio bounded by $2 - 2/n$.

We seek further improvement by looking for larger independent sets. We first show that for graphs with no short odd cycles, finding a larger independent set is possible. Consider the algorithm given in Figure 8.8.

Lemma 8.2.9 *For a graph* G *of* n *vertices with no odd cycles of length less than or equal to* $2k - 1$*, where* k *is an integer satisfying* $(2k - 1)^k \geq n$*, the algorithm* **Large-IS**(G, k) *runs in time* $O(nm)$ *and constructs an independent set* I *of size at least* $n/(2k)$.

```
Algorithm. Large-IS($G, k$)
INPUT:  a graph $G$ of $n$ vertices that hsa no odd cycles of length $\leq 2k-1$, where $k$
        is an integer satisfying $(2k-1)^k \geq n$
OUTPUT:  an independent set $I$ in $G$
1.  $I = \emptyset$;
2.  while ($G$ is not empty) do
        pick any vertex $v$ in $G$ and apply BFS starting from $v$;
        let $L_0$, $L_1$, ..., $L_k$ be the first $k+1$ levels of vertices in the BFS tree;
        let $D_{2t} = \bigcup_{i=0}^{t} L_{2i}$ and $D_{2t+1} = \bigcup_{i=0}^{t} L_{2i+1}$,  for $t = 0, 1, \ldots$;
        let $s$ be the smallest index satisfying $|D_s| \leq (2k-1)|D_{s-1}|$;
        $I = I \cup D_{s-1}$;
        remove all vertices in $D_s \cup D_{s-1}$ from the graph $G$;
3.  return $I$.
```

Figure 8.8: Finding an independent set in graphs without short odd cycles.

PROOF. First we show that it is always possible to fnd the index s such that $|D_s| \leq (2k-1)|D_{s-1}|$. Suppose such an index does not exist. Then we have $|D_i| > (2k-1)|D_{i-1}|$ for all $i = 1, \ldots, k$. Therefore (note $|D_0| = 1$ and $(2k-1)^k \geq n$),

$$|D_k| > (2k-1)|D_{k-1}| > (2k-1)^2|D_{k-2}| > \cdots > (2k-1)^k|D_0| \geq n.$$

This is impossible, since D_k is a subset of vertices in the graph G while G has n vertices. Therefore, the index s always exists.

Since $|D_s| \leq (2k-1)|D_{s-1}|$, we have $|D_{s-1}| \geq (|D_s| + |D_{s-1}|)/(2k)$. Therefore, each time when we remove $|D_s| + |D_{s-1}|$ vertices from the graph G, we include $|D_{s-1}| \geq (|D_s| + |D_{s-1}|)/(2k)$ vertices in the set I. As a result, the set I constructed by the algorithm **Large-IS**(G, k) has at least $n/(2k)$ vertices.

What remains is to show that the set I is an independent set in G. For a BFS tree, every edge in G either connects two vertices at the same level, or connects two vertices in the adjacent levels [30]. Therefore, no edge is between two vertices that belong to different levels in the set D_{s-1} (note that D_{s-1} contains either only odd levels or only even levels in the BFS tree). Moreover, any edge connecting two vertices at the same level in D_{s-1} would form an odd cycle of length $\leq 2k-1$ (recall $s \leq k$), which contradicts the assumption that the graph G has no odd cycles of length $\leq 2k-1$. In conclusion, no two vertices in the set D_{s-1} are adjacent and the set D_{s-1} is an independent set. Since in each execution of the body of the **while**-loop in step 2, we also remove vertices in the set D_s, there is also no edge between the two sets D_{s_1-1} and D_{s_2-1} constructed by two different executions of the body of the **while**-loop. Thus, the set I returned by the algorithm **Large-IS**(G, k) is an independent set in the graph G.

For the algorithm complexity, each execution of the **while** loop body is a BFS on the graph G, which takes time $O(m)$, and removes at least one vertex from the graph G. Therefore, the algorithm runs in time $O(nm)$. $\square$

The conditions in Lemma 8.2.9 are bit too strong. We need to take care of the situation where graphs contain short odd cycles. Suppose that the vertices v_1, v_2, and v_3 form a triangle in a graph G. Then we observe that *every* minimum vertex cover of G must contain at least two of these three vertices. Therefore, if our objective is an approximation ratio larger than 1.5, then intuitively it will not hurt if we include all three vertices in our vertex cover since the "local" approximation ratio for this inclusion is 1.5. In general, for a subgraph H of h vertices in G, if we know the ratio $h/Opt(H)$ is not larger than our objective ratio, where $Opt(H)$ is the size of a minimum vertex cover for the subgraph H, then it seems reasonable to simply include all vertices in the subgraph H and remove H from G. This intuition is confirmed by the following lemma.

Lemma 8.2.10 *Let G be a graph and H be a subgraph induced by h vertices in G. Let $G^- = G - H$. Suppose that C^- is a vertex cover of the graph G^-. Then $C^- \cup H$ is a vertex cover of the graph G and*

$$\frac{|C^- \cup H|}{Opt(G)} \le \max\left\{\frac{|C^-|}{Opt(G^-)}, \frac{h}{Opt(H)}\right\}.$$

PROOF. Let $[u, v]$ be an edge in the graph G. If one of u and v is in the graph H, then certainly $[u, v]$ is covered by $C^- \cup H$. If none of u and v is in H, then $[u, v]$ is an edge in G^- and must be covered by C^-. Therefore, $C^- \cup H$ is a vertex cover of the graph G.

Let $C_{\min}$ be a minimum vertex cover of the graph G. Let $C_{\min}^-$ be the set of vertices in $C_{\min}$ that are in the graph G^-, and let $C_{\min}^H$ be the set of vertices in $C_{\min}$ that are in H. Then $C_{\min}^-$ is a vertex cover of the graph G^- and $C_{\min}^H$ is a vertex cover of the graph H. Therefore, we have

$$\begin{aligned} \frac{|C^- \cup H|}{Opt(G)} &= \frac{|C^- \cup H|}{|C_{\min}|} = \frac{|C^-| + h}{|C_{\min}^-| + |C_{\min}^H|} \\ &\le \frac{|C^-| + h}{Opt(G^-) + Opt(H)} \le \max\left\{\frac{|C^-|}{Opt(G^-)}, \frac{h}{Opt(H)}\right\}, \end{aligned}$$

here we have used the obvious inequalities that $|C^-_{\min}| \geq Opt(G^-)$, that $|C^H_{\min}| \geq Opt(H)$, and that $(a+b)/(c+d) \leq \max\{a/c, b/d\}$ for any positive numbers a, b, c, and d. □

If the subgraph H is a cycle of length $h = 2k-1$, obviously we have $h/Opt(H) = (2k-1)/k = 2 - 1/k$. According to Lemma 8.2.10, if our objective approximation ratio is not smaller than $2 - 1/k$, then we can remove the cycle H from the graph by simply including all vertices in H in the vertex cover. Repeating this procedure, we will result in a graph G' with no short odd cycles. Now applying the algorithm **Large-IS** on G' gives a larger independent set I, from which a better vertex cover is obtained. These ideas are implemented in the algorithm given in Figure 8.9.

```
Algorithm. VC-Apx-III
INPUT:  a graph $G$ of $n$ vertices
OUTPUT:  a vertex cover $C$ of $G$
1.  $C_1 = \emptyset$;
2.  let $k$ be the smallest integer such that $(2k-1)^k \geq n$;
3.  while ($G$ contains an odd cycle of length $\leq 2k-1$) do
        find an odd cycle $X$ of length $\leq 2k-1$;
        add all vertices of $X$ to $C_1$;
        delete all vertices of $X$ from the graph $G$;
4.  apply Theorem 8.2.6 to $G$ to construct the vertex sets $C_0$ and $V_0$ in $G$;
5.  call Large-IS($G(V_0), k$) to construct an independent set $I$ in $G(V_0)$;
6.  $C_2 = C_0 \cup (V_0 - I)$;
7.  return $C = C_1 \cup C_2$.
```

Figure 8.9: Approximating vertex cover III.

Theorem 8.2.11 *The algorithm* **VC-Apx-III** *for* VERTEX COVER *runs in time $O(nm)$, and has an approximation ratio $2 - \log\log n/(2\log n)$.*

PROOF. The time complexity of all steps, except step 3, of the algorithm has been discussed and is bounded by $O(nm)$. To find an odd cycle of length bounded by $2k-1$ in step 3, we pick any vertex v and apply BSF starting from v for at most $k+1$ levels. Either we will find an edge connecting two vertices at the same level, which gives an odd cycle of length bounded by $2k-1$, or we do not find such an odd cycle. In the former case, the cycle will be removed from the graph G, while in the latter case, the vertex v is not contained in any odd cycle of length bounded by $2k-1$. Therefore, the vertex v can be removed from the graph in the latter search for odd cycles.

In any case, each BFS removes at least one vertex from the graph. Thus, at most n BFS's are performed in step 3. Since each BFS takes time $O(m)$, the time complexity of step 3 is $O(nm)$. Summarizing all these, we conclude that the time complexity of the algorithm **VC-Apx-III** is $O(nm)$.

We prove that the approximation ratio of the algorithm **VC-Apx-III** is bounded by $2 - 1/k$, where k is defined in step 2 of the algorithm.

Let H be the subgraph of G consisting of all the odd cycles removed in step 3. Since each cycle X in H has length $2j - 1$, where $j \le k$, we have $(2j-1)/Opt(X) = (2j-1)/j = 2-1/j \le 2-1/k$. Since all cycles in H are disjoint, we have $h/Opt(H) \le 2 - 1/k$, where h is the number of vertices in the subgraph H. Note that at step 4, the graph G is the original graph G with all vertices in H removed. To avoid confusion, rename the graph G at step 4 by G_4. By Lemma 8.2.10, to prove that the algorithm **VC-Apx-III** has an approximation ratio bounded by $2 - 1/k$, it suffices to proved that the set C_2 constructed in step 6 is a vertex cover of the graph G_4 satisfying $|C_2|/Opt(G_4) \le 2 - 1/k$.

By Lemma 8.2.9, the independent set I in the graph $G(V_0)$ constructed in step 5 has at least $|V_0|/(2k)$ vertices. Therefore, $V_0 - I$ is a vertex cover of $G(V_0)$ with at most $|V_0| - |V_0|/(2k) = |V_0|(1 - 1/(2k))$ vertices. Therefore,

$$\frac{|V_0 - I|}{Opt(G(V_0))} \le \frac{|V_0|(1 - 1/(2k))}{Opt(G(V_0))} \le \frac{|V_0|(1 - 1/(2k))}{|V_0|/2} = 2 - \frac{1}{k}.$$

From this and Corollary 8.2.7, the set $C_2 = C_0 \cup (V_0 - I)$ is a vertex cover of the graph G_4 satisfying

$$\frac{|C_2|}{Opt(G_4)} \le \frac{|V_0 - I|}{Opt(G(V_0))} \le 2 - \frac{1}{k}.$$

Now the inequality $|C|/Opt(G) \le 2-1/k$ follows from Lemma 8.2.10. Thus, the approximation ratio of the algorithm **VC-Apx-III** is bounded by $2 - 1/k$. Since k is the smallest integer satisfying $(2k - 1)^k \ge n$, we can derive using elementary mathematics that $k \le (2 \log n)/(\log \log n)$. This completes the proof of the theorem. $\Box$

The ratio in Theorem 8.2.11 is the best known result for polynomial-time approximation algorithms for the Vertex Cover problem. We point out that the above techniques can be extended to design approximation algorithms with the same ratio for the weighted version of the Vertex Cover problem. Interested readers are referred to [11].

8.3 Maximum satisfiability

Let $X = \{x_1, \ldots, x_n\}$ be a set of boolean variables. A *literal* in X is either a boolean variable x_i or its negation $\overline{x}_i$, for some $1 \leq i \leq n$. A *clause* on X is a disjunction, i.e., an OR, of a set of literals in X. We say that a truth assignment to $\{x_1, \ldots, x_n\}$ *satisfies* a clause if the assignment makes at least one literal in the clause TRUE, and we say that a set of clauses is *satisfiable* if there is an assignment that satisfies all clauses in the set.

SATISFIABILITY (SAT)

INPUT: a set $F = \{C_1, C_2, \ldots, C_m\}$ of clauses on $\{x_1, \ldots, x_n\}$

QUESTION: is F satisfiable?

The SAT problem is the first NP-complete problem, according to the famous Cook's Theorem (see Theorem 1.4.2 in Chapter 1).

If we have further restrictions on the number of literals in each clause, we obtain an interesting subproblem for SAT.

k-SATISFIABILITY (k-SAT)

INPUT: a set $F = \{C_1, C_2, \ldots, C_m\}$ of clauses on $\{x_1, \ldots, x_n\}$ such that each clause has at most k literals

QUESTION: is F satisfiable?

It is well-known that the k-SAT problem remains NP-complete for $k \geq 3$, while the 2-SAT problem can be solved in polynomial time (in fact, in linear time). Interested readers are referred to [30] for details.

As the SAT problem plays a fundamental role in the study of NP-completeness theory, an optimization version of the SAT problem, the MAX-SAT problem, plays a similar role in the study of approximation algorithms.

MAXIMUM SATISFIABILITY (MAX-SAT)

INPUT: a set $F = \{C_1, C_2, \ldots, C_m\}$ of clauses on $\{x_1, \ldots, x_n\}$

OUTPUT: a truth assignment on $\{x_1, \ldots, x_n\}$ that satisfies the maximum number of the clauses in F

The optimization version for the k-SAT problem is defined similarly.

MAXIMUM k-SATISFIABILITY (MAX-kSAT)

INPUT: a set $F = \{C_1, C_2, \ldots, C_m\}$ of clauses on $\{x_1, \ldots, x_n\}$ such that each clause has at most k literals

OUTPUT: a truth assignment on $\{x_1, \ldots, x_n\}$ that satisfies the maximum number of the clauses in F

It is easy to see that the SAT problem can be reduced in polynomial time to the MAX-SAT problem: a set $\{C_1, \ldots, C_m\}$ of clauses is a yes-instance for the SAT problem if and only if when it is regarded as an instance of the MAX-SAT problem, its optimal value is m. Therefore, the MAX-SAT problem is NP-hard. Similarly, the k-SAT problem for $k \geq 3$ can be reduced in polynomial time to the MAX-kSAT problem so the MAX-kSAT problem is NP-hard for $k \geq 3$.

Since the 2-SAT problem can be solved in linear time, one may expect that the corresponding optimization problem MAX-2SAT is also easy. However, the following theorem gives a bit surprising result.

Theorem 8.3.1 *The* MAX-2SAT *problem is* NP*-hard.*

PROOF. We show that the NP-complete problem 3-SAT can be reduced in polynomial time to the MAX-2SAT problem.

Let $F = \{C_1, \ldots, C_m\}$ be an instance of the 3-SAT problem, where each C_i is a clause of at most three literals in $\{x_1, \ldots, x_n\}$. The set F may contain clauses with fewer than three literals. We first show how to convert F into an instance of 3-SAT in which all clauses have exactly three literals.

If a clause C_i in F has exactly two literals: $C_i = (l_1 \vee l_2)$, then we replace C_i by two clauses of three literals $(l_1 \vee l_2 \vee y_1)$ and $(l_1 \vee l_2 \vee \bar{y}_1)$, where y_1 is a new boolean variable; if a clause C_j in F has exactly one literal: $C_j = (l_3)$, then we replace C_j by four clauses of three literals $(l_3 \vee y_2 \vee y_3)$, $(l_3 \vee y_2 \vee \bar{y}_3)$, $(l_3 \vee \bar{y}_2 \vee y_3)$, and $(l_3 \vee \bar{y}_2 \vee \bar{y}_3)$, where y_2 and y_3 are new variables. The resulting set F' of clauses is still an instance for 3-SAT in which each clause has exactly three literals. It is straightforward to see that the instance F is satisfiable if and only if the instance F' is satisfiable.

Thus, we can assume, without loss of generality, that each clause in the given instance F for the 3-SAT problem has exactly three literals.

Consider a clause $C_i = (a_i \vee b_i \vee c_i)$ in F, where a_i, b_i, and c_i are literals in $\{x_1, \ldots, x_n\}$. We construct a set of ten clauses:

$$\begin{aligned} F_i \;=\; & \{(a_i),\;\; (b_i),\;\; (c_i),\;\; (y_i),\;\; (\bar{a}_i \vee \bar{b}_i),\;\; (\bar{a}_i \vee \bar{c}_i), \\ & (\bar{b}_i \vee \bar{c}_i),\;\; (a_i \vee \bar{y}_i),\;\; (b_i \vee \bar{y}_i),\;\; (c_i \vee \bar{y}_i)\} \end{aligned} \qquad (8.2)$$

where y_i is a new variable. It is easy to verify the following facts:

- if all a_i, b_i, c_i are set FALSE, then any assignment to y_i can satisfy at most 6 clauses in F_i;

- if at least one of a_i, b_i, c_i is set TRUE, then there is an assignment to y_i that satisfies 7 clauses in F_i, and no assignment to y_i can satisfy more than 7 clauses in F_i.

Let $F'' = F_1 \cup F_2 \cup \cdots \cup F_m$ be the set of the $10m$ clauses constructed from the m clauses in F using the formula given in (8.2). The set F'' is an instance of the MAX-2SAT problem. It is easy to see that the set F'' can be constructed in polynomial time from the set F.

Suppose that F is a yes-instance of the 3-SAT problem. Then there is an assignment S_x to $\{x_1, \ldots, x_n\}$ that satisfies at least one literal in each C_i of the clauses in F. According to the analysis given above, this assignment S_x plus a proper assignment S_y to the new variable set $\{y_1, \ldots, y_m\}$ will satisfy 7 clauses in the set F_i, for each $i = 1, \ldots, m$. Thus, the assignment $S_x + S_y$ to the boolean variables $\{x_1, \ldots, x_n, y_1, \ldots, y_m\}$ satisfies $7m$ clauses in F''. Since no assignment can satisfy more than 7 clauses in each set F_i, we conclude that in this case the optimal value for the instance F'' of MAX-2SAT is $7m$.

Now suppose that F is a no-instance of the 3-SAT problem. Let S' be any assignment to $\{x_1, \ldots, x_n, y_1, \ldots, y_m\}$. The assignment S' can be decomposed into an assignment S'_x to $\{x_1, \ldots, x_n\}$ and an assignment S'_y to $\{y_1, \ldots, y_m\}$. Since F is a no-instance for the 3-SAT problem, for at least one clause C_i in F, the assignment S'_x makes all literals false. According to our previous analysis, any assignment to y_i plus the assignment S'_x can satisfy at most 6 clauses in the corresponding set F_i. Moreover, since no assignment to $\{x_1, \ldots, x_n, y_1, \ldots, y_m\}$ can satisfy more than 7 clauses in each set F_j, for $j = 1, \ldots, m$, we conclude that the assignment S' can satisfy at most $7(m-1)+6 = 7m-1$ clauses in F''. Since S' is arbitrary, we conclude that in this case, no assignment to $\{x_1, \ldots, x_n, y_1, \ldots, y_m\}$ can satisfy more than $7m-1$ clauses in F''. Thus, in this case the optimal value for the instance F'' of MAX-2SAT is at most $7m-1$.

Summarizing the discussion above, we conclude that the set F of m clauses of three literals is a yes-instance for the 3-SAT problem if and only if the optimal value for the instance F'' of MAX-2SAT is $7m$. Consequently, the 3-SAT problem is polynomial time reducible to the MAX-2SAT problem. We conclude that the MAX-2SAT problem is NP-hard. □

8.3.1 Johnson's algorithm

Now we present an approximation algorithm for the MAX-SAT problem, due to David Johnson [74]. Consider the algorithm given in Figure 8.10, where for a clause C_i, we use $|C_i|$ to denote the number of literals in C_i.

```
Algorithm. Johnson
Input:  a set of clauses $F = \{C_1, \ldots, C_m\}$ on $\{x_1, \ldots, x_n\}$
Output:  a truth assignment $\tau$ to $\{x_1, \ldots, x_n\}$

1.   for (each clause $C_j$) do  $w(C_j) = 1/2^{|C_j|}$;
2.   $L = \{C_1, \ldots, C_m\}$;
3.   for $t = 1$ to $n$ do
3.1      find all clauses $C_1^T, \ldots, C_q^T$ in $L$ that contain $x_t$;
         find all clauses $C_1^F, \ldots, C_s^F$ in $L$ that contain $\overline{x}_t$;
         if $\sum_{i=1}^q w(C_i^T) \geq \sum_{i=1}^s w(C_i^F)$
3.2      then  $\tau(x_t) =$ TRUE;  delete $C_1^T, \ldots, C_q^T$ from $L$;
                 for $i = 1$ to $s$ do  $w(C_i^F) = 2w(C_i^F)$;
3.3      else  $\tau(x_t) =$ FALSE;  delete $C_1^F, \ldots, C_s^F$ from $L$;
                 for $i = 1$ to $q$ do  $w(C_i^T) = 2w(C_i^T)$.
```

Figure 8.10: Johnson's Algorithm.

The algorithm **Johnson** obviously runs in polynomial time. We analyze the approximation ratio for the algorithm.

Lemma 8.3.2 *If each clause in the input instance F contains at least k literals, then the algorithm* **Johnson** *constructs an assignment that satisfies at least $m(1 - 1/2^k)$ clauses in F, where m is the number of clauses in F.*

Proof. In the algorithm **Johnson**, once a literal in a clause is set to TRUE, i.e., once the clause is satisfied, the clause is removed from the set L (see steps 3.2 and 3.3 of the algorithm). Therefore, the number of clauses that are not satisfied by the constructed assignment τ is equal to the number of clauses left in the set L at the end of the algorithm.

Each clause C_i is associated with a weight value $w(C_i)$. Initially, we have $w(C_i) = 1/2^{|C_i|}$ for all C_i. By our assumption, each clause C_i contains at least k literals. So initially we have

$$\sum_{C_i \in L} w(C_i) = \sum_{i=1}^{m} w(C_i) = \sum_{i=1}^{m} 1/2^{|C_i|} \leq \sum_{i=1}^{m} 1/2^k = m/2^k.$$

In step 3, we update the set L and the weight for the clauses in L. It can be easily seen that we never increase the value $\sum_{C_i \in L} w(C_i)$: each time we update the set L, we remove a heavier set of clauses from L and double the weight for a lighter set of clauses remaining in L. Therefore, at end of the algorithm we should still have

$$\sum_{C_i \in L} w(C_i) \leq m/2^k. \tag{8.3}$$

At the end of the algorithm, all boolean variables $\{x_1, \ldots, x_n\}$ have been assigned a value. A clause C_i left in the set L has been considered by the algorithm exactly $|C_i|$ times and each time the corresponding literal in C_i was assigned FALSE. Therefore, for each literal in C_i, the weight of the clause C_i is doubled once. Since initially the clause C_i has weight $1/2^{|C_i|}$ and its weight is doubled exactly $|C_i|$ times in the algorithm, we conclude that at the end of the algorithm, the clause C_i left in L has weight 1. Combining this with the inequality (8.3), we conclude that at the end of the algorithm, the number of clauses in the set L is bounded by $m/2^k$. In other words, the number of clauses satisfied by the constructed assignment τ is at least $m - m/2^k = m(1 - 1/2^k)$. The lemma is proved. □

The observation given in Lemma 8.3.2 derives the following bound on the approximation ratio for the algorithm **Johnson** immediately.

Theorem 8.3.3 *The algorithm* **Johnson** *for the* MAX-SAT *problem has its approximation ratio bounded by* 2.

PROOF. According to Lemma 8.3.2, on an input F of m clauses, each containing at least k literals, the algorithm **Johnson** constructs an assignment that satisfies at least $m(1 - 1/2^k)$ clauses in F. Since a clause in the input F contains at least one literal, i.e., $k \geq 1$, we derive that for any instance F for MAX-SAT, the assignment constructed by the algorithm **Johnson** satisfies at least $m(1 - 1/2) = m/2$ clauses in F. Since the optimal value for the instance F is obviously bounded by m, the approximation ratio for the algorithm must be bounded by $\frac{m}{m/2} = 2$. □

The algorithm **Johnson** has played an important role in the study of approximation algorithms for the MAX-SAT problem. In particular, it is an excellent illustration for the probabilistic method, which has been playing a more and more important role in the design and analysis of approximation algorithms for NP-hard optimization problems. We will re-consider the algorithm **Johnson** in the next section from a different point of view.

8.3.2 Revised analysis on Johnson's algorithm

Theorem 8.3.3 claims that the algorithm **Johnson** has approximation ratio bounded by 2. Is the bound 2 tight for the algorithm? In this subsection, we provide a more careful analysis on the algorithm and show that the approximation ratio of the algorithm is actually 1.5. Readers may skip this subsection in their first reading.

In order to analyze the algorithm **Johnson**, we may need to "flip" a boolean variable x_t, i.e., interchange x_t and $\overline{x}_t$, in an instance for MAX-SAT. This may change the set of clauses satisfied by the assignment τ constructed by the algorithm. In order to take care of this abnormality, we will augment the algorithm **Johnson** by a Boolean array $b[1..n]$. The augmented Boolean array $b[1..n]$ will be part of the input to the algorithm. We call such an algorithm the *augmented Johnson's algorithm.* Our first analysis will be performed on the augmented Johnson's algorithm with an *arbitrarily* augmented Boolean array. The bound on the approximation ratio for the augmented Johnson's algorithm will imply the same bound for the original algorithm **Johnson**.

The augmented Johnson's algorithm is given in Figure 8.11.

```
Augmented Johnson's Algorithm.
INPUT:  a set $F$ of clauses on $\{x_1, \ldots, x_n\}$, and a Boolean array $b[1..n]$
OUTPUT:  a truth assignment $\tau$ to $\{x_1, \ldots, x_n\}$
1.   for (each clause $C_j$ in $F$) do  $w(C_j) = 1/2^{|C_j|}$;
2.   $L = F$;
3.   for $t = 1$ to $n$ do
3.1      find all clauses $C_1^T, \ldots, C_q^T$ in $L$ that contain $x_t$;
         find all clauses $C_1^F, \ldots, C_s^F$ in $L$ that contain $\overline{x}_t$;
         case 1. $\left(\sum_{i=1}^q w(C_i^T) > \sum_{i=1}^s w(C_i^F)\right)$ or
                 $\left(\sum_{i=1}^q w(C_i^T) = \sum_{i=1}^s w(C_i^F)$ and $b[t] = \text{TRUE}\right)$
                   $\tau(x_t) = \text{TRUE}$;  delete $C_1^T, \ldots, C_q^T$ from $L$;
                   for $i = 1$ to $s$ do $w(C_i^F) = 2w(C_i^F)$;
         case 2. $\left(\sum_{i=1}^q w(C_i^T) < \sum_{i=1}^s w(C_i^F)\right)$ or
                 $\left(\sum_{i=1}^q w(C_i^T) = \sum_{i=1}^s w(C_i^F)$ and $b[t] = \text{FALSE}\right)$
                   $\tau(x_t) = \text{FALSE}$;  delete $C_1^F, \ldots, C_s^F$ from $L$;
                   for $i = 1$ to $q$ do $w(C_i^T) = 2w(C_i^T)$.
```

Figure 8.11: The augmented Johnson's algorithm.

The only difference between the original algorithm **Johnson** and the augmented Johnson's algorithm is that in case $\sum_{i=1}^q w(C_i^T) = \sum_{i=1}^s w(C_i^F)$, the original algorithm **Johnson** assigns $\tau(x_t) = \text{TRUE}$ while the augmented Johnson's algorithm assigns $\tau(x_t) = b[t]$.

In the following, we prove a lemma for the augmented Johnson's algorithm. To do this, we need to introduce some terminologies and notations.

A literal is a *positive literal* if it is a Boolean variable x_i for some i, and a *negative literal* if it is the negation $\overline{x}_i$ of a Boolean variable.

Fix an instance $F = \{C_1, \ldots, C_m\}$ for MAX-SAT and let $b[1..n]$ be any fixed Boolean array. Let r be the maximum number of literals in a clause in

F. Apply the augmented Johnson's algorithm on F and $b[1..n]$. Consider a fixed moment in the execution of the augmented Johnson's algorithm. We say that a literal is still *active* if it has not been assigned a truth value yet. A clause C_j in F is *satisfied* if at least one literal in C_j has been assigned value TRUE. A clause C_j is *killed* if all literals in C_j are assigned value FALSE. A clause C_j is *negative* if it is neither satisfied nor killed, and all active literals in C_j are negative literals.

Definition 8.3.1 Fix a t, $0 \leq t \leq n$, and suppose that we are at the end of the t-th iteration of the **for** loop in step 3 of the augmented Johnson's algorithm. Let $S^{(t)}$ be the set of satisfied clauses, $K^{(t)}$ be the set of killed clauses, and $N_i^{(t)}$ be the set of negative clauses with exactly i active literals.

For a set S of clauses, denote by $|S|$ the number of clauses in S, and let $w(S) = \sum_{C_j \in S} w(C_j)$.

Lemma 8.3.4 *For all t, $0 \leq t \leq n$, the sets $S^{(t)}$, $K^{(t)}$, and $N_i^{(t)}$ satisfy the following condition:*

$$|S^{(t)}| \geq 2|K^{(t)}| + \sum_{i=1}^{r} \frac{|N_i^{(t)}|}{2^{i-1}} - A_0,$$

where $A_0 = \sum_{i=1}^{r} |N_i^{(0)}|/2^{i-1}$.

PROOF. The proof proceeds by induction on t. For $t = 0$, since $S^{(0)} = K^{(0)} = \emptyset$, and $\sum_{i=1}^{r} |N_i^{(t)}|/2^{i-1} = A_0$, the lemma holds true.

Suppose $t > 0$. We need to introduce two more notations. At the end of the t-th iteration for the **for** loop in step 3 of the augmented Johnson's algorithm, let $P_{i,j}$ be the set of clauses that contain the positive literal x_{t+1} such that each clause in $P_{i,j}$ contains exactly i active literals, of which exactly j are positive, and let $N_{i,j}$ be the set of clauses that contain the negative literal $\overline{x}_{t+1}$ such that each clause in $N_{i,j}$ contains exactly i active literals, of which exactly j are positive. Note that according to the augmented Johnson's algorithm, if at this moment a clause C_h has exactly i active literals, then the weight value $w(C_h)$ equals exactly $1/2^i$.

Case 1. Suppose that the augmented Johnson's algorithm assigns $\tau(x_{t+1}) =$ TRUE. Then according to the algorithm, regardless of the value $b[t]$ we must have

$$\sum_{i=1}^{r} \sum_{j=1}^{i} w(P_{i,j}) \geq \sum_{i=1}^{r} \sum_{j=0}^{i-1} w(N_{i,j}).$$

This is equivalent to

$$\sum_{i=1}^{r} \frac{\sum_{j=1}^{i} |P_{i,j}|}{2^i} \geq \sum_{i=1}^{r} \frac{\sum_{j=0}^{i-1} |N_{i,j}|}{2^i}. \tag{8.4}$$

Now we have

$$\begin{aligned}
N_1^{(t+1)} &= (N_1^{(t)} - N_{1,0}) \cup N_{2,0}, \\
N_2^{(t+1)} &= (N_2^{(t)} - N_{2,0}) \cup N_{3,0}, \\
&\cdots \\
N_{r-1}^{(t+1)} &= (N_{r-1}^{(t)} - N_{r-1,0}) \cup N_{r,0}, \\
N_r^{(t+1)} &= (N_r^{(t)} - N_{r,0}).
\end{aligned}$$

This gives

$$\begin{aligned}
& |N_1^{(t+1)}| + \frac{1}{2}|N_2^{(t+1)}| + \cdots + \frac{1}{2^{r-1}}|N_r^{(t+1)}| \\
= \; & |N_1^{(t)}| + \frac{1}{2}|N_2^{(t)}| + \cdots + \frac{1}{2^{r-1}}|N_r^{(t)}| \qquad (8.5) \\
& \quad -|N_{1,0}| + \frac{1}{2}|N_{2,0}| + \frac{1}{2^2}|N_{3,0}| + \cdots + \frac{1}{2^{r-1}}|N_{r,0}| \\
= \; & \sum_{i=1}^{r} \frac{|N_i^{(t)}|}{2^{i-1}} + \sum_{i=1}^{r} \frac{|N_{i,0}|}{2^{i-1}} - 2|N_{1,0}|.
\end{aligned}$$

On the other hand, we have

$$S^{(t+1)} = S^{(t)} \cup \bigcup_{i=1}^{r} \bigcup_{j=1}^{i} P_{i,j} \quad \text{and} \quad K^{(t+1)} = K^{(t)} \cup N_{1,0}. \tag{8.6}$$

Combining relations (8.4)-(8.6), and using the inductive hypothesis, we get

$$\begin{aligned}
|S^{(t+1)}| &= |S^{(t)}| + \sum_{i=1}^{r} \sum_{j=1}^{i} |P_{i,j}| \\
&\geq 2|K^{(t)}| + \sum_{i=1}^{r} \frac{|N_i^{(t)}|}{2^{i-1}} - A_0 + \sum_{i=1}^{r} \frac{\sum_{j=1}^{i} |P_{i,j}|}{2^{i-1}} \\
&\geq 2|K^{(t)}| + \sum_{i=1}^{r} \frac{|N_i^{(t)}|}{2^{i-1}} - A_0 + \sum_{i=1}^{r} \frac{\sum_{j=0}^{i-1} |N_{i,j}|}{2^{i-1}} \\
&\geq 2(|K^{(t)}| + |N_{1,0}|) + \sum_{i=1}^{r} \frac{|N_i^{(t)}|}{2^{i-1}} + \sum_{i=1}^{r} \frac{|N_{i,0}|}{2^{i-1}} - 2|N_{1,0}| - A_0
\end{aligned}$$

$$= \quad 2|K^{(t+1)}| + \sum_{i=1}^{r} \frac{|N_i^{(t+1)}|}{2^{i-1}} - A_0.$$

Therefore, the induction goes through in this case.

Case 2. Suppose that the augmented Johnson's algorithm assigns $\tau(x_{t+1}) = \text{FALSE}$. The proof for this case is similar but slightly more complicated. We will concentrate on describing the differences.

According to the augmented Johnson's algorithm, we have

$$\sum_{i=1}^{r} \frac{\sum_{j=1}^{i} |P_{i,j}|}{2^i} \le \sum_{i=1}^{r} \frac{\sum_{j=0}^{i-1} |N_{i,j}|}{2^i}. \tag{8.7}$$

Based on the relations

$$\begin{aligned}
N_1^{(t+1)} &= (N_1^{(t)} - N_{1,0}) \cup P_{2,1}, \\
N_2^{(t+1)} &= (N_2^{(t)} - N_{2,0}) \cup P_{3,1}, \\
&\cdots \\
N_{r-1}^{(t+1)} &= (N_{r-1}^{(t)} - N_{r-1,0}) \cup P_{r,1}, \\
N_r^{(t+1)} &= (N_r^{(t)} - N_{r,0}),
\end{aligned}$$

we get

$$\begin{aligned}
& |N_1^{(t+1)}| + \frac{1}{2}|N_2^{(t+1)}| + \cdots + \frac{1}{2^{r-1}}|N_r^{(t+1)}| \qquad (8.8)\\
= \quad & \sum_{i=1}^{r} \frac{|N_i^{(t)}|}{2^{i-1}} + \sum_{i=2}^{r} \frac{|P_{i,1}|}{2^{i-2}} - \sum_{i=1}^{r} \frac{|N_{i,0}|}{2^{i-1}}.
\end{aligned}$$

Moreover, we have

$$S^{(t+1)} = S^{(t)} \cup \bigcup_{i=1}^{r} \bigcup_{j=0}^{i-1} N_{i,j} \quad \text{and} \quad K^{(t+1)} = K^{(t)} \cup P_{1,1}. \tag{8.9}$$

Combining relations (8.8) and (8.9) and using the inductive hypothesis,

$$\begin{aligned}
& 2|K^{(t+1)}| + \sum_{i=1}^{r} \frac{|N_i^{(t+1)}|}{2^{i-1}} - A_0 \\
= \quad & 2|K^{(t)}| + 2|P_{1,1}| + \sum_{i=1}^{r} \frac{|N_i^{(t)}|}{2^{i-1}} + \sum_{i=2}^{r} \frac{|P_{i,1}|}{2^{i-2}} - \sum_{i=1}^{r} \frac{|N_{i,0}|}{2^{i-1}} - A_0
\end{aligned}$$

$$\begin{aligned}
&\leq |S^{(t)}| + \sum_{i=1}^{r} \frac{|P_{i,1}|}{2^{i-2}} - \sum_{i=1}^{r} \frac{|N_{i,0}|}{2^{i-1}} \\
&= |S^{(t)}| + \sum_{i=1}^{r}\sum_{j=0}^{i-1} |N_{i,j}| + \sum_{i=1}^{r} \frac{|P_{i,1}|}{2^{i-2}} - \sum_{i=1}^{r} \frac{|N_{i,0}|}{2^{i-1}} - \sum_{i=1}^{r}\sum_{j=0}^{i-1} |N_{i,j}|.
\end{aligned}$$

Now according to equation (8.9),

$$|S^{(t+1)}| = |S^{(t)}| + \sum_{i=1}^{r}\sum_{j=0}^{i-1} |N_{i,j}|.$$

Moreover, since

$$\begin{aligned}
&\sum_{i=1}^{r} \frac{|N_{i,0}|}{2^{i-1}} + \sum_{i=1}^{r}\sum_{j=0}^{i-1} |N_{i,j}| \geq |N_{1,0}| + |N_{1,0}| + \sum_{i=2}^{r}\sum_{j=0}^{i-1} |N_{i,j}| \\
\geq\ & 2|N_{1,0}| + \sum_{i=2}^{r} \frac{\sum_{j=0}^{i-1} |N_{i,j}|}{2^{i-2}} = \sum_{i=1}^{r} \frac{\sum_{j=0}^{i-1} |N_{i,j}|}{2^{i-2}} \geq \sum_{i=1}^{r} \frac{\sum_{j=1}^{i} |P_{i,j}|}{2^{i-2}} \\
\geq\ & \sum_{i=1}^{r} \frac{|P_{i,1}|}{2^{i-2}},
\end{aligned}$$

where the third inequality above follows from relation (8.7), we conclude

$$2|K^{(t+1)}| + \sum_{i=1}^{r} \frac{|N_i^{(t+1)}|}{2^{i-1}} - A_0 \leq |S^{(t+1)}|.$$

Thus, the induction also goes through in this case.

The lemma now follows directly from the inductive proof. □

Now we are ready to prove our main theorem. Let us come back to the original algorithm **Johnson**.

Theorem 8.3.5 *The approximation ratio for the algorithm* **Johnson** *given in Figure 8.10 for the* Max-Sat *problem is* 1.5. *This bound is tight.*

Proof. Let F be an instance of the Max-Sat problem. Let τ_o be an arbitrary optimal assignment to F. We construct another instance F' for Max-Sat, as follows. Starting with F, if for a Boolean variable x_t, we have $\tau_o(x_t) =$ false, then we "flip" x_t (i.e., interchange x_t and $\overline{x}_t$) in F. Thus, there is a one-to-one correspondence between the set of clauses in F and the set of clauses in F'. It is easy to see that the sets F and F', as instances for MAX-SAT, have the same optimal value. In particular, the assignment τ'_o on F' such that $\tau'_o(x_t) = \text{TRUE}$ for all t is an optimal assignment for the instance F'.

We let a Boolean array $b[1..n]$ be such that $b[t] = \tau_o(x_t)$ for all t.

We show that the assignment constructed by the original algorithm **Johnson** on the instance F and the assignment constructed by the augmented Johnson's algorithm on the instance F' augmented by the Boolean array $b[1..n]$ satisfy exactly the same set of clauses.

Inductively, suppose that for the first $(t-1)$-st iterations of the **for** loop in step 3, both algorithms satisfy exactly the same set of clauses. Now consider the t-th iteration of the algorithms.

If x_t in F is not flipped in F', then $b[t] = \text{TRUE}$. Thus, the augmented Johnson's algorithm assigns $\tau(x_t) = \text{TRUE}$ and makes the clauses $C_1^T, \ldots, C_q^T$ satisfied during the t-th iteration if and only if $\sum_{i=1}^q w(C_i^T) \geq \sum_{i=1}^s w(C_i^F)$, where $C_1^T, \ldots, C_q^T$ are the clauses in L containing x_t and $C_1^F, \ldots, C_s^F$ are the clauses in L containing $\overline{x}_t$. On the other hand, if x_t in F is flipped in F', then $b[t] = \text{FALSE}$, and the augmented Johnson's algorithm assigns $\tau(x_t) = \text{FALSE}$ and makes the clauses $C_1^F, \ldots, C_s^F$ satisfied if and only if $\sum_{i=1}^q w(C_i^T) \leq \sum_{i=1}^s w(C_i^F)$. Note that if x_t is not flipped, then $\{C_1^T, \ldots, C_q^T\}$ is exactly the set of clauses containing x_t in the t-th iteration of the original algorithm **Johnson** for the instance F, while if x_t is flipped, then $\{C_1^F, \ldots, C_s^F\}$ is exactly the set of clauses containing x_t in the t-th iteration of the original algorithm **Johnson** for the instance F. Therefore, in the t-th iteration, the set of the clauses satisfied by the augmented Johnson's algorithm on F' and $b[1..n]$ corresponds exactly to the set of clauses satisfied by the original algorithm **Johnson** on F. In conclusion, the assignment constructed by the original algorithm **Johnson** on the instance F and the assignment constructed by the augmented Johnson's algorithm on the instance F' and the Boolean array $b[1..n]$ satisfy exactly the same set of clauses.

Therefore, we only need to analyze the approximation ratio of the augmented Johnson's algorithm on the instance F' and the Boolean array $b[1..n]$.

Let $K^{(t)}$, $S^{(t)}$, and $N_i^{(t)}$ be the sets defined before for the augmented Johnson's algorithm on the instance F' and the Boolean array $b[1..n]$. According to Lemma 8.3.4, we have

$$|S^{(t)}| \geq 2|K^{(t)}| + \sum_{i=1}^{r} \frac{|N_i^{(t)}|}{2^{i-1}} - A_0, \tag{8.10}$$

for all $0 \leq t \leq n$, where $A_0 = \sum_{i=1}^r |N_i^{(0)}|/2^{i-1}$.

At the end of the augmented Johnson's algorithm, i.e., $t = n$, $S^{(n)}$ is exactly the set of clauses satisfied by the assignment constructed by the algorithm, and $K^{(n)}$ is exactly the set of clauses not satisfied by the assignment. Moreover, $N_i^{(n)} = \emptyset$ for all $i \geq 1$.

By (8.10), we have

$$|S^{(n)}| \geq 2|K^{(n)}| - A_0. \tag{8.11}$$

Also notice that

$$A_0 = \sum_{i=1}^{r} \frac{|N_i^{(0)}|}{2^{i-1}} \leq \sum_{i=1}^{r} |N_i^{(0)}|. \tag{8.12}$$

Combining relations (8.11) and (8.12), we get

$$3|S^{(n)}| \geq 2(|S^{(n)}| + |K^{(n)}|) - \sum_{i=1}^{r} |N_i^{(0)}|. \tag{8.13}$$

Since $S^{(n)} \cup K^{(n)}$ is the whole set $\{C_1, \ldots, C_m\}$ of clauses in F', we have $|S^{(n)}| + |K^{(n)}| = m$. Moreover, the assignment $\tau_o'(x_t) = \text{TRUE}$ for all $1 \leq t \leq n$ is an optimal assignment to the instance F', which satisfies all clauses in F' except those in $N_i^{(0)}$, for $1 \leq i \leq r$. Thus, the optimal value of the instance F', i.e., the number of clauses satisfied by an optimal assignment to F' is equal to

$$Opt(F') = m - \sum_{i=1}^{r} |N_i^{(0)}|. \tag{8.14}$$

Now combining the relations (8.14) and (8.13), we get

$$3|S^{(n)}| \geq m + Opt(F') \geq 2 \cdot Opt(F').$$

The set $S^{(n)}$ is the set of clauses satisfied by the assignment constructed by the augmented Johnson's algorithm. Since the original algorithm **Johnson** and the augmented Johnson's algorithm satisfy the same set of clauses and since $Opt(F) = Opt(F')$, we conclude that the approximation ratio of the algorithm **Johnson** for the MAX-SAT problem is $Opt(F')/|S^{(n)}| \leq 1.5$.

To see that the bound 1.5 is tight for the algorithm **Johnson**, consider the following instance F_h of $3h$ clauses for MAX-SAT, where h is any integer larger than 0.

$$F_h = \{(x_{3k+1} \vee x_{3k+2}), (x_{3k+1} \vee x_{3k+3}), (\overline{x}_{3k+1}) \mid 0 \leq k \leq h-1\}.$$

It is easy to verify that the algorithm **Johnson** assigns $x_t = \text{TRUE}$ for all $1 \leq t \leq 3h$, and this assignment satisfies exactly $2h$ clauses in F_h. On the other hand, the assignment $x_{3k+1} = \text{FALSE}$, $x_{3k+2} = x_{3k+3} = \text{TRUE}$ for all $0 \leq k \leq h-1$ obviously satisfies all $3h$ clauses in F_h. $\square$

Theorem 8.3.5 gives an example in which the precise approximation ratio for a simple algorithm is difficult to derive. The next question is whether the approximation ratio 1.5 on the MAX-SAT problem can be further improved by a "better" algorithm for the problem. This has been a very active research topic in the past decade. In the next chapter, we will develop new techniques that give better approximation algorithms for the MAX-SAT problem.