

## Chapter 5

### NP Problems and Approximations Algorithms

- We have studied many useful and efficient algorithms for solving many different and difficult problems. Many
- Many of algorithms that we have studied so far have fairly low complexities with orders ranging from logarithmic  $O(\lg n)$ , linear  $O(n)$ , linear times logarithmic  $O(n \lg n)$ , quadratic  $O(n^2)$ , cubic  $O(n^3)$ , etc. These algorithms are called **tractable**, or not so hard
- All these require fairly low to medium computation time for reasonably large  $n$  on a modern computer. But the power of algorithms is not unlimited, and there are some problems that either cannot be solved by any algorithm, or that require astronomical times even on the fastest computer.
- There also combinatorial problems that are **hard** or intractable and whose complexity is exponential (e.g.  $O(2^n)$ ). A modern computer may need days, months or even years solve these hard problems.

**Example:** Consider a Boolean function  $F$  in  $n$  variables  $A_1, A_2, \dots, A_n$ , e.g.

$$F = A_1 \cdot A_2 + A_2 \cdot \bar{A}_3 \cdot A_4 \cdot A_{10} + \dots + \bar{A}_6 \cdot A_9 \cdot \bar{A}_n.$$

Find the values of  $A_1, A_2, \dots, A_n$  that would produce a true for  $F$  ( i.e.  $F = 1$ ).

- Since there are  $n$  variables there will be  $2^n$  cases to consider (i.e the truth table has  $2^n$  rows), and the complexity of this algorithm is  $O(2^n)$ .

**Other Examples:** Binomial, Travelling Salesman, Knapsack, Partition, etc.

### 6.3 Class P Problems

A problem is Class P, if there exists an algorithm for it whose complexity is at most a polynomial in the input size  $n$ , i.e.  $O(n^\alpha)$  where  $\alpha$  is a finite constant integer. In this case we say that the algorithm is polynomial bounded.

**Note 1 :** Polynomial complexity can be quite bad, thus every problem in class P does not necessarily have an efficient or acceptable solution. However, a problem not in P is usually impossible to solve in practice.

**Example:** Hamiltonian Cycle problem- In a graph  $G(V,E)$  find a simple cycle that contains every vertex. This problem does not have a polynomial bounded algorithm, and is thus very hard.

**Note 2.** Any algorithm built from several polynomial bounded algorithms is also polynomial bounded.

## 6.4 Optimization and Decision Problems

Many of the “hard” problems are optimization problems, but can also be stated as decision problems. Below we give the statements of several of these problems in both optimization and decision forms

### Graph Coloring Problem

**Optimization Problem:** Given a graph  $G$ , determine the minimum number of colors such that adjacent vertices have different colors.

**Decision Problem:** Given  $G$ , and a positive integer  $k$ , is there a coloring of  $G$  that uses at most  $k$  colors?

**Example:** Let  $E$  be pair of courses whose time slots must be different to allow a student take both of them. Let  $V$  be the number of courses to be scheduled, then the graph coloring problem is the scheduling of courses without a time conflict.

## Bin Packing Problem

Suppose we have many bins of capacity one, and  $n$  objects of sizes  $S_1, S_2, \dots, S_n$  each less than one.

**Optimization Problem:** Determine the smallest number of bins into which the objects can be packed.

**Decision Problem:** Given an integer  $k$ , do the objects fit into  $k$  bins?

**Examples:** Packing data (files) in computer memory (storage), filling orders for a product (e.g. fabric, lumber) to be cut from large size pieces.

## Knapsack (Burglar) Problem

Suppose we have a bag of capacity  $C$  and  $n$  objects with sizes  $S_1, S_2, \dots, S_n$ , with values  $w_1, w_2, \dots, w_n$ .

**Optimization Problem:** Find the objects with the largest total value that can fit into the bag.

**Decision Problem:** Given  $k$ , is there a subset of the objects that fit in the bag and has a total value of at least  $k$ .

## Travelling Salesman Problem

The salesman wants to minimize the total travelling cost (time, distance, or fare).

*Optimization Problem* : Given a complete graph wieghted graph, find a minimum total weight cycle that visits ever vertex exactly once.

*Decision Problem* : Given a complete, weighted graph and an integer  $k$ , is there a cycle that visits every vertex exactly once with a total weight of at most  $k$ ?

## Independent Set problem

Given a graph  $G=(V,E)$ , we say a set of nodes  $S \subseteq V$  is independent if no two nodes in  $S$  are joined by an edge.

*Optimization Problem*: Give a graph find an independent set that is as large as possible.

*Decision Problem*: Given a graph and a number  $k$ , is there an independent set of size at least equal to  $k$ ?

## Subset Sum Problem

There are  $n$  objects with sizes  $S_1, S_2, \dots, S_n$ , and given constant  $C$ .

***Optimization Problem*** : Among subsets of the objects with total sizes (sum) of at most  $C$ , what is the largest subset sum?

***Decision Problem***: Is there a subset of objects whose sizes add up to exactly  $C$ ?

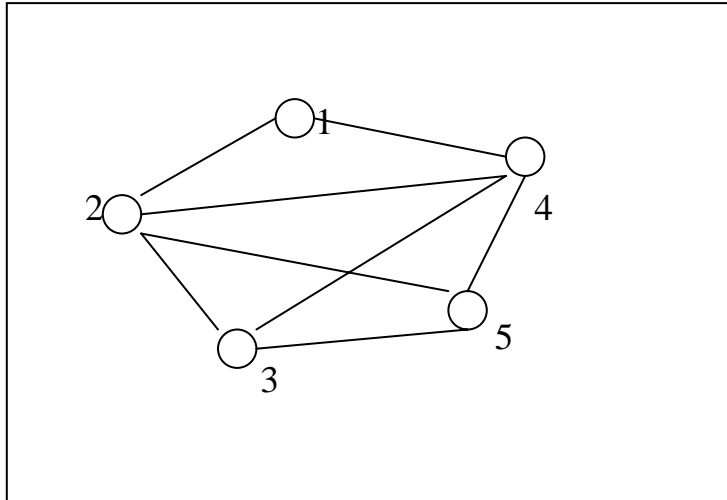
### **The NP Class**

- A non-deterministic polynomial bounded (NP) is a the class of decision problems for which a given proposed solution can be checked for correctness quickly, i.e. in polynomial time (complexity).
- A decision problem has yes/no answer.
- The NP problems need non-deterministic algorithms for their solutions.

A ***non-deterministic algorithm*** has two phases:

- The non-deterministic or guessing phase which provides a guess at the solution.
- The deterministic or verifying phase in which a deterministic (ordinary) routine begins execution, and eventually returns a true or false, or it may get in an infinite loop. If the verifying phase returns true, the algorithm outputs yes, otherwise there is no output.

**Example:** Can the following graph be colored using only four different colors? Let the colors be denoted by R (red), B (blue), G (green) and Y (yellow), and the node be denoted as 1, 2, 3, 4 and 5.



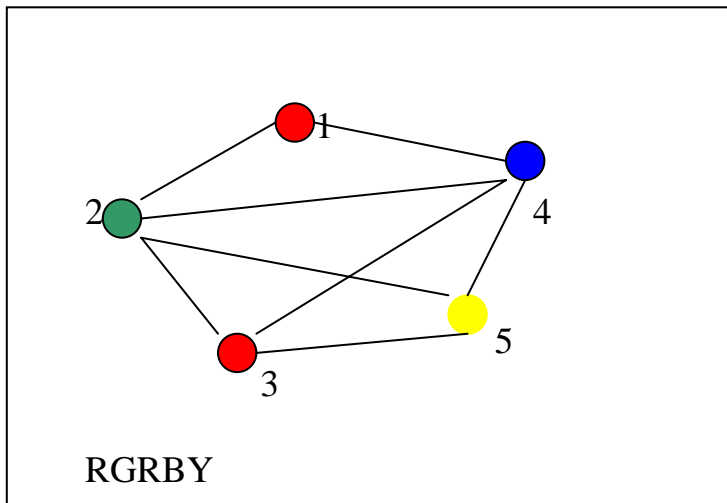
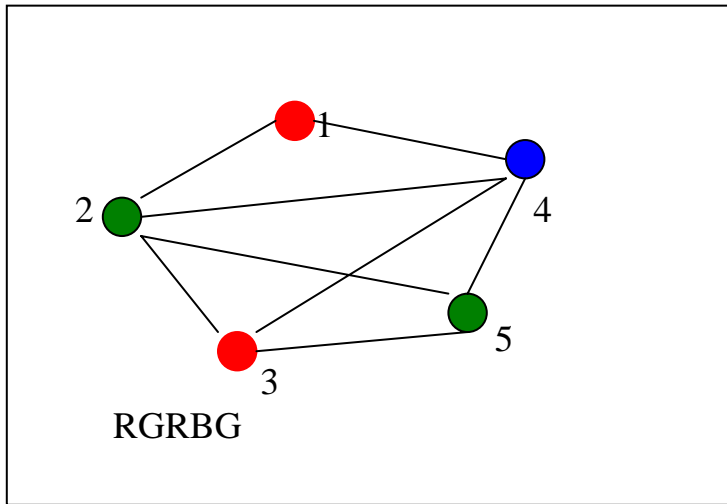
A possible non-deterministic algorithm for this problem is:

1. The generate strings.
2. Check that there are 4 colors in the strings.
3. Assign the colors to vertices
4. scan the adjacency matrix (list) and for each edge check that the colors of vertices incident to the edge are different.

Suppose that somehow the following strings are generated (guessed), where for example RGBY denotes the color gives to nodes 1 through 5 respectively.

String	Output	Reason
--------	--------	--------

<b>RGRBG</b>	<b>false</b>	<b>adjacent vertices 2 &amp; 5 are both green</b>
<b>RGRB</b>	<b>false</b>	<b>not all vertices are colored</b>
<b>RBYGO</b>	<b>false</b>	<b>too many colors are used (O is not defined)</b>
<b>RGRBY</b>	<b>true</b>	<b>a valid four coloring</b>
<b>R&amp;8*B9</b>	<b>false</b>	<b>undefined colors</b>



**Since there is at least one possible computation that returns a true, the answer of the non-deterministic algorithm is yes.**

- NP (non-deterministic polynomially bounded) problem are a class of **decision** problems for which there is polynomially bounded **non-deterministic** algorithms.

**Note:** Graph coloring, bin packing, subset sum, knapsack and the travelling salesman problem all belong to the class of NP problems. The reason is that the answers can be checked in polynomial time.

**Note:** The class P problems is a subsets of class NP problems. In other words an ordinary (deterministic) algorithm for a decision problem is with minor modification a special case of non-deterministic algorithm. This is true since instead of guessing in the first phase, a deterministic algorithm will provide a systematic method of finding the answer, if it exists.

**Important Question:** Are class P and NP problems the same (i.e.  $P = NP$ ), or is class P a subset of Class NP, i.e.  $P \subseteq NP$ ?

- In other words, are there problems that can be solved in polynomial time using a non-deterministic guesser that cannot be solved in polynomial time by an ordinary (deterministic) algorithm?
- If a problem is in NP, we can give yes/no answer if we check all guessed solutions in polynomial time. The trouble is that there are too many (usually exponential in  $n$ ) solutions to check.

- It is possible to come up with a clever algorithm that does not have to examine all solutions. For example, when sorting we do not check each of the  $n!$  permutations of the given  $n$  keys which puts the keys in order.
- The difficulty of the “hard” problems is that the known algorithms either examine all possibilities or clever short cuts are not good enough to give polynomial bounded algorithms.
- It is believed that NP is a larger class than P, but no NP problem has yet been found that would not be in P.

## NP- Complete Problems

NP-complete is the term used to describe the set of decision problems that are the hardest in NP class in the sense that if there were an  $O(n^p)$  algorithm for one of these NP-complete problems, then there would be an  $O(n^p)$  algorithm for each of them.

Some of the problems in Section 5.2 may seem easier than others, and in fact the worst case complexity of the algorithms found for them do differ (e.g. they are  $O(2^{\sqrt{n}})$ ,  $O(2^n)$ ,  $O(n!)$ , etc.), but they are NP-complete.

**Formal Definition:** A decision problem D is said to be NP-complete if:

1. It belong to class NP, and
2. Every problem in NP is polynomially reducible to D.

**Note:** It can be shown that graph coloring, Hamiltonian cycle, bin packing, subset sum, knapsack and travelling salesman problems are NP-complete. This means that if there is a polynomially bounded algorithm for one of them, then the others can also be solved by a polynomially bounded algorithm.

**Note:** The notion of NP-completeness requires polynomial reducibility of **all** problems in NP, both known and unknown, to the problem in question.

- To prove that a problem P is NP-complete, we must show:
  - (a) That the problem is NP, i.e. a randomly generated string can be checked in polynomial time to determine whether or not it represent a solution to the problem, and
  - (b) Every problem in NP is reducible to the problem in question in polynomial time. Alternatively, this can be done by showing that a known NP-complete problem can be transformed to the problem in question in polynomial time.

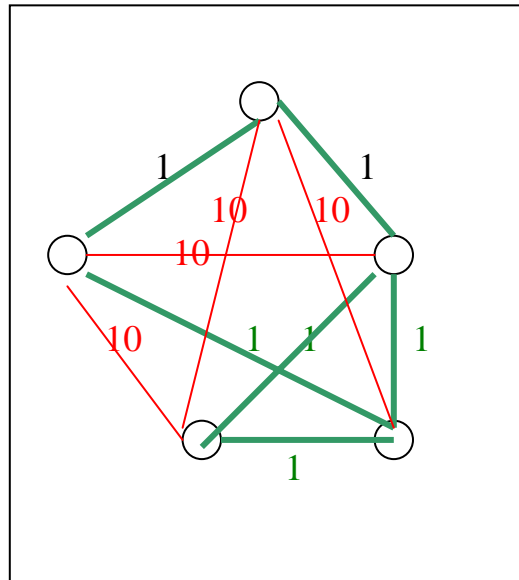
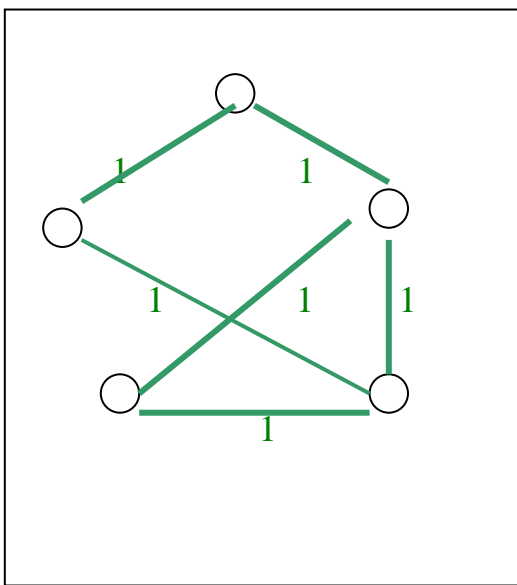
**Example:** Suppose we want to show that the travelling salesman problem (TSP) is NP-complete. This problem is stated as follows: Given a graph  $G(V, E)$  with edge costs and an integer  $K$ , is there a simple cycle that visits

all vertices and has a total cost less than or equal to  $K$ ?  
 Suppose that we know that the Hamiltonian cycle problem (HCP) is NP-complete.

We show that HCP is reducible to TSP in polynomial time. Note that in TSP all  $n_E = \frac{n_V}{2}(n_V + 1)$  edges are present and the graph is weighted. To transform HCP to TSP, construct a graph  $G_1$  where  $G_1$  has the same vertices as  $G$ , and all possible edges. Assign weights to the edges  $uv$  in  $G_1$  as follows

weight  $(uv) = 1$  if  $uv$  is also an edge in  $G$   
 $= 2$  otherwise

Choose  $K = n_V$ . Now  $G$  has a HC if and only if  $G_1$  has a TS with a total cost of  $n_V$ .



If the total cost of visiting all vertices in  $G_1$  is  $n_v$ , we have traveled only on  $n$  edges each with a cost of 1.

## Decision, Optimization and Solution

**Restriction of Structure:** If the set of inputs or the structure of a graph is restricted then an NP-complete problem may become a P problem.

- **Example 1:** HCP can be solved in poly-time (i.e. is in P) if the degree of every vertex is 1 or 2.
- **Example 2:** The GCP is in P, if degree of every vertex  $\leq 3$ . Thus it is the presence of vertices with high degree that make these problems hard (i.e. NP-complete).
- Sometimes a slight difference between the statement of a problem can change it from P to NP-complete and vice versa.
- **Example:** Shortest path is of  $O(m \lg n)$ . However, the longest path problem is NP-complete. The decision formulation of the two problems includes an integer  $k$  as input and asks if there is a path shorter than  $k$  or longer than  $k$ , respectively.
- There is no simple generalization for why a problem is NP-complete. There are still many open questions, the main one being is  $P = NP$ ?

**Recall** :There are two alternative statement of a problem, i.e. decision (yes/no answer) and optimization (finding the optimal value).

**Example**: In graph coloring problem, if  $k$  = number of colors, we can ask if the graph can be colored with  $k$  colors (yes/no), where  $k$  is given. The optimization problem is finding the minimum value of  $k$ . Neither of these two statements seek a solution (i.e. the color of each vertex).

- Thus we have three kinds of problems with increasing difficulty:

1. Decision problem
2. Optimal value
3. Optimal solution

- If for the coloring problem we have  $k$ , it is easy to check the solution, but it is very difficult to find the actual solution.
- It is easier to work with NP-completeness for the decision problems. Thus the **optimization problems** are sometimes called **NP-hard problems**, since they are harder than NP-complete decision problems.
- This is true since no polynomial time verification algorithm is known that can determine if a proposed solution is an optimal solution.

- Suppose that  $P = NP$ . If we had polynomial time algorithm for the decision problems, can we then find the optimal value in polynomial time? In many cases we can.

**Example:** Suppose we have a polynomial time Boolean function  $\text{CanColor}(G,k)$  which returns true if and only if  $G$  can be colored with  $k$  colors. Then the following will find the optimal value in polynomial time:

```
ColorNumber(G)
{
    for(k=1; k<= n, k++) // n = # vertices
        if(CanColor(G,k))
            break;
    return k
}
```

## 6.8 Approximation and Heuristic Algorithms

- Many application problems are NP-complete. What can we do if we want to solve one of these problems?
- There may be a fast (polynomial bounded) algorithm, that even though is **not** optimal, may provide a near optimal solution. Such an algorithm is called an **approximation or a heuristic algorithm**.
- A heuristic algorithm is one that uses “rule of thumb” and usually is some idea that makes sense, but it may not be possible to prove its good performance. An approximation solution is usually good enough for many applications.

- The strategies, or heuristics, used by approximation algorithms are generally simple, yet they provide good results. Many of them use “**greedy heuristics**”.

**Greedy Algorithms:** These algorithms work in phases. In each phase a good (local optimum) choice is made.

At the completion of the phases we will have

- global optimal solution,
- a **near optimal solution**, or
- a sub-optimal solution.

**Example 1:** Change a large bill using minimum number of bills and coins. To do this, repeatedly dispense largest denominations. This solution is optimal for the U.S. currency system. However, it is not optimal if the denominations are 100, 40, 25, 5, 1, since to change 100 using largest denomination each time, we will have  $100 = 40+40+5+5+5+5$  (a total of six bills). The optimal solution is  $100 = 25+25+25+25$  (four bills).

**Example 2:** Route selection based on local optimal solution does not result in a global optimal solution.

### Approximation and Heuristic Algorithms

Many application problems are NP-complete. What can we do if we want to solve one of these problems? There may be a fast (polynomial bounded) algorithm, that even though is not optimal, may provide a near optimal solution. Such an algorithm is called an approximation or a heuristic algorithm.

A heuristic algorithm is one that uses “rule of thumb” and usually is some idea that makes sense, but it may not be possible to prove its good

performance. An approximation solution is usually good enough for many applications. The strategies, or heuristics, used by approximation algorithms are generally simple, yet they provide good results. Many of them use “greedy heuristics”.

**Greedy Algorithms:** These algorithms work in phases. In each phase a good (local optimum) choice is made. At the completion of the phases we will have global optimal solution, a near optimal solution, or a sub-optimal solution.

**Example:** Change a large bill using minimum number of bills and coins. To do this, repeatedly dispense largest denominations. This solution is optimal for the U.S. currency system. However, it is not optimal if the denominations are 100, 40, 25, 5, 1, since to change 100 using largest denomination each time, we will have  $100 = 40+40+5+5+5+5$  (a total of six bills). The optimal solution is  $100 = 25+25+25+25$  (four bills).

**Example:** Route selection based on local optimal solution does not result in a global optimal solution.

If the output of an algorithm is an approximation of the actual optimal solution, we would like to know how good is this approximation. Suppose that we want to maximize some function  $f$ , and we get an approximate solution  $s_{\text{apx}}$  whereas the exact solution is  $s_{\text{ext}}$ . We use the accuracy ratio

For max problems 
$$R = \frac{f(s_{\text{apx}})}{f(s_{\text{ext}})} \quad (1)$$

This ratio is equal to 1 when the approximate solution is the same as the exact solution, and in other cases it is less than 1 (**Why?**), so  $R$  ranges as  $0 \leq R \leq 1$ . The closeness of  $R$  to 1 indicates the accuracy of the solution. In case we want to minimize function  $f$ , we take the ratio of

For min problems: 
$$R = \frac{f(s_{\text{ext}})}{f(s_{\text{apx}})} \quad (2)$$

Note that we want this ratio to be close to 1 not just for one set of inputs or one instance of the problem, but over all instances. The ratio in (1) and (2) is called the **performance ratio** if it is taken over all instances of the problem. In general we do not know  $s_{\text{ext}}$ , however an estimate of closeness

of the approximate solution to the optimal solution can be obtained using the following concept.

We say that a polynomial-time approximation algorithm is a **K-approximation** algorithm if  $f(s_{\text{apx}}) \leq K f(s_{\text{ext}})$  for all instances of the problem.

**Note:** Although most combinatorial NP-complete problem have very similar complexity, approximation algorithms for these problems may have much different complexities.

## Approximation Algorithms for Traveling Salesman Problem (TSP)

We consider two simple approximation for TSP.

**Nearest Neighbor Algorithm:** The heuristic is to start on any city, and go to the nearest unvisited city until all cities are visited.

Example: In the following graph, starting at vertex A, the nearest neighbor gives the approximate solution as  $s_{\text{apx}} : A \rightarrow B \rightarrow C \rightarrow D \rightarrow A$  with a total length of 10. The optimal (exact) solution can be found by exhaustive search  $s_{\text{ext}} : A \rightarrow B \rightarrow D \rightarrow C \rightarrow A$  which has a total length of 8. The accuracy

ration is  $R = \frac{f(s_{\text{ext}})}{f(s_{\text{apx}})} = 0.8$ . This number is not bad, however, nothing can

be said in general about the accurately of solutions since the nearest neighbor algorithm can force us to traverse a very long edge. In fact the edge weight  $AD = 5$  is changed to a large value, say 1000, then the algorithm will yield  $s_{\text{apx}} : A \rightarrow B \rightarrow C \rightarrow D \rightarrow A$  with a length of 1005, while the optimal

still remains 8. Thus  $R = \frac{f(s_{\text{ext}})}{f(s_{\text{apx}})} = 0.008$  which is a very low number.

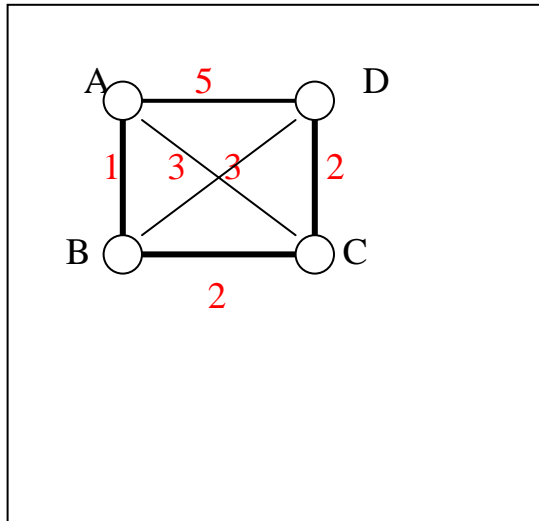
However, if the distances satisfy the Euclidean distance properties, namely

Triangular inequality:  $\text{dist}(i, j) \leq \text{dist}(i, k) + \text{dist}(k, j)$   
Symmetry:  $\text{dist}(i, j) = \text{dist}(j, i)$

The accuracy ratio can now be shown to satisfy

$$R = \frac{f(s_{\text{apxt}})}{f(s_{\text{ext}})} \leq \frac{1}{2} (\lceil \lg n \rceil + 1)$$

where  $n$  is the number of cities.



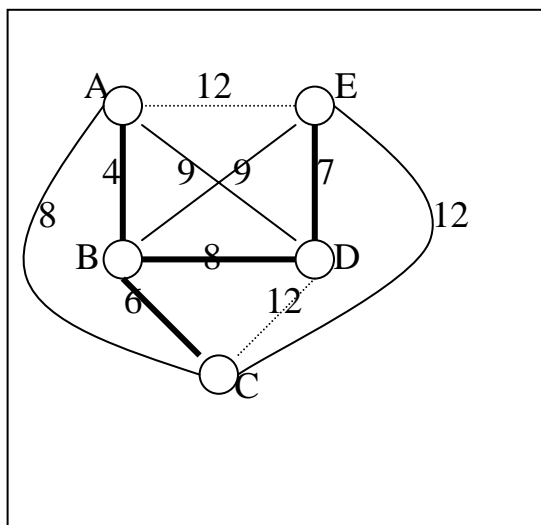
**Using Minimum Spanning Tree:** This algorithm assumes Euclidean distances, and is outlined as follows:

1. Construct a minimum spanning tree (MST) of the graph for the TSP.
2. Start at a vertex, visit vertices in MST
3. Scan the list of visited vertices in Step 2, and delete all duplicate vertices except the starting vertex. The remaining vertices are the answer.

**Example:**

1. The MST of the graph shown consists of edges AB, BC, BD, DE.
2. Traversing MST starting from A produces A, B, C, B, D, E, D, B, A.
3. Since B is repeated, we eliminate it to get A, B, C, D, E, A

The answer obtained above is not optimal. In order to estimate the closeness of the solution to the optimal one, we have the following result which we present without a proof.



**Result:** The TSP solution using MST is a 2-approximation for Euclidean distances, i.e.

$$f(s_{\text{apx}}) \leq 2f(s_{\text{ext}})$$

Now we ask the question: Can we hope to come up with a polynomial time  $K$ -approximation algorithm ( $K < \infty$ ) for TSP for all instances of the problem? The following result shows that this is not possible unless  $P=NP$ .

**Result 2:** If  $P \neq NP$ , then there exists no  $K$ -approximation algorithm for the TSP so that for all instances of this problem

$$f(s_{\text{apx}}) \leq Kf(s_{\text{ext}})$$

for some constant  $K$ .

**Proof:** We prove the result by contradiction. Suppose that such an approximation algorithm and a constant  $K$  exist. We show that this approximation algorithm can be used to solve the Hamiltonian cycle problem (HCP). Since HCP can be transformed to TSP (see Section 5.6), this means that HCP can be solved in polynomial time. However, we know that HCP is an NP-complete problem, and therefore we have a contradiction unless  $P = NP$ .

## Approximation Algorithms for Knapsack Problem

The knapsack problem (KSP) is an NP-complete problem, that we have encountered several times before. Suppose we have a bag of capacity  $S$  and  $n$  objects with sizes  $s_1, s_2, \dots, s_n$ , with values  $w_1, w_2, \dots, w_n$ . Find the objects with the largest total value that can fit into the bag.

**Greedy Algorithms for KSP:** One greedy algorithm is to sort and take the items in decreasing order of their size, but larger items may not be the most valuable ones. Alternatively, we can pick the items in decreasing order according to their value, but there is no guarantee that the space will be used most efficiently. However by finding the value to size ratio  $w_i/s_i$  and selecting the items in decreasing order of this ratio, we may be able to solve the KSP. The algorithm is

1. Compute the ratios  $w_i/s_i, i=1, 2, \dots, n$ .
2. Sort items in non-increasing order of the ratios.
3. If the current item fits into knapsack, place it otherwise proceed to the next item until no item is left.

Unfortunately, this greedy algorithm does not always give an optimal solution as the following example shows. (if it did we would have a polynomial time algorithm for the NP problem).

Item	size	value	ratio	
A	1	2	2	knapsack capacity =100
B	100	100	1	

According to the above algorithm, only item A with a value of 2 is picked (**Why?**), whereas the optimal solution is item B (**Why?**). Note that the

performance ratio  $R = \frac{f(s_{\text{ext}})}{f(s_{\text{apx}})} = \frac{2}{100}$ , and can approach zero if the size =

value = very large number.

The above greedy algorithm can be improved as follows: Take the items using the above method, or take a single most valuable item which ever

gives a better performance. It can be shown that the ratio  $R = \frac{f(s_{\text{ext}})}{f(s_{\text{apx}})} \geq 0.5$

implying that the improved algorithm never takes less than half the optimal value.

Greedy Algorithm for Continuous KSP: In this version we are permitted to take any fraction of the item.

- 1 Compute the ratios  $w_i / s_i$ ,  $i=1, 2, \dots, n$ .
- 2 Sort items in non-increasing order of the ratios.
- 3 If the current item fits into knapsack, place it and go to the next item, otherwise takes the largest fraction to fill the knapsack entirely
- 4 Repeat 3 until knapsack is filled or there is no item left.

It is to be noted that this algorithm always produces an optimal solution to the continuous KSP (Why/How?).

## Approximation Algorithms Bin Packing problem

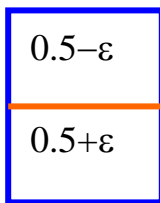
**The bin packing problem (BPP) is: Given the objects of sizes  $S_1, S_2, \dots, S_n$  where  $0 < S_i \leq 1$ , and bins of unit capacity each, pack the objects in fewest number of bins.**

**This is an NP-complete problem and we consider approximation (near optimal) solutions.**

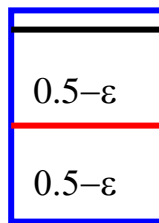
## On-line bin packing algorithms

**Here we have to pack items as they arrive, i.e. we must place an item in a bin before processing the next item.**

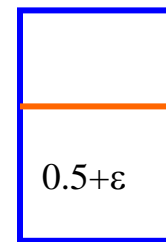
The on-line algorithm does not give an optimal solution even if unlimited computation is allowed. To see this, consider  $n/2$  small items of size  $0.5 - \varepsilon$ , arriving first and  $n/2$  slightly larger items of size  $0.5 + \varepsilon$  arriving after the small items where  $\varepsilon$  is a small number and the total number of items is  $n$ . If we were given a chance to examine all items we would place one item of size  $0.5 - \varepsilon$  and another of size  $0.5 + \varepsilon$  in the same bin, as shown below, requiring a total of  $m$  bins. This is the optimal number of bins  $b_{\text{opt}}$  used where  $b_{\text{opt}} = n/2$



**Optimal packing**  
 $n/2$  bins



**On-line packing**  
 $n/4 + n/2 = 3n/4$  bins



Using on-line packing and assuming  $n/2$  is even, we would put the first  $m$  items into  $n/2$  bins, and each of the second  $m$  items into a separate bins resulting a total of  $b = \frac{n}{4} + \frac{n}{2} = \frac{3n}{4}$  which requires 50% more bins than the optimal, that is  $b = 1.5 b_{\text{opt}}$ .

**Exercise:** Show that if  $n/2$  is odd we would still need  $b = \frac{3n}{4}$  bins.

Based on the above example, we can state the following result:

**Result:** There are input that force the on-line bin packing algorithm to use 1.5 times the optimal number of bins.

In the following we give three on-line algorithms that would always ensure  $b < 2b_{\text{opt}}$  for any input.

### (a) Next Fit Strategy

The sketch of this algorithm is:

if item fits in the bin where the last item was placed  
    place item there  
else  
    create a new bin, and put the item there

This algorithm is simple and its complexity is  $O(n)$ , where  $n$  is the number of items.

**Theorem:** Let  $b_{\text{opt}}$  be the optimal number of bins required to pack  $n$  items. Then the next fit algorithm uses  $b \leq 2b_{\text{opt}}$  bins. Furthermore, there exist sequences such that next fit uses  $b = 2b_{\text{opt}} - 2$  bins.

**Proof:** Let  $B_i, B_{i+1}$  be two adjacent bins with sizes  $s_i, s_{i+1}$ . We know that  $(s_i + s_{i+1}) > 1$  since otherwise they would have been both placed in  $B_i$ . Therefore no more

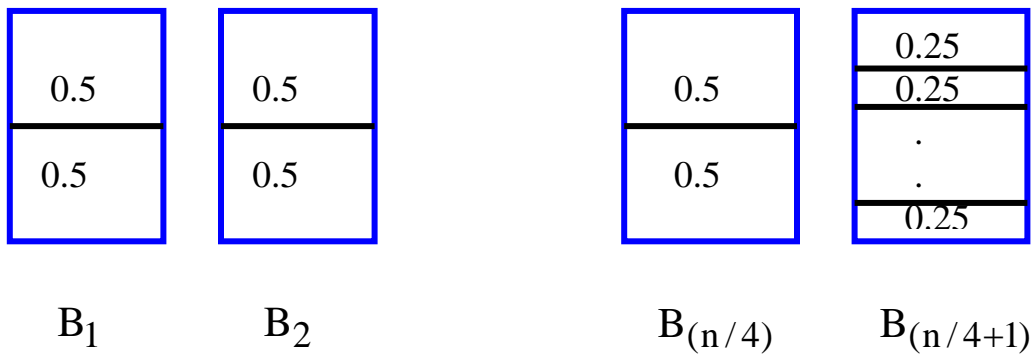
than half the space is wasted, and thus the next fit uses at most  $2b_{\text{opt}}$  bins.

To show that there exists a sequence that will use  $b = 2b_{\text{opt}} - 2$  bins, consider the following sequence

$$\begin{cases} s_i = 0.5 & \text{for } i \text{ odd} \\ s_i = \frac{2}{n} & \text{for } i \text{ even} \end{cases}$$

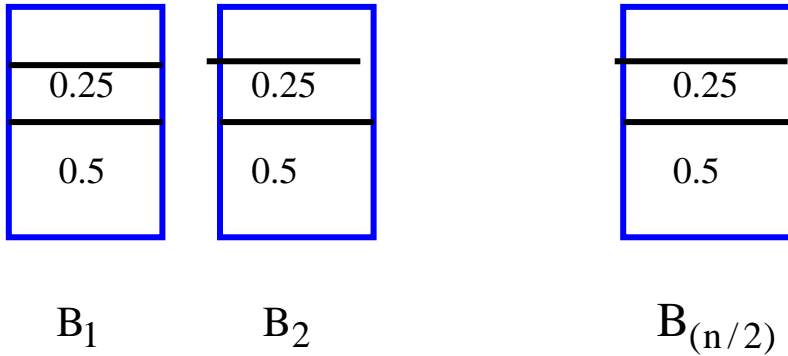
For example if the number of items  $n = 8$ , then the sequence of items would be  $.5, .25, .5, .25, .5, .25, .5, .25$ .

Assuming  $n$  is divisible by 4, then the optimal packing would be



The optimal number of bins is  $b_{\text{opt}} = \frac{n}{4} + 1$ .

Now for the next fit



**The number of bins required by next fit is**

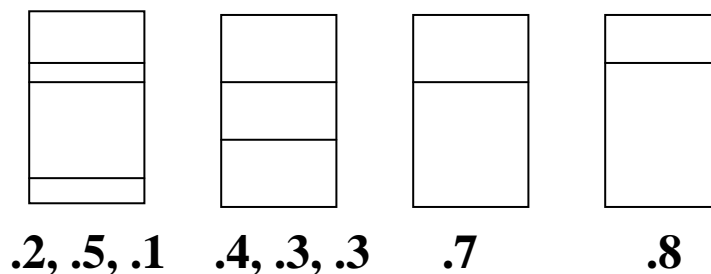
$b = \frac{n}{2} = 2b_{opt} - 2$ , **using the above equation.**

**(a) First Fit Strategy**

**The algorithm is as follows**

- scan existing bins in order starting from first**
- if current item fits into any existing bin**
- place it**
- else**
- create a new bin and place it.**

**Example:** Item sizes are .2, .5, .4, .7, .1, .3, .8, .3. **The result of first fit is shown below:**



Since for each item, all the currently unfilled bins are scanned, and there are  $n$  items, the complexity of the algorithm is  $O(n^2)$  in the worst case.

Note that at any point, at most one bin can be more than half empty, and the other bins are at least half full. Thus the number bin used in first fit is  $b \leq 2b_{\text{opt}}$ . The following results, stated here without proof, indicate that a better result is possible.

**Theorem:** First fit never uses more than  $\lceil 1.7b_{\text{opt}} \rceil$  bins.

**Result:** If  $n$  is large, and the sizes are uniformly distributed, then the number of bins used by the first fit strategy is  $b \cong 1.02b_{\text{opt}}$ .

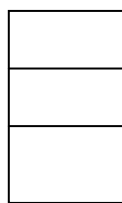
### (b) Best Fit Strategy

This strategy requires that the new items be placed in the tightest fit space among the current bins.

**Example:** Let the sizes be  $.2, .5, .4, .7, .1, .3, .8$ . The result of best fit is shown below.



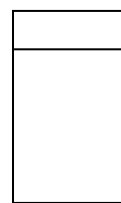
$.2, .5, .1$



$.4$



$.7, .3$



$.8$

This algorithm processes each new item by scanning down the bins sequentially, and therefore its complexity is  $O(n^2)$ .

It can be shown that best fit uses  $b \leq 1.7b_{\text{opt}}$ .

### Off-line bin packing algorithms

Here we can process all items before packing. As a result performances closer to the optimal can be achieved. The items are sorted in decreasing item size. Either first fit or best fit can be used, and it is easy to see that

$$b_{\text{opt}} \leq b \leq 2b_{\text{opt}}$$

We define the number of extra bins as  $b_{\text{ext}} = b - b_{\text{opt}}$  which is the number of bins used in excess of the minimum possible.

**Lemma 1:** Let items be sorted in decreasing size so that  $s_{j+1} \leq s_j$ . Then all items that first-fit decreasing algorithm places in the extra bins have size at most  $\frac{1}{3}$ .

**Lemma 2:** The number of objects placed in extra bins is  $\leq b_{\text{opt}}$ .

**Proof:** Suppose that the Lemma 2 is incorrect, and that at least  $b_{\text{opt}}$  can be place in the extra bins. Suppose the sizes of objects in the first  $b_{\text{opt}}$  bins are  $u_1, u_2, \dots, u_{b_{\text{opt}}}$  and the sizes of objects in the extra bins are  $v_1, v_2, \dots, v_{b_{\text{opt}}}$ . Then we have

$$\sum_{i=1}^n s_i \geq \sum_{j=1}^{b_{\text{opt}}} u_j + \sum_{j=1}^{b_{\text{opt}}} v_j = \sum_{j=1}^{b_{\text{opt}}} (u_j + v_j)$$

Now  $u_j + v_j > 1$  since otherwise  $v_j$  would not have been placed in the extra bin. It follows from the above summation inequality that

$$\sum_{i=1}^n s_i > \sum_{j=1}^{b_{\text{opt}}} 1 > b_{\text{opt}}$$

This is a contradiction, since it is assumed that all items can be placed in the optimal number of bins.

**Theorem:** The total number of bins used by first fit decreasing algorithm is  $b \leq \frac{4b_{\text{opt}} + 1}{3}$ .

**Proof:** There are  $b_{\text{opt}}$  original bins plus  $(b_{\text{opt}} - 1)$  objects of size  $\frac{1}{3}$  giving

$$b \leq b_{\text{opt}} + \left\lceil \frac{1}{3}(b_{\text{opt}} - 1) \right\rceil \leq \frac{4b_{\text{opt}} + 1}{3}$$

**Note:** It is also possible to show that no more than  $(\frac{9}{11}b_{\text{opt}} + 4)$  are needed in the off-line first fit decreasing algorithm.

**In conclusion, bin packing is an example of using heuristic algorithms for obtaining a good sub-optimal solution with a low complexity for a hard (NP-complete) problem.**