CHAPTER 35

Approximation Algorithms

Coping with NP-completeness: With NP-completeness we have seen that there are many important optimization problems that are likely to be quite hard to solve exactly. Since these are important problems, we cannot simply give up at this point, since people do need solutions to these problems. How do we cope with NP-completeness:

• ***Use brute-force search:*** Even on the fastest parallel computers this approach is viable only for the smallest instances of these problems.

• ***Heuristics:*** A *heuristic* is a strategy for producing a valid (*feasible*) solution, but there are no guarantees how close it is to optimal. This is worthwhile if all else fails, or if lack of optimality is not really an issue.

• ***General Search Methods:*** There are a number of very powerful techniques for solving general combinatorial optimization problems that have been developed in the areas of AI and operations research. These go under names such as *branch-and-bound*, *A* -search*, *simulated annealing*, and *genetic algorithms*. The performance of these approaches varies considerably from one problem to the next.

• ***Approximation Algorithms:*** This is an algorithm that runs in polynomial time (ideally), and produces a solution that is within a guaranteed factor of the optimum solution.

Performance Bounds

• Most NP-complete problems have been stated as decision problems for theoretical reasons. However underlying most of these problems is a natural optimization problem.

• For example,

  • the TSP optimization problem is to find a simple cycle of minimum cost in a digraph,
  • the VC optimization problem is to find the vertex cover of minimum size,
  • the clique optimization problem is to find the clique of maximum size.

• Note that sometimes we are minimizing and sometimes we are maximizing.

• An approximation algorithm is one that returns a legitimate answer (*feasible solution*), but not necessarily one of the smallest size.

• How do we measure how good an approximation algorithm is?

• We define the ratio bound of an approximation algorithm as follows. Given an instance I of our problem, let $C(I)$ be the cost of the solution produced by our approximation algorithm, and let $C^*(I)$ be the optimal solution. We will assume that costs are strictly positive values.

• For a minimization problem we want $C(I)/C^*(I)$ to be small, and for a maximization problem we want $C^*(I)/C(I)$ to be small. For any input size n, we say that the approximation algorithm achieves ratio bound $\rho(n)$, if for all I, $|I| = n$ we have

$$\max\left(\frac{C(I)}{C^*(I)}, \frac{C^*(I)}{C(I)}\right) \leq \rho(n).$$

• Observe that $\rho(n)$ is always greater than or equal to 1, and it is equal to 1 if and only if the approximate solution is the true optimum solution.

Approximability

• Some NP-complete problems can be approximated closely. Such an algorithm is given both the input, and a real value $\varepsilon > 0$, and returns an answer whose ratio bound is at most $(1+\varepsilon)$. Such an algorithm is called a ***polynomial time approximation scheme*** (or ***PTAS*** for short). The running time is a function of both *n* and ε .

• As ε approaches 0, the running time increases beyond polynomial time. For example, the running time might be $O(n^{\lceil 1/\varepsilon \rceil})$. If the running time depends only on a polynomial function of $1/\varepsilon$ then it is called a ***fully polynomial-time approximation scheme***. For example, a running time like $O((1/\varepsilon)^2 n^3)$ would be such an example.

• Although NP-complete problems are equivalent with respect to whether they can be solved exactly in polynomial time in the worst case, their approximability varies considerably.

• For some NP-complete problems, it is very unlikely that any approximation algorithm exists. For example, if the graph TSP problem had an approximation algorithm with a ratio bound of any value less than ∞ , then **P = NP**.

• Many NP-complete problems can be approximated, but the ratio bound is a (slow growing) function of *n*. For example, the set cover problem (a generalization of the vertex cover problem), can be approximated to within a factor of *log n*.

• Some NP-complete problems can be approximated to within a fixed constant factor. We will discuss two examples below.

• Some NP-complete problems have PTAS's. One example is the subset problem (which we haven't discussed, but is described in CLRS) and the Euclidean TSP problem.

• In fact, much like NP-complete problems, there are collections of problems which are ``believed" to be hard to approximate and are equivalent in the sense that if any one can be approximated in polynomial time then they all can be. This class is called Max-SNP complete. We will not discuss this further. Suffice it to say that the topic of approximation algorithms would fill another course.

Vertex Cover

- We begin by showing that there is an approximation algorithm for vertex cover with a ratio bound of 2, that is, this algorithm will be guaranteed to find a vertex cover whose size is at most twice that of the optimum.

- Recall that the vertex cover problem seeks to find a set of vertices such that every edge in the graph touches one of these vertices.

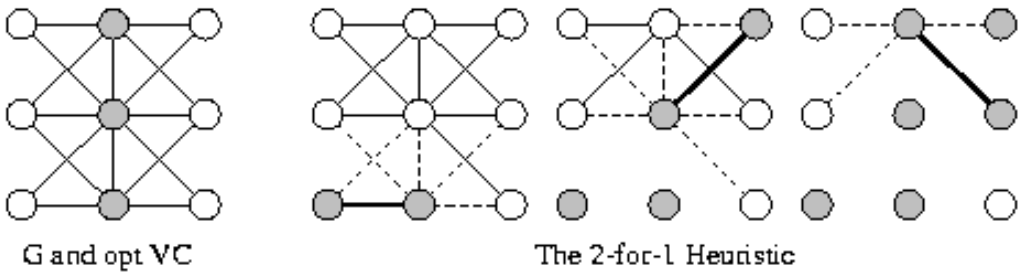
- How does one go about finding an approximation algorithm. The first approach is to try something that seems like a ``reasonably'' good strategy, a heuristic. It turns out that many simple heuristics, when not optimal, can often be proved to be close to optimal.

- Here is a very simple algorithm, that guarantees an approximation within a factor of 2 for the vertex cover problem.

- It is based on the following observation. Consider an arbitrary edge *(u, v)* in the graph. One of its two vertices must be in the cover, but we do not know which one.

- The idea of this heuristic is to simply put both vertices into the vertex cover. (You cannot get much stupider than this!) Then we remove all edges that are incident to *u* and *v* (since they are now all covered), and recurse on the remaining edges.

- For every one vertex that must be in the cover, we put 2 into our cover, so it is easy to see that the cover we generate is at most twice the size of the optimum cover.



The 2-for-1 heuristic for vertex cover.

Vertex Cover: correctness

```
ApproxVC {
    C = empty-set;
    while (E is nonempty) do {
        let (u,v) be any edge of E;   (*)
        add both u and v to C;
        remove from E all edges incident to either u or v;
    }
    return C;
}
```

Claim: ApproxVC yields a factor-2 approximation for Vertex Cover.

Proof: Consider the set *C* output by ApproxVC. Let *C** be the optimum VC. Let *A* be the set of edges selected by the line marked with (*) above. Observe that the size of *C* is exactly *2|A|* because we add two vertices for each such edge. However note that in the optimum VC one of these two vertices must have been added to the VC, and thus the size of *C** is at least |A|. Thus we have:

$$\frac{|C|}{2} = |A| \leq |C^*| \quad \Rightarrow \quad \frac{|C|}{|C^*|} \leq 2.$$

• This proof illustrates one of the main features of the analysis of any approximation algorithm.

• Namely, that we need some way of finding a bound on the optimal solution. (For minimization problems we want a lower bound, for maximization problems an upper bound.) The bound should be related to something that we can compute in polynomial time. In this case, the bound is related to the set of edges *A*, which form a maximal independent set of edges.

Traveling Salesman Problem

• In the Traveling Salesperson Problem (TSP) we are given a complete undirected graph with nonnegative edge weights, and we want to find a simple cycle that visits all vertices and is of minimum cost.

• Let *c(u,v)* denote the weight on edge *(u,v).* Given a set of edges *A* forming a tour we define *c(A)* to be the sum of edge weights in *A*.

• Last time we mentioned that TSP (posed as a decision problem) is **NP-complete**.

• For many of the applications of TSP, the problem satisfies something called the triangle inequality. Intuitively, this says that the direct path from *u* to *w*, is never longer than an indirect path. More formally, for all *u,v,w* from *V*

$$c(u, w) \leq c(u, v) + c(v, w).$$

• There are many examples of graphs that satisfy the triangle inequality. For example, given any weighted graph, if we define *c(u,v)* to be the shortest path length between *u* and *v* (computed, say by Floyd's algorithm), then it will satisfy the triangle inequality.

• Another example is if we are given a set of points in the plane, and define a complete graph on these points, where *c(u,v)* is defined to be the Euclidean distance between these points, then the triangle inequality is also satisfied.

• When the underlying cost function satisfies the triangle inequality there is a approximation algorithm for TSP with a ratio-bound of 2.

• Thus, although this algorithm does not produce an optimal tour, the tour that it produces cannot be worse than twice the cost of the optimal tour.

Traveling Salesman Problem (cont.)

• The key insight is to observe that a TSP with one edge removed is a spanning tree. However it is not necessarily a minimum spanning tree.

• Therefore, the cost of the minimum TSP tour is at least as large as the cost of the MST.

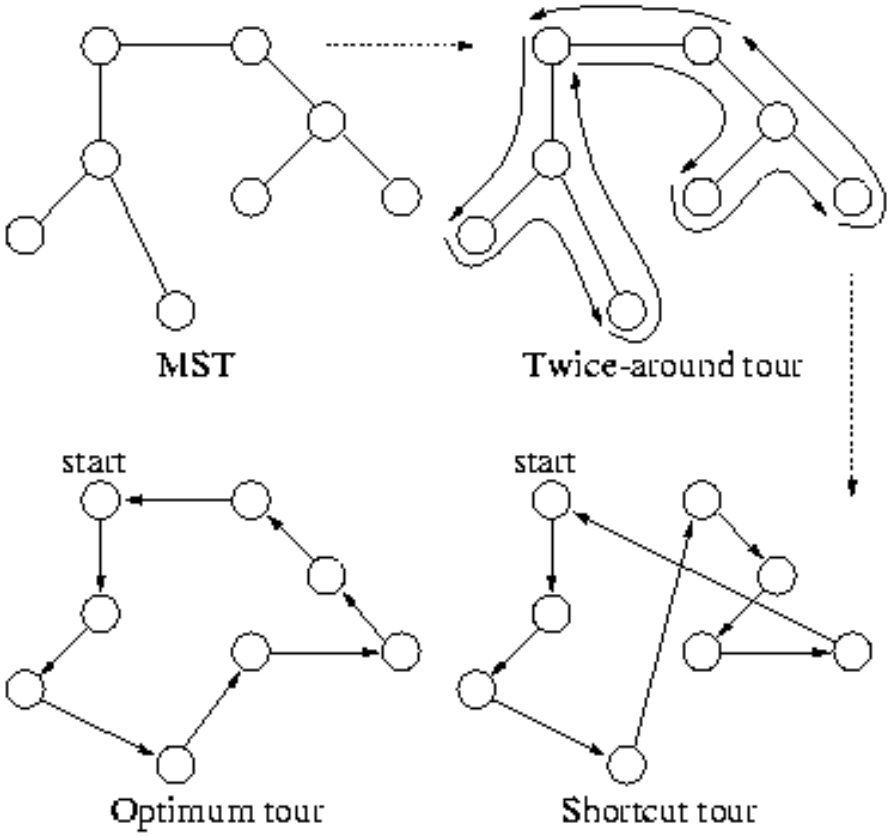
• We can compute MST's efficiently using either Kruskal's or Prim's algorithm.

• If we can find some way to convert the MST into a TSP tour while increasing its cost by at most a constant factor, then we will have an approximation for TSP.

• Here is how the algorithm works. Given any free tree there is a tour of the tree called a twice around tour that traverses the edges of the tree twice, once in each direction. The figure ➔➔➔➔➔ shows an example of this.

• This path is not simple because it revisits vertices, but we can make it simple by short-cutting, that is, we skip over previously visited vertices. Notice that the final order in which vertices are visited using the short-cuts is exactly the same as a preorder traversal of the MST.

• The triangle inequality assures us that the path length will not increase when we take short-cuts.



Traveling Salesman Problem: correctness

```
ApproxTSP(G=(V,E)) {
    T = minimum spanning tree for G;
    r = any vertex;
    L = list of vertices visited by a preorder walk of T
        starting with r;
    return L;
}
```

Claim: Approx-TSP has a ratio bound of 2.

Proof: Let H denote the tour produced by this algorithm and let H^* be the optimum tour.

Let T be the minimum spanning tree. As we said before, since we can remove any edge of H^* resulting in a spanning tree, and since T is the minimum cost spanning tree we have

$$c(T) \le c(H^*).$$

Now observe that the twice around tour of T has cost $2c(T)$, since every edge in T is hit twice.

By the triangle inequality, when we short-cut an edge of T to form H we do not increase the cost of the tour, and so we have

$$c(H) \le 2c(T).$$

Combining these we have

$$\frac{c(H)}{2} \le c(T) \le c(H^*) \quad \Rightarrow \quad \frac{c(H)}{c(H^*)} \le 2.$$

General Traveling Salesman Problem

• Assume now that the cost function *c* does not satisfy the triangle inequality.

Theorem: If **P≠NP,** then for any constant $\rho \geq 1$, there is no polynomial-time approximation algorithm with approximation ratio ρ for the general traveling-salesman problem.

Proof: (By contradiction) Assume there is a polynomial time algorithm *A* with approximation ratio ρ. W.l.o.g. assume ρ is an integer (we can round it up if necessary).

Then, we can show how to use *A* to solve instances of the Hamiltonian-Cycle (HC) problem in polynomial time (see the reduction below).

Since the Hamiltonian-Cycle problem is NP-complete and we assumed that **P≠NP,** a contradiction will arise.

Reduction: Let *G=(V,E)* be an instance of the HC problem. We wish to determine efficiently whether *G* contains a Hamiltonian cycle. We turn *G* into an instance of the traveling salesman problem as follows. Consider a complete graph *G'=(V,E')* on *V*, i.e. for any different *u,v* in *V*, *(u,v)* is an edge of *G'*. Assign an integer cost to each edge of *G'* as follows:

$$c(u,v) = \begin{cases} 1 & \text{if } (u,v) \in E \\ \rho |V| + 1 & \text{otherwise} \end{cases}$$

Clearly, this is a polynomial time reduction

Now, it is easy to see that

• *G* has a Hamiltonian cycle if and only if *(G',c)* has a tour of cost |V|.

• *G* has no Hamiltonian cycle if and only if any tour of *(G',c)* has cost > ρ|V|.

Since *A* is guaranteed to return a tour of cost no more than ρ times the cost of an optimal tour, if *G* has a Hamiltonian cycle, then *A* must return it. Also, if *G* has no Hamiltonian cycle, then *A* returns a tour of cost more that ρ|V|. Hence, *A* can be used to solve the HC problem in polynomial time, that is impossible unless **P=NP**.

More Approximations: Set Cover and Bin Packing

• The theorem of the approximation bound proven here is a bit weaker from the one in CLRS, but it is easier to understand. The material on the bin packing problem is presented as an exercise in CLRS.

Set Cover

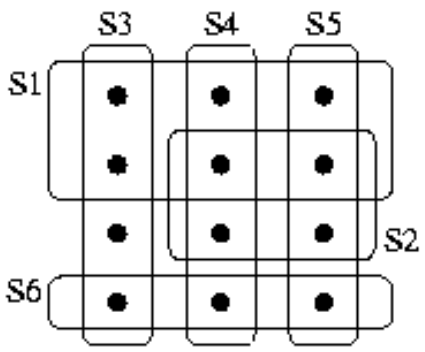
• The set cover problem is a very important optimization problem. You are given a pair *(X, F)* where $X = \{x_1, x_2, ..., x_m\}$ is a finite set (a domain of elements) and $F = \{S_1, S_2, ..., S_n\}$ is a family of subsets of *X*, such that every element of *X* belongs to at least one set of *F* .

• Consider a subset $C \subseteq F$. (This is a collection of sets over *X*.) We say that *C* covers the domain if every element of *X* is in some set of *C*, that is

$$X = \bigcup_{S_i \in C} S_i .$$

• The problem is to find the minimum-sized subset *C* of *F* that covers *X*.

• Consider the example shown below. The optimum set cover consists of the three sets *{S3, S4, S5}*.



Set cover can be applied to a number of applications. For example, suppose you want to set up security cameras to cover a large art gallery. From each possible camera position, you can see a certain subset of the paintings. Each such subset of paintings is a set in your system. You want to put up the fewest cameras to see all the paintings.

Complexity of Set Cover

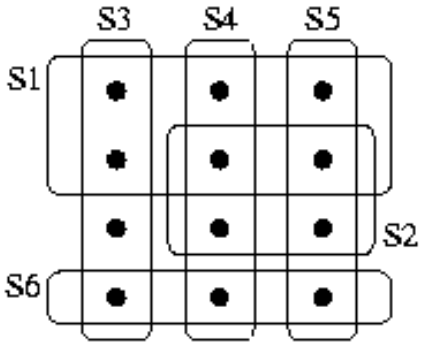
- We have seen special cases of the set cover problems that are NP-complete.
- For example, vertex cover is a type of set cover problem. The domain to be covered are the edges, and each vertex covers the subset of incident edges.
- Therefore the set cover problem is NP-complete as well.
- Unfortunately, although we have a factor-2 approximation for the vertex cover problem, it relied on the fact that each element of the domain (an edge) is in exactly 2 sets.
- Unfortunately, this is not true for the general set cover problem. Recently it has been proved that there is no constant factor approximation to the set cover problem (unless P = NP).
- This is unfortunate, because set cover is one of the most powerful NP-complete problems.
- Today we will show that there is a reasonable approximation algorithm, the greedy heuristic, which achieves an approximation bound of $\ln m$, where m is the size of the underlying domain X.
- (The book proves a somewhat stronger result, that the approximation factor of $\ln m'$ where $m' \leq m$ is the size of the largest set in F . However, their proof is more complicated.)

Greedy Set Cover

• A simple greedy approach to set cover works by at each stage selecting the set that covers the greatest number of ``uncovered'' elements.

```
Greedy-Set-Cover(X, F) {
    U = X;                                  // U are the items to be covered
    C = empty;                              // C will be the sets in the cover
    while (U is nonempty) {                 // there is someone left to cover
        select S in F that covers the most elements of U;
        add S to C;
        U = U - S;
    }
    return C;
}
```

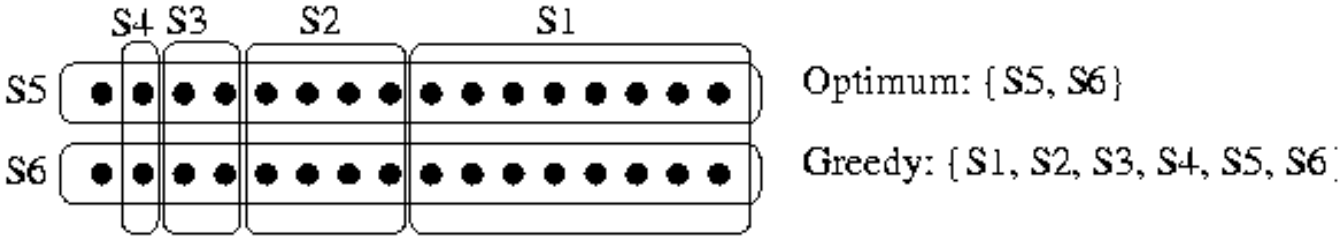
• For the example given earlier the greedy-set cover algorithm would select *S1* (since it covers 6 out of 12 elements), then *S6* (since it covers 3 out of the remaining 6), then *S2* (since it covers 2 of the remaining 3) and finally *S3*. Thus, it would return a set cover of size 4, whereas the optimal set cover has size 3.



What is the Approximation Factor?

• The problem with the greedy set cover algorithm is that it can be ``fooled" into picking the wrong set, over and over again.

• Consider the following example. The optimal set cover consists of sets S5 and S6 . But (if ties are broken in the worst possible way) the greedy algorithm will select sets *S1* (size 16), then *S2* (size 8), then *S3* (size 4), then *S4* (size 2), and finally *S5* and *S6*) (size 1 each). It is easy to see that this example can be generalized so that the ratio bound will be roughly *(log m)/2*.



• However we will show that the greedy set cover heuristic never performs worse than a factor of *ln m*, where $m = |X|$. (Note that this is natural log, not base 2.)

• Before giving the proof, we need one important mathematical inequality.

Lemma: For all $c > 0$, $(1-\frac{1}{c})^c \leq \frac{1}{e}$, where e is the base of the natural logarithm.

Proof: The book gives a proof, but here is one that I like. It requires the fact that for all x, $1+x \leq e^x$. (The two functions are equal when $x = 0$.) Now, if we substitute $-1/c$ for x we have $1-1/c \leq e^{-1/c}$,

and if we raise both sides to the cth power, we have the desired result.

Approximation Factor Theorem

Theorem: Greedy set cover has the ratio bound of at most *ln m* where $m = |X|$.

Proof: Let *c* denote the size of the optimum set cover, and let *g* denote the size of the greedy set cover minus 1. We will show that $g/c \leq \ln m$. (This is not quite what we wanted, but we are correct to within 1 set.)

• Initially, there are $m_0 = m$ elements left to be covered. We know that there is a cover of size *c* (the optimal cover) and therefore by the *pigeonhole* principle, there must be at least one set that covers at least m_0/c elements. (Since otherwise, if every set covered less than m_0/c elements, then no collection of *c* sets could cover all m_0 elements.)

• Since the greedy algorithm selects the largest set, it will select a set that covers at least this many elements. The number of elements that remain to be covered is at most $m_1 = m_0 - m_0/c = m_0(1-1/c)$.

• Applying the argument again, we know that we can cover these m_1 elements with a cover of size *c* (the optimal cover), and hence there exists a subset that covers at least m_1/c elements, leaving at most $m_2 = m_1 - m_1/c = m_0(1-1/c)^2$ elements remaining.

• If we apply this argument *g* times, each time we succeed in covering at least a fraction of *(1-1/c)* of the remaining elements.

• Then the number of elements that remain uncovered after *g* sets have been chosen by the greedy algorithm is at most $m_g = m_0(1-1/c)^g$.

Proof: cont.

• How long can this go on? Consider the largest value of g such that after removing all but the last set of the greedy cover, we still have some element remaining to be covered.

• Thus, we are interested in the largest value of g such that

$$1 \leq m\,(1 - 1/c)^g.$$

• We can rewrite this as

$$1 \leq m\left[(1 - 1/c)^c\right]^{g/c}.$$

• By the inequality above we have

$$1 \leq m\left[1/e\right]^{g/c}.$$

• Now, if we multiply by $e^{g/c}$ and take natural *log*s we get that g satisfies:

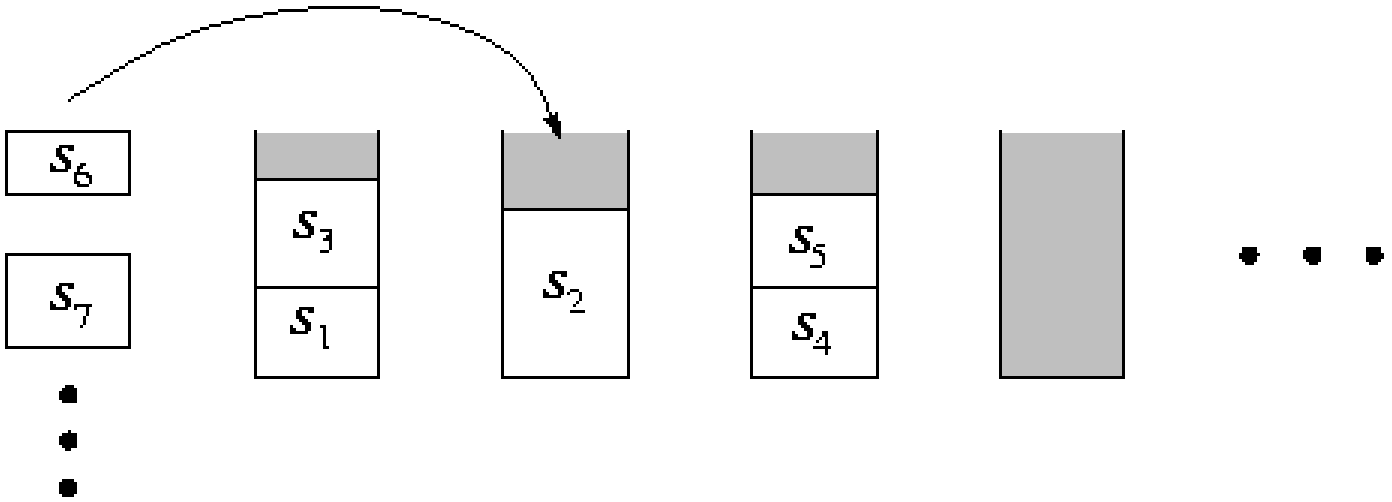
$$e^{g/c} \leq m \quad \Rightarrow \quad \frac{g}{c} \leq \ln m.$$

• This completes the proof.

Bin Packing

- Bin packing is another well-known NP-complete problem, which is a variant of the knapsack problem.
- We are given a set of *n* objects, where s_i denotes the *size* of the *i*th object.
- It will simplify the presentation to assume that $0 < s_i < 1$.
- We want to put these objects into a set of bins. Each bin can hold a subset of objects whose total size is at most 1.
- The problem is to partition the objects among the bins so as to use the fewest possible bins. (Note that if your bin size is not 1, then you can reduce the problem into this form by simply dividing all sizes by the size of the bin.)
- Bin packing arises in many applications. Many of these applications involve not only the size of the object but their geometric shape as well. For example, these include packing boxes into a truck, or cutting the maximum number of pieces of certain shapes out of a piece of sheet metal. However, even if we ignore the geometry, and just consider the sizes of the objects, the decision problem is still NP-complete. (The reduction is from the knapsack problem.)
- Here is a simple heuristic algorithm for the bin packing problem, called the *first-fit* heuristic.
- We start with an unlimited number of empty bins. We take each object in turn, and find the first bin that has space to hold this object. We put this object in this bin. The algorithm is illustrated in the figure on the next slide.
- We claim that *first-fit* uses at most *twice* as many bins as the optimum, that is, if the optimal solution uses b^* bins, and first-fit uses b bins, then $b/b^* \leq 2$.

First-Fit Heuristic: Example



First-fit Heuristic.

Approximation Factor Theorem

Theorem: The *first-fit* heuristic achieves a ratio bound of 2.

Proof: Consider an instance $\{s_1, s_2, ..., s_n\}$ of the bin packing problem. Let $S = \sum_i s_i$ denote the sum of all the object sizes. Let *b** denote the optimal number of bins, and *b* denote the number of bins used by first-fit.

• First observe that $b^* \geq S$. This is true, since no bin can hold a total capacity of more than 1 unit, and even if we were to fill each bin exactly to its capacity, we would need at least *S* (in fact $\lceil S \rceil$ bins).

• Next, we claim that $b \leq 2S$. To see this, let t_i denote the total size of the objects that first-fit puts into bin *i*. Consider bins *i* and *i + 1* filled by first-fit. Assume that indexing is cyclical, so if *i* is the last index ($i = b$) then $i + 1 = 1$. We claim that

$$t_i + t_{i+1} \geq 1.$$

• If not, then the contents of bins *i* and *i + 1* could both be put into the same bin, and hence first-fit would never have started to fill the second bin, prefering to keep everything in the first bin. Thus we have:

$$\sum_{i=1}^{b} (t_i + t_{i+1}) \geq b.$$

• But this sum adds up all the elements twice, so it has a total value of *2S*. Thus we have $b \leq 2S$. Combining this with the fact that $b^* \geq S$ we have

$$b \leq 2S \leq 2b^*,$$

implying that $b/b^* \leq 2$, as desired.