Coping with Hardness

Lecturer: Kai Wu

Coping with Hardness

- **Backtracking**: suitable for those problems that exhibit good average time complexity. This methodology is based on a methodic examination of the implicit state space induced by the problem instance under study. In the process of exploring the state space of the instance, some pruning takes place.

- **Randomized Algorithms**: Based on the probabilistic notion of accuracy.

- **Approximation Algorithms**: Compromise on the quality of solution in return for faster solutions.

Backtracking

- In many real world problems, a solution can be obtained by exhaustively searching through a large but finite number of possibilities. Hence, the need arose for developing systematic techniques of searching, with the hope of cutting down the search space to possibly a much smaller space.

- Here, we present a general technique for organizing the search known as ***backtracking***. This algorithm design technique can be described as an organized exhaustive search which often avoids searching all possibilities.

The 3-Coloring Problem

- Given an undirected graph $G=(V, E)$, it is required to color each vertex in V with one of three colors, say 1, 2, and 3, such that no two adjacent vertices have the same color. We call such a coloring **legal**; otherwise, if two adjacent vertices have the same color, it is **illegal**.

- A coloring can be represented by an n-tuple $(c_1, c_2, \ldots, c_n)$ such that $c_i \in \{1, 2, 3\}$, $1 \le i \le n$.

- For example, (1, 2, 2, 3, 1) denotes a coloring of a graph with five vertices.

The 3-Coloring Problem

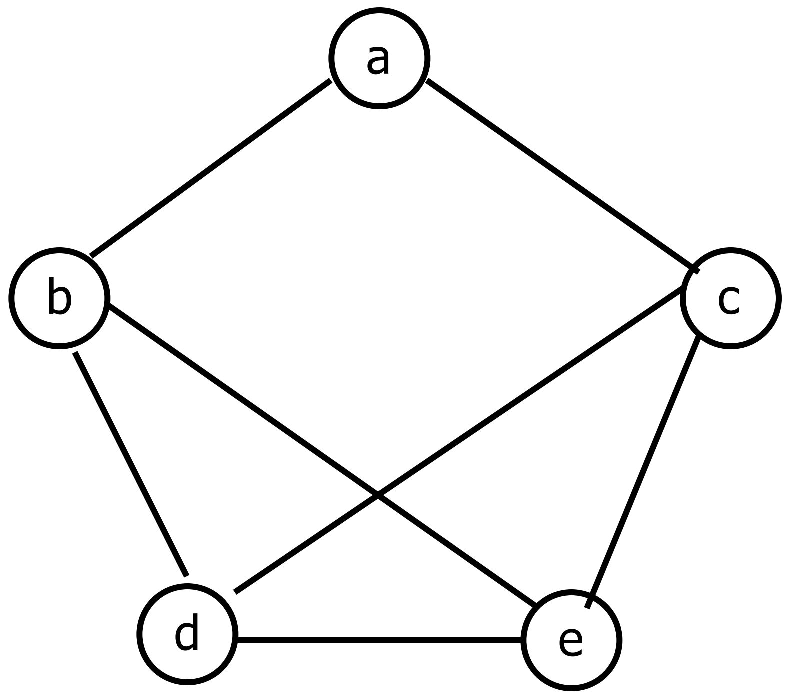
- There are 3^n possible colorings (legal and illegal) to color a graph with n vertices.
- The set of all possible colorings can be represented by a complete ternary tree called the ***search tree***. In this tree, each path from the root to a leaf node represents one coloring assignment.
- An incomplete coloring of a graph is ***partial*** if no two adjacent colored vertices have the same color.
- Backtracking works by generating the underlying tree one node at a time.
- If the path from the root to the current node corresponds to a legal coloring, the process is terminated (unless more than one coloring is desired).

The 3-Coloring Problem

- If the length of this path is less than *n* and the corresponding coloring is partial, then one child of the current node is generated and is marked as the current node.
- If, on the other hand, the corresponding path is not partial, then the current node is marked as a ***dead node*** and a new node corresponding to another color is generated.
- If, however, all three colors have been tried with no success, the search backtracks to the parent node whose color is changed, and so on.

The 3-Coloring Problem

- Example:



The 3-Coloring Problem

There are two important observations to be noted, which generalize to all backtracking algorithms:

(1) The nodes are generated in a depth-first-search manner.

(2) There is no need to store the whole search tree; we only need to store the path from the root to the current active node. In fact, no physical nodes are generated at all; the whole tree is implicit. We only need to keep track of the color assignment.

The 3-Coloring Problem

Recursive Algorithm

Input: An undirected graph $G=(V, E)$.

Output: A 3-coloring $c[1\ldots n]$ of the vertices of G, where each $c[j]$ is 1, 2, or 3.

```
1. for $k \leftarrow 1$ to $n$
2.     $c[k] \leftarrow 0$;
3. end for;
4. $flag \leftarrow$ false;
5. $graphcolor(1)$;
6. if $flag$ then output $c$;
7. else output “no solution”;

$graphcolor(k)$
1. for $color=1$ to 3
2.     $c[k] \leftarrow color$;
3.     if $c$ is a legal coloring then set $flag \leftarrow$ true and exit;
4.     else if $c$ is partial then $graphcolor(k+1)$;
5. end for;
```

The 3-Coloring Problem

Iterative Algorithm

Input: An undirected graph $G=(V, E)$.

Output: A 3-coloring $c[1\ldots n]$ of the vertices of G, where each $c[j]$ is 1, 2, or 3.

```
1. for $k \leftarrow 1$ to $n$
2.     $c[k] \leftarrow 0$;
3. end for;
4. $flag \leftarrow$ false;
5. $k \leftarrow 1$;
6. while $k \geq 1$
7.     while $c[k] \leq 2$
8.         $c[k] \leftarrow c[k]+1$;
9.         if $c$ is a legal coloring then set $flag \leftarrow$ true and exit from the two while loops;
10.        else if $c$ is partial then $k \leftarrow k+1$;
11.    end while;
12.    $c[k] \leftarrow 0$;
13.    $k \leftarrow k-1$;
14. end while;
15. if $flag$ then output $c$;
16. else output "no solution";
```

The 8-Queens Problem

- How can we arrange 8 queens on an 8×8 chessboard so that no two queens can attack each other?
- Two queens can attack each other if they are in the same row, column or diagonal.
- The n-queens problem is defined similarly, where in this case we have n queens and an $n \times n$ chessboard for an arbitrary value of $n \geq 1$.

The 8-Queens Problem

- Consider a chessboard of size 4×4. Since no two queens can be put in the same row, each queen is in a different row. Since there are four positions in each row, there are 4^4 possible configurations.
- Each possible configuration can be described by a vector with four components $\boldsymbol{x}=(x_1, x_2, x_3, x_4)$.
- For example, the vector (2, 3, 4, 1) corresponds to a configuration.

The 8-Queens Problem

- Input: none;
- Output: A vector x[1…4] corresponding to the solution of the 4-queens problem.

```
1. for $k \leftarrow 1$ to 4
2.     $x[k] \leftarrow 0$;
3. end for;
4. $flag \leftarrow$ false;
5. $k \leftarrow 1$;
6. while $k \geq 1$
7.     while $x[k] \leq 3$
8.         $x[k] \leftarrow x[k]+1$;
9.         if $x$ is a legal placement then set $flag \leftarrow$ true and exit from the two while loops;
10.        else if $x$ is partial then $k \leftarrow k+1$;
11.    end while;
12.    $x[k] \leftarrow 0$;
13.    $k \leftarrow k-1$;
14. end while;
15. if $flag$ then output $x$;
16. else output "no solution";
```

The General Backtracking Method

- The general backtracking algorithm can be described as a systematic search method that can be applied to a class of search problems whose solution consists of a vector $(x_1, x_2, \ldots x_i)$ satisfying some predefined constraints. Here, i is dependent on the problem formulation. In 3-Coloring and the 8-queens problems, i was fixed.

- In some problems, i may vary from one solution to another.

The General Backtracking Method

- Consider a variant of the PARTITION problem defined as follows. Given a set of n integers $X=\{x_1, x_2, \dots, x_n\}$ and an integer y, find a subset Y of X whose sum is equal to y.

- For instance if $X=\{10, 20, 30, 40, 50, 60\}$, and $y=60$, then there are three solutions of different lengths: {10, 20, 30}, {20, 40}, and {60}.

- Actually, this problem can be formulated in another way so that the solution is a boolean vector of length n in the obvious way. The above three solutions may be expressed by the boolean vectors {1, 1, 1, 0, 0, 0}, {0, 1, 0, 1, 0, 0}, and {0, 0, 0, 0, 0, 1}.

The General Backtracking Method

- In backtracking, each x_i in the solution vector belongs to a finite linearly ordered set X_i. Thus, the backtracking algorithm considers the elements of the cartesian product $X_1 \times X_2 \times \ldots X_n$ in lexicographic order.
- Initially, the algorithm starts with the empty vector. It then chooses the least element of X_1 as x_1. If (x_1) is a partial solution, then algorithm proceeds by choosing the least element of X_2 as x_2. If (x_1, x_2) is a partial solution, then the least element of X_3 is included; otherwise x_2 is set to the next element in X_2.
- In general, suppose that the algorithm has detected the partial solution $(x_1, x_2, \ldots, x_j)$. It then considers the vector $v=(x_1, x_2, \ldots, x_j, x_{j+1})$. We have the following cases:

The General Backtracking Method

(1) If v represents a final solution to the problem, the algorithm records it as a solution and either terminates in case only one solution is desired or continues to find other solutions.

(2) If v represents a partial solution, the algorithm advances by choosing the least element in the set X_{j+2}.

(3) If v is neither a final nor a partial solution, we have two subcases:

(a) If there are still more elements to choose from in the set X_{j+1}, the algorithm sets x_{j+1} to the next member of X_{j+1}.
(b) If there are no more elements to choose from in the set X_{j+1}, the algorithm backtracks by setting x_j to the next member of X_j. If again there are no more elements to choose from in the set X_j, the algorithm backtracks by setting x_{j-1} to the next member of X_{j-1}, and so on.

Branch and Bound

- Branch and bound design technique is similar to backtracking in the sense that it generates a search tree and looks for one or more solutions.

- However, while backtracking searches for a solution or a set of solutions that satisfy certain properties (including maximization or minimization), branch-and-bound algorithms are typically concerned with only maximization or minimization of a given function.

- Moreover, in branch-and-bound algorithms, a bound is calculated at each node *x* on the possible value of any solution given by nodes that may later be generated in the subtree rooted at *x*. If the bound calculated is worse than the previous bound, the subtree rooted at *x* is blocked.

Branch and Bound

- Henceforth, we will assume that the algorithm is to minimize a given cost function; the case of maximization is similar. In order for branch and bound to be applicable, the cost function must satisfy the following property.
- For all partial solutions $(x_1, x_2, \ldots, x_{k-1})$ and their extensions $(x_1, x_2, \ldots, x_k)$, we must have

$$cost(x_1, x_2, \ldots, x_{k-1}) \leq cost(x_1, x_2, \ldots, x_k)$$

- Given this property, a partial solution $(x_1, x_2, \ldots, x_k)$ can be discarded once it is generated if its cost is greater than or equal to a previously computed solution.
- Thus, if the algorithm finds a solution whose cost is c, and there is a partial solution whose cost is at least c, no more extensions of this partial solution are generated.

Branch and Bound

Traveling Salesman Problems (TSPs):

Given a set of cities and a cost function that is defined on each pair of cities, find a tour of minimum cost. Here a tour is a closed path that visits each city exactly once. The cost function may be the distance, travel time, air fare, etc.

An instance of the TSP is given by its cost matrix whose entries are assumed to be nonnegative.

∞	17	7	35	18
9	∞	5	14	19
29	24	∞	30	12
27	21	25	∞	48
15	16	28	18	∞

Branch and Bound

With each partial solution ($x_1, x_2, \ldots, x_k$), we associate a lower bound y, which is the cost of any complete tour that visits the cities $x_1, x_2, \ldots, x_k$ in this order must be at least y.

Observations:

We observe that each complete tour must contain exactly one edge and its associated cost from each row and each column of the cost matrix.

We also observe that if a constant r is subtracted from every entry in any row or column of the cost matrix A, the cost of any tour under the new matrix is exactly r less than the cost of the same tour under A. This motivates the idea of reducing the cost matrix so that each row or column contains at least one entry that is equal to 0. We will refer to such a matrix as the reduction of the original matrix.

Branch and Bound

- Let $(r_1, r_2, \ldots, r_n)$ and $(c_1, c_2, \ldots, c_n)$ be the amounts subtracted from rows 1 to n and columns 1 to n, respectively, in an $n \times n$ cost matrix A. Then, y defined as follow is a lower bound on the cost of any complete tour.

$$y = \sum_{i=1}^{n} r_i + \sum_{i=1}^{n} c_i$$

∞	17	7	35	18
9	∞	5	14	19
29	24	∞	30	12
27	21	25	∞	48
15	16	28	18	∞

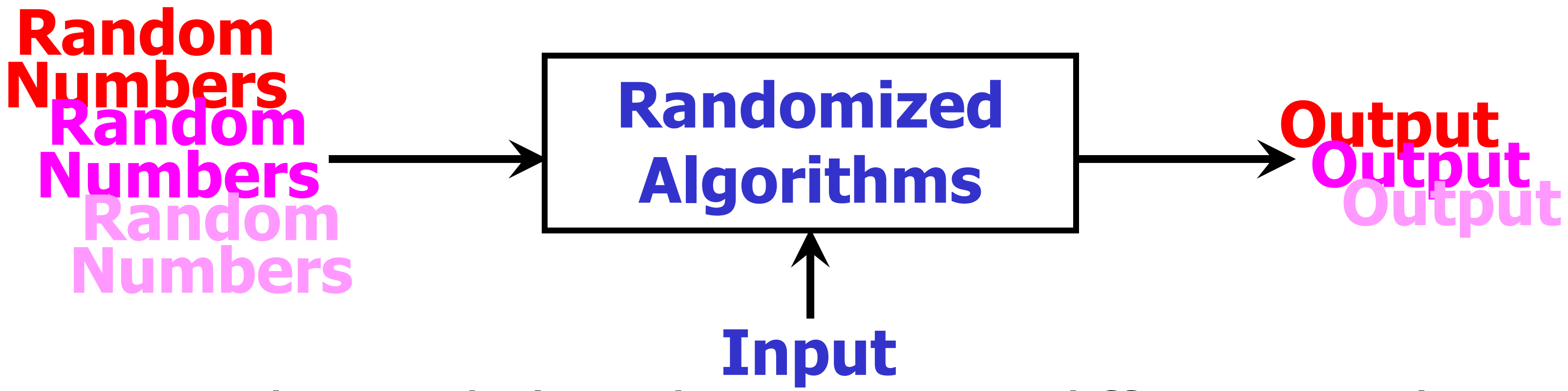
Randomized Algorithms

- One form of algorithm design in which we relax the condition that an algorithm must solve the problem correctly for all possible inputs, and demand that its possible incorrectness is something that can safely be ignored due to its very low likelihood of occurrence.

- Also, we will not demand that the output of an algorithm must be the same in every run on a particular input.

Randomized Algorithms

- A **randomized algorithm** can be defined as one that receives, in addition to its input, a stream of random bits that it can use in the course of its action for the purpose of making random choices.



- A randomized algorithm may give different results when applied to the same input in different rounds. It follows that the execution time of a randomized algorithm may vary from one run to another when applied to the same input.

Randomized Algorithms

Randomized algorithms can be classified into two categories:

- The first category is referred to as ***Las Vegas*** algorithms. It constitutes those randomized algorithms that always give a correct answer, or do not give an answer at all.

- The second category is referred to as ***Monte Carlo*** algorithms. It always gives an answer, but may occasionally produce an answer that is incorrect. However, the probability of producing an incorrect answer can be make arbitrarily small by running the algorithm repeatedly with independent random choices in each run.

Randomized Selection

- **Input**: An array $A[1\ldots n]$ of n elements and an integer k, $1\leq k\leq n$;
- **Output**: The kth smallest element in A;
- 1. $rselect(A, 1, n, k)$;

```
rselect(A, low, high, k)
1. $v \leftarrow random(low, high)$;
2. $x \leftarrow A[v]$;
3. Partition $A[low\ldots high]$ into three arrays:
    $A_1=\{a|a<x\}$, $A_2=\{a|a=x\}$, $A_3=\{a|a>x\}$;
4. case
    $|A_1|\geq k$: return $select(A_1, 1, |A_1|, k)$;
    $|A_1|+|A_2|\geq k$: return $x$;
    $|A_1|+|A_2|<k$: return $select(A_3, 1, |A_3|, k-|A_1|-|A_2|)$;
5. end case;
```

Testing String Equality

- Suppose that two parties A and B can communicate over a communication channel, which we will assume to be very reliable. A has a very long string *x* and B has a very long string *y*, and they want to determine whether *x*=*y*.

- Obviously, A can send *x* to B, who in turn can immediately test whether *x*=*y*. But this method would be extremely expensive, in view of the cost of using the channel.

Testing String Equality

- Another alternative would be for A to derive from x a much shorter string that could serve as a "***fingerprint***" of x and send it to B.
- B then would use the same derivation to obtain a fingerprint for y, and then compare the two fingerprints.
- If they are equal, then B would assume that $x=y$; otherwise he would conclude that $x\neq y$. B than notifies A of the outcome of the test.
- This method requires that transmission of a much shorter string across the channel.

Testing String Equality

- For a string w, let $I(w)$ be the integer represented by the bit string w. One method of fingerprinting is to choose a prime number p and then use the fingerprint function

$$I_p(x)=I(x) \text{ (mod } p)$$

- If p is not too large, then the fingerprint $I_p(x)$ can be sent as a short string. If $I_p(x)\neq I_p(y)$, then obviously $x\neq y$. However, the converse is not true. That is, if $I_p(x)=I_p(y)$, then it is not necessarily the case that $x=y$. We refer to this phenomenon as a ***false match***.

- In general, a false match occurs if $x\neq y$, but $I_p(x)=I_p(y)$, i.e., p divides $I(x)-I(y)$.

Testing String Equality

- The weakness of this method is that, for fixed p, there are certain pairs of strings x and y on which the method will always fail. Then, ***what's the probability?***

Testing String Equality

- Let n be the number of bits of the binary strings of x and y, and p be a prime number which is smaller than $2n^2$. The probability of false matching is $1/n$.

Testing String Equality

- Thus, we choose p at random every time the equality of two strings is to be checked, rather than agreeing on p in advance. Moreover, choosing p at random allows for resending another fingerprint, and thus increasing the confidence in the case $x=y$.

1. A chooses p at random from the set of primes less than M.
2. A sends p and $I_p(x)$ to B.
3. B checks whether $I_p(x)=I_p(y)$ and confirms the equality or inequality of the two strings x and y.

Pattern Matching

0 1 a b c

- Given a string of text $X=x_1x_2\ldots x_n$ and a pattern $Y=y_1y_2\ldots y_m$, where $m\leq n$, determine whether or not the pattern appears in the text. Without loss of generality, we will assume that the text alphabet is $\Sigma=\{0, 1\}$.

- The most straightforward method for solving this problem is simply to move the pattern across the entire text, and in every position compare the pattern with the portion of the text of length m.

- This brute-force method leads to an $O(mn)$ running time in the worst case.

Pattern Matching

- Here we will present a simple and efficient Monte Carlo algorithm that achieves a running time of $O(n+m)$.

- The algorithm follows the same brute-force algorithm of sliding the pattern Y across the text X, but instead of comparing the pattern with each block $X(j)=x_j x_{j+1}\ldots x_{j+m-1}$, we will compare the fingerprint $I_p(Y)$ of the pattern with the fingerprints $I_p(X(j))$ of the blocks of text.

p

Pattern Matching

- The key observations that when we shift from one block of text to the text, the fingerprint of the new block $X(j+1)$ can easily be computed from the fingerprint of $X(j)$.

$$I_p(X(j+1))=(2I_p(X(j))-2^m x_j+x_{j+m}) \pmod{p}$$

If we let $W_p=2^m \pmod{p}$, then we have the recurrence

$$I_p(X(j+1))=(2I_p(X(j))-W_p x_j+x_{j+m}) \pmod{p}$$

Pattern Matching

Input: A string of text X and a pattern Y of length n and m, respectively.

Output: The first position of Y in X if Y occurs in X; otherwise 0.

```
1. Choose $p$ at random from the set of primes less than $M$;
2. $j \leftarrow 1$;
3. Compute $W_p=2^m$ (mod $p$), $I_p(Y)$ and $I_p(X_j)$;
4. while $j \leq n-m+1$
5.      if $I_p(X_j)=I_p(Y)$ then return $j$ ; \\A match is found (probably)
6.      Compute $I_p(X_j)$ using previous equation;
7.      $j \leftarrow j+1$;
8. end while;
9. return 0; //$Y$ does not occur in $X$ (definitely)
```

Pattern Matching

- The time complexity of the previous pattern matching algorithm is $O(m+n)$.
- Let p be a prime number which is smaller than $2mn^2$. The probability of false matching is $1/n$.

Pattern Matching

- To convert the algorithm into a Las Vegas algorithm is easy.
- Whenever the two fingerprints $I_p(Y)$ and $I_p(X(j))$ match, the two strings are tested for equality.
- Thus, we end with an efficient pattern matching algorithm that always gives the correct result, and the time complexity is still $O(m+n)$.

Approximation Algorithms

- There are many hard combinatorial optimization problems that cannot be solved efficiently using backtracking or randomization.
- An alternative in this case for tacking some of these problems is to devise an approximation algorithm, given that we will be content with a “reasonable” solution that approximates an optimal solution.

Approximation Algorithms

- Associated with each approximation algorithm, there is a performance bound that guarantees that the solution to a given instance will not be far away from the neighborhood of the exact solution.
- A marking characteristic of (most of) approximation algorithms is that they are fast, as they are mostly greedy algorithm.

Approximation Algorithms

- A *combinatorial optimization problem* Π is either a *minimization problem* or a *maximization problem*. It consists of three components:

(1) A set D_Π of instances.

(2) For each instance $I \in D_\Pi$, there is a finite set $S_\Pi(I)$ of *candidate solutions* for I.

(3) Associated with each solution $\sigma \in S_\Pi(I)$ to an instance I in D_Π, there is a value $f_\Pi(\sigma)$ called the *solution value* for σ.

Approximation Algorithms

- If Π is a minimization problem, then an optimal solution σ^* for an instance $I \in D_\Pi$ has the property that for all $\sigma \in S_\Pi(I)$, $f_\Pi(\sigma^*) \leq f_\Pi(\sigma)$. An optimal solution for a maximization problem is defined similarly. We will denote by $OPT(I)$ the value $f_\Pi(\sigma^*)$.

- An ***approximation algorithm*** *A* for an optimization problem Π is a (polynomial time) algorithm such that given an instance $I \in D_\Pi$, it outputs some solution $\sigma \in S_\Pi(I)$. We will denote by $A(I)$ the value $f_\Pi(\sigma)$.

Difference Bounds

- The most we can hope from an approximation algorithm is that the difference between the value of the optimal solution and the value of the solution obtained by the approximation algorithm is always constant.

- In other words, for all instances I of the problem, the most desirable solution that can be obtained by an approximation algorithm A is such that $|A(I)-OPT(I)|\leq K$, for some constant K.

Difference Bounds

Planar Graph Coloring

Let $G=(V, E)$ be a planar graph. By the Four Color Theorem, every planar graph is four-colorable. It is fairly easy to determine whether a graph is 2-colorable or not. On the other hand, to determine whether it is 3-colorable is NP-complete.

Difference Bounds

Planar Graph Coloring

Given an instance *I* of *G*, an approximation algorithm *A* may proceed as follows:

Assume *G* is nontrivial, i.e. it has at least one edge. Determine if the graph is 2-colorable. If it is, then output 2; otherwise output 4. If *G* is 2-colorable, then $|A(I)-OPT(I)|=0$. If it is not 2-colorable, then $|A(I)-OPT(I)\leq 1$. This is because in the latter case, *G* is either 3-colorable or 4-colorable.

Difference Bounds

Counterexample: Knapsack Problems

There is no approximation algorithms with difference bounds for knapsack problems.

Relative Performance Bounds

- Clearly, a difference bound is the best bound guaranteed by an approximation algorithm.

- However, it turns out that very few hard problems possess such a bound. So we will discuss another performance guarantee, namely the *relative performance guarantee*.

Relative Performance Bounds

- Let Π be a minimization problem and I an instance of Π. Let A be an approximation algorithm to solve Π. We define the approximation ratio $R_A(I)$ to be

$$R_A(I) = \frac{A(I)}{OPT(I)}$$

- If Π is a maximization problem, then we define R_A(I) to be

$$R_A(I) = \frac{OPT(I)}{A(I)}$$

- Thus the approximation ratio is always greater than or equal to one.

Relative Performance Bounds

The Bin Packing Problem

- Given a collection of items u_1, u_2, …, u_n of sizes s_1, s_2, …, s_n, where such s_j is between 0 and 1, we are required to pack these items into the minimum number of bins of unit capacity.

Relative Performance Bounds

The Bin Packing Problem

- We list here one heuristic method:

First Fit (FF): The bins are indexed as 1, 2, … All bins are initially empty. The items are considered for packing in the order u_1, u_2, …, u_n. To pack item u_i, find the least index j such that bin j contains at most 1-s_i, and add item u_i to the items packed in bin j. Then, we have

$$R_{FF}(I) = \frac{FF(I)}{OPT(I)} < 2$$