2 Approximation Algorithms for Knapsack

Recall the optimization problem KNAPSACK. Although this problem is **NP**-complete, meaning that it does not have a polynomial-time algorithm unless **P** = **NP**, there are still many different approaches to attack this problem, giving slow exact solutions or fast approximation solutions. We first use the technique of dynamic programming to find an exact solution for KNAPSACK.

To simplify the description of the algorithm, we first define some notations. For any subset $I \subseteq \{1, \ldots, n\}$, let S_I denote the sum $\sum_{k \in I} s_k$. For each pair (i, j), with $1 \le i \le n, 0 \le j \le S$, let $c(i, j)$ be a subset of the index set $\{1, \ldots, i\}$ such that $\sum_{k \in c(i,j)} c_k = j$ and $S_{c(i,j)} \le S$ if such an index subset exists. If such an index subset does not exist, then we say that $c(i, j)$ is undefined, and write $c(i, j) = nil$.

Using the above notation, it is clear that $c^* = \max\{j \mid c(n, j) \ne nil\}$. Therefore, it suffices to compute all values of $c(i, j)$. The following algorithm is based on this idea.

Algorithm C (*Exact Algorithm for* KNAPSACK)
input: Positive integers $S, s_1, c_1, s_2, c_2, \ldots, s_n, c_n$.

```
(1) Let $c_{sum} \leftarrow \sum_{i=1}^{n} c_i$.
(2) For $j \leftarrow 0$ to $c_{sum}$ do $c(1,j) \leftarrow \begin{cases} \emptyset & \text{if } j = 0, \\ \{1\} & \text{if } j = c_1, \\ nil & \text{otherwise;} \end{cases}$
(3) For $i \leftarrow 2$ to $n$ do
        for $j \leftarrow 0$ to $c_{sum}$ do
            if $[c(i-1, j-c_i) \ne nil]$ and $[S_{c(i-1,j-c_i)} \le S - s_i]$
              and $[c(i-1, j) \ne nil \Rightarrow S_{c(i-1,j)} > S_{c(i-1,j-c_i)} + s_i]$
                then $c(i, j) \leftarrow c(i-1, j-c_i) \cup \{i\}$
                else $c(i, j) \leftarrow c(i-1, j)$;
(4) Output $c^* \leftarrow \max\{j \mid c(n, j) \ne nil\}$.                    ■
```

To show the correctness of Algorithm C, let us define some more notations. For $1 \le i \le n$ and $0 \le j \le S$, let $\mathcal{I}_{i,j}$ be the collection of all subsets $I \subseteq \{1, \ldots, i\}$ such that $\sum_{k \in I} c_k = j$ and $S_I \le S$. Also define $S^*_{i,j} = \min\{S_I \mid I \in \mathcal{I}_{i,j}\}$. Then, we note that each $c(i, j)$ obtained in the algorithm has an additional property:

$$\text{If } \mathcal{I}_{i,j} \ne \emptyset \text{ then } c(i,j) \in \mathcal{I}_{i,j} \text{ and } S_{c(i,j)} = S^*_{i,j}. \tag{2}$$

This property can be proved by induction on i. For $i = 1$, it is trivial. For $i \ge 2$, we fix a pair (i, j) and consider two cases:

Case 1. $c(i-1,j) \in \mathcal{I}_{i,j}$ and $S_{c(i-1,j)} = S^*_{i,j}$. Then, in step (3) of Algorithm C, the third condition of the **if**-statement is false with respect to the pair (i,j), and so it sets $c(i,j) \leftarrow c(i-1,j)$ and property (2) holds.

Case 2. Not Case 1; that is, either $c(i-1,j) = nil$ or $c(i-1,j) \in \mathcal{I}_{i-1,j}$ but $S_{c(i-1,j)} > S^*_{i,j}$. Assume that $\mathcal{I}_{i,j} \neq \emptyset$. From the inductive hypothesis, we know that if $c(i-1,j) \neq nil$ then $S^*_{i-1,j} = S_{c(i-1,j)} > S^*_{i,j}$. Therefore, for all $I \in \mathcal{I}_{i,j}$ with $S_I = S^*_{i,j}$, I must contain i, and hence $I - \{i\} \in \mathcal{I}_{i-1,j-c_i}$. In addition, for such a set I, we have $S_{I-\{i\}} = S^*_{i-1,j-c_i}$, and $S^*_{i,j} = S^*_{i-1,j-c_i} + s_i$.

Now, by the inductive hypothesis, we know that $c(i-1,j-c_i) \in \mathcal{I}_{i-1,j-c_i}$ and $S_{c(i-1,j-c_i)} = S^*_{i-1,j-c_i}$. It is easy to verify that, in step (3) of Algorithm C, the three conditions of the **if**-statement, with respect to the pair (i,j), hold true, and therefore it sets $c(i,j) = c(i-1,j-c_i) \cup \{i\}$. This set $c(i,j)$ satisfies $S_{c(i,j)} = S_{c(i-1,j-c_i)} + s_i = S^*_{i-1,j-c_i} + s_i = S^*_{i,j}$, and hence satisfies property (2).

From property (2), we see immediately that $c^* = \max\{j \mid \mathcal{I}_{n,j} \neq \emptyset\} = \max\{j \mid c(n,j) \neq nil\}$. Thus, step (4) correctly finds the optimum solution. This completes the proof of the correctness of Algorithm C.

Next, let us consider the time complexity of this algorithm. It is easy to see that, for any $I \subseteq \{1,\ldots,n\}$, it takes time $O(n \lg S)$ to compute S_I. Thus, Algorithm C runs in time $O(n^3 M \lg(MS))$ where $M = \max\{c_k \mid 1 \leq k \leq n\}$ (note that $c_{sum} = O(nM)$). Since the input size of the problem is $n \log M + \lg S$, we see that Algorithm C is *not* a polynomial-time algorithm. It is actually a *pseudo* polynomial-time algorithm, in the sense that it runs in time polynomial in the maximum input value but not necessarily polynomial in the input size. Since the input value could be very large, a pseudo polynomial-time algorithm is usually not considered as an efficient algorithm.

As a compromise, we might find a faster approximation algorithm more useful. For instance, the following is such an approximation algorithm, which uses a simple greedy strategy that selects the *heaviest* item (i.e., the item with the greatest *density* c_i/s_i) first.

Algorithm D (*Greedy Algorithm for* KNAPSACK)

Input: Same as in Algorithm C.

```
(1) Sort all items in the decreasing order of $c_i/s_i$. Without loss of generality,
    assume that $\frac{c_1}{s_1} \geq \frac{c_2}{s_2} \geq \cdots \geq \frac{c_n}{s_n}$.
(2) If $\sum_{i=1}^{n} s_i \leq S$ then output $c_G \leftarrow \sum_{i=1}^{n} c_i$
    else find the maximum $k$ such that $\sum_{i=1}^{k} s_i \leq S < \sum_{i=1}^{k+1} s_i$
        and output $c_G \leftarrow \max\{c_{k+1}, \sum_{i=1}^{k} c_i\}$.   ■
```

It is clear that this greedy algorithm runs in time $O(n \log(nMS))$, and hence is very efficient. The following theorem shows that it produces an approximate solution not very far from the optimal.

Theorem 2.1 *Let c^* be the optimal solution to the problem* KNAPSACK *and c_G the approximate solution obtained by* Algorithm D. *Then $c^* \leq 2c_G$.*

Proof. If $\sum_{i=1}^{n} s_i \leq S$ then $c_G = c^*$. Thus, we may assume $\sum_{i=1}^{n} s_i > S$. Let k be the integer found by Algorithm D in step (2). We caim that

$$\sum_{i=1}^{k} c_i \leq c^* < \sum_{i=1}^{k+1} c_i. \tag{3}$$

The first half of the above inequality holds trivially. For the second half, we note that, in step (1), we sorted the items according to their density c_i/s_i. Therefore, if we are allowed to cut each item into small pieces, then the most efficient way to use the knapsack is to load the first k items, plus a portion of the $(k+1)$st item, because replacing any protion of these items by other items decreases the total density of the knapsack. This shows that the maximum total value we can get is less than $\sum_{i=1}^{k+1} c_i$, which is therefore an upper bound for c^*. □

The above two algorithms demonstrate an interesting tradeoff between the running time and accuracy of an algorithm: If we sacrifice a little in the accuracy of the solution, we may get a much more efficient algorithm. Indeed, we can further generalize the above greedy algorithm and get even better approximate solutions—though with worse running time. The idea is as follows: We divide all items into two groups: those with values $c_i \leq a$ and those with $c_i > a$, where a is a fixed parameter. Note that in any feasible solution $I \subseteq \{1, 2, \ldots, n\}$, there can be at most $c^*/a \leq 2c_G/a$ items that have values c_i greater than a. So, we perform an exhaustive search over all index subsets of size at most $2c_G/a$ from the second group, and use the greedy strategy to find an approximate solution from the first group. From Theorem 2.1, we know that our error is bounded by the value of a single item of the first group, which is at most a. In addition, we note that there are at most $n^{2c_G/a}$ index subsets of the second group to be searched through, and so the running time is still a polynomial function in the input size.

In the following, we write $|A|$ to denote the size of a finite set A.

Algorithm E (*Generalized Greedy Algorithm for* KNAPSACK)

Input: Same as Algorithm C; and a constant $0 < \varepsilon < 1$.

(1) Run Algorithm D on the input to get value c_G.

(2) Let $a \leftarrow \varepsilon c_G$.

(3) Let $I_a \leftarrow \{i \mid 1 \leq i \leq n, c_i \leq a\}$. (Without loss of generality, assume that $I_a = \{i, \ldots, m\}$, where $m \leq n$.)

(4) Sort the items in I_a in the decreasing order of c_i/s_i. Without loss of generality, assume that $\frac{c_1}{s_1} \geq \frac{c_2}{s_2} \geq \cdots \geq \frac{c_m}{s_m}$.

```
(5) For each $I \subseteq \{m+1, m+2, \cdots, n\}$ with $|I| \le 2/\varepsilon$ do
        if $\sum_{i \in I} s_i > S$ then let $c(I) \leftarrow 0$
        else if $\sum_{i=1}^{m} s_i \le S - \sum_{i \in I} s_i$
                then let $c(I) \leftarrow \sum_{i=1}^{m} c_i + \sum_{i \in I} c_i$
                else find the maximum $k$ such that
                        $\sum_{i=1}^{k} s_i \le S - \sum_{i \in I} s_i < \sum_{i=1}^{k+1} s_i$
                    and let $c(I) \leftarrow \sum_{i=1}^{k} c_i + \sum_{i \in I} c_i$.
(6) Output $c_{GG} \leftarrow \max\{c(I) \mid I \subseteq \{m+1, m+2, \cdots, n\}, |I| \le 2/\varepsilon\}$.   ■
```

Theorem 2.2 *Let c^* be the optimal solution to* Knapsack *and c_{GG} the approximation obtained by Algorithm* E. *Then $c^* \le (1+\varepsilon)c_{GG}$. Moreover, Algorithm* E *runs in time $O(n^{1+2/\varepsilon}\log(nMS))$.*

Proof. Let I^* be the optimal index set; that is, $\sum_{i \in I^*} c_i = c^*$ and $\sum_{i \in I^*} s_i \le S$. Define $\overline{I} = \{i \in I^* \mid c_i > a\}$. We have already shown that $|\overline{I}| \le c^*/a \le 2c_G/a = 2/\varepsilon$. Therefore, in step (5) of Algorithm E, the index set I will eventually be set to $\overline{I}$, and the greedy strategy, as shown in the proof of Theorem 2.1, will find $c(\overline{I})$ with the property

$$c(\overline{I}) \le c^* \le c(\overline{I}) + a.$$

Since c_{GG} is the maximum of $c(I)$'s, we get

$$c(\overline{I}) \le c_{GG} \le c^* \le c(\overline{I}) + a \le c_{GG} + a.$$

It follows that

$$c^* \le (1+\varepsilon)c_{GG}.$$

Note that there are at most $n^{2/\varepsilon}$ index sets I of size $|I| \le 2/\varepsilon$. Therefore, the running time of Algorithm E is $O(n^{1+2/\varepsilon}\log(nMS))$. □

By Theorem 2.2, for any fixed $\varepsilon > 0$, Algorithm E runs in polynomial time: $O(n^{1+2/\varepsilon}\log(nMS))$. As ε decreases to zero, however, the running time increases exponentially with respect to $1/\varepsilon$. Can we slow down the speed of increase of the running time with respect to $1/\varepsilon$? The answer is yes. The following is one of such approximation algorithms.

Algorithm F (*Polynomial Tradoff Approximation for* Knapsack)
Input: Same as Algorithm C; and an integer $h > 0$.
(1) **For** each $k \leftarrow 1, \cdots, \lfloor c_k n(h+1)/M \rfloor$, where $M = \max\{c_i \mid 1 \le i \le n\}$.

(2) Run Algorithm C on the following instance of KNAPSACK:

$$\begin{aligned}\text{maximize} \quad & c'_1x_1 + c'_2x_2 + \cdots + c'_nx_n \\ \text{subject to} \quad & s_1x_1 + s_2x_2 + \cdots + s_nx_n \le S, \\ & x_1, x_2, \ldots, x_n \in \{0,1\}.\end{aligned}$$

Let $(x_1^*, \ldots, x_n^*)$ be the optimal solution found by Algorithm C (i.e., the index set corresponding to the optimal solution $(c')^*$).

(3) Output $c_{PT} \leftarrow c_1x_1^* + \cdots + c_nx_n^*$. ■

Theorem 2.3 *The solution obtained by Algorithm* F *satisfies the relation*

$$\frac{c^*}{c_{PT}} \le 1 + \frac{1}{h},$$

where c^* is the optimal solution to the input instance.

Proof. Let I^* be the optimal index set of the input instance; that is, $c^* = \sum_{k \in I^*} c_k$. Also, let $J^* = \{k \mid 1 \le k \le n, x_k^* = 1\}$. Then, we have

$$\begin{aligned} c_{PT} &= \sum_{k \in J^*} c_k = \sum_{k \in J^*} \frac{c_k n(h+1)}{M} \cdot \frac{M}{n(h+1)} \\ &\ge \sum_{k \in J^*} \left\lfloor \frac{c_k n(h+1)}{M} \right\rfloor \cdot \frac{M}{n(h+1)} \\ &= \frac{M}{n(h+1)} \sum_{k \in J^*} c'_k \ge \frac{M}{n(h+1)} \sum_{k \in I^*} c'_k \\ &\ge \frac{M}{n(h+1)} \sum_{k \in I^*} \left(\frac{c_k n(h+1)}{M} - 1 \right) \\ &\ge c^* - \frac{M}{h+1} \ge c^* \left(1 - \frac{1}{h+1} \right). \end{aligned}$$

In the above, the second inequality holds because J^* is the optimal solution to the modified instance of KNAPSACK; and the last inequality holds because $M = \max_{1 \le i \le n}\{c_i\} \le c^*$. Thus,

$$\frac{c^*}{c_{PT}} \le \frac{1}{1 - 1/(h+1)} = 1 + \frac{1}{h}. \qquad \square$$

We note that Algorithm C runs on the modified instance in time $O(n^3 M' \log(M'S))$, where $M' = \max\{c'_k \mid 1 \le k \le n\} \le n(h+1)$. That is, the running time of Algorithm F is $O(n^4 h \log(nhS))$, which is a polynomial function with respect to n, $\log S$, and $h = 1/\varepsilon$. Thus, the tradeoff between running time and approximation ratio of Algorithm F is better than that of the generalized greedy algorithm.

3 Performance Ratios

From the example of Knapsack, we see that the two most important criteria in the study of approximation algorithms are efficiency and the performance ratio. By efficiency, we mean polynomial-time computability. By performance ratio, we mean the ratio of the objective function values between the approximate and the optimal solutions. More precisely, for any optimization problem Π and any input instance I, let $OPT(I)$ denote the objective function value of the optimal solution to instance I, and $A(I)$ the objective function value produced by an approximation algorithm A on instance I. Then, for a minimization problem, we define the *performance ratio* of an approximation algorithm A to be

$$r(A) = \sup_I \frac{A(I)}{OPT(I)},$$

and, for a maximization problem, we define it to be

$$r(A) = \sup_I \frac{OPT(I)}{A(I)},$$

where I ranges over all possible input instances. Thus, for any approximation algorithm A, $r(A) \geq 1$, and, in general, the smaller the performance ratio is, the better the approximation algorithm is.

For instance, consider the maximization problem Knapsack again. Let $c^*(I)$ be the maximum value of the objective function on input instance I, and $c_G(I)$ and $c_{GG}(I)$ the objective function values obtained by Algorithms B and C, respectively, on instance I. Then, by Theorems 2.1 and 2.2, the performance ratios of these two algorithms are

$$r(\mathrm{D}) = \sup_I \frac{c^*(I)}{c_G(I)} \leq 2,$$

and

$$r(\mathrm{E}) = \sup_I \frac{c^*(I)}{c_G(I)} \leq 1 + \varepsilon.$$

That is, both of these algorithms achieve a constant approximation ratio, but Algorithm E has a better ratio.

As another example, consider the famous Traveling Salesman Problem (TSP) defined in the last section. We assume that the distance between any two vertices is positive. In addition, we assume that the given distance function d satisfies the *triangle inequality*, that is,

$$d(a, b) + d(b, c) \geq d(a, c)$$

for any three vertices a, b and c. Then, there is a simple approximation algorithm for TSP that finds a tour (i.e., a Hamiltonian circuit) with the total distance within twice of the optimal. This algorithm uses two basic linear-time algorithms on graphs:

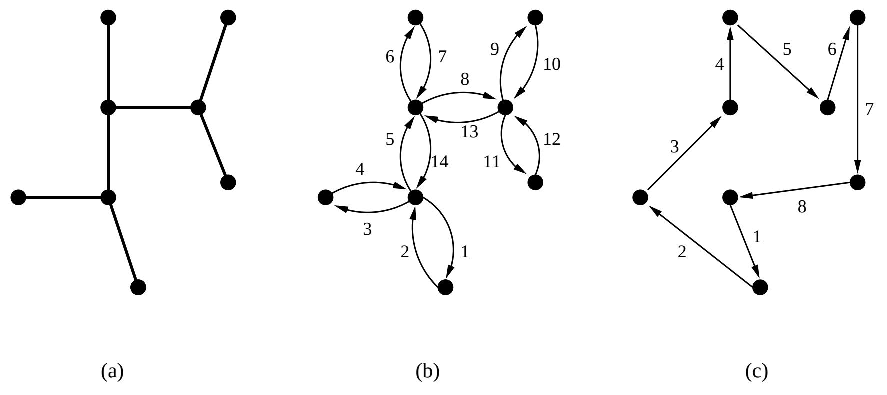

Figure 1: Algorithm G: (a) the minimum spanning tree, (b) the Euler tour, and (c) the shortcut.

> *Minimum Spanning Tree Algorithm*: Given a connected graph G with a distance function d on all edges, this algorithm finds a minimum spanning tree T of the graph G. (T is a *minimum spanning tree* of G if T is a connected subgraph of G with the minimum total distance.)
>
> *Euler Tour Algorithm*: Given a connected graph G in which each vertex has an even degree, this algorithm finds an Euler tour, i.e., a cycle that passes through each edge in G exactly once.

Algorithm G (*Approximation Algorithm for* TSP *with* Δ*-Inequality*)

Input: A complete graph $G = (V, E)$, where $V = \{1, 2, \ldots, n\}$, and a distance function $d : V \times V \to \mathbb{N}$ which satisfies the triangle inequality.

(1) Find a minimum spanning tree T of G.

(2) Change each edge e in T to two (parallel) edges between the same pair of vertices. Call the resulting graph H.

(3) Find an Euler tour P of H.

(4) Output the Hamiltonian circuit Q that is obtained by visiting each vertex once in the order of their first occurence in P. (That is, Q is the *shortcut* of P which skips a vertex if it has already been visited. See Figure 1.) ■

We first note that, after step (2), each vertex in graph H has an even degree and hence the Euler Tour Algorithm can find an Euler tour of H in linear time. Thus, Algorithm G is well-defined. Next, we verify that its performance ratio is bounded by two. This is easy to see from the following three observations: (a) The total distance of the minimum spanning tree T must be less than that

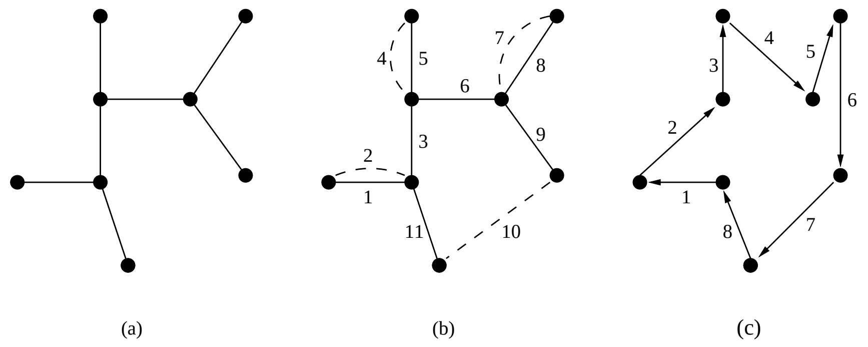

Figure 2: Christofides's approximation: (a) The minimum spanning tree, (b) the minimum matching (shown in broken lines) and the Euler tour, and (c) the shortcut.

of any Hamiltonian cicuit C, since we can obtain a spanning tree by removing an edge from C. (b) The total distance of P is exactly twice of that of T, and so at most twice of that of the optimal solution. (c) By the triangle inequality, the total distance of the shortcut Q is no greater than that of tour P.

Christofides [1976] introduced a new idea into this approximation algorithm and improved the performance ratio to 3/2. This new idea requires another basic graph algorithm:

> *Minimum Perfect-Matching Algorithm*: Given a complete graph G of an even number of vertices and a distance function d on edges, this algorithm finds a perfect matching with the minimum total distance. (A *matching* of a graph G is a subset M of the edges such that each vertex occurs in at most one edge in M. A *perfect matching* of a matching M in which each vertex occurs in exactly one edge.)

Algorithm H (*Christofides's Algorithm for* TSP *with* Δ*-Inequality*)
Input: Same as Algorithm G.

(1) Find a minimum spanning tree $T = (V, E_T)$ of G.

(2) Let V' be the set of all vertices in T of odd degrees, and let $G' = (V', E')$ be the subgraph of G induced by vertex set V'. Find a minimum perfect matching M for G'. Add the edges in M to tree T (with possible parallel edges between two vertices) to form a new graph H'. (See Figure 2.)

(3) Find an Euler tour P' of H'.

(4) Output the shortcut Q of the tour P' as in Step (4) of Algorithm G. ■

To understand Christofides's algorithm, we first note that, in step (2), $|V'|$ must be even, since the total degree of all vertices must be even. Thus, the minimum perfect matching M exists. Next, we observe that after adding the matching M to tree T, each vertex in graph H' has an even degree. Therefore, step (3) of Algorithm H is also well-defined. Now, we note that the total distance of the matching M is at most one half of that of a minimum Hamiltonian circuit C in G', since we can remove alternating edges from C to obtain a matching. Also, by the triangle inequality, the total distance of the minimum Hamiltonian circuit in G' is smaller than that of the minimum Hamiltonian circuit in G. Therefore, the total distance of the tour P', as well as that of Q, is at most $3/2$ of the optimal solution. That is, the performance ratio of Algorithm H is bounded by $3/2$.

Actually, the performance ratio of Christofides's approximation can be shown to be exactly $3/2$. Consider the graph G of Figure 3. Graph G has $2n+1$ vertices $v_0, v_1, \ldots, v_{2n}$ on the Euclidean space $\mathbb{R}^2$, with the distance $d(v_i, v_{i+1}) = 1$ for $i = 0, 1, \ldots, 2n-1$, and $d(v_i, v_{i+2}) = 1 + a$ for $i = 0, 1, \ldots, 2n-2$, where $0 < a < 1$. It is clear that the minimum spanning tree T of G is the path from v_0 to v_{2n} containing all edges of distance 1. There are only two vertices, v_0 and v_{2n}, having odd degrees in tree T. Thus, the traveling salesman tour produced by Christofides's algorithm is the cycle $(v_0, v_1, v_2, \ldots, v_{2n}, v_0)$, whose total distance is $2n + n(1+a) = 3n + na$. Moreover, it is easy to see that the minimum traveling salesman tour consists of all horizontal edges plus the two outside non-horizontal edges, whose total distance is $(2n-1)(1+a) + 2 = 2n + 1 + (2n-1)a$. Thus, in this example,

$$\frac{\mathrm{H}(I)}{OPT(I)} = \frac{3n + na}{2n + 1 + (2n-1)a},$$

which appoarches $3/2$ as a goes to 0 and n goes to infinity. Thus, $r(\mathrm{H}) = 2/3$.

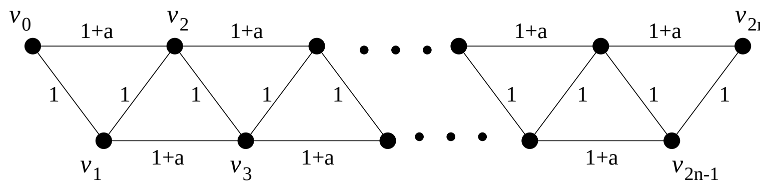

Figure 3: Worst cases of Christofides's approximation.

Theorem 3.1 *For the subproblem of* TSP *with the triangle inequality, as well as the subproblem of* TSP *on Euclidean space, the Christofides's approximation* H *has the performace ratio* $r(\mathrm{H}) = 3/2$.

For simplicity, we say an approximation algorithm A is an α-approximation if $r(A) \leq \alpha$ for some constant $\alpha \geq 1$. Thus, we say Christofides's algorithm is a (3/2)-approximation for TSP with the triangle inequality, but not an α-approximation for any $\alpha < 3/2$.

An approximation algorithm with a constant performance ratio is also called a *bounded approximation* or a *linear approximation.* An optimization problem Π is said to have a *polynomial-time approximation scheme* (PTAS) if for any $k > 0$, there exists a polynomial-time approximation algorithm A_k for Π with performance ratio $r(A_k) \leq 1 + 1/k$. Furthermore, if the running time of the algorithm A_k in the approximation scheme is a polynomial function in $n + 1/k$, where n is the input size, then the scheme is called a *fully* polynomial-time approximation scheme (FPTAS). For instance, the generalized greedy algorithm (Algorithm E) is a PTAS, and the polynomial tradeoff approximation (Algorithm F) is a FPTAS for KNAPSACK.

In the study of intractable optimization problems, our main concern is to find efficient approximations with the best performance ratios. However, some optimization problems are so hard that they don't even have any polynomial-time bounded algorithms. In these cases, we also need to prove that such approximations do not exist. Since most optimization problems are **NP**-complete, and hence have polynomial-time optimal solutions if $\mathbf{P} = \mathbf{NP}$. So, when we prove that a bounded approximation does not exist, we must assume that $\mathbf{P} \neq \mathbf{NP}$. Very often, we simply prove that the problem of finding a bounded approximation (or, an α-approximation for some fixed constant α) itself is **NP**-complete. The following is a simple example.

Theorem 3.2 *If $\mathbf{P} \neq \mathbf{NP}$, then there is no polynomial-time approximation algorithm for* TSP *(without the restriction of the triangle inequality) with a constant performance ratio.*

Proof. For any fixed integer $K > 1$, we will construct a *reduction* from the problem HC to the problem of finding a K-approximation for TSP.[1] That is, we will construct a mapping from each instance G of the problem HC to an instance (H, d) of TSP, such that the question of whether G has a Hamiltonian circuit can be determined from any traveling salesman tour for (H, d) whose total distance is within K times of the optimal tour.

For any graph $G = (V, E)$, with $|V| = n$, let H be the complete graph over vertex set V. Define the distance between two vertices $u, v \in V$ as follows:

$$d(u,v) = \begin{cases} 1 & \text{if } \{u,v\} \in E, \\ n(K+1) & \text{otherwise.} \end{cases}$$

Now, assume that C is a traveling salesman tour of the instance (H, d) whose total distance is at most K times of the optimal tour. If the total distance of C is less than $n(K+1)$, then we know that all edges in C are of

[1] Note that TSP is not a decision problem. So, the reduction here has a more general form than that defined in Section 1.

distance one and so they are all in E. Thus, C is a Hamiltonian circuit of G. On the other hand, if the total distance of C is greater than or equal to $n(K+1)$, this implies that the minimum traveling salesman tour has total distance greater than $n(K+1)/K > n$. That is, it must contain an edge not in E. Thus, G has no Hamiltonian circuit.

Thus, if there is a polynomial-time K-approximation for TSP, we can then use it to solve the problem HC, which is **NP**-complete. It follows that **P** $=$ **NP**. $\square$