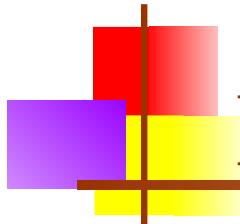


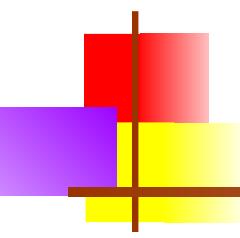
Greedy Algorithms





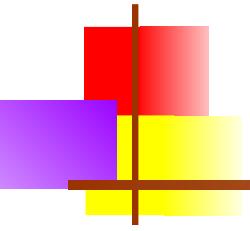
List of Algorithms' Categories (Brief)

- Algorithms' types we will consider include:
 - Simple Recursive Algorithms
 - Backtracking Algorithms
 - Divide and Conquer Algorithms
 - Dynamic Programming Algorithms
- ➡ ■ Greedy Algorithms
 - Branch and bound Algorithms
 - Brute Force Algorithms
 - Randomized Algorithms



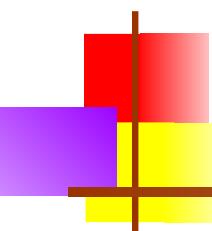
Optimization Problems

- An **Optimization Problem** is one in which we want to find, not just *a* solution, but the *best* solution
- A “Greedy Algorithm” sometimes works well for optimization problems
- A **Greedy Algorithm** works in phases. At each phase:
 - We take the best we can get right now, without regard for future consequences
 - We hope that by choosing a *local* optimum at each step, we will end up at a *global* optimum



The Greedy Algorithms

- **The greedy method** is a general algorithm design paradigm, built on the following elements:
 - **Configurations**: different choices/collections/values to find
 - **Objective Function**: a score assigned to configurations, which we want to either maximize or minimize
- It works best (efficiently) when applied to problems with the **greedy-choice** property:
 - Globally-optimal solution can always be found by a series of local improvements from a starting configuration.

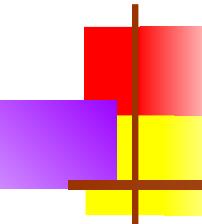


Greedy Algorithms:

Many real-world problems are *optimization* problems for which we need to find an optimal solution among many possible *candidate* solutions. A familiar scenario is the change-making problem that we often encounter at a cash register: receiving the fewest numbers of coins to make change after paying the bill for a purchase. For example, the purchase is worth \$5.27, how many coins and what coins does a cash register return after paying a \$6 bill?

The Make-Change Algorithm:

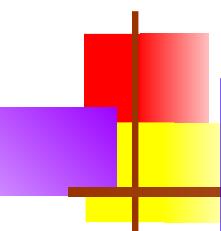
For a given amount (e.g. \$0.73), use as many quarters (\$0.25) as possible without exceeding the amount. Use as many dimes (\$.10) for the remainder, then use as many nickels (\$.05) as possible. Finally, use the pennies (\$.01) for the rest.



Example: To make change for the amount $x = 67$ (cents).

Use $q = (x/25) = 2$ quarters. The remainder is $x - 25q = 17$, which we use $d = (17/10) = 1$ dime. Then the remainder is $17 - 10d = 7$, therefore we can use $n = (7/5) = 1$ nickel. So, the remainder is $7 - 5n = 2$, which requires $p = \lfloor 2/1 \rfloor = 2$ pennies. Total number of coins used is $q + d + n + p = 6$.

Note: The above algorithm is optimal i.e. it uses the fewest number of coins among all possible ways to make change for a given amount. (This fact can be proven formally). However, this is dependent on the denominations of the US currency system. For example, try a system that uses denominations of 1-cent, 6-cent, and 7-cent coins, and try to make change for $x = 18$ cents. The greedy strategy uses two 7-cents and four 1-cents, for a total of 6 coins. However, the optimal solution is to use three 6-cent coins.



A Generic Greedy Algorithm:

- (1) Initialize C to be the set of candidate solutions
- (2) Initialize a set $S = \emptyset$ (the set is to be the optimal solution we are constructing). (3)
- While $C \neq \emptyset$ and S is (still) not a solution do
 - (3.1) select x from set C using a greedy strategy
 - (3.2) delete x from C (3.3) if $\{x\} \cup S$ is a *feasible* solution, then $S = S \cup \{x\}$
 - (i.e., add x to set S) (4) if S is a solution then
return S
- (5) else return *failure*

In general, the greedy algorithm is efficient because it makes a sequence of (local) decisions and never backtracks. However, the solution is not always optimal.

Coin changing

Goal. Given U. S. currency denominations { 1, 5, 10, 25, 100 }, devise a method to pay amount to customer using fewest coins.

Ex. 34¢.



Cashier's algorithm. At each iteration, add coin of the largest value that does not take us past the amount to be paid.

Ex. \$2.89.



Cashier's algorithm

At each iteration, add coin of the largest value that does not take us past the amount to be paid.

CASHIERS-ALGORITHM (x, c_1, c_2, \dots, c_n)

SORT n coin denominations so that $0 < c_1 < c_2 < \dots < c_n$.

$S \leftarrow \emptyset$. ← multiset of coins selected

WHILE ($x > 0$)

$k \leftarrow$ largest coin denomination c_k such that $c_k \leq x$.

IF (no such k)

RETURN “no solution.”

ELSE

$x \leftarrow x - c_k$.

$S \leftarrow S \cup \{ k \}$.

RETURN S .

Cashier's algorithm (for arbitrary coin denominations)

- Q. Is cashier's algorithm optimal for any set of denominations?
- A. No. Consider U.S. postage: 1, 10, 21, 34, 70, 100, 350, 1225, 1500.
- Cashier's algorithm: $140\text{¢} = 100 + 34 + 1 + 1 + 1 + 1 + 1$.
 - Optimal: $140\text{¢} = 70 + 70$.



- A. No. It may not even lead to a feasible solution if $c_1 > 1$: 7, 8, 9.
- Cashier's algorithm: $15\text{¢} = 9 + ?$.
 - Optimal: $15\text{¢} = 7 + 8$.

Properties of any optimal solution (for U.S. coin denominations)

Property. Number of pennies ≤ 4 .

Pf. Replace 5 pennies with 1 nickel.

Property. Number of nickels ≤ 1 .

Property. Number of quarters ≤ 3 .

Property. Number of nickels + number of dimes ≤ 2 .

Pf.

- Recall: ≤ 1 nickel.
- Replace 3 dimes and 0 nickels with 1 quarter and 1 nickel;
- Replace 2 dimes and 1 nickel with 1 quarter.



dollars
(100¢)



quarters
(25¢)



dimes
(10¢)



nickels
(5¢)



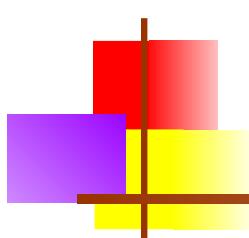
Optimality of cashier's algorithm (for U.S. coin denominations)

Theorem. Cashier's algorithm is optimal for U.S. coins { 1, 5, 10, 25, 100 }.

Pf. [by induction on amount to be paid x]

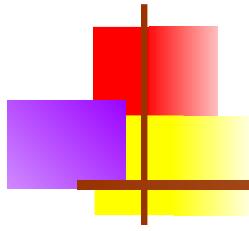
- Consider optimal way to change $c_k \leq x < c_{k+1}$: greedy takes coin k .
- We claim that any optimal solution must take coin k .
 - if not, it needs enough coins of type c_1, \dots, c_{k-1} to add up to x
 - table below indicates no optimal solution can do this
- Problem reduces to coin-changing $x - c_k$ cents, which, by induction, is optimally solved by cashier's algorithm. ▀

k	c_k	all optimal solutions must satisfy	max value of coin denominations c_1, c_2, \dots, c_{k-1} in any optimal solution
1	1	$P \leq 4$	-
2	5	$N \leq 1$	4
3	10	$N + D \leq 2$	$4 + 5 = 9$
4	25	$Q \leq 3$	$20 + 4 = 24$
5	100	<i>no limit</i>	$75 + 24 = 99$



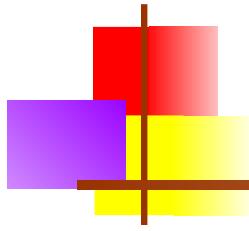
Example: Counting Money

- Suppose we want to count out a certain amount of money, using the fewest possible bills and coins
- A “Greedy Algorithm” would do the following:
At each step, take the largest possible bill or coin that does not overshoot
 - Example: To make \$6.39, we can choose:
 - a \$5 bill
 - a \$1 bill, to make \$6
 - a 25¢ coin, to make \$6.25
 - A 10¢ coin, to make \$6.35
 - four 1¢ coins, to make \$6.39
- For US money, the greedy algorithm always gives the optimum solution



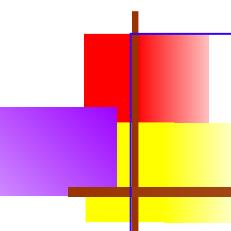
A Failure of the Greedy Algorithm

- In some monetary system, for example: “Rupees” come in Rs. 1, Rs. 2, Rs. 5, and Rs. 10 coins
- Using a Greedy Algorithm to count Rs. 15, we would get
 - One 10 Rs.; Five 1 Rs., for a total of 15 Rs.; requires Six coins
 - Three 5 Rs. for a total of 15 Rs.; Requires Three coins only
- A better solution would be to use one 5 Rs. piece and one 10 Rs piece for a total of Rs. 15
 - This requires Two coins only
- The Greedy Algorithm results in a solution, but not in finding an optimal solution



Example: Text Compression

- Given a string of text characters X, efficiently encode X into a smaller string of characters Y
 - Saves memory and/or bandwidth
- A good approach: **Huffman Encoding**
 - Compute frequency $f(c)$ for each character c.
 - Encode high-frequency characters with short code words
 - No code word is a prefix for another code
 - Use an optimal encoding tree to determine code words



Huffman Codes:

Suppose we wish to save a text (ASCII) file on the disk or to transmit it through a network using an encoding scheme that minimizes the number of bits required. Without *compression*, characters are typically encoded by their ASCII codes with 8 bits per character. We can do better if we have the freedom to design our own encoding.

Example. Given a text file that uses only 5 different letters (a, e, i, s, t), the space character, and the newline character. Since there are 7 different characters, we could use 3 bits per character because that allows 8 bit patterns ranging from 000 through 111 (so we still one pattern to spare). The following table shows the encoding of characters, their frequencies, and the size of encoded (compressed) file.

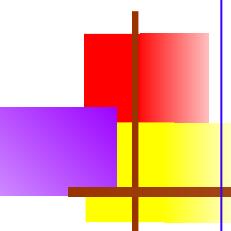
Character	Frequency	Code	Total bits
a	10	000	30
e	15	001	45
i	12	010	
36	s	011	
9	t	100	
12	space	101	
39	newline	110	
3			
Total	58		174

Fixed-length encoding

Code	Total bits
001	30
01	30
10	24
00000	15
0001	16
11	26
00001	5
	146

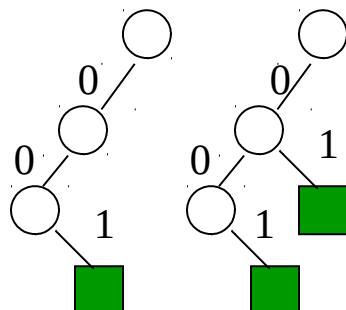
Variable-length encoding

If we can use variable lengths for the codes, we can actually compress more as shown in the above. However, the codes must satisfy the property that no code is the prefix of another code; such code is called a *prefix code*.

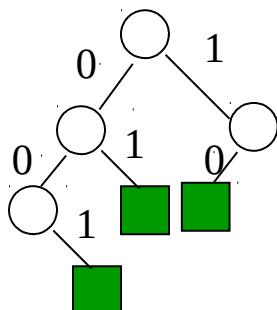


How to design an optimal prefix code (i.e., with minimum total length) for a given file?

We can depict the codes for the given collection of characters using a binary tree as follows: reading each code from left to right, we construct a binary tree from the root following the left branch when encountering a ‘0’, right branch when encountering a ‘1’. We do this for all the codes by constructing a single combined binary tree. For example,

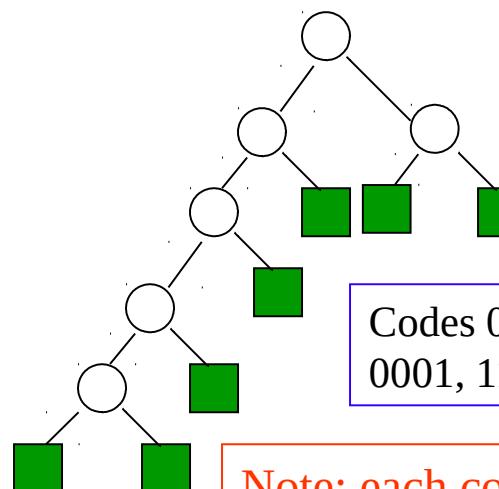


Code 001



Codes 001
and 01

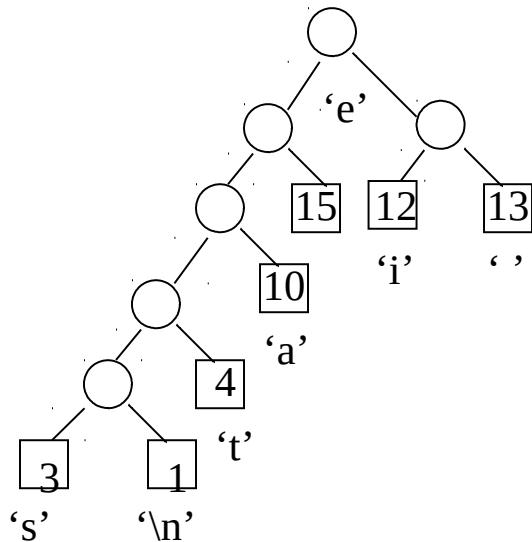
Codes 001,
01, and 10



Codes 001, 01, 10, 00000,
0001, 11, and 00001

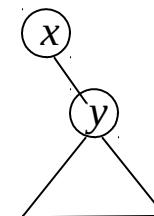
Note: each code terminates at a
leaf node, by the prefix property.

We note that the encoded file size is equal to the total weighted external path lengths if we assign the frequency to each leaf node. For example,

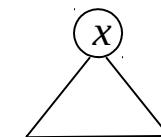


Total file size = $3*5 + 1*5 + 4*4 + 10*3 + 15*2 + 12*2 + 13*2 = 146$, which is exactly the total weighted external path lengths.

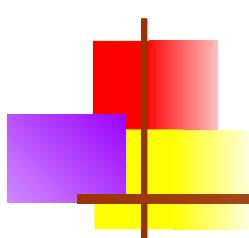
We also note that in an optimal prefix code, each node in the tree has either no children or has two. Thus, the optimal binary merge tree algorithm finds the optimal code (Huffman code).



Node x has only one child y



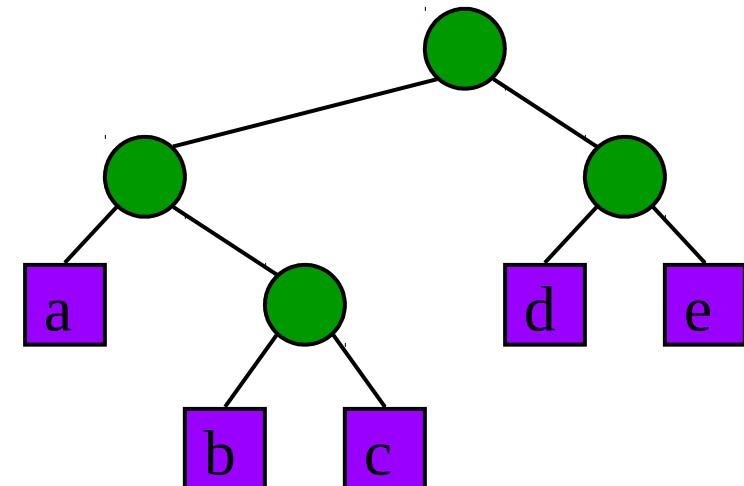
Merge x and y, reducing total size

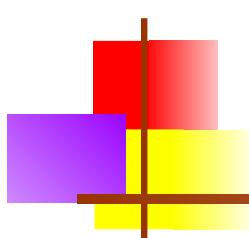


Encoding Tree Example

- A **code** is a mapping of each character of an alphabet to a binary code-word
- A **prefix code** is a binary code such that no code-word is the prefix of another code-word
- An **encoding tree** represents a prefix code
 - Each external node stores a character
 - The code word of a character is given by the path from the root to the external node storing the character (0 for a left child and 1 for a right child)

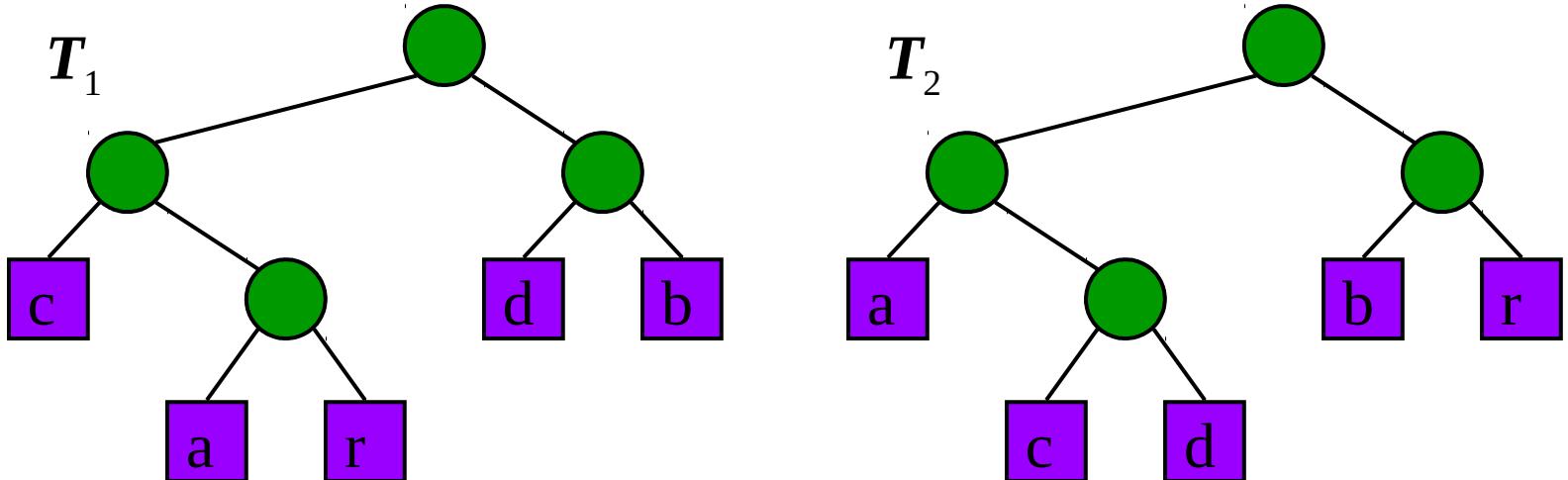
00	010	011	10	11
a	b	c	d	e

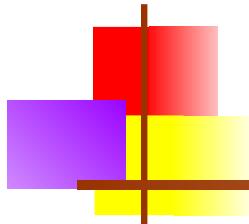




Encoding Tree Optimization

- Given a text string X , we want to find a prefix code for the characters of X that yields a small encoding for X
 - Frequent characters should have long code-words
 - Rare characters should have short code-words
- Example
 - $X = \text{abracadabra}$
 - T_1 encodes X into 29 bits
 - T_2 encodes X into 24 bits





Huffman's Algorithm

- Given a string X , Huffman's Algorithm constructs a prefix code that minimizes the size of the encoding of X
- It runs in time $O(n + d \log d)$, where n is the size of X and d is the number of distinct characters of X
- A heap-based priority queue is used as an auxiliary structure

Algorithm *HuffmanEncoding*(X)

Input string X of size n

Output optimal encoding trie for X

$C \leftarrow \text{distinctCharacters}(X)$

$\text{computeFrequencies}(C, X)$

$Q \leftarrow$ new empty heap

for all $c \in C$

$T \leftarrow$ new single-node tree storing c

$Q.\text{insert}(\text{getFrequency}(c), T)$

while $Q.\text{size}() > 1$

$f_1 \leftarrow Q.\text{minKey}()$

$T_1 \leftarrow Q.\text{removeMin}()$

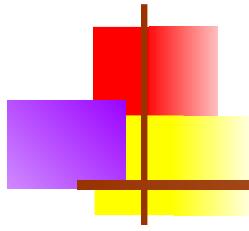
$f_2 \leftarrow Q.\text{minKey}()$

$T_2 \leftarrow Q.\text{removeMin}()$

$T \leftarrow \text{join}(T_1, T_2)$

$Q.\text{insert}(f_1 + f_2, T)$

return $Q.\text{removeMin}()$



Example for Huffman Encoding

$X = \text{abracadabra}$

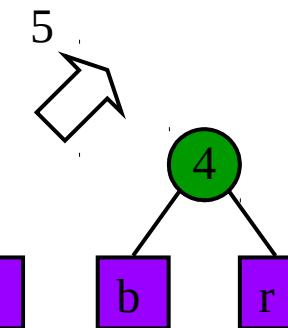
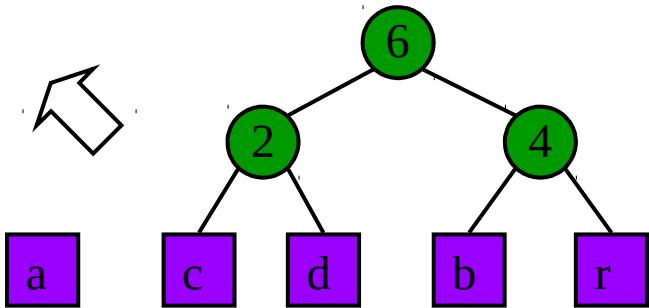
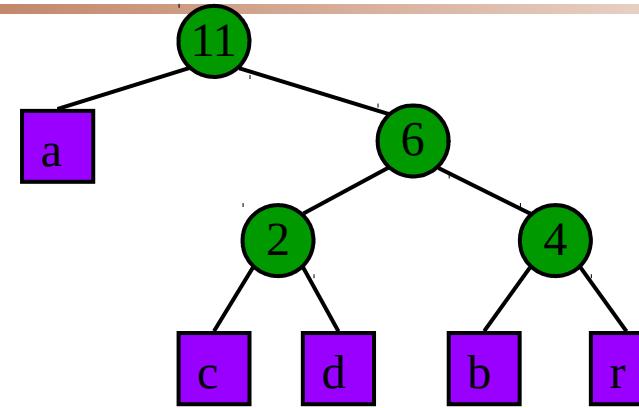
Frequencies

a	b	c	d	r
5	2	1	1	2

a b c d r
5 2 1 1 2



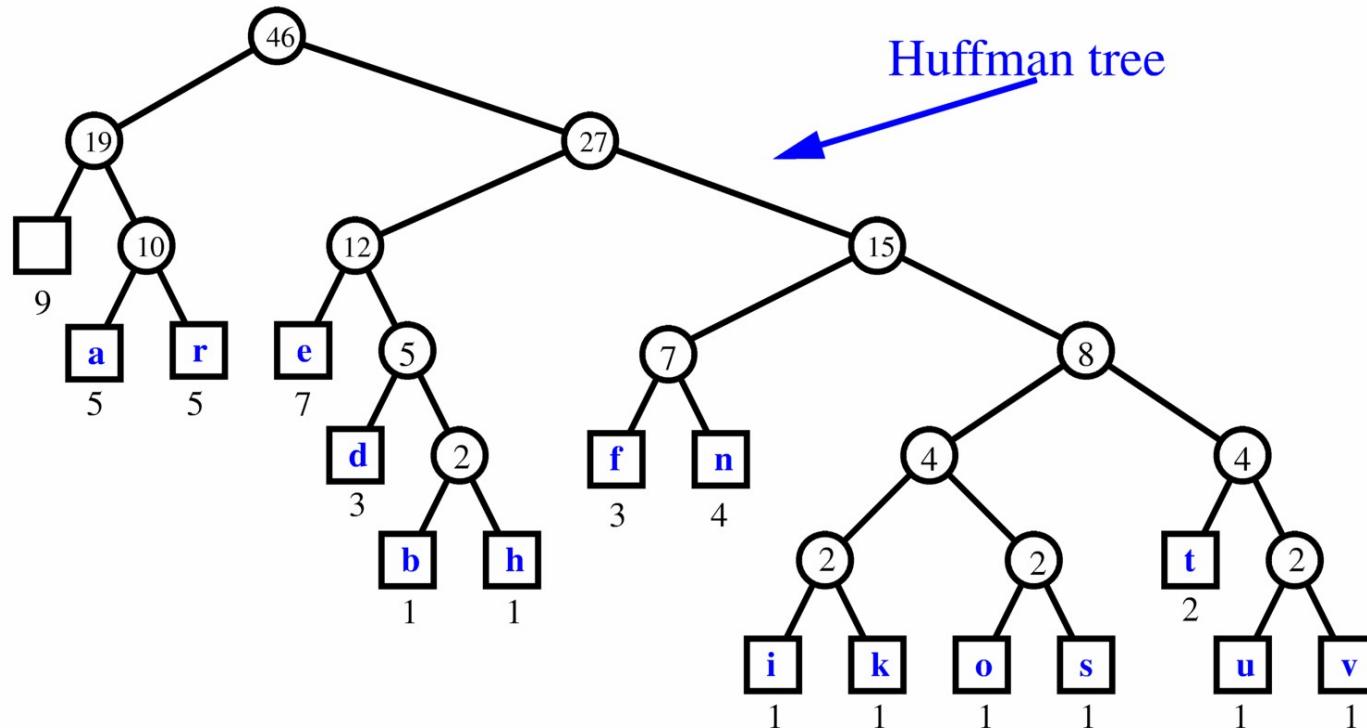
a b c d r
5 2 2

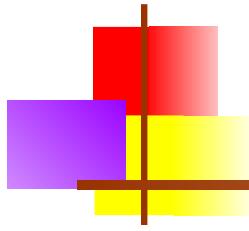


Extended Huffman Tree Example

String: **a fast runner need never be afraid of the dark**

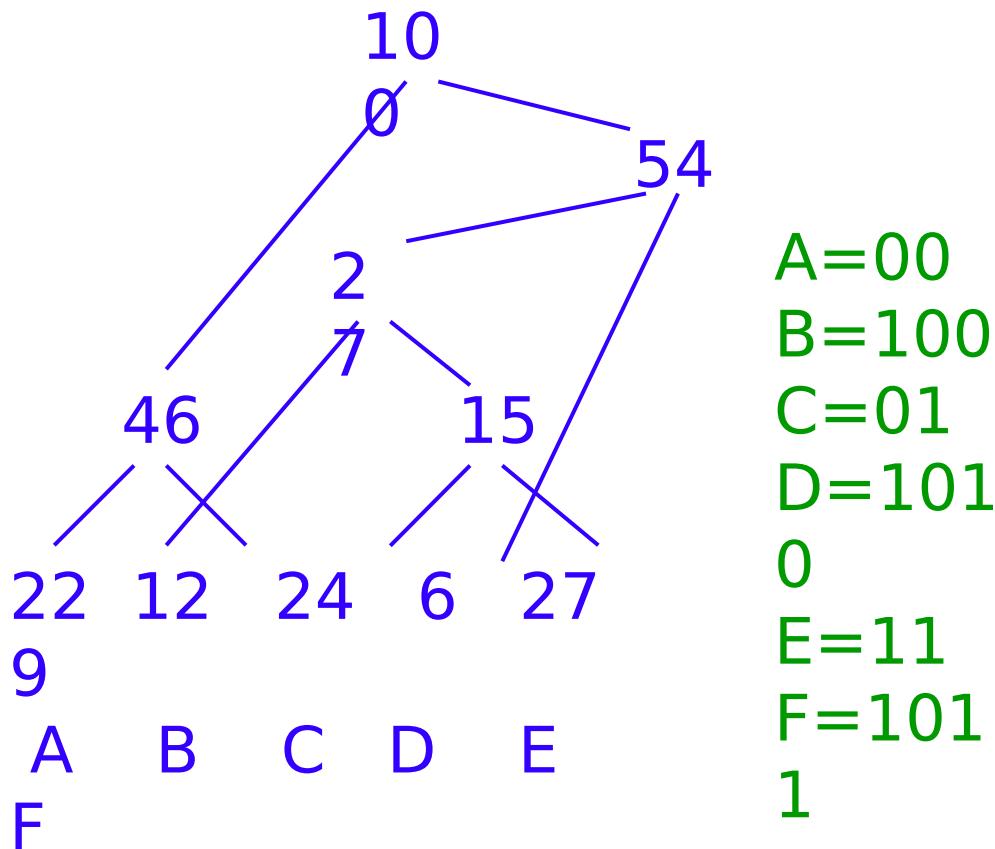
Character	a	b	d	e	f	h	i	k	n	o	r	s	t	u	v
Frequency	9	5	1	3	7	3	1	1	4	1	5	1	2	1	1





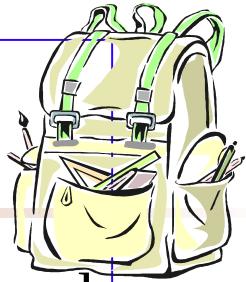
Example for Huffman Encoding

- The Huffman encoding algorithm is a greedy algorithm
- We always pick the two smallest numbers to combine



- Average bits/char:
 $0.22*2 + 0.12*3$
+
 $0.24*2 + 0.06*4$
+
 $0.27*2 + 0.09*4$
 $= 2.42$
- The Huffman algorithm finds an optimal solution

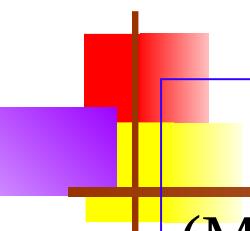
The Knapsack Problem:



Given n objects each have a *Weight* w_i and a *Value* v_i , and given a knapsack of total *Capacity* W . The problem is to pack the knapsack with these objects in order to maximize the total value of those objects packed without exceeding the knapsack's capacity. More formally, let x_i denote the fraction of the object i to be included in the knapsack, $0 \leq x_i \leq 1$, for $1 \leq i \leq n$. The problem is to find the values for the x_i such that $\sum_{i=1}^n x_i w_i \leq W$ and $\sum_{i=1}^n x_i v_i$ is maximized.

$$\sum_{i=1}^n x_i w_i \leq W \quad \text{and} \quad \sum_{i=1}^n x_i v_i \rightarrow \text{maximized}$$

Note that we may assume that $\sum_{i=1}^n w_i > W$ otherwise, we would choose $x_i = 1$ for each i which would be considered as an obvious optimal solution.



There seem to be 3 obvious greedy strategies:

(Max value) Sort the objects from the highest value to the lowest, then pick them in that order.

(Min weight) Sort the objects from the lowest weight to the highest, then pick them in that order.

(Max value/weight ratio) Sort the objects based on the value to weight ratios, from the highest to the lowest, then select.

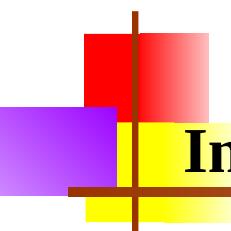
Example: Given $n = 5$ objects and a knapsack capacity $W = 100$ as shown in Table I. The three solutions are given in Table II.

v	20	30	66	40	60
w	10	20	30	40	50
v/w	2.0	1.5	2.2	1.0	
1.2					

Table I

select	x_i	value	$(\sum x_i v_i)$
Max v_i	0 0 1 0.5 1		146
Min w_i	1 1 1 1 0		156
Max v_i/w_i	1 1 1 0 0.8		164

Table II



The Optimal Knapsack Algorithm:

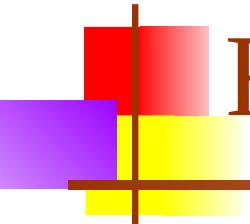
Input: An integer n , positive values w_i and v_i , for $1 \leq i \leq n$, and another positive value W .

Output: n values x_i such that $0 \leq x_i \leq 1$ and

$$\sum_{i=1}^n x_i w_i \leq W \text{ and } \sum_{i=1}^n x_i v_i \text{ is maximized.}$$

Algorithm (of time complexity $O(n \log n)$)

- (1) Sort ' n ' objects from large to small based on the ratios v_i/w_i . Assume the arrays $w[1..n]$ and $v[1..n]$. Store the weights and values after sorting.
- (2) initialize array $x[1..n]$ to zeros.
- (3) weight = 0; $i = 1$
- (4) while ($i \leq n$ and weight < W) do
 - (4.1) if weight + $w[i] \leq W$ then $x[i] = 1$
 - (4.2) else $x[i] = (W - \text{weight}) / w[i]$
 - (4.3) weight = weight + $x[i] * w[i]$
 - (4.4) $i++$



Fractional Knapsack Problem



- Given: A set S of n items, with each item i having
 - b_i - a positive benefit
 - w_i - a positive weight
- Goal: Choose items with maximum total benefit but with weight at most W .
- If we are allowed to take fractional amounts (broken items), then this is known as the **Fractional Knapsack Problem**.
 - In this case, we let x_i denote the amount we take of item i

- Objective: maximize $\sum_{i \in S} b_i(x_i / w_i)$

$$\sum_{i \in S} x_i \leq W$$

- Constraint:

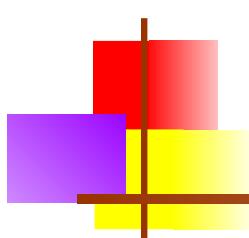
Example



- Given: A set S of n items, with each item i having
 - b_i - a positive benefit
 - w_i - a positive weight
- Goal: Choose items with maximum total benefit but with weight at most W .

Items:	1	2	3	4	5	“Knapsack” Solution:
Weight:	4 ml	8 ml	2 ml	6 ml	1 ml	<ul style="list-style-type: none">1 ml of 52 ml of 36 ml of 41 ml of 2
Benefit:	\$12	\$32	\$40	\$30	\$50	
Value: (\$ per ml)	3	4	20	5	50	

A large blue cylindrical container on the right is labeled "10 ml".



Fractional Knapsack Algorithm



- Greedy choice: Keep taking item with highest **value** (benefit to weight ratio)
 - Since $\sum_{i \in S} b_i(x_i / w_i) = \sum_{i \in S} (b_i / w_i)x_i$
 - Run time: $O(n \log n)$. Why?
- Correctness: Suppose there is a better solution
 - there is an item i with higher value than a chosen item j , but $x_i < w_i$, $x_j > 0$ and $v_i < v_j$
 - If we substitute some i with j , we get a better solution
 - How much of i : $\min\{w_i - x_i, x_j\}$
 - Thus, there is no better solution than the greedy one

Algorithm *fractionalKnapsack(S, W)*

Input: set S of items w/ benefit b_i and weight w_i ; max. weight W

Output: amount x_i of each item i to maximize benefit w/ weight at most W

for each item i in S

$x_i \leftarrow 0$

$v_i \leftarrow b_i / w_i$ {value}

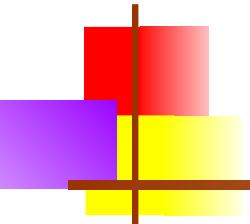
$w \leftarrow 0$ {total weight}

while $w < W$

remove item i w/ highest v_i

$x_i \leftarrow \min\{w_i, W - w\}$

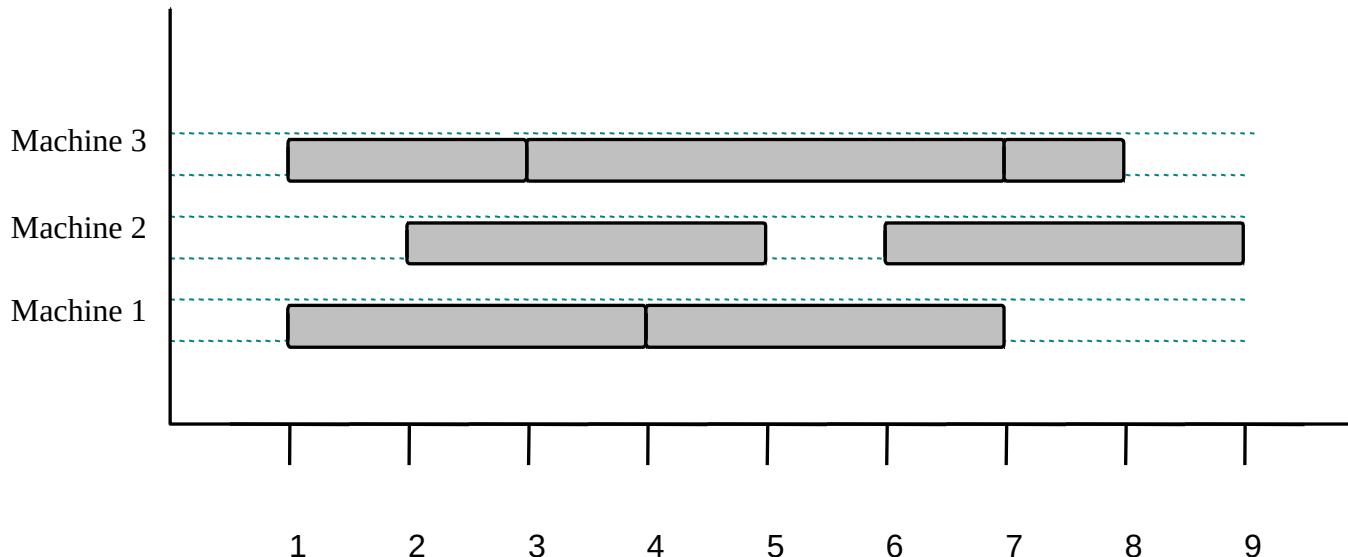
$w \leftarrow w + \min\{w_i, W - w\}$



Task Scheduling



- Given: a set T of n tasks, each having:
 - A start time, s_i
 - A finish time, f_i (where $s_i < f_i$)
- Goal: Perform all the tasks using a minimum number of “machines.”



Task Scheduling Algorithm

- Greedy choice: consider tasks by their start time and use as few machines as possible with this order.
 - Run time: $O(n \log n)$. Why?
- Correctness: Suppose there is a better schedule.
 - We can use $k-1$ machines
 - The algorithm uses k
 - Let i be first task scheduled on machine k
 - Machine i must conflict with $k-1$ other tasks
 - But that means there is no non-conflicting schedule using $k-1$ machines



Algorithm *taskSchedule(T)*

Input: set T of tasks w/ start time s_i and finish time f_i

Output: non-conflicting schedule with minimum number of machines

$m \leftarrow 0$ {no. of machines}

while T is not empty

remove task i w/ smallest s_i

if *there's a machine j for i* **then**
schedule i on machine j

else

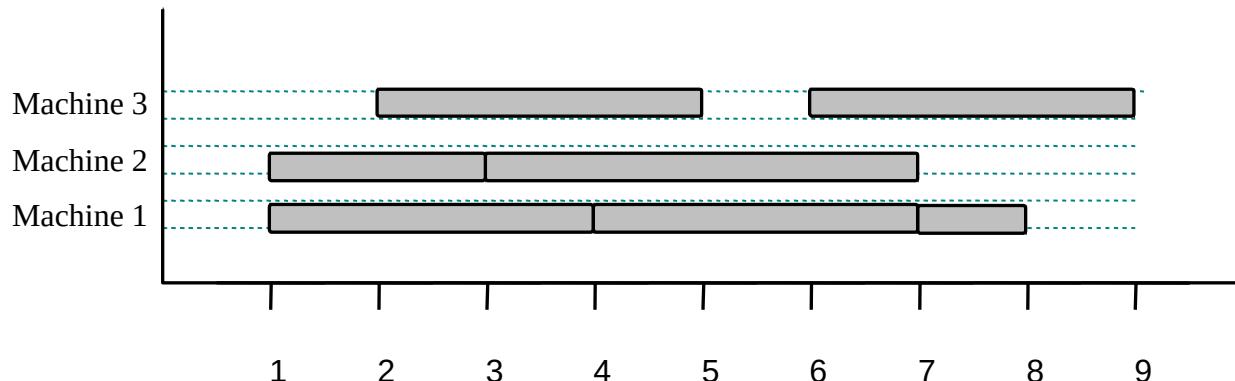
$m \leftarrow m + 1$

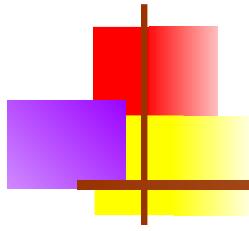
schedule i on machine m

Example for Task Scheduling



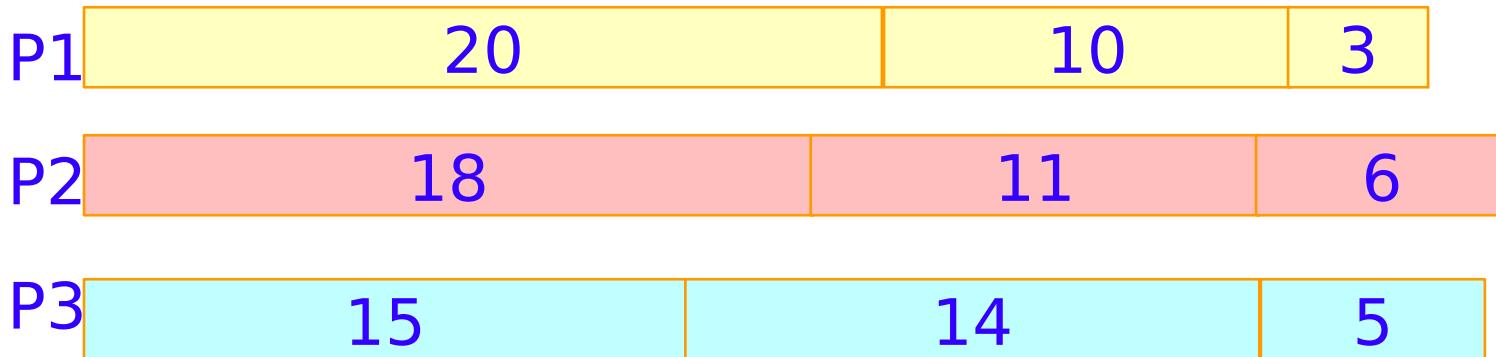
- Given: a set T of n tasks, each having:
 - A start time, s_i
 - A finish time, f_i (where $s_i < f_i$)
 - [1,4], [1,3], [2,5], [3,7], [4,7], [6,9], [7,8] (ordered by start)
- Goal: Perform all tasks on min. number of machines



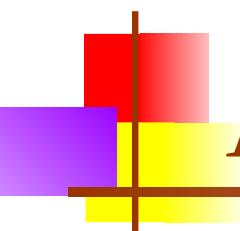


Job Scheduling Problem

- We have to run nine jobs, with running times of 3, 5, 6, 10, 11, 14, 15, 18, and 20 minutes
- We have three processors on which we can run these jobs
- We decide to do the longest-running jobs first, on whatever processor is available

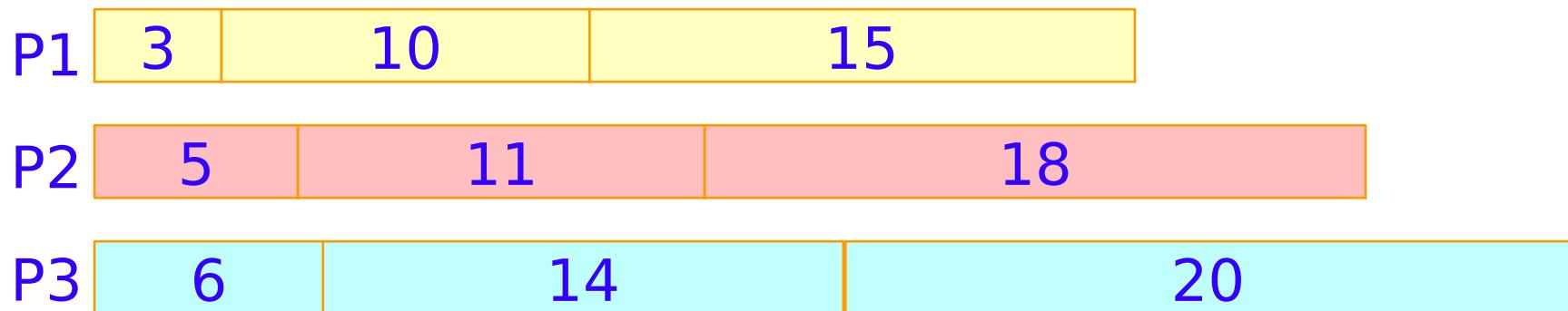


- Time for completion: $18 + 11 + 6 = 35$ minutes
- This solution isn't that bad, but we might be able to do better

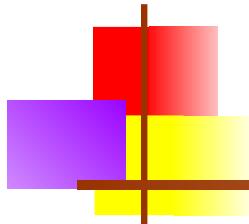


Another Approach for Job Scheduling

- What would be the result if we ran the *shortest* job first?
- Again, the running times are 3, 5, 6, 10, 11, 14, 15, 18, and 20 minutes

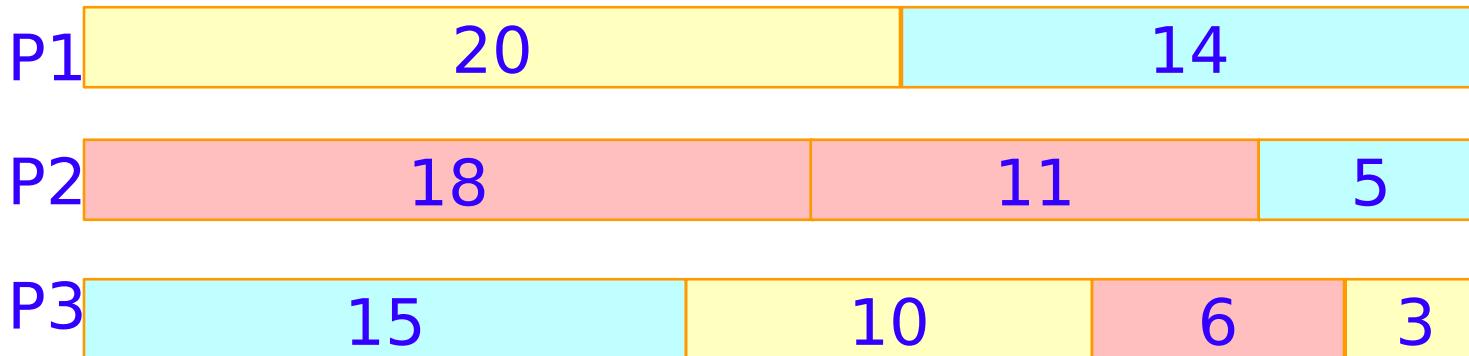


- That wasn't such a good idea; time to completion is now $6 + 14 + 20 = 40$ minutes
- Note, however, that the greedy algorithm itself is fast
 - All we had to do at each stage was pick the minimum or maximum

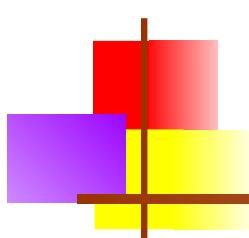


An Optimum Solution

- Better Solutions do exist:



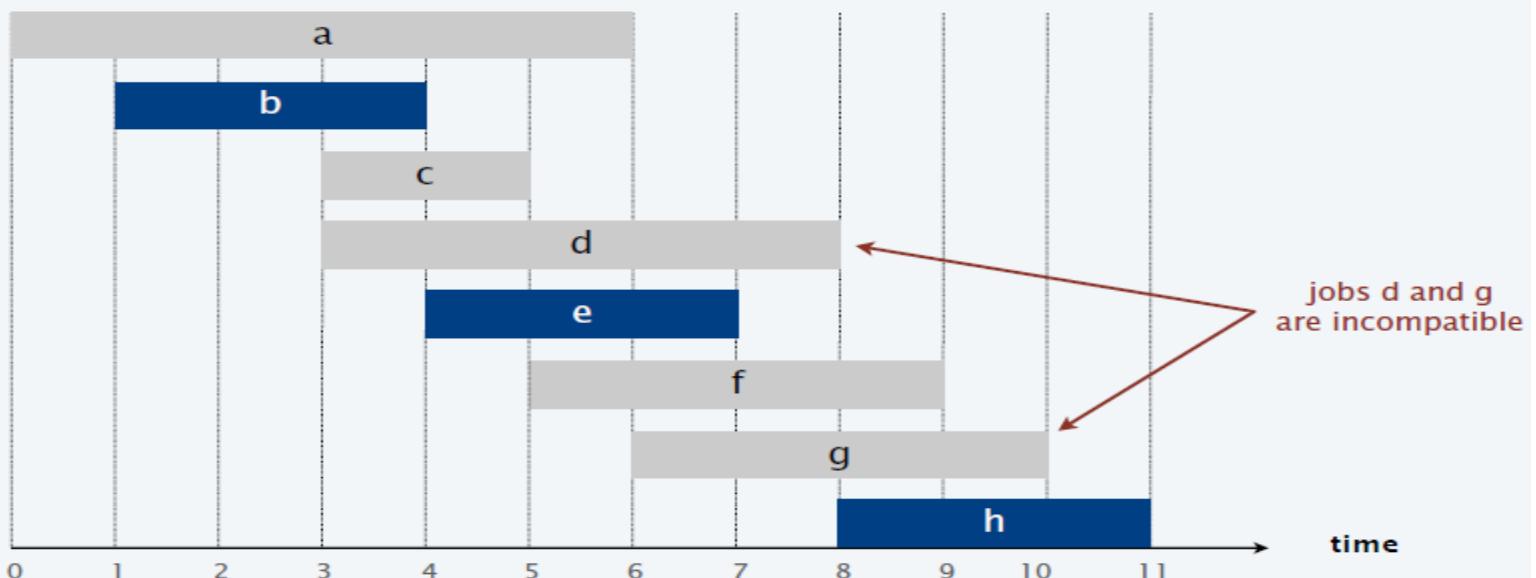
- This solution is clearly optimal (why?)
- Clearly, there are other optimal solutions (why?)
- How do we find such a solution?
 - One way: Try all possible assignments of jobs to processors
 - Unfortunately, this approach can take exponential time

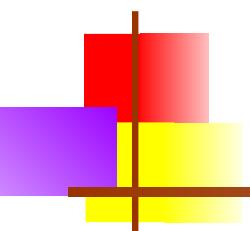


Interval Scheduling

Interval scheduling

- Job j starts at s_j and finishes at f_j .
- Two jobs are **compatible** if they don't overlap.
- Goal: find maximum subset of mutually compatible jobs.





Interval Scheduling

Interval scheduling: earliest-finish-time-first algorithm

EARLIEST-FINISH-TIME-FIRST ($n, s_1, s_2, \dots, s_n, f_1, f_2, \dots, f_n$)

SORT jobs by finish times and renumber so that $f_1 \leq f_2 \leq \dots \leq f_n$.

$S \leftarrow \emptyset$. ← set of jobs selected

FOR $j = 1$ TO n

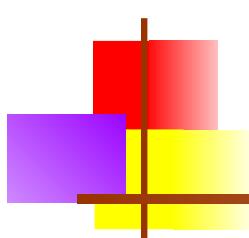
IF (job j is compatible with S)

$S \leftarrow S \cup \{ j \}$.

RETURN S .

Proposition. Can implement earliest-finish-time first in $O(n \log n)$ time.

- Keep track of job j^* that was added last to S .
- Job j is compatible with S iff $s_j \geq f_{j^*}$.
- Sorting by finish times takes $O(n \log n)$ time.



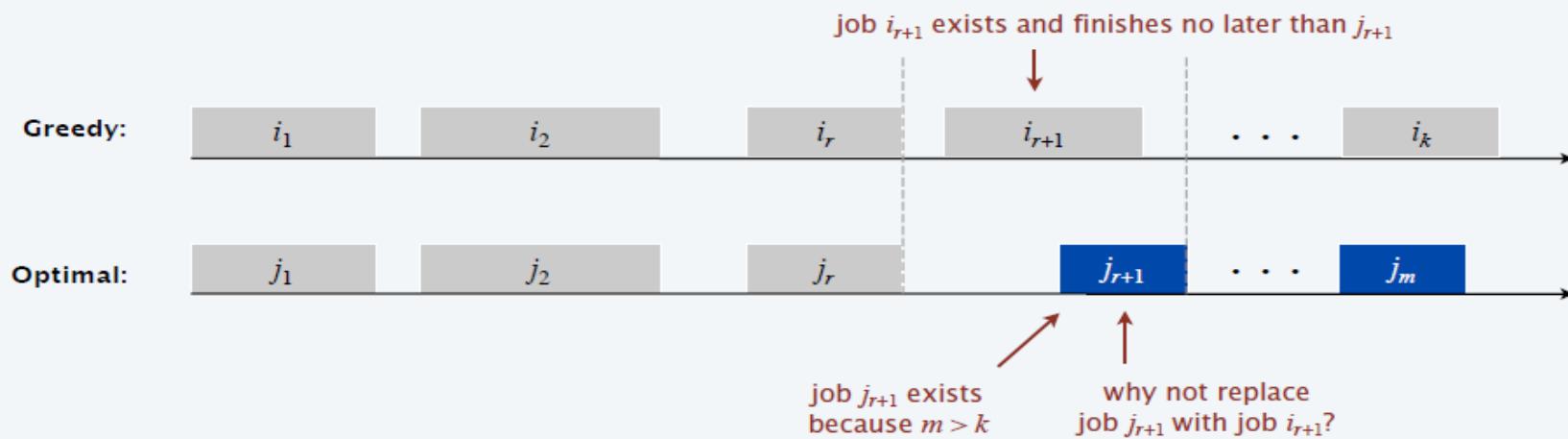
Interval Scheduling

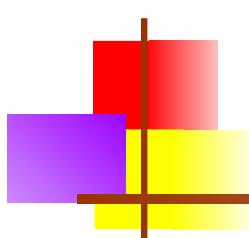
Interval scheduling: analysis of earliest-finish-time-first algorithm

Theorem. The earliest-finish-time-first algorithm is optimal.

Pf. [by contradiction]

- Assume greedy is not optimal, and let's see what happens.
- Let i_1, i_2, \dots, i_k denote set of jobs selected by greedy.
- Let j_1, j_2, \dots, j_m denote set of jobs in an optimal solution with $i_1 = j_1, i_2 = j_2, \dots, i_r = j_r$ for the largest possible value of r .





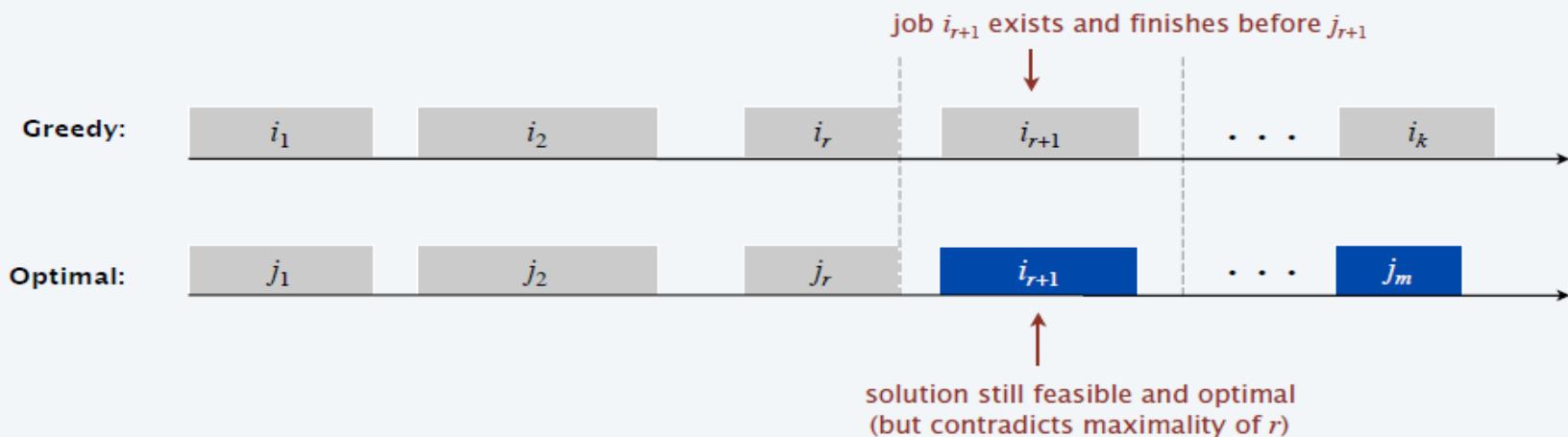
Interval Scheduling

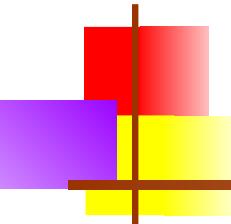
Interval scheduling: analysis of earliest-finish-time-first algorithm

Theorem. The earliest-finish-time-first algorithm is optimal.

Pf. [by contradiction]

- Assume greedy is not optimal, and let's see what happens.
- Let i_1, i_2, \dots, i_k denote set of jobs selected by greedy.
- Let j_1, j_2, \dots, j_m denote set of jobs in an optimal solution with $i_1 = j_1, i_2 = j_2, \dots, i_r = j_r$ for the largest possible value of r .



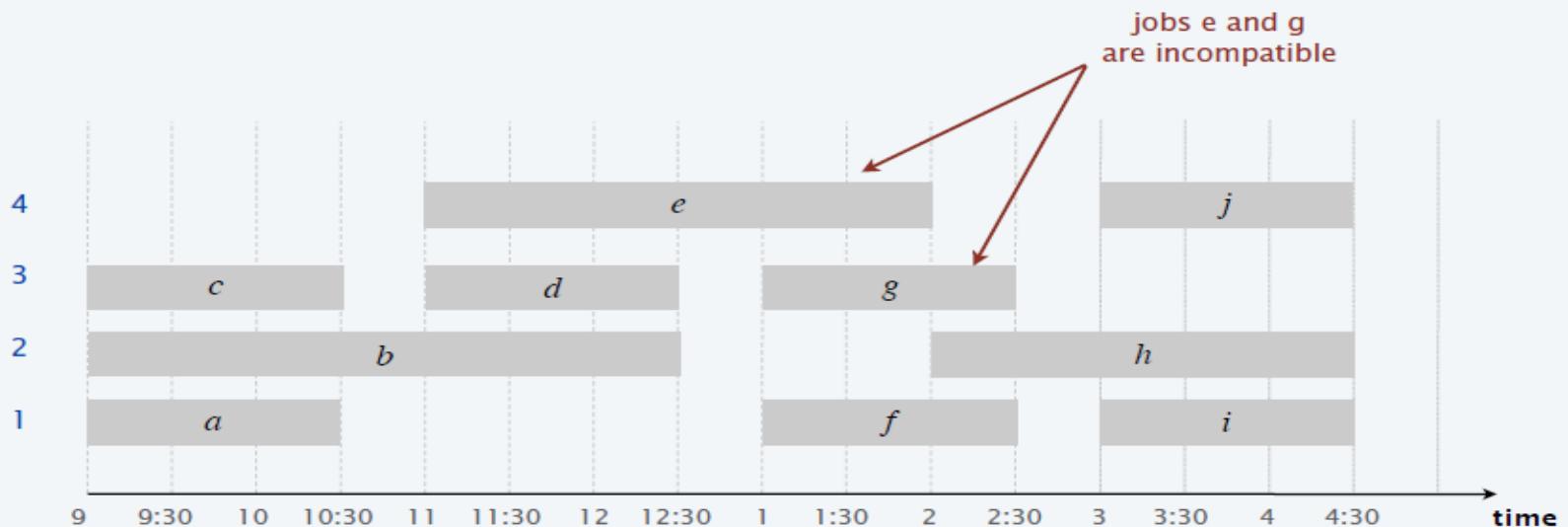


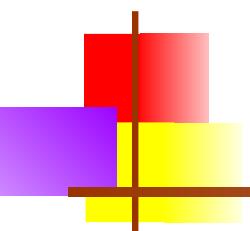
Interval Partitioning

Interval partitioning

- Lecture j starts at s_j and finishes at f_j .
- Goal: find minimum number of classrooms to schedule all lectures so that no two lectures occur at the same time in the same room.

Ex. This schedule uses 4 classrooms to schedule 10 lectures.



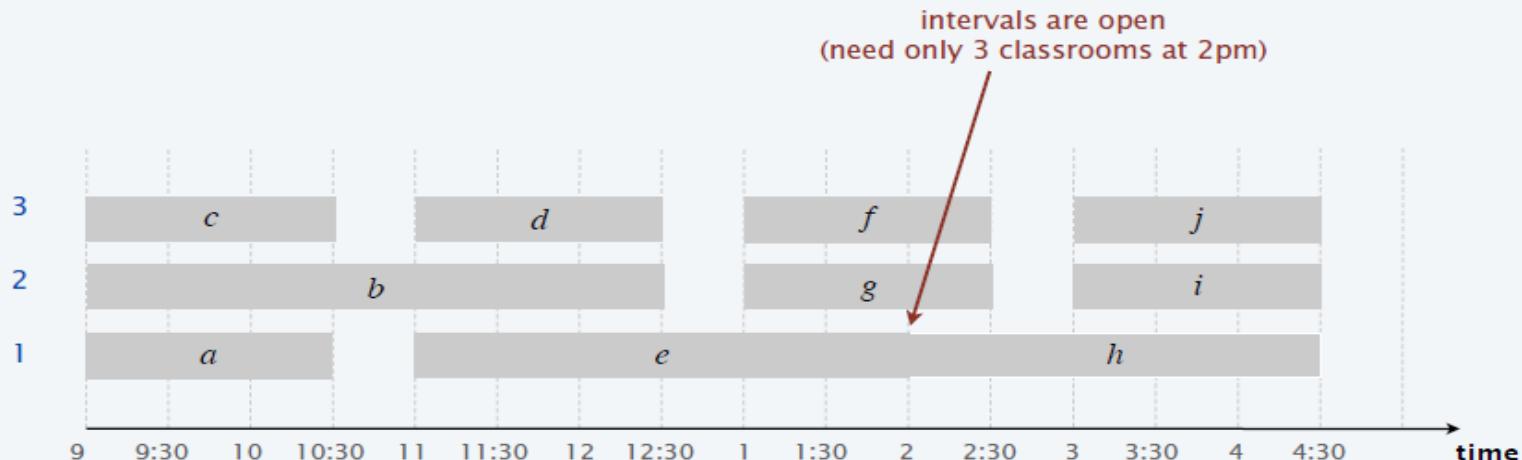


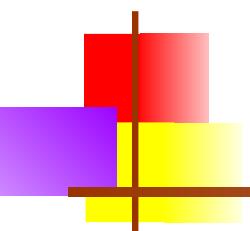
Interval Partitioning

Interval partitioning

- Lecture j starts at s_j and finishes at f_j .
- Goal: find minimum number of classrooms to schedule all lectures so that no two lectures occur at the same time in the same room.

Ex. This schedule uses 3 classrooms to schedule 10 lectures.





Interval Partitioning

Interval partitioning: earliest-start-time-first algorithm

EARLIEST-START-TIME-FIRST ($n, s_1, s_2, \dots, s_n, f_1, f_2, \dots, f_n$)

SORT lectures by start times and renumber so that $s_1 \leq s_2 \leq \dots \leq s_n$.

$d \leftarrow 0$. ← number of allocated classrooms

FOR $j = 1$ TO n

 IF (lecture j is compatible with some classroom)

 Schedule lecture j in any such classroom k .

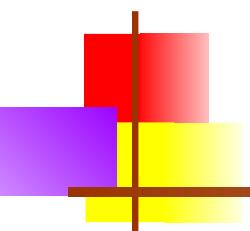
 ELSE

 Allocate a new classroom $d + 1$.

 Schedule lecture j in classroom $d + 1$.

$d \leftarrow d + 1$.

RETURN schedule.



Interval Partitioning

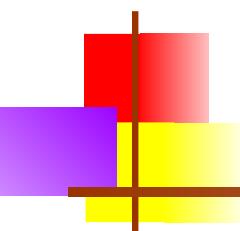
Interval partitioning: earliest-start-time-first algorithm

Proposition. The earliest-start-time-first algorithm can be implemented in $O(n \log n)$ time.

Pf.

- Sorting by start times takes $O(n \log n)$ time.
- Store classrooms in a **priority queue** (key = finish time of its last lecture).
 - to allocate a new classroom, **INSERT** classroom onto priority queue.
 - to schedule lecture j in classroom k , **INCREASE-KEY** of classroom k to f_j .
 - to determine whether lecture j is compatible with any classroom, compare s_j to **FIND-MIN**
- Total # of priority queue operations is $O(n)$; each takes $O(\log n)$ time. ▀

Remark. This implementation chooses a classroom k whose finish time of its last lecture is the **earliest**.



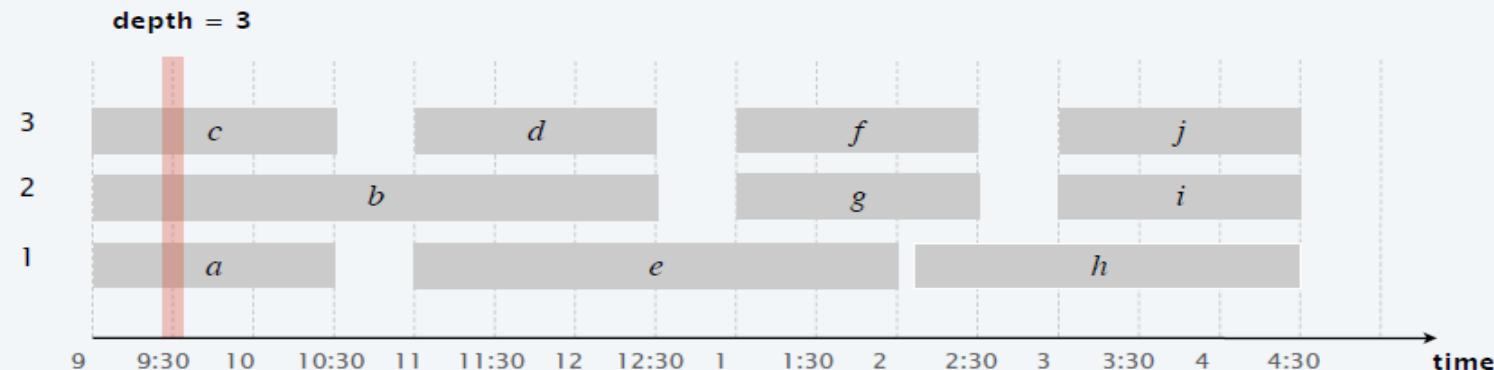
Interval Partitioning

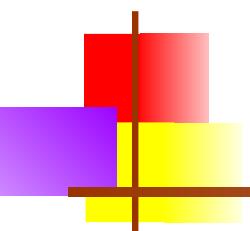
Interval partitioning: lower bound on optimal solution

Def. The **depth** of a set of open intervals is the maximum number of intervals that contain any given point.

Key observation. Number of classrooms needed \geq depth.

- Q.** Does minimum number of classrooms needed always equal depth?
- A.** Yes! Moreover, earliest-start-time-first algorithm finds a schedule whose number of classrooms equals the depth.





Interval Partitioning

Interval partitioning: analysis of earliest-start-time-first algorithm

Observation. The earliest-start-time first algorithm never schedules two incompatible lectures in the same classroom.

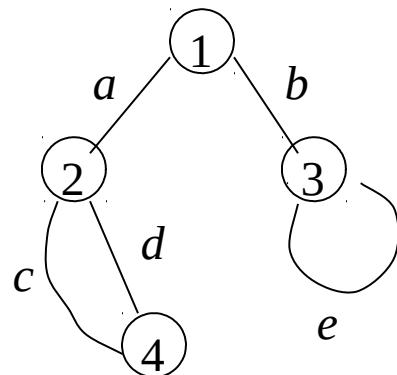
Theorem. Earliest-start-time-first algorithm is optimal.

Pf.

- Let d = number of classrooms that the algorithm allocates.
- Classroom d is opened because we needed to schedule a lecture, say j , that is incompatible with a lecture in each of $d - 1$ other classrooms.
- Thus, these d lectures each end after s_j .
- Since we sorted by start time, each of these incompatible lectures start no later than s_j .
- Thus, we have d lectures overlapping at time $s_j + \varepsilon$.
- Key observation \Rightarrow all schedules use $\geq d$ classrooms. ▀

Greedy Strategies Applied to Graph Problems:

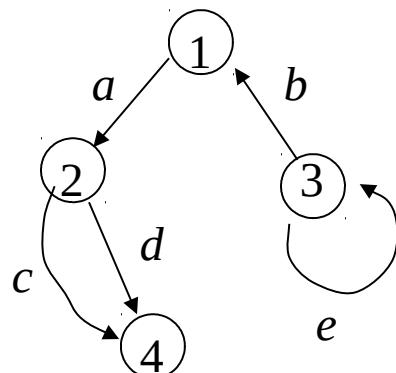
We first review some notations and terms about graphs. A graph consists of vertices (nodes) and edges (arcs, links), in which each edge “connects” two vertices (not necessarily distinct). More formally, a graph $G = (V, E)$, where V and E denote the sets of vertices and edges, respectively.



In this example, $V = \{1, 2, 3, 4\}$, $E = \{a, b, c, d, e\}$. Edges c and d are parallel edges; edge e is a self-loop. A path is a sequence of “adjacent” edges, e.g., path $abeb$, path $acdab$.

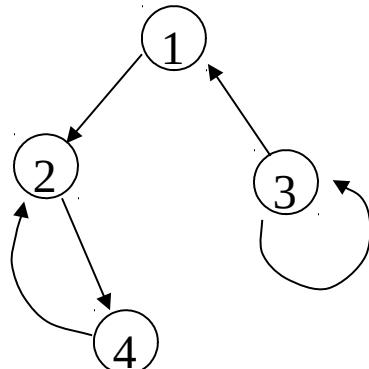
Directed Graphs vs. Un-directed Graphs:

If every edge has an orientation, e.g., an edge starting from node x terminating at node y , the graph is called a directed graph, or digraph for short. If all edges have no orientation, the graph is called an undirected graph, or simply, a graph. When there are no parallel edges (two edges that have identical end points), we could identify an edge with its two end points, such as edge $(1,2)$, or edge $(3,3)$. In an undirected graph, edge $(1,2)$ is the same as edge $(2,1)$. We will assume no parallel edges unless otherwise stated.



A directed graph. Edges c and d are parallel (directed) edges. Some directed paths are ad , $ebac$.

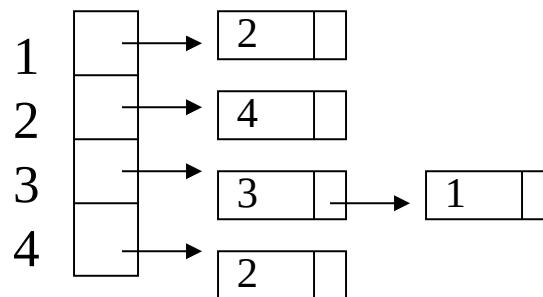
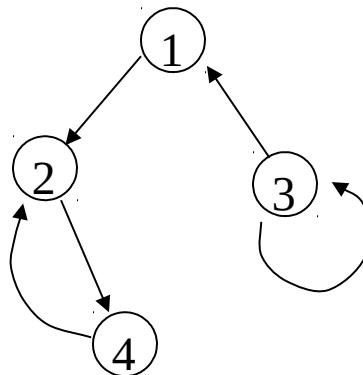
Both directed and undirected graphs appear often and naturally in many scientific (call graphs in program analysis), business (query trees, entity-relation diagrams in databases), and engineering (CAD design) applications. The simplest data structure for representing graphs and digraphs is using 2-dimensional arrays. Suppose $G = (V, E)$, and $|V| = n$. Declare an array $T[1..n][1..n]$ so that $T[i][j] = 1$ if there is an edge $(i, j) \in E$; 0 otherwise. (Note that in an undirected graph, edges (i, j) and (j, i) refer to the same edge.)



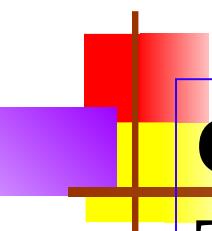
	j			
i	1	2	3	4
1	0	1	0	0
2	0	0	0	1
3	1	0	1	0
4	0	1	0	0

A 2-dimensional array for the digraph, called the *adjacency matrix*.

Sometimes, edges of a graph or digraph are given a positive *weight* or *cost* value. In that case, the adjacency matrix can easily modified so that $T[i][j] =$ the weight of edge (i, j) ; 0 if there is no edge (i, j) . Since the adjacency matrix may contain many zeros (when the graph has few edges, known as *sparse*), a space-efficient representation uses linked lists representing the edges, known as the *adjacency list* representation.



The adjacency lists for the digraph, which can store edge weights by adding another field in the list nodes.



Graph (and Digraph) Traversal Techniques:

Given a (directed) graph $G = (V, E)$, determine all nodes that are connected from a given node v via a (directed) path.

There are essentially two graph traversal algorithms, known as *Breadth-first search* (BFS) and *depth-first search* (DFS), both of which can be implemented efficiently.

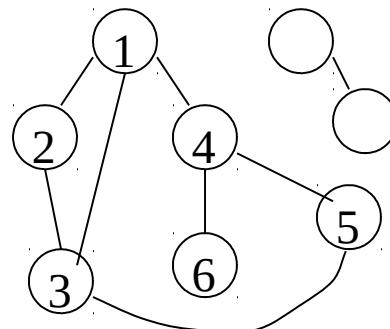
BFS: From node v , visit each of its neighboring nodes in sequence, then visit their neighbors, etc., while avoiding repeated visits.

DFS: From node v , visit its first neighboring node and all its neighbors using recursion, then visit node v 's second neighbor applying the same procedure, until all v 's neighbors are visited, while avoiding repeated visits.

Breadth-First Search (BFS):

BFS(v) // visit all nodes reachable from node v
(1) Create an empty FIFO queue Q , add node v to Q
(2) Create a Boolean array $\text{visited}[1..n]$, initialize all values
to false except for $\text{visited}[v]$ to true (3)
while Q is not empty
node w from Q (3.1) delete a
adjacent from node w (3.2) for each node z
then if $\text{visited}[z]$ is false
to true add node z to Q and set $\text{visited}[z]$

The time complexity is $O(n+e)$ with n nodes and e edges, if the adjacency lists are used. This is because in the worst case, each node is added once to the queue ($O(n)$ part), and each of its neighbors gets considered once ($O(e)$ part).



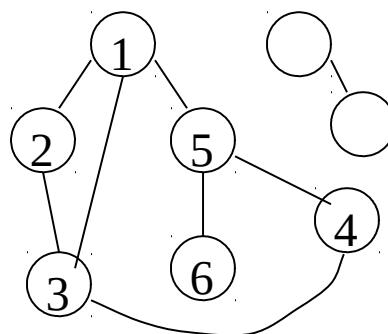
Node search order starting with node 1, including two nodes not reached

Depth-First Search (DFS):

(1) Create a Boolean array $\text{visited}[1..n]$, initialize all values to false except for $\text{visited}[v]$ to true
(2) Call $\text{DFS}(v)$ to visit all nodes reachable via a path

```
DFS(v)
neighboring nodes w of v do
false then
DFS(w) // recursive call
for each
if visited[w] is
set visited[w] to true; call
```

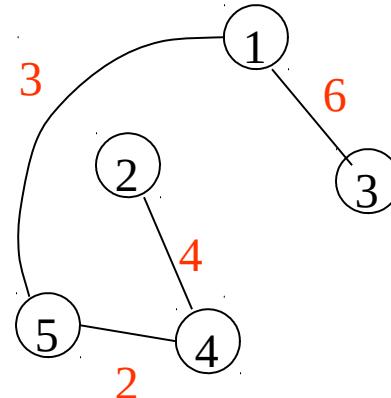
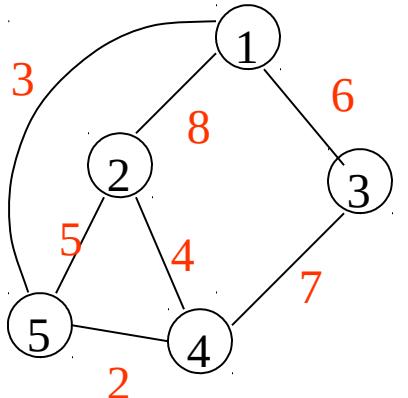
The algorithm's time complexity is also $O(n+e)$ using the same reasoning as in the BFS algorithm.



Node search order starting with node 1, including two nodes not reached

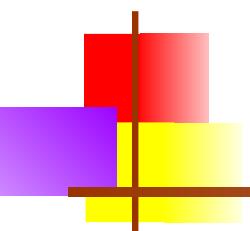
The Minimum Spanning Tree (MST) Problem:

Given a weighted (undirected) graph $G = (V, E)$, where each edge e has a positive weight $w(e)$. A *spanning tree* of G is a *tree* (connected graph without cycles, or circuits) which has V as its vertex set, i.e., the tree connects all vertices of the graph G . If $|V| = n$, then the tree has $n - 1$ edges (this is a fact which can be proved by induction). A *minimum spanning tree* of G is a spanning tree that has the minimum total edge weight.



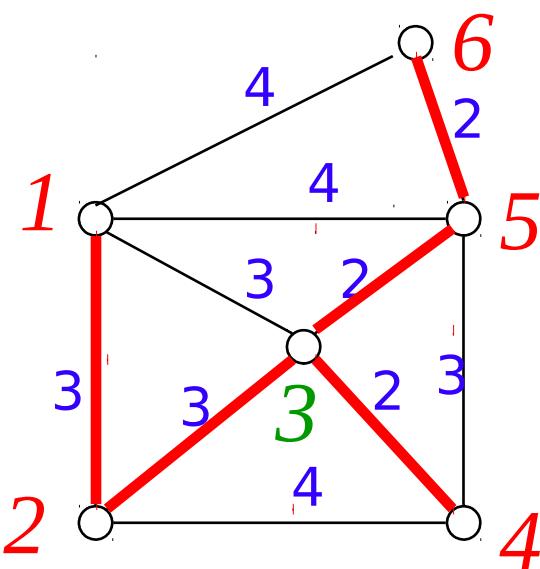
A minimum spanning tree (of 4 edges), weight = $3 + 2 + 4 + 6 = 15$.

A weighted graph of no parallel edges or self-loops

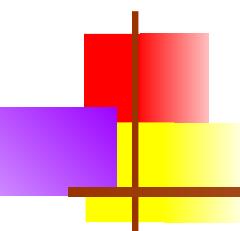


Minimum Spanning Tree (MST)

- A minimum spanning tree is a least-cost subset of the edges of a graph that connects all the nodes
 - Start by picking any node and adding it to the tree
 - Repeatedly: Pick any *least-cost* edge from a node in the tree to a node not in the tree, and add the edge and new node to the tree
 - Stop when all nodes have been added to the tree

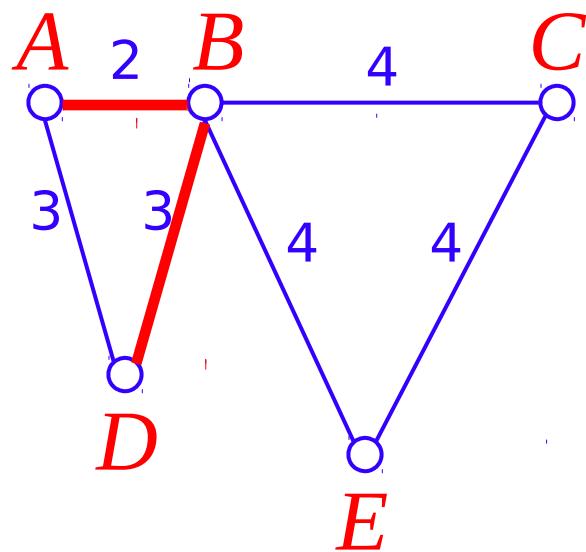


- The result is a least-cost ($3+3+2+2+2=12$) spanning tree
- If we think some other edge should be in the spanning tree:
 - Try adding that edge
 - Note that the edge is part of a cycle
 - To break the cycle, we must remove the edge with the greatest cost
 - This will be the edge we just

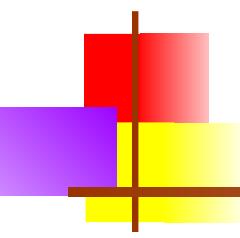


Traveling Salesperson Problem (TSP)

- A salesperson must visit every city (starting from city **A**), and wants to cover the least possible distance
 - He(she) can revisit a city (and reuse a road) if necessary
- He(she) does this by using a greedy algorithm: He(she) goes to the next nearest city from wherever he(she) is

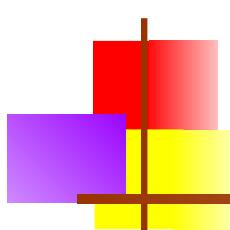


- From **A** he(she) goes to **B**
- From **B** he(she) goes to **D**
- This is *not* going to result in a shortest path!
- The best result he(she) can get now will be **ABDBCE**, at a cost of **16**
- An actual least-cost path from **A** is **ADBCE**, at a cost of **14**



Time Complexity Analysis

- A Greedy Algorithm typically makes (approximately) n choices for a problem of size n
 - (The first or last choice may be forced)
- Hence the expected running time is:
 $O(n * O(\text{choice}(n)))$, where $\text{choice}(n)$ is making a choice among n objects
 - Counting: Must find largest useable coin from among k sizes of coin (k is a constant), an $O(k)=O(1)$ operation;
 - Therefore, coin counting is (n)
 - Huffman: Must sort n values before making n choices
 - Therefore, Huffman is $O(n \log n) + O(n) = O(n \log n)$
 - Minimum spanning tree: At each new node, must include new edges and keep them sorted, which is $O(n \log n)$ overall
 - Therefore, MST is $O(n \log n) + O(n) = O(n \log n)$



Greedy Analysis Strategies

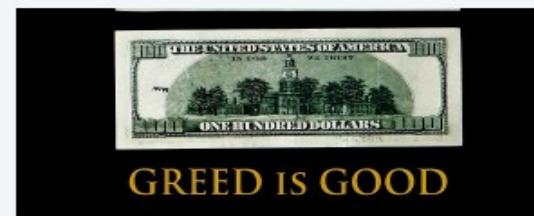
Greedy analysis strategies

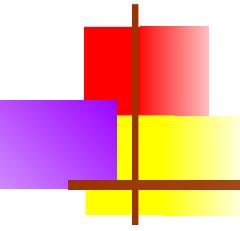
Greedy algorithm stays ahead. Show that after each step of the greedy algorithm, its solution is at least as good as any other algorithm's.

Structural. Discover a simple “structural” bound asserting that every possible solution must have a certain value. Then show that your algorithm always achieves this bound.

Exchange argument. Gradually transform any solution to the one found by the greedy algorithm without hurting its quality.

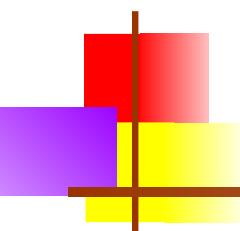
Other greedy algorithms. Gale–Shapley, Kruskal, Prim, Dijkstra, Huffman, ...





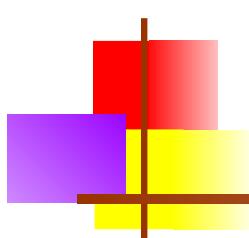
Other Greedy Algorithms

- Dijkstra's Algo for finding the shortest path in a graph
 - Always takes the *shortest* edge (path) connecting a known node to an unknown node
- Kruskal's Algo for finding a minimum-cost spanning tree
 - Always tries the *lowest-cost* remaining edge
- Prim's Algo for finding a minimum-cost spanning tree
 - Always takes the *lowest-cost* edge between nodes in the spanning tree and nodes not yet in the spanning tree



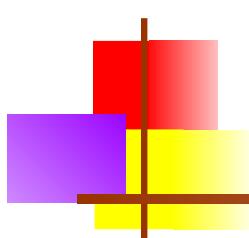
Dijkstra's Shortest-path Algorithm

- Dijkstra's algorithm finds the shortest paths from a given node to all other nodes in a graph
 - Initially,
 - Mark the given node as *known* (path length is zero)
 - For each out-edge, set the distance in each neighboring node equal to the *cost* (length) of the out-edge, and set its *predecessor* to the initially given node
 - Repeatedly (until all nodes are known),
 - Find an unknown node containing the smallest distance
 - Mark the new node as known
 - For each node adjacent to the new node, examine its neighbors to see whether the estimated distance can be reduced (distance to known node + cost of out-edge)
 - If so, also reset the predecessor of the new node



Analysis of Dijkstra's Algorithm I

- Assume that the *average* out-degree of a node is some constant k
 - Initially,
 - Mark the given node as *known* (path length is zero)
 - This takes $O(1)$ (constant) time
 - For each out-edge, set the distance in each neighboring node equal to the *cost* (length) of the out-edge, and set its *predecessor* to the initially given node
 - If each node refers to a list of k adjacent node/edge pairs, this takes $O(k) = O(1)$ time, that is, constant time
 - Notice that this operation takes *longer* if we have to extract a list of names from a hash table



Analysis of Dijkstra's Algorithm II

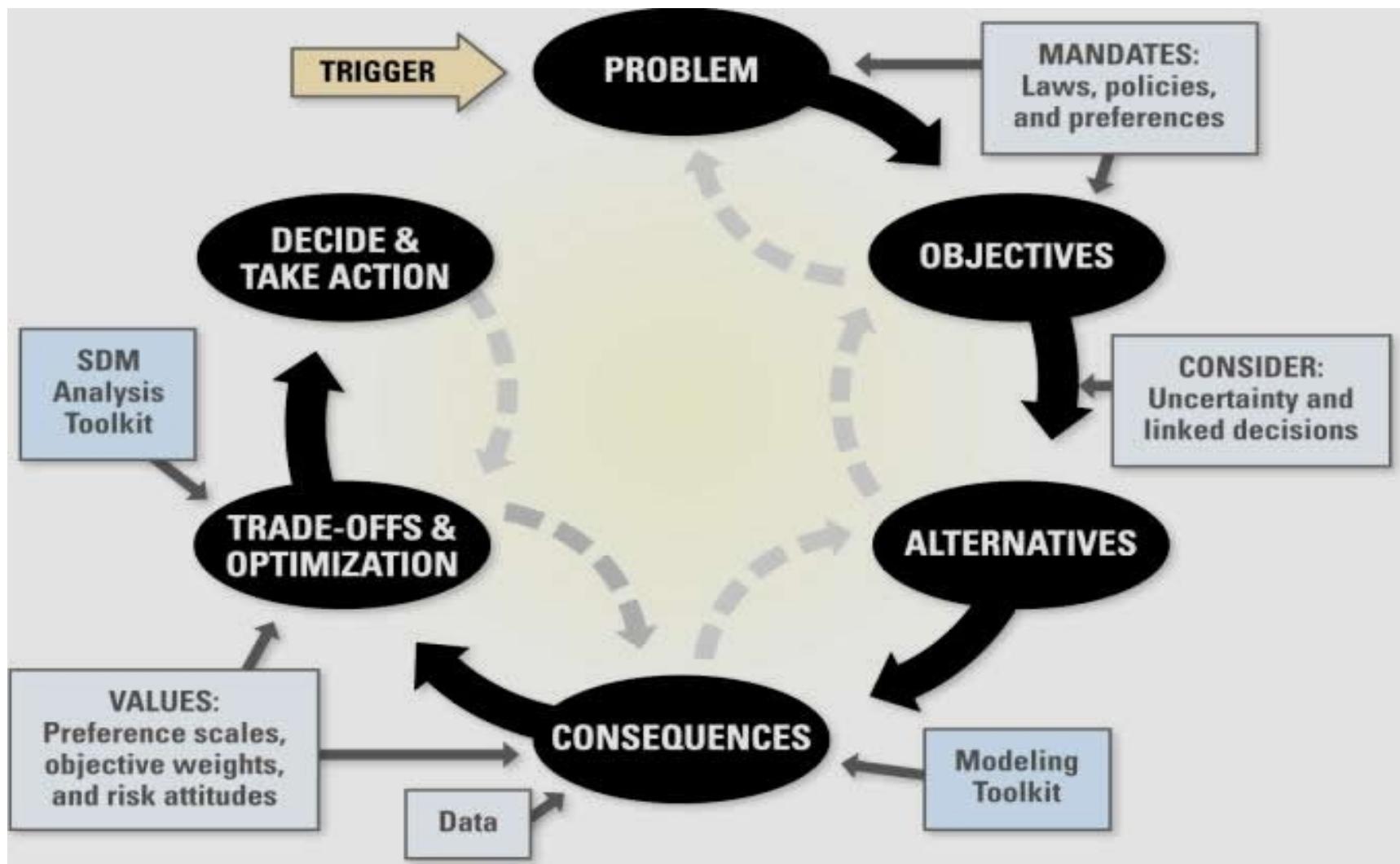
- Repeatedly (until all nodes are known), (n times)
 - Find an unknown node containing the smallest distance
 - Probably the best way to do this is to put the unknown nodes into a priority queue; this takes $k * O(\log n)$ time each time a new node is marked “known” (and this happens n times)
 - Mark the new node as known -- $O(1)$ time
 - For each node adjacent to the new node, examine its neighbors to see whether their estimated distance can be reduced (distance to known node plus cost of out-edge)
 - If so, also reset the predecessor of the new node
 - There are k adjacent nodes (on average), operation requires constant time at each, therefore $O(k)$ (constant) time
 - Combining all the parts, we get:
 $O(1) + n*(k*O(\log n)+O(k))$, that is, $O(nk \log n)$ time



Topics in Computational Sustainability: Introduction to Linear Programming



Core Elements of Strategic Decision Making (SDM)



Inputs



Computational Problems

Problem

Outputs

Decision Problems

(e.g., Yes or No decision: Can it be done and how?)

or

Optimization Problems

(e.g., What's the optimal solution and how?)

Decision Problem Graph Coloring

Can we color a map with 4 colors?



Decision Problem Set Covering

Can we protect the forest with 10 forest rangers?



Optimization Problem Linear Programming

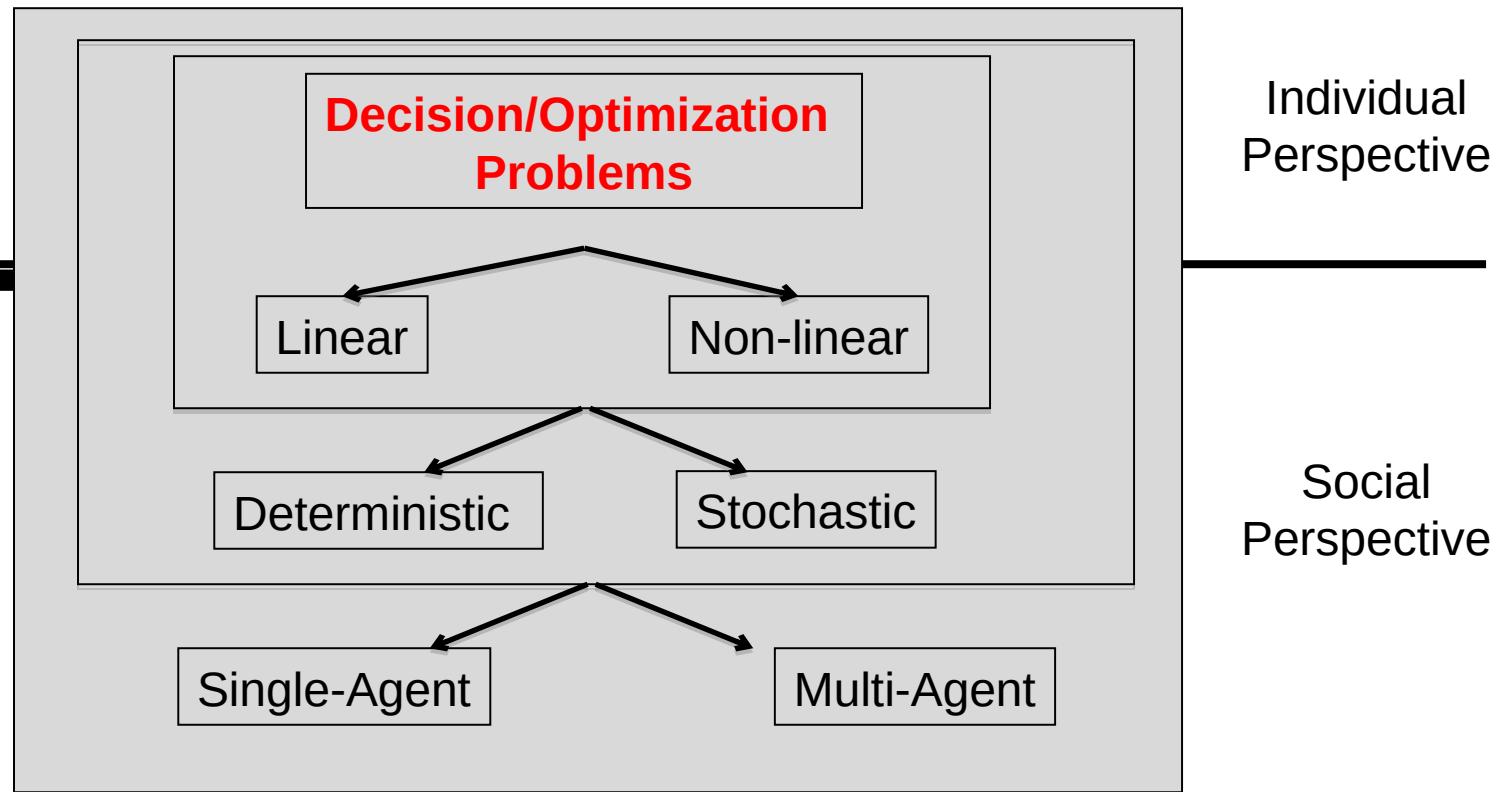
How to minimize Pulp mill pollution?



Optimization Problem Knapsack

What Renewable Energy Investments should the Green Power Company pick to maximize profit?





Solution Methods

Exact or Complete

Linear/Mixed/ Integer/Quadratic Programming	Network Flow (e.g.. Shortest path)	Approximation Algorithms Simulated annealing Local Search Genetic algorithms	Markov Chain Monte Carlo Methods Agent Based Simulation Computable General Equilibrium
Dynamic Programming	Branch and Bound	Sampling Based Methods	Meta-heuristic Methods
Stochastic Programming	Others...	Particle Swarm Optimization	Others...

Incomplete, Sampling, Simulation

Optimization: Mathematical Program

- Optimization Problem in which the objective and constraints are given as mathematical functions and functional relationships.

Minimize $f(x_1, x_2, \dots, x_n)$

Subject to:

$$g_1(x_1, x_2, \dots, x_n) = , \geq, \leq b_1$$

$$g_2(x_1, x_2, \dots, x_n) = , \geq, \leq b_2$$

...

$$g_m(x_1, x_2, \dots, x_n) = , \geq, \leq b_m$$

Linear Programming

Linear Programming (LP)

- Linear – All the functions are linear

Ex: $f(x_1, x_2, \dots, x_n) = c_1x_1 + c_2x_2 + \dots c_nx_n$

- Programming – does not refer to computer programming but rather “planning” - planning of activities to obtain an optimal result i.e., it reaches the specified goal (best solution) (according to the mathematical model) among all feasible alternatives.

Components of a Linear Programming Model

- A Linear Programming (LP) Model Consists of:
 - A Set of Decision Variables
 - A (Linear) Objective Function
 - A Set of (Linear) Constraints

- Linear Programming (LP) Problems

Both objective function and constraints are linear.
Solutions are highly structured and can be rapidly obtained.

Linear Programming (LP)

- LP has gained widespread industrial acceptance since the 1950s for on-line optimization, blending etc.
- Linear Constraints can arise due to:
 1. Production Limitation e.g. Equipment Limitations, Storage Limits, Market Constraints.
 2. Raw Material Limitation
 3. Safety Restrictions, e.g. Allowable Operating Ranges for
Temperature and Pressures.
 4. Physical Property Specifications e.g. Product Quality Constraints when a blend property can be calculated as an average of pure component properties:
$$P = \sum_{i=1}^n y_i P_i \leq \alpha$$
 5. Material and Energy Balances
 - Tend to yield equality constraints.
 - Constraints can change frequently (daily or hourly).

• Effect of Inequality Constraints

- Consider the Linear and Quadratic Objective Functions on the next slide.
-

- Note that for the LP problem, the optimum must lie on one or more constraints.

• Generic Statement of the LP Problem:

$$\max f = \sum_{i=1}^n c_i x_i$$

subject to (s. t.):

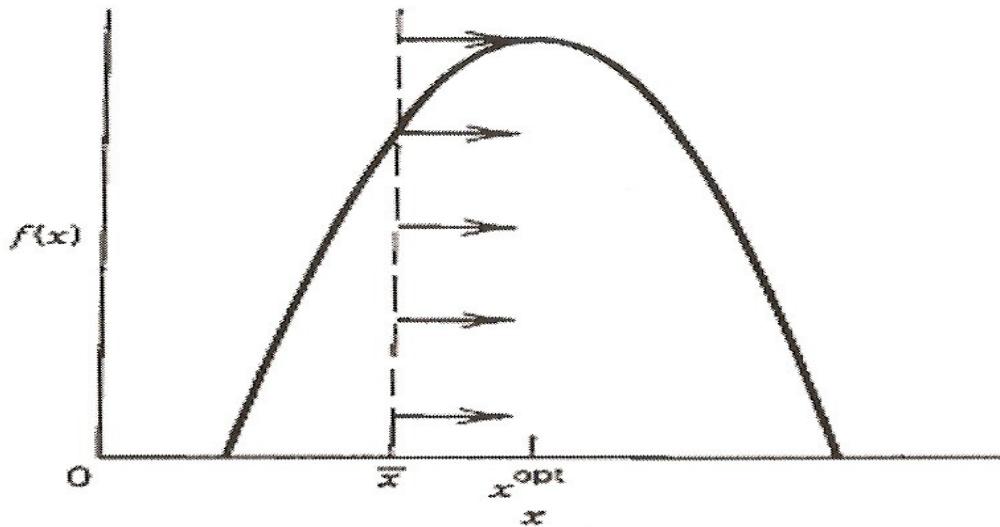
$$x_i \geq 0 \quad i = 1, 2, \dots, n$$

$$\sum_{j=1}^n a_{ij} x_j \leq b_i \quad i = 1, 2, \dots, n$$

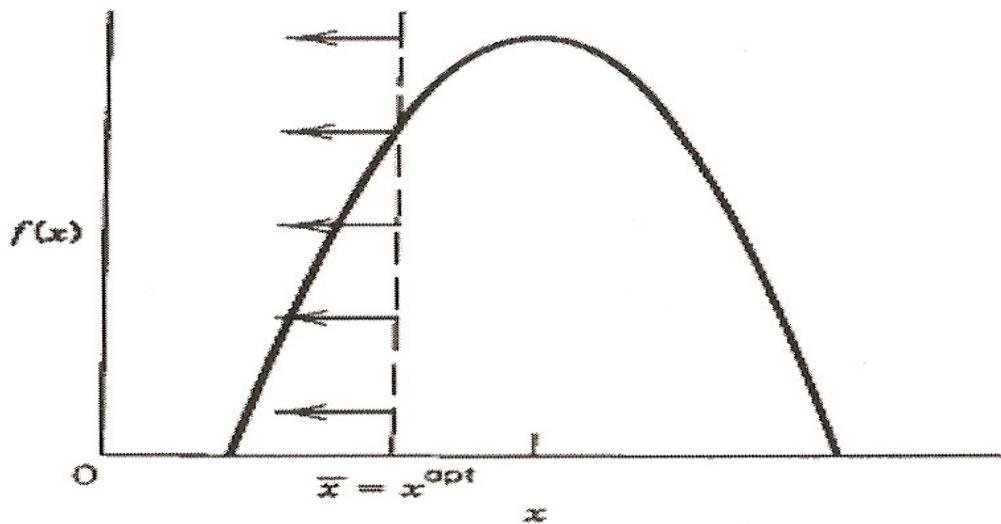
• Solution of LP Problems

- Simplex Method (Dantzig, 1947)
- Examine only constraint boundaries
- Very efficient, even for large problems

Linear Programming



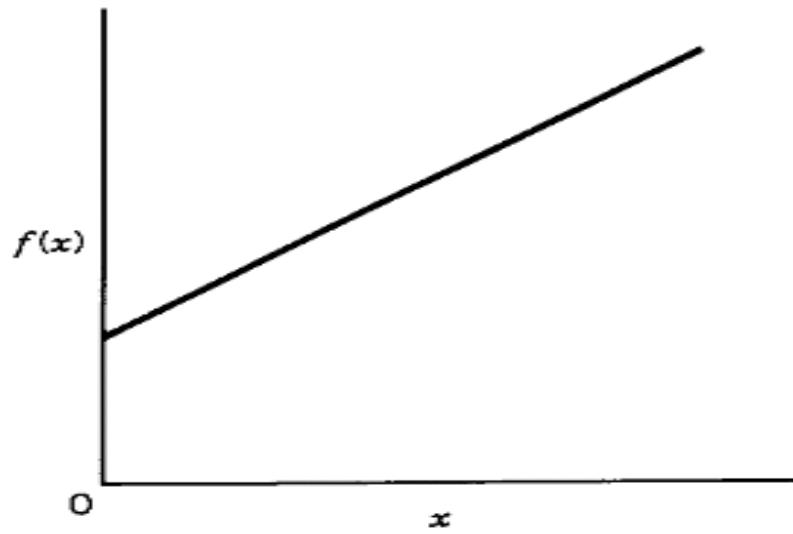
(a) Constrained case ($x \geq \bar{x}$), $x^{\text{opt}} = \frac{-a_1}{2a_2}$



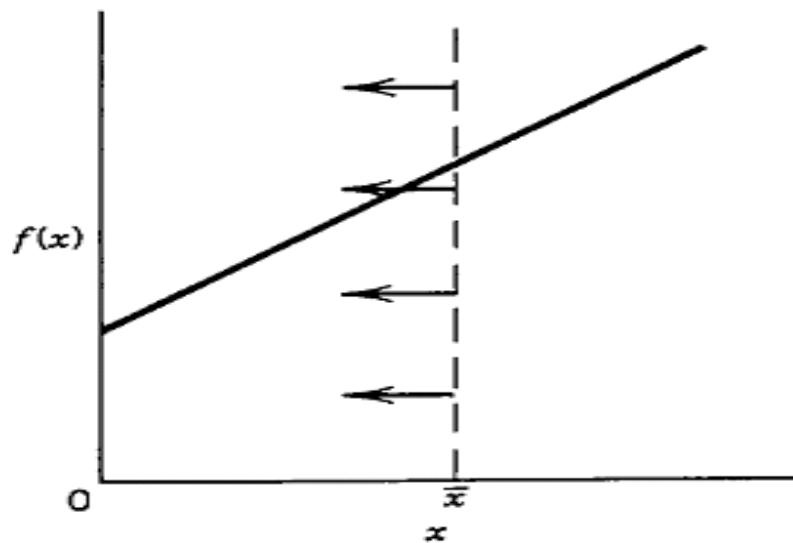
(b) Constrained case ($x \leq \bar{x}$), $x^{\text{opt}} = \bar{x}$

Figure: The effect of an inequality constraint on the maximum of quadratic function, $f(x) = a_0 + a_1x + a_2x^2$. The arrows indicate the allowable values of x .

Linear Programming



(a) Unconstrained case, $x^{\text{opt}} \rightarrow \infty$



(b) Constrained case ($x \leq \bar{x}$), $x^{\text{opt}} = \bar{x}$

The effect of a linear constraint
on the maximum of linear objective function,
 $f(x) = a_0 + a_1x$.

Steps in Setting up a LP

1. Determine and label the *Decision Variables*.
2. Determine the Objective and use the Decision Variables to write an expression for the *Objective Function*.
3. Determine the Constraints - *Feasible Region*.
 1. Determine the *Explicit Constraints* and write a functional expression for each of them.
 2. Determine the *Implicit Constraints (Nonnegativity Constraints)*.

Nature Connection: Recreational Sites

Nature Connection is planning two new public recreational sites: **a forested wilderness area** and a **sightseeing and hiking park**. They own **80 hectares of forested wilderness area** and **20 hectares** suitable for the sightseeing and hiking park but they don't have enough resources to make the entire areas available to the public. They have a **budget of \$120K** per year. They estimate a yearly management and maintenance cost of **\$1K per hectare** for the forested wilderness area, and **\$4K per hectare** for the sightseeing and hiking park. The expected average number of visiting hours a day per hectare are: **10 for the forest** and **20 for the sightseeing and hiking park**.

Question: How many hectares should Nature Connection allocate to the public sightseeing and hiking park and to the public forested wilderness area, in order to maximize the amount of recreation, (in average number of visiting hours a day for the total area to be open to the public, for both sites) given their budget constraint?

Formulation of the Problem as a Linear Program

1 Decision Variables

x_1 – # hectares to allocate to the public forested wilderness area

x_2 – # hectares to allocate the public sightseeing and hiking park

2 Objective Function

$$\text{Max } 10x_1 + 20x_2$$

3 Constraints

$$x_1 \leq 80$$

$$x_2 \leq 20$$

$$x_1 + 4x_2 \leq 120$$

$x_1 \geq 0; x_2 \geq 0$ Non-negativity constraints

Formulation of the Problem as a Linear Program

Nature Connection is planning two new public recreational sites: a forested wilderness area and a sightseeing and hiking park. They own 80 hectares of forested wilderness area and 20 hectares suitable for the sightseeing and hiking park but they don't have enough resources to make the entire areas available to the public. They have a budget of \$120K per year. They estimate a yearly management and maintenance cost of \$1K per hectare, for the forested wilderness area, and \$4K per hectare for the sightseeing and hiking park. The expected average number of visiting hours a day per hectare are: 10 for the forest and 20 for the sightseeing and hiking park.

Question: How many hectares should Nature Connection allocate to the public sightseeing and hiking park and to the public forested wilderness area, in order to maximize the amount of recreation, (in average number of visiting hours a day for the total area to be open to the public, for both sites) given their budget constraint?

1 Decision Variables

~~x1 # hectares to allocate to the public forested wilderness area~~

X2 – # hectares to allocate the public sightseeing and hiking park

2 Objective Function

$$\text{Max } 10x_1 + 20x_2$$

3 Constraints

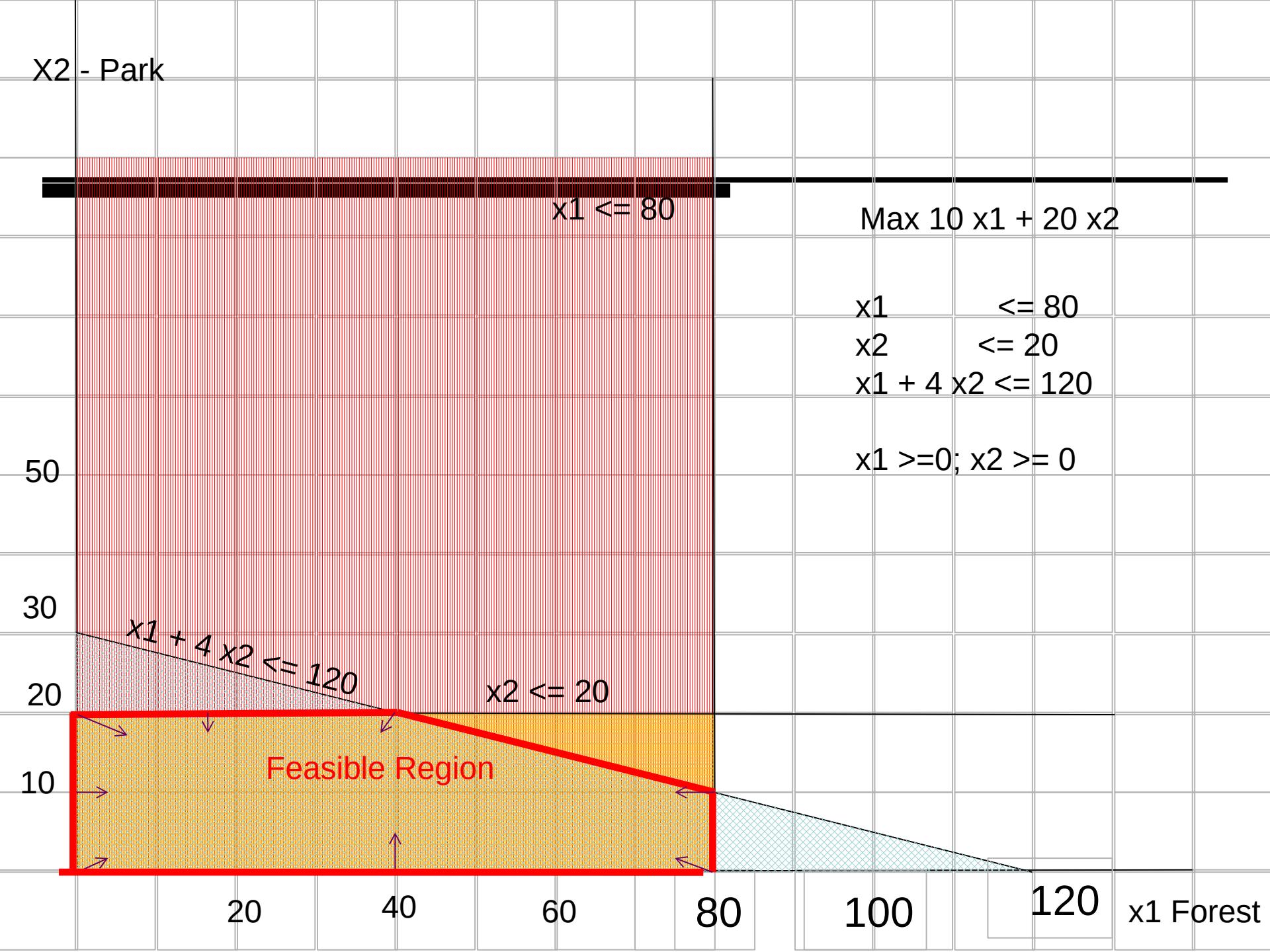
$$x_1 \leq 80 \text{ Land for forest}$$

$$x_2 \leq 20 \text{ Land for Park}$$

$$x_1 + 4x_2 \leq 120 \text{ Budget}$$

$$x_1 \geq 0; x_2 \geq 0 \text{ Non-negativity constraints}$$

x_2 - Park



$$x_1 \leq 80$$

$$\text{Max } 10x_1 + 20x_2$$

$$x_1 \leq 80$$

$$x_2 \leq 20$$

$$x_1 + 4x_2 \leq 120$$

$$x_1 \geq 0; x_2 \geq 0$$

$$x_1 + 4x_2 \leq 120$$

$$x_2 \leq 20$$

Feasible Region

20

40

60

80

100

120

x_1 Forest

x₂ - Park

The vector representing the gradient of the objective function is given by:

$$\begin{bmatrix} 10 \\ 20 \end{bmatrix}$$

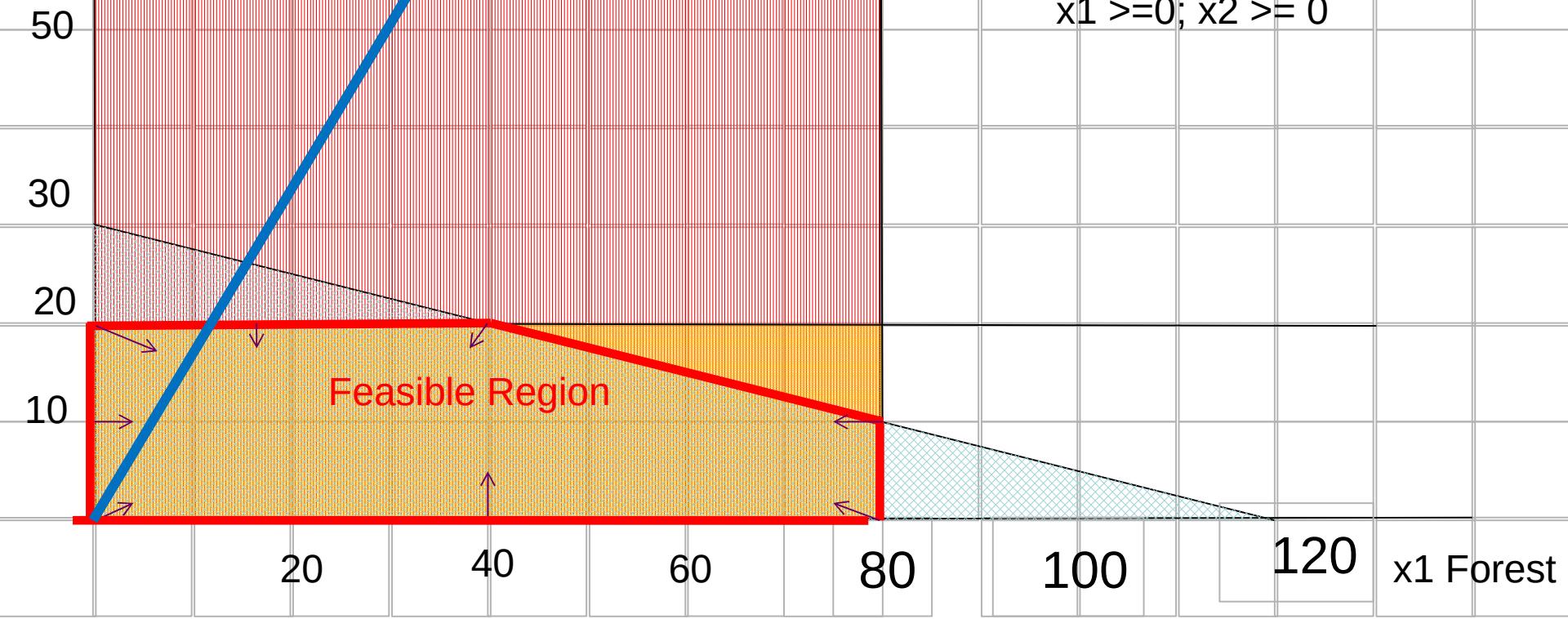
$$\text{Max } 10x_1 + 20x_2$$

$$x_1 \leq 80$$

$$x_2 \leq 20$$

$$x_1 + 4x_2 \leq 120$$

$$x_1 \geq 0; x_2 \geq 0$$



x_2 - Park

The vector representing the gradient of the objective function is given by:

$$\begin{bmatrix} 10 \\ 20 \end{bmatrix}$$

$$\text{Max } 10x_1 + 20x_2$$

$$x_1 \leq 80$$

$$x_2 \leq 20$$

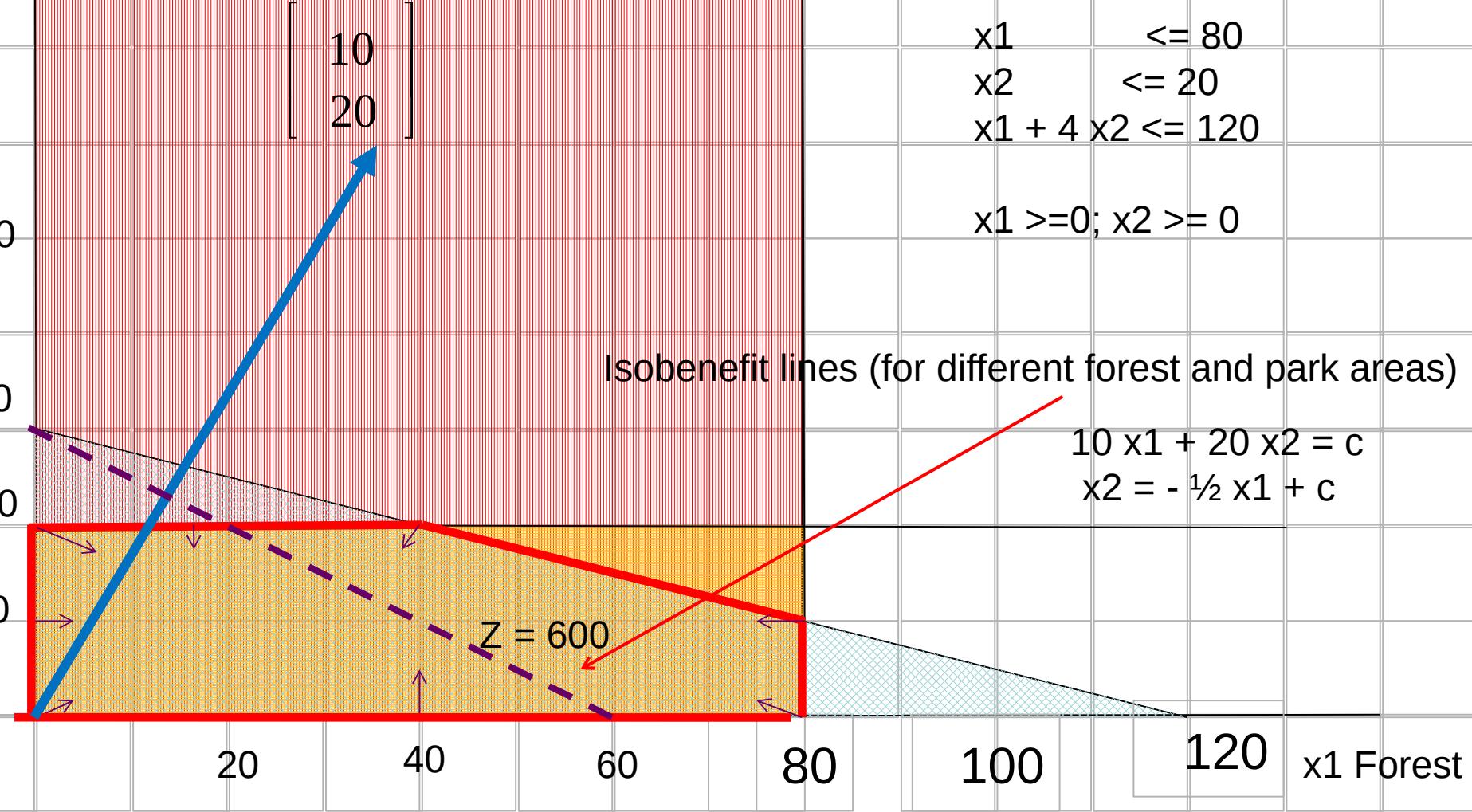
$$x_1 + 4x_2 \leq 120$$

$$x_1 \geq 0; x_2 \geq 0$$

Isobenefit lines (for different forest and park areas)

$$10x_1 + 20x_2 = c$$

$$x_2 = -\frac{1}{2}x_1 + c$$



x_2 - Park

$$\text{Max } 10x_1 + 20x_2$$

$$x_1 \leq 80$$

$$x_2 \leq 20$$

$$x_1 + 4x_2 \leq 120$$

$$x_1 \geq 0; x_2 \geq 0$$

The vector representing the gradient of the objective function is given by:

$$\begin{bmatrix} 10 \\ 20 \end{bmatrix}$$

Isobenefit lines (for different forest and park areas)

$$x_2 = -\frac{1}{2}x_1 + c$$

$$Z^* = 10(80) + 20(10) = 1000$$

$$Z = 200$$

$$Z = 600$$

20

40

60

80

100

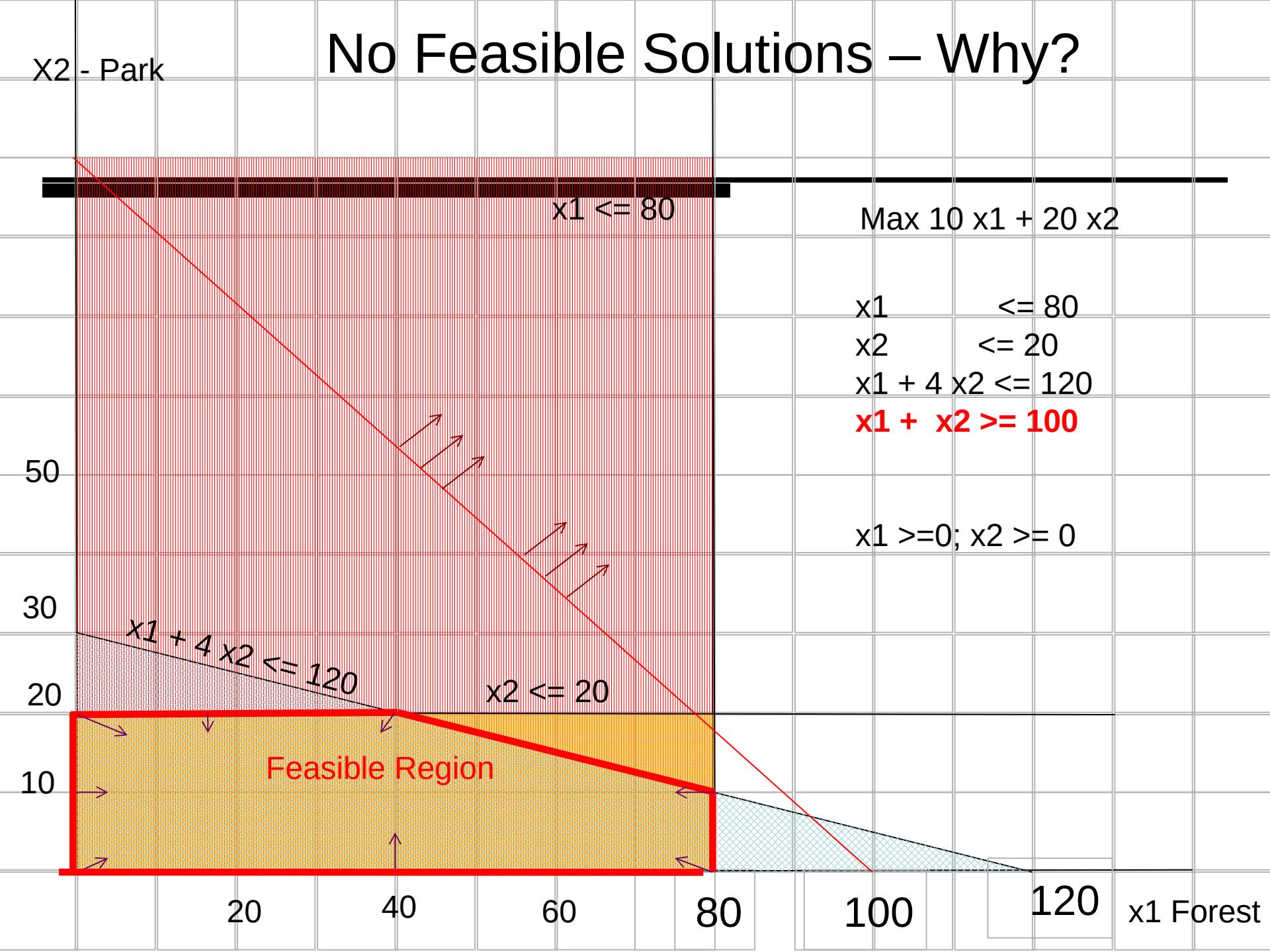
120

x_1 Forest

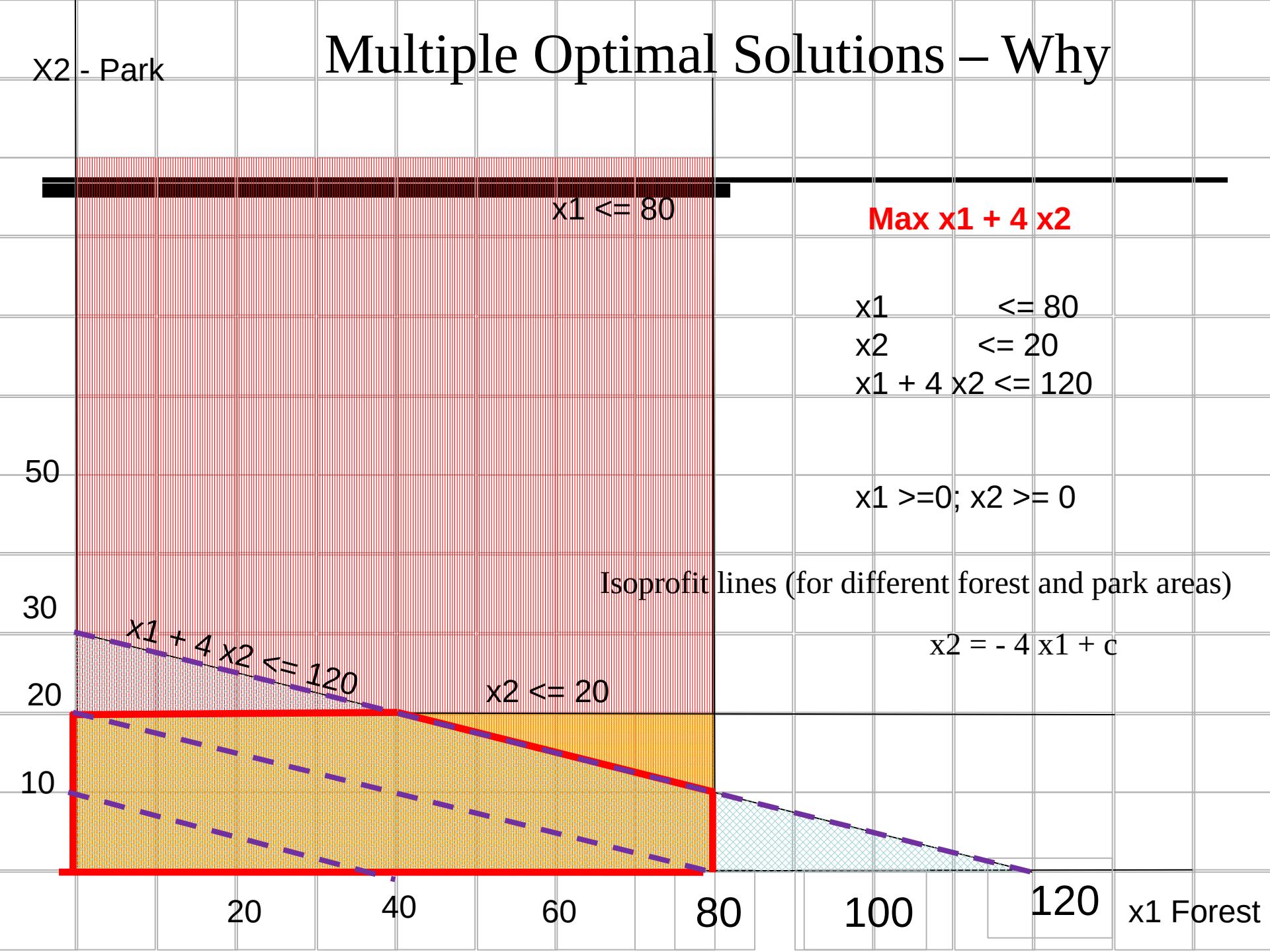
Summary of the Graphical Method

- Draw the constraint boundary line for each constraint. Use the origin (or any point not on the line) to determine which side of the line is permitted by the constraint.
- Find the feasible region by determining where all constraints are satisfied simultaneously.
- Determine the slope of one objective function line (perpendicular to its gradient vector). All other objective function lines will have the same slope.
- Move a straight edge with this slope through the feasible region in the direction of improving values of the objective function (direction of the gradient). Stop at the last instant that the straight edge still passes through a point in the feasible region. This is the optimal objective function line.
- A feasible point on the optimal objective function line is an optimal solution.

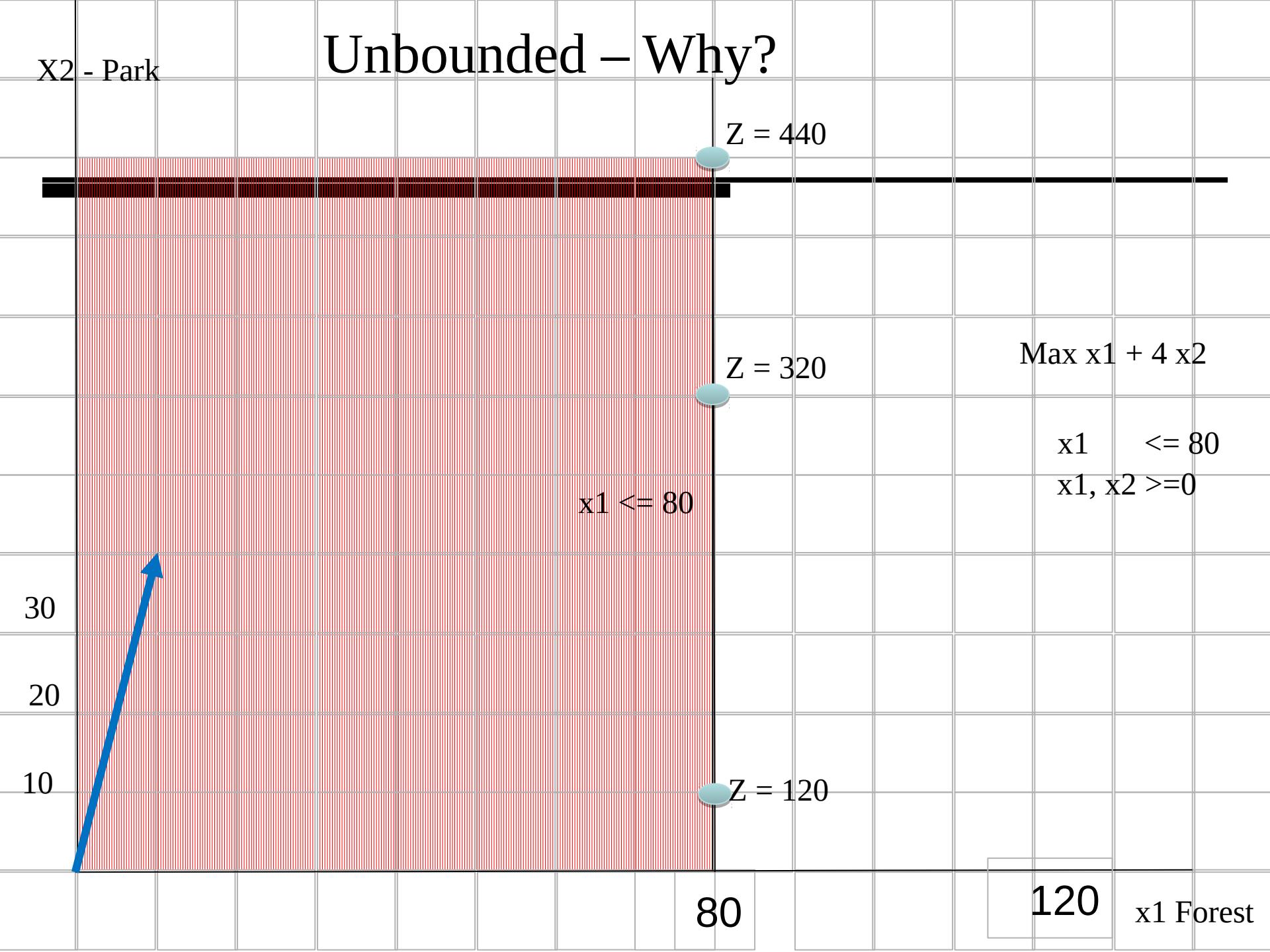
No Feasible Solutions – Why?



Multiple Optimal Solutions – Why



Unbounded – Why?



Key Categories of LP Problems:

- Resource-Allocation Problems
- Cost-benefit-trade-off Problems
- Distribution-Network Problems
- Mixed Problems

Second Example: Keeping the River Clean

Cost-benefit-trade-off problems

Choose the mix of levels of various activities to achieve minimum acceptable levels for various benefits at a minimum cost.

Second Example: Keeping the River Clean

A pulp mill in Maine makes mechanical and chemical pulp, polluting the river in which it spills its spent waters. This has created several problems, leading to a change in management.

The previous owners felt that it would be too expensive to reduce pollution, so they decided to sell the pulp mill. The mill has been bought back by the employees and local businesses, who now own the mill as a cooperative. The new owners have several objectives:

- 1 – to keep **at least 300 people employed at the mill** (300 workers a day);
- 2 – to generate at least \$40,000 of revenue a day

They estimate that this will be enough to pay operating expenses and yield a return that will keep the mill competitive in the long run. Within these limits, everything possible should be done to **minimize pollution**.

Both chemical and mechanical pulp require the labor of one worker for 1 day (1 workday, wd) per ton produced;

Mechanical pulp sells at \$100 per ton; Chemical pulp sells at \$200 per ton;

Pollution is measured by the biological oxygen demand (BOD). One ton of mechanical pulp produces 1 unit of BOD; One ton of chemical pulp produces 1.5 units of BOD.

The maximum capacity of the mill to make mechanical pulp is 300 tons per day; for chemical pulp is 200 tons per day. The two manufacturing processes are independent (i.e., the mechanical pulp line cannot be used to make chemical pulp and vice versa).

- Pollution, employment, and revenues result from the production of both types of pulp. So a natural choice for the variables is:

Decision Variables

- X_1 amount of mechanical pulp produced (in tons per day, or t/d) and
- X_2 amount of chemical pulp produce (in tons per day, or t/d)
- $\text{Min } Z = 1 X_1 + 1.5 X_2$
 $(\text{BOD/day}) \quad (\text{BOD/t}) \quad (\text{t/d}) \quad (\text{BOD/t}) \quad (\text{t/d})$

Subject to (s.t.):

$$\begin{aligned}
 & 1 X_1 + 1 X_2 \geq 300 \text{ Workers/day} \\
 & (\text{wd/t}) \quad (\text{t/d}) \quad (\text{wd/t})(\text{t/d}) \\
 & 100 X_1 + 200 X_2 \geq 40,000 \text{ revenue/day} \\
 & (\$/\text{t}) \quad (\text{t/d}) \quad (\$/\text{t})(\text{t/d}) \quad \$/\text{d} \\
 & X_1 \leq 300 \text{ (mechanical pulp)} \\
 & (\text{t/d}) \quad (\text{t/d}) \\
 & X_2 \leq 200 \text{ (chemical pulp)} \\
 & (\text{t/d}) \quad (\text{t/d}) \\
 & X_1 \geq 0; X_2 \geq 0
 \end{aligned}$$

Distribution Network Problems

Distribution-Network Problem

- The International Hospital Share Organization is a non-profit organization that refurbishes a variety of used equipment for hospitals of developing countries at two international factories (F1 and F2). One of its products is a large X-Ray machine.
- Orders have been received from three large communities for the X-Ray machines.

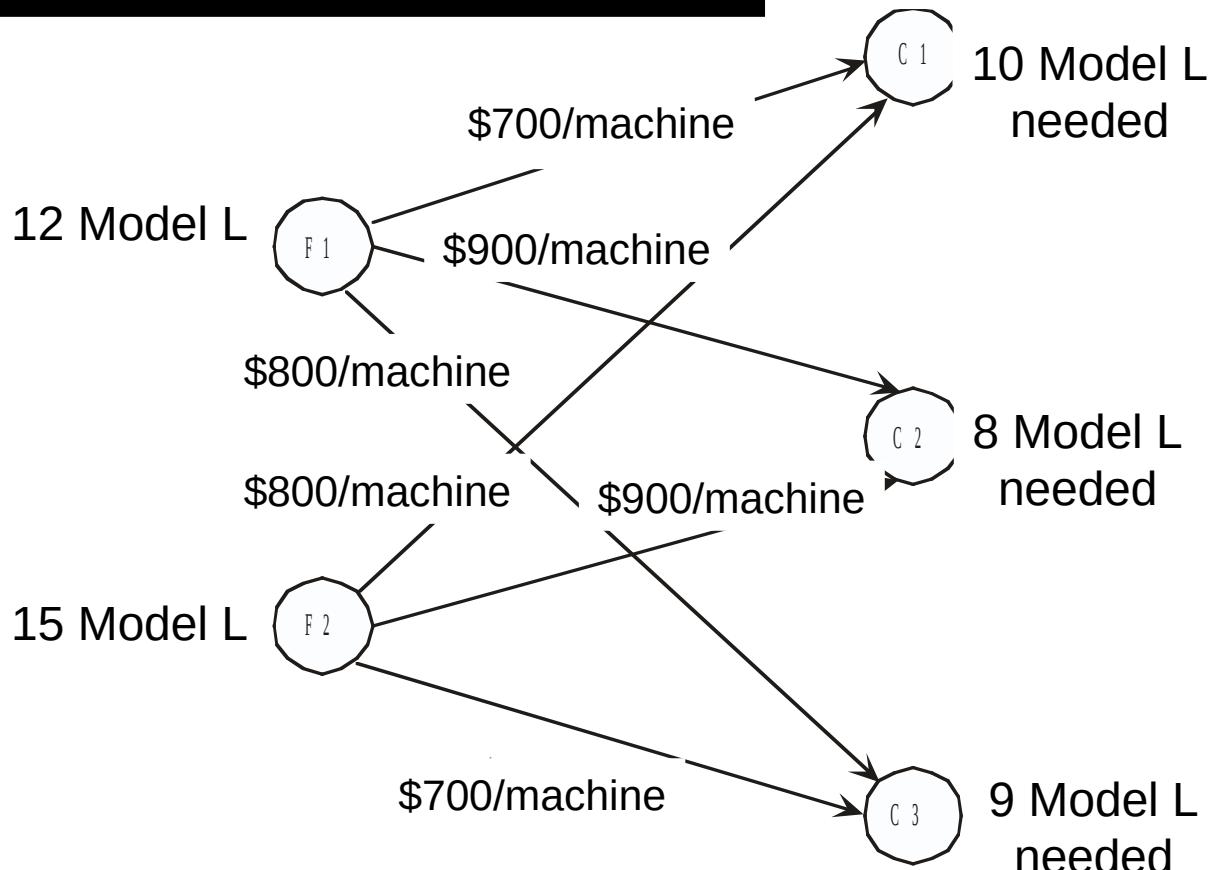
Some Data

Shipping Cost for Each Machine (Model L)

To	Community 1	Community 2	Community 3	From	Output
Factory 1	\$700	\$900	\$800	12 X-ray machines	
Factory 2	800	900	700	15 X-Ray machines	
Order Size	10 X-ray machines	8 X-Ray machines	9 X-Ray machines		

Question: How many X-Ray machines (model L) should be shipped from each factory to each hospital so that shipping costs are minimized?

The Distribution Network



Question: How many machines (model L) should be shipped from each factory to each customer so that shipping costs are minimized?

-
- Activities – shipping lanes (not the level of production which has already been defined)
 - Level of each activity – number of machines of model L shipped through the corresponding shipping lane.
→ Best mix of shipping amounts

Example:

- Requirement 1: Factory 1 must ship 12 machines
- Requirement 2: Factory 2 must ship 15 machines
- Requirement 3: Customer 1 must receive 10 machines
- Requirement 4: Customer 2 must receive 8 machines
- Requirement 5: Customer 3 must receive 9 machines

Algebraic Formulation

Let S_{ij} = Number of machines to ship from i to j ($i = F1, F2$; $j = C1, C2, C3$).

$$\begin{aligned} \text{Minimize Cost} = & \$700S_{F1-C1} + \$900S_{F1-C2} + \$800S_{F1-C3} \\ & + \$800S_{F2-C1} + \$900S_{F2-C2} + \$700S_{F2-C3} \end{aligned}$$

subject to

$$S_{F1-C1} + S_{F1-C2} + S_{F1-C3} = 12$$

$$S_{F2-C1} + S_{F2-C2} + S_{F2-C3} = 15$$

$$S_{F1-C1} + S_{F2-C1} = 10$$

$$S_{F1-C2} + S_{F2-C2} = 8$$

$$S_{F1-C3} + S_{F2-C3} = 9$$

and

$$S_{ij} \geq 0 \quad (i = F1, F2; j = C1, C2, C3).$$

Algebraic Formulation

Let S_{ij} = Number of machines to ship from i to j ($i = F1, F2$; $j = C1, C2, C3$).

$$\begin{aligned} \text{Minimize Cost} = & \$700S_{F1-C1} + \$900S_{F1-C2} + \$800S_{F1-C3} \\ & + \$800S_{F2-C1} + \$900S_{F2-C2} + \$700S_{F2-C3} \end{aligned}$$

subject to

$$\text{Factory 1: } S_{F1-C1} + S_{F1-C2} + S_{F1-C3} = 12$$

$$\text{Factory 2: } S_{F2-C1} + S_{F2-C2} + S_{F2-C3} = 15$$

$$\text{Customer 1: } S_{F1-C1} + S_{F2-C1} = 10$$

$$\text{Customer 2: } S_{F1-C2} + S_{F2-C2} = 8$$

$$\text{Customer 3: } S_{F1-C3} + S_{F2-C3} = 9$$

and

$$S_{ij} \geq 0 \quad (i = F1, F2; j = C1, C2, C3).$$

Terminology and Notations

Terminology of Solutions in LP Model

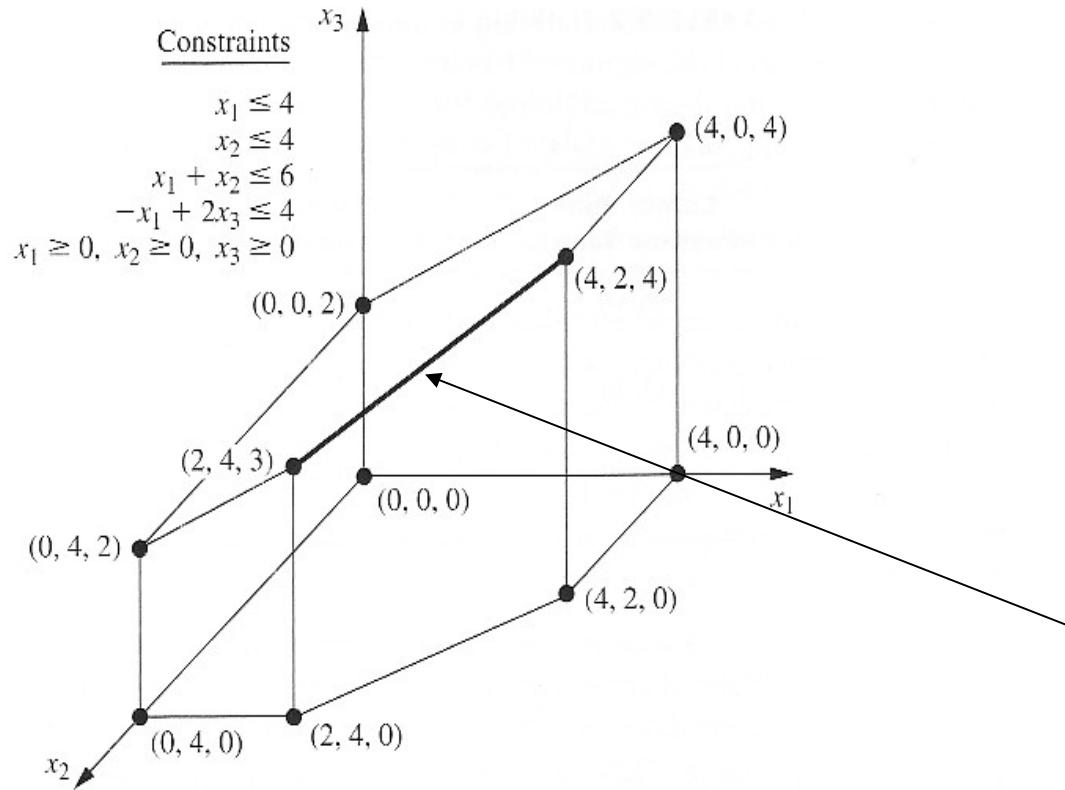
- Solution – not necessarily the final answer to the problem!!!
- ~~Feasible Solution~~ – Solution that satisfies all the constraints
- Infeasible Solution – Solution for which at least one of the constraints is violated
- Feasible Region – Set of all points that satisfies all constraints (possible to have a problem without any feasible solutions)
- Binding Constraint – The left-hand side (LHS) and the right-hand side (RHS) of the constraint are equal, i.e., constraint is satisfied in equality. Otherwise the constraint is nonbinding.
- Optimal Solution – Feasible Solution that has the Best Value of the Objective Function.

Largest Value \square Maximization Problems

Smallest Value \square Minimization Problems

- Multiple Optimal Solutions, no optimal solutions, unbounded Z

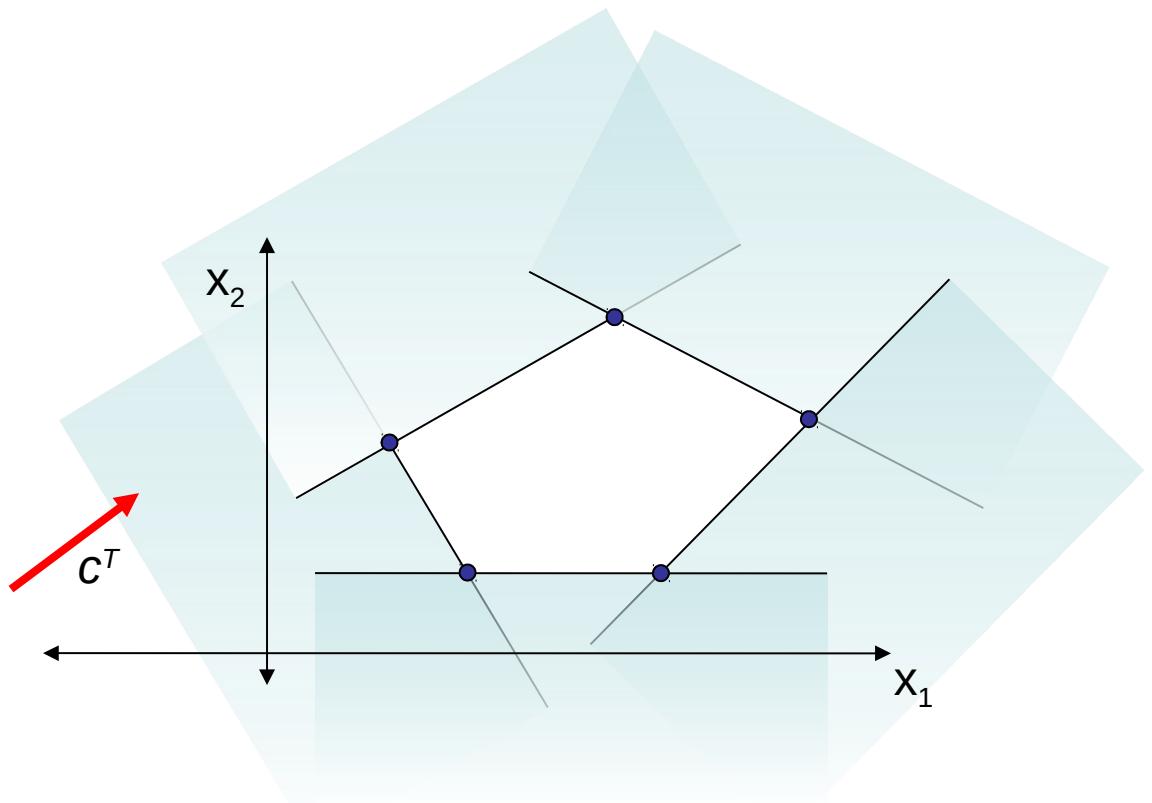
3D Feasible Region



Solving Linear Programs

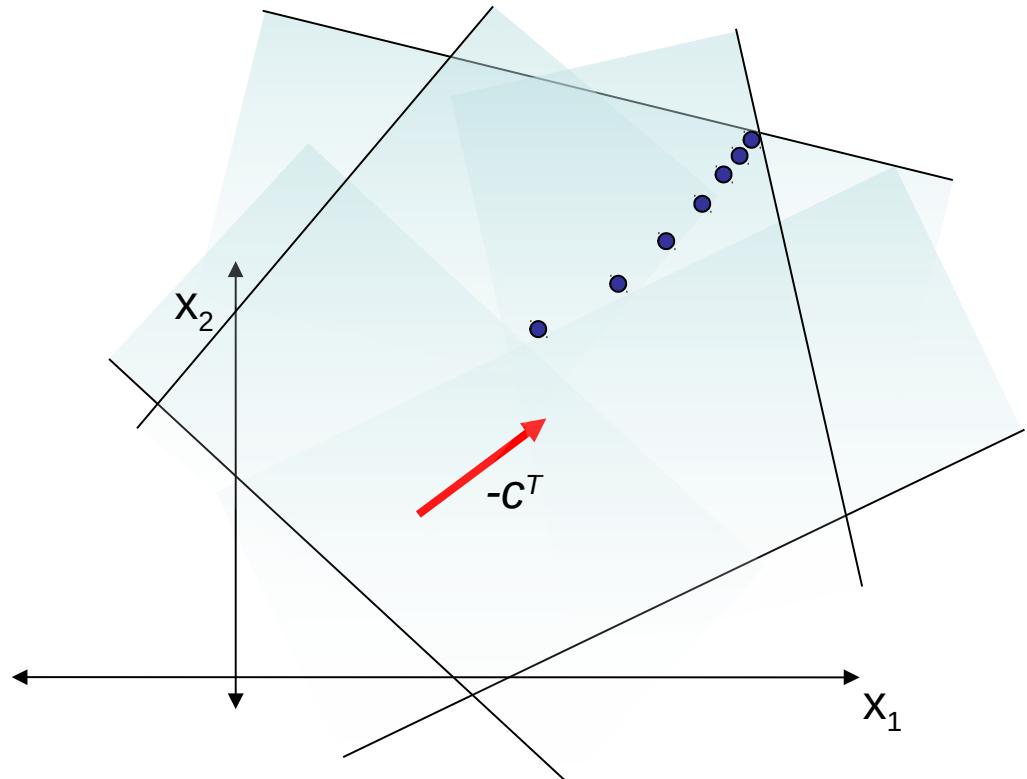
Solution Methods for Linear Programs

- Simplex Method
 - Optimum must be at the intersection of constraints
 - Intersections are easy to find, change inequalities to equalities



Solution Methods for Linear Programs

- Interior Point Methods
- Benefits
 - Scales Better than Simplex



Standard Form of the LP Model

minimize $Z = c_1x_1 + c_2x_2 + \dots + c_nx_n,$

subject to the restrictions

$$a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n \leq b_1$$

$$a_{21}x_1 + a_{22}x_2 + \dots + a_{2n}x_n \leq b_2$$

⋮

$$a_{m1}x_1 + a_{m2}x_2 + \dots + a_{mn}x_n \leq b_m,$$

$$x_i \geq 0, (i=1,2,\dots,n)$$

Other Variants:

Maximize Z (instead of minimizing Z ; but $\text{Min } Z = -\text{Max } -Z$)

Some constraints have other signs ($=$; and \geq)

Some variables have unrestricted sign, i.e., they are not subject to the non-negativity constraints

Solving Linear Programs

- To solve LPs, typically need to put them in standard form:

$$\begin{array}{ll}\text{minimize}_z & c^T z \\ \text{subject to} & Az \leq b\end{array}$$

- $z \in \mathbb{R}^n$, $A \in \mathbb{R}^{N_i \times n}$, $b \in \mathbb{R}^{N_i}$
- For absolute loss LP

$$z = \begin{bmatrix} \theta \\ \nu \end{bmatrix}, \quad c = \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \quad A = \begin{bmatrix} \Phi & -I \\ -\Phi & -I \end{bmatrix}, \quad b = \begin{bmatrix} y \\ -y \end{bmatrix}$$

$$\underset{\theta}{\text{minimize}} \quad \sum_{i=1}^m |\theta^T \phi(x_i) - y_i|$$

$$\begin{array}{ll}\underset{\theta, \nu}{\text{minimize}} & \sum_{i=1}^m \nu_i \\ \text{subject to} & -\nu_i \leq \theta^T \phi(x_i) - y_i \leq \nu_i\end{array}$$

Violates Divisibility Assumption of LP

- **Divisibility Assumption of Linear Programming (LP):** Decision variables in a Linear Programming model are allowed to have *any* values, including *fractional (noninteger)* values, that satisfy the functional and nonnegativity constraints i.e., activities can be run at *fractional levels..*

**When divisibility assumption is violated
then apply Integer Linear Programming!!!**



Topics in Computational Sustainability: Integer Linear Programming



Violates Divisibility Assumption of LP

- **Divisibility Assumption of Linear Programming (LP):** Decision variables in a Linear Programming model are allowed to have *any* values, including *fractional (noninteger)* values, that satisfy the functional and nonnegativity constraints i.e., activities can be run at *fractional levels..*

When divisibility assumption is violated then apply Integer Linear Programming!!!

Why Integer Linear Programming?

- Advantages of restricting the variables to take on integer values
 - More realistic
 - More flexibility
- Disadvantages
 - More difficult to model
 - Can be much more difficult to solve

Integer Linear Programming

- When are “non-integer” solutions okay?
 - Solution is naturally divisible
 - e.g., pounds, hours
 - Solution represents a rate
 - e.g., units per week
- When is rounding okay?
 - When numbers are large
 - e.g., rounding 114.286 to 114 is probably okay.
- When is rounding not okay?
 - When numbers are small
 - e.g., rounding 2.6 to 2 or 3 may be a problem.
 - Binary variables
 - yes-or-no decisions

Graphical Method for Integer Programming

- When a Binary Integer Programming problem has just **two decision variables**, its optimal solution can be found by applying the *Graphical Method for Linear Programming* with just one change at the end.
- Let us begin by graphing the feasible region for the LP relaxation, determining the slope of the objective function lines, and moving a straight edge with this slope through this feasible region in the direction of improving values of the objective function.
- However, rather than stopping at the last instant the straight edge passes through this feasible region, now stop at the last instant the straight edge passes through an integer point that lies within this feasible region.
- This integer point is the optimal solution.

Green Power Company Example: Capital Budgeting Allocation Problem

- A Green Power Company would like to consider 6 investments. The cash required from each investment as well as the total electricity production of the investment is given next. The cash available for the investments is \$14M. Danish Green Power wants to maximize electricity production.
What is the optimal strategy?
- An investment can be selected or not. One cannot select a fraction of an investment.

Data for the Problem

Investment budget = \$14M

Investment	1	2	3	4	5	6
Cost (millions dollars)	5	7	4	3	4	6
Total production (100KW)	16	22	12	8	11	19

Capital Budgeting Allocation Problem (one resource) (Mapped to the Knapsack Problem)

- Why is a problem with the characteristics of the previous problem called the Knapsack Problem?
- It is an abstraction, considering the simple problem:

A hiker tries to fill knapsack to maximum total value. Each item we consider taking with us has a certain value and a certain weight. An overall weight limitation gives the single constraint.

Practical Applications:

- Project selection and capital budgeting allocation
- Storing a warehouse to maximum value given the indivisibility of goods and space limitations

Integer Programming Formulation

What are the Decision Variables?

$$x_i = \begin{cases} 1, & \text{if you choose item } i = 1, \dots, 6, \\ 0, & \text{else} \end{cases}$$

Objective Function and Constraints?

$$\text{Max } 16x_1 + 22x_2 + 12x_3 + 8x_4 + 11x_5 + 19x_6$$

$$5x_1 + 7x_2 + 4x_3 + 3x_4 + 4x_5 + 6x_6 \leq 14$$

$x_j \in \{0,1\}$ for each $j = 1$ to 6

- The previous constraints represent “**economic indivisibilities**”, either a project is selected, or not. It is not possible to select a fraction of a project.
- Similarly, integer variables can model **logical requirements** (e.g., if item 2 is selected, then so is item 1.)

How to model “logical” constraints

- Exactly 3 items are selected.
- If item 2 is selected, then so is item 1.
- If item 1 is selected, then item 3 is not selected.
- Either item 4 is selected or item 5 is selected, but not both.

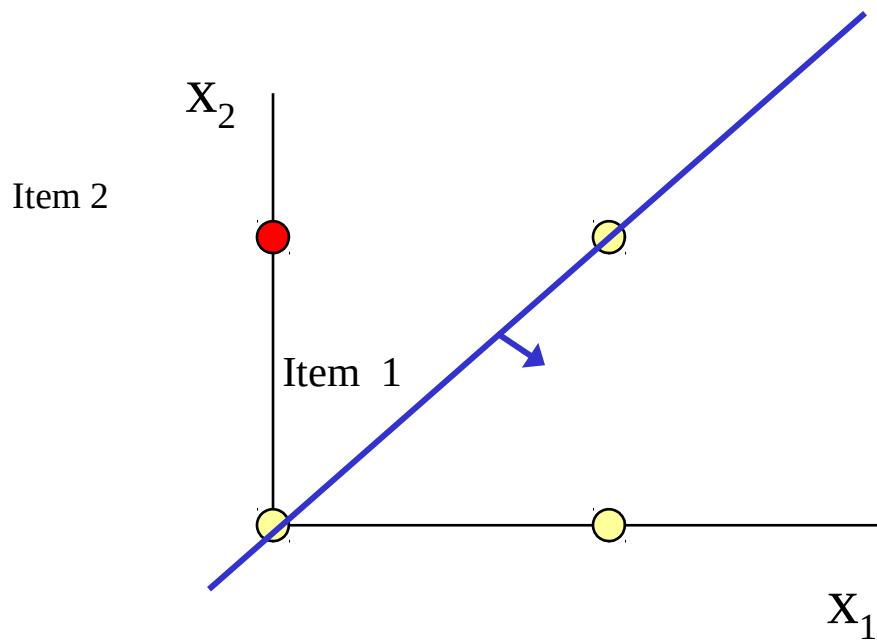
Formulating Constraints

- Exactly 3 items are selected

$$x_1 + x_2 + x_3 + x_4 + x_5 + x_6 = 3$$

If item 2 is selected then so is item 1

A 2-dimensional representation

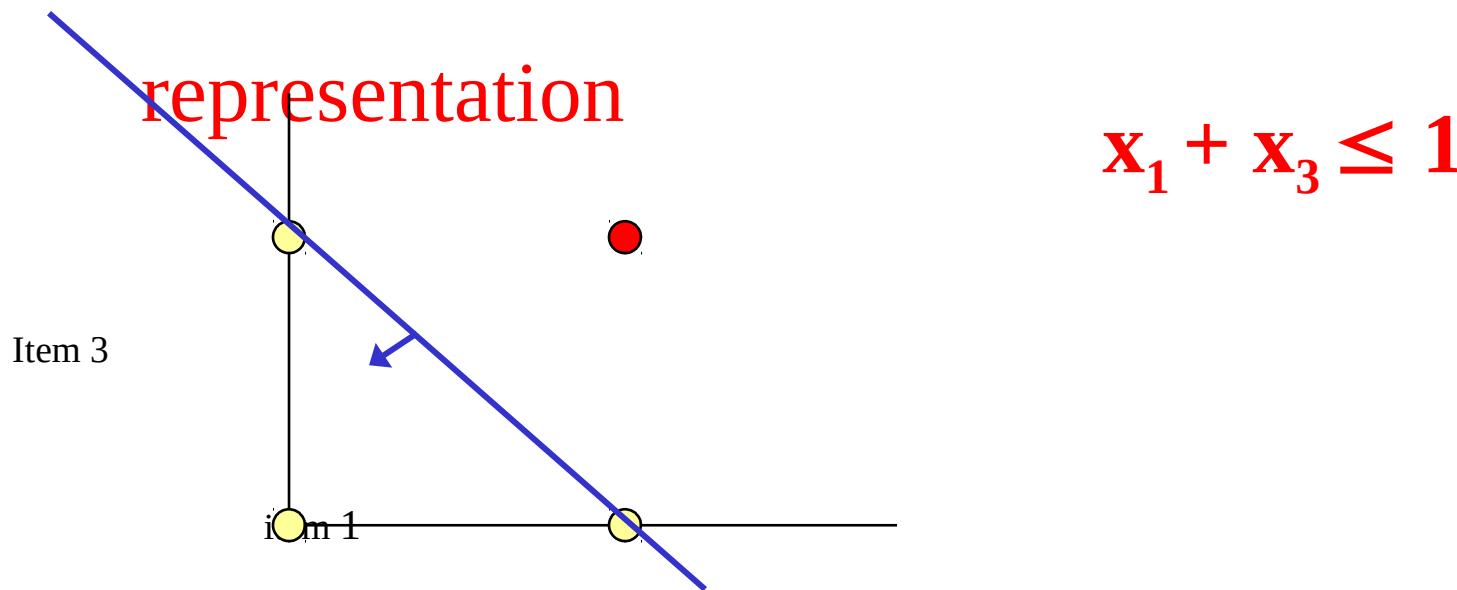


The integer
programming
constraint:

$$x_1 \geq x_2$$

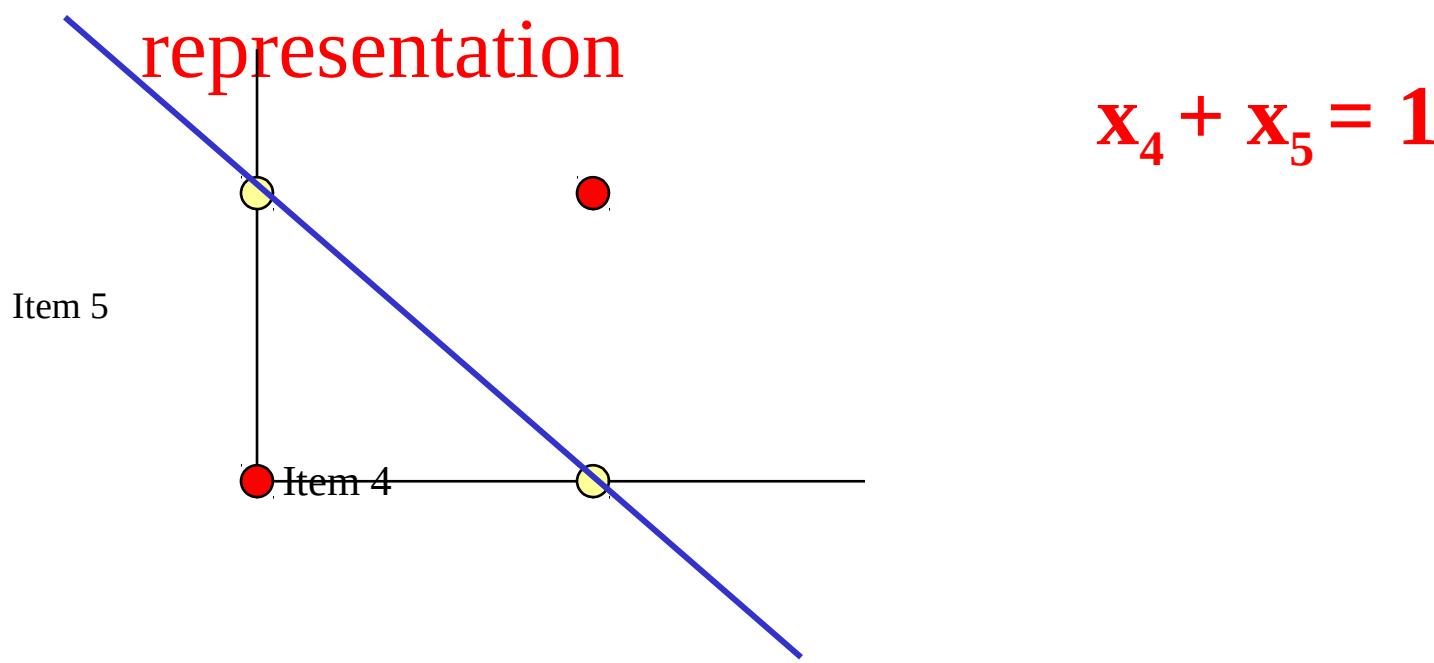
If item 1 is selected then item 3 is
not selected

The integer programming constraint:
A 2-dimensional



Either item 4 is selected or item 5
is selected, but not both.

The integer programming constraint:
A 2-dimensional



How to model “logical” constraints

- Exactly 3 items are selected. $x_1 + x_2 + x_3 + x_4 + x_5 + x_6 = 3$
- If item 2 is selected, then so is item 1. $x_1 \geq x_2$
- If item 1 is selected, then item 3 is not selected.

$$x_1 + x_3 \leq 1 \quad \square x_1 \leq 1 - x_3$$

- Either item 4 is selected or item 5 is selected, but not both.

$$x_4 + x_5 = 1$$

Modeling Fixed Charge Problems

If a product is produced, a factory is built \square must incur a fixed setup cost.

→ The problem is non-linear.

x – quantity of product to be manufactured

$x = 0 \square$ cost = 0;

$x > 0 \square$ cost = $C_1x + C_2$

→ How to model it? Using an *indicator* variable y

$y = 1 \square$ x is produced; $y = 0 \square$ x is not produced

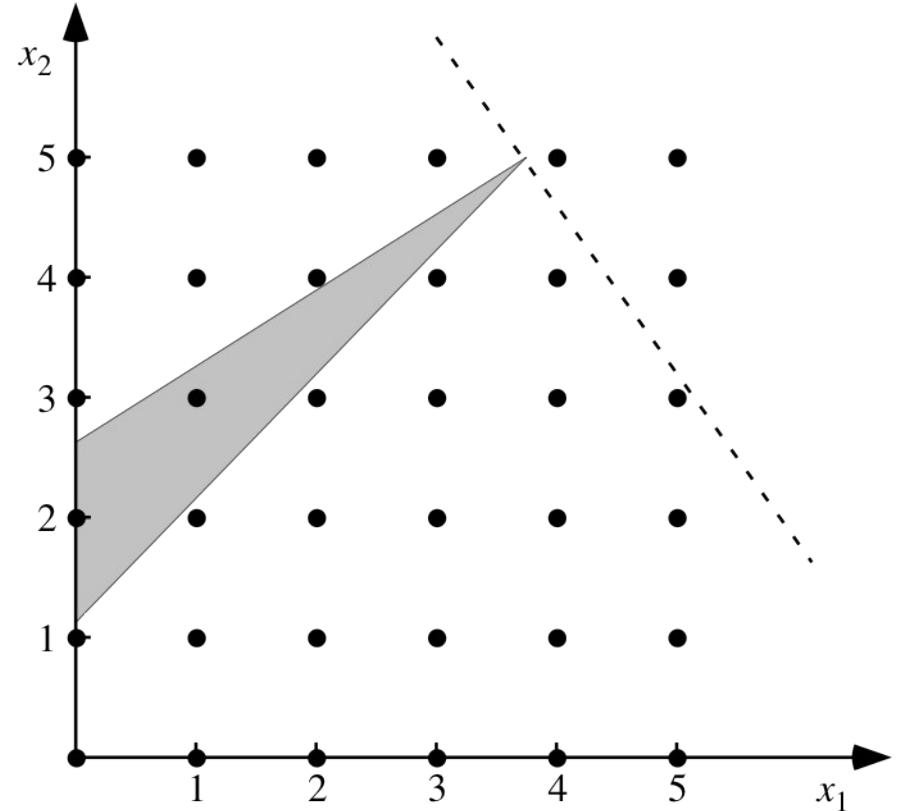
Objective function becomes $\square C_1x + C_2y$

Additional Constraint $\square x \leq My$ M is a big number

Solving Integer Programs

The Challenges of Rounding

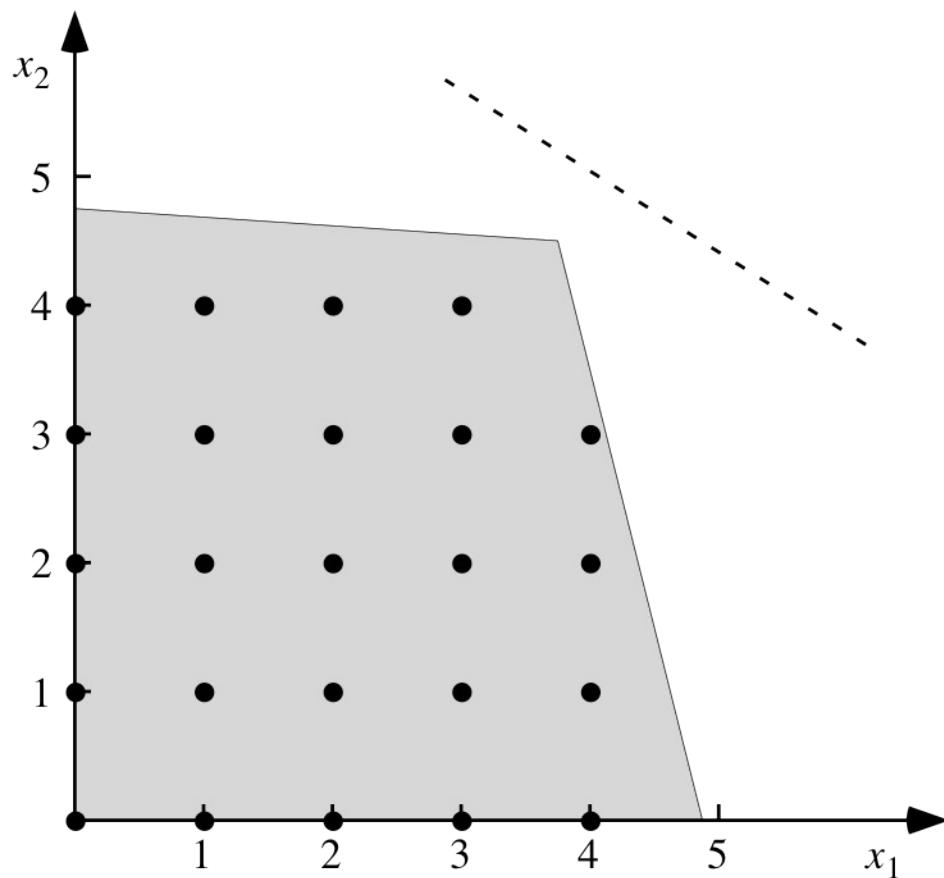
- Rounded Solution may not be feasible.
- Rounded solution may not be close to optimal.
- There can be *many* rounded solutions.
 - Example: Consider a problem with 30 variables that are non-integer in the LP-solution. How many possible rounded solutions are there?



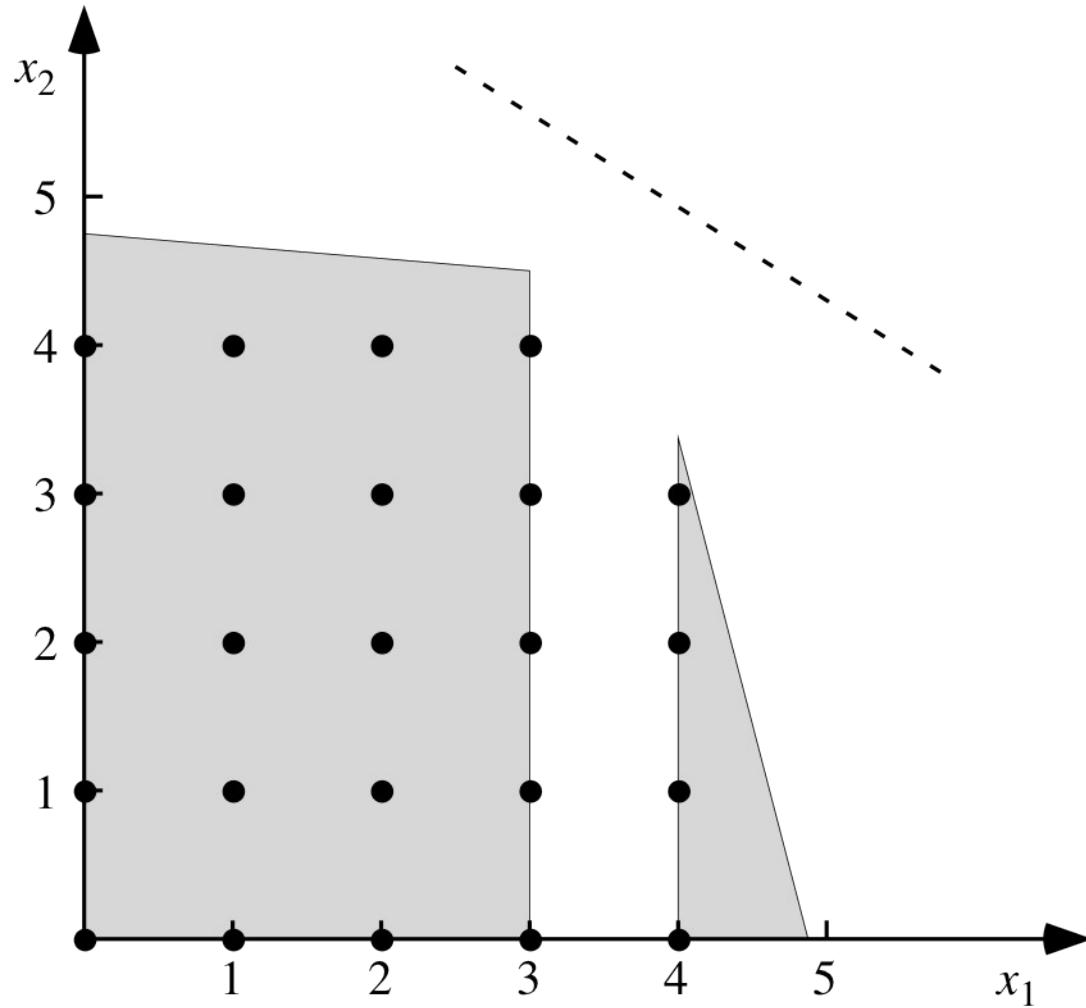
Overview of Techniques for Solving Integer Programs

- Enumeration Techniques
 - Complete Enumeration
 - list all “solutions” and choose the best
 - Branch and Bound
 - Implicitly search all solutions, but cleverly eliminate the vast majority before they are even searched
- Cutting Plane Techniques
 - Use Linear Programming (LP) to solve integer programs by adding constraints to eliminate the fractional solutions.
- Hybrid Approaches (e.g., Branch and Cut)

How Integer Programs are Solved: Cuts



How Integer Programs are Solved



Branch and Bound

- Implicit enumeration of all the solutions:
 - It is the starting point for all solution techniques for integer programming
- Lots of research has been carried out over the past 40 years to make it more and more efficient.
- It is an art form to make it efficient (We shall get a sense why).
- Integer programming is intrinsically difficult.

Knapsack Problem: Binary Integer Programming Formulation

What are the decision variables

$$x_i = \begin{cases} 1, & \text{if you choose item } i = 1, \dots, 6, \\ 0, & \text{else} \end{cases}$$

$$\text{Max } 16x_1 + 22x_2 + 12x_3 + 8x_4 + 11x_5 + 19x_6$$

$$5x_1 + 7x_2 + 4x_3 + 3x_4 + 4x_5 + 6x_6 \leq 14$$

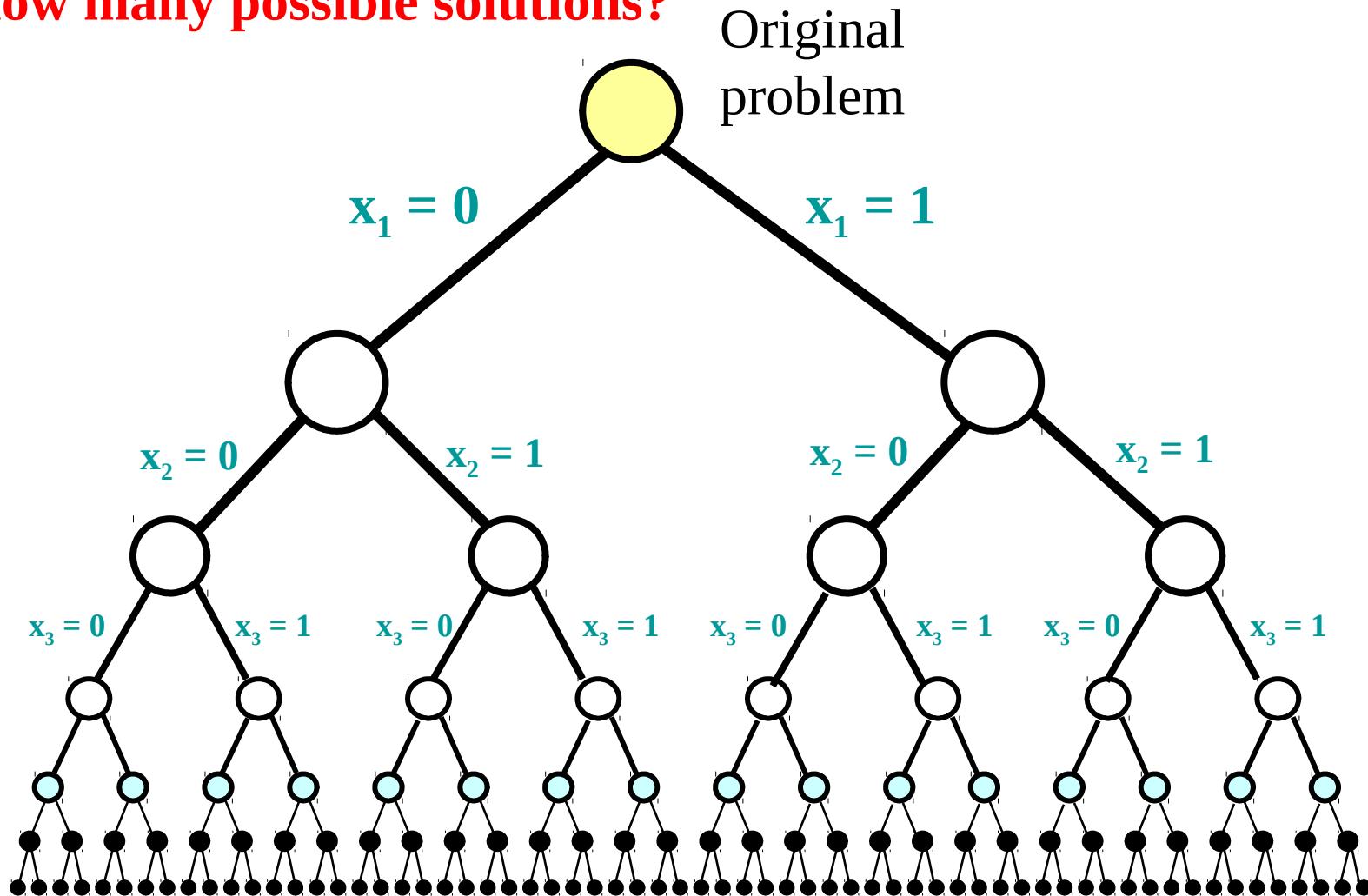
$$x_j \in \{0,1\} \text{ for each } j = 1 \text{ to } 6$$

Complete Enumeration

- Systematically considers all possible values of the decision variables.
 - If there are n binary variables, there are 2^n different ways.
- Usual idea: iteratively break the problem in two. At the first iteration, we consider separately the case that $x_1 = 0$ and $x_1 = 1$.

An Enumeration Tree

How many possible solutions?



On Complete Enumeration

- Suppose that we could evaluate 1 billion solutions per second.
- Let n = number of binary variables
- Solutions times (worst case, approx.)
 - $n = 30$, 1 second
 - $n = 40$, 18 minutes
 - $n = 50$ 13 days
 - $n = 60$ 31 years

On Complete Enumeration

- Suppose that we could evaluate 1 trillion solutions per second, and instantaneously eliminate 99.999999% of all solutions as not worth considering
- Let n = number of binary variables
- Solutions times
 - $n = 70$, 1 second
 - $n = 80$, 17 minutes
 - $n = 90$ 11.6 days
 - $n = 100$ 31 years

Branch and Bound

The essential idea: search the enumeration tree, but at each node

1. Solve the Linear Program (LP) at the node (by relaxing the integrality constraints)
2. Eliminate (Prune) the subtree if
 1. The solution is integer (there is no need to go further- why?) or
 2. The best solution in the subtree cannot be as good as the best available solution (the incumbent- how does that happen?) or
 3. There is no feasible solution

Branch and Bound

1 44 3/7

Node 1 is the Original LP Relaxation

maximize $16x_1 + 22x_2 + 12x_3 + 8x_4 + 11x_5 + 19x_6$

subject to $5x_1 + 7x_2 + 4x_3 + 3x_4 + 4x_5 + 6x_6 \leq 14$

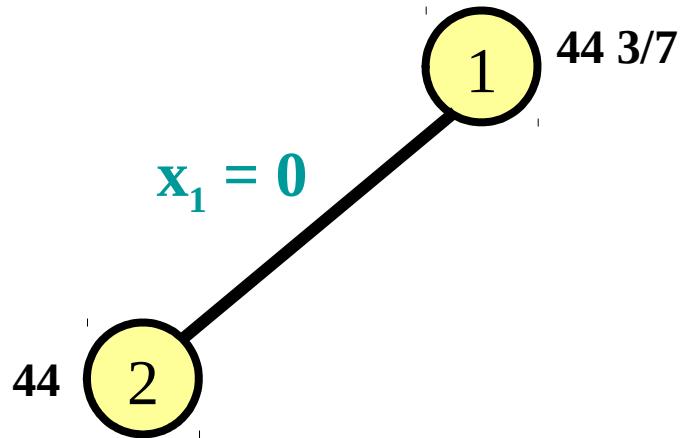
$0 \leq x_j \leq 1$ for $j = 1$ to 6

Solution at node 1:

$x_1 = 1$ $x_2 = 3/7$ $x_3 = x_4 = x_5 = 0$ $x_6 = 1$ $z = 44 \frac{3}{7}$

The IP cannot have value higher than $44 \frac{3}{7}$.

Branch and Bound



Node 2 is the original LP Relaxation plus the constraint $x_1 = 0$.

maximize $16x_1 + 22x_2 + 12x_3 + 8x_4 + 11x_5 + 19x_6$

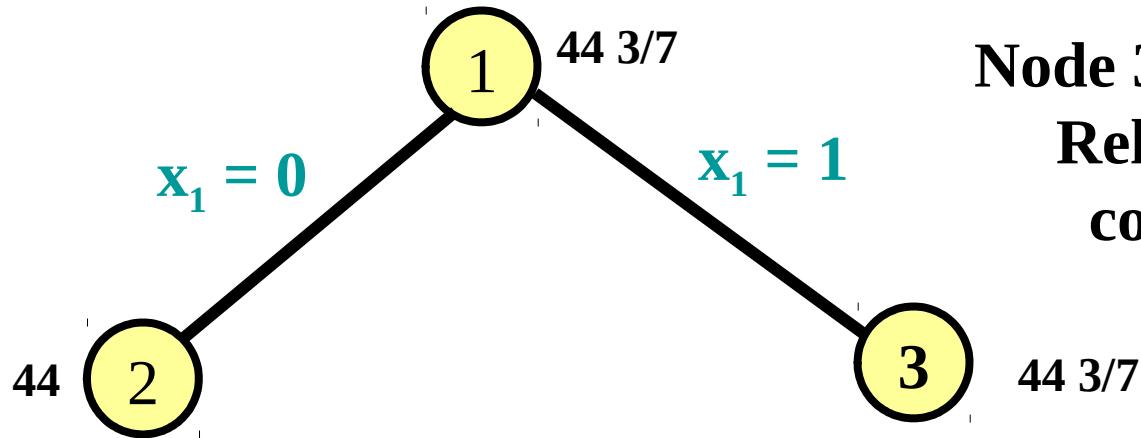
subject to $5x_1 + 7x_2 + 4x_3 + 3x_4 + 4x_5 + 6x_6 \leq 14$

$0 \leq x_j \leq 1$ for $j = 1$ to 6 , $x_1 = 0$

Solution at node 2:

$x_1 = 0$ $x_2 = 1$ $x_3 = 1/4$ $x_4 = x_5 = 0$ $x_6 = 1$ $z = 44$

Branch and Bound



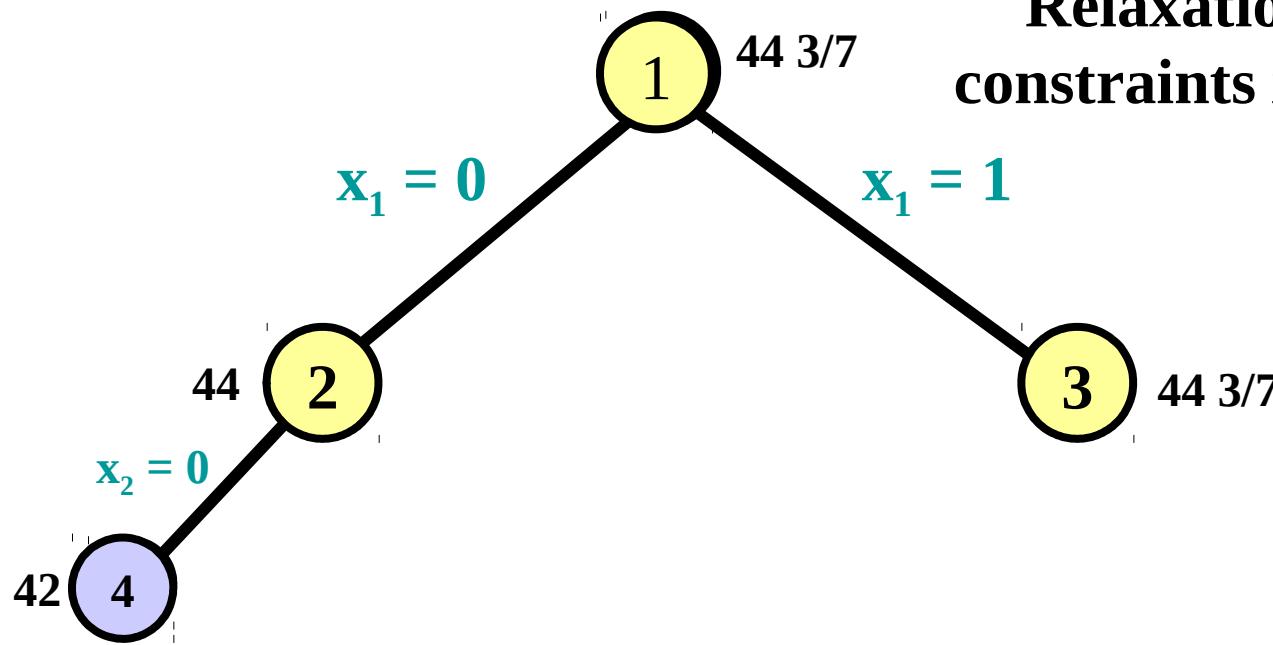
Node 3 is the original LP Relaxation plus the constraint $x_1 = 1$.

The solution at node 1 was

$$x_1 = 1 \quad x_2 = 3/7 \quad x_3 = x_4 = x_5 = 0 \quad x_6 = 1 \quad z = 44 \frac{3}{7}$$

Note: it was the best solution with no constraint on x_1 . So, it is also the solution for node 3. (If you add a constraint, and the old optimal solution is feasible, then it is still optimal.)

Branch and Bound



Node 4 is the original LP Relaxation plus the constraints $x_1 = 0, x_2 = 0$.

Solution at node 4: 0 0 1 0 1 1 $z = 42$

Our first *incumbent* solution!

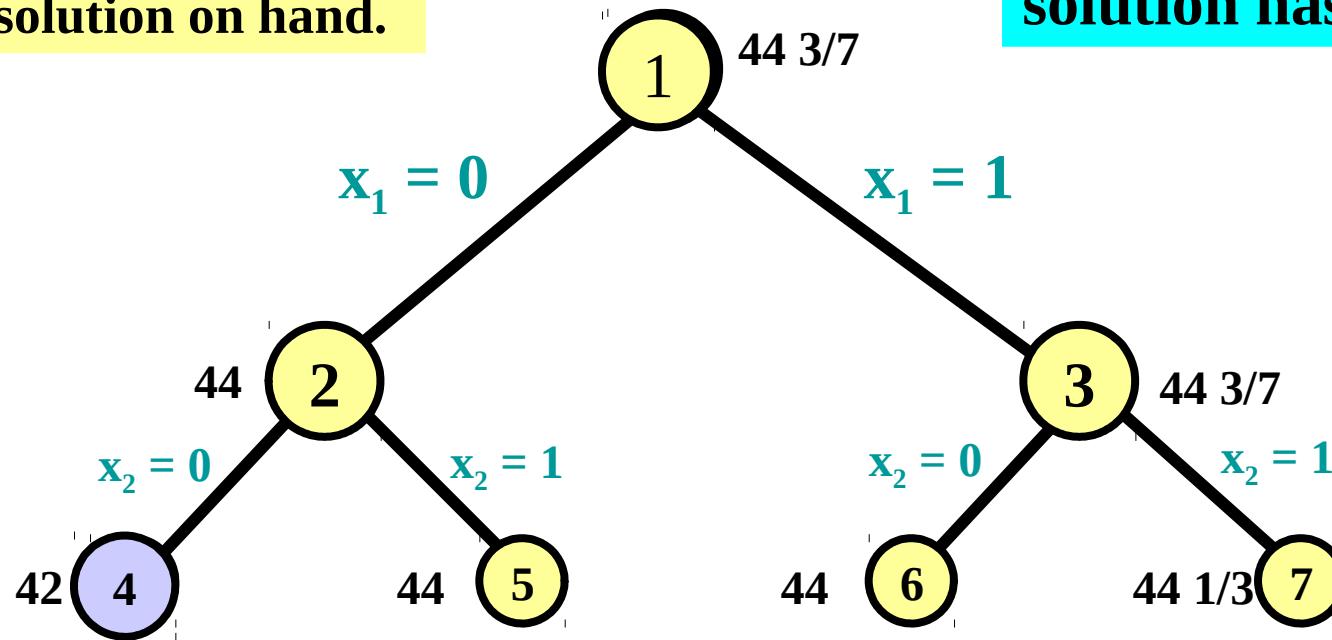
No further searching from node 4 because there cannot be a better integer solution.

No solution in the subtree can have a value better than 42.

Branch and Bound

The incumbent is the best solution on hand.

The incumbent solution has value 42



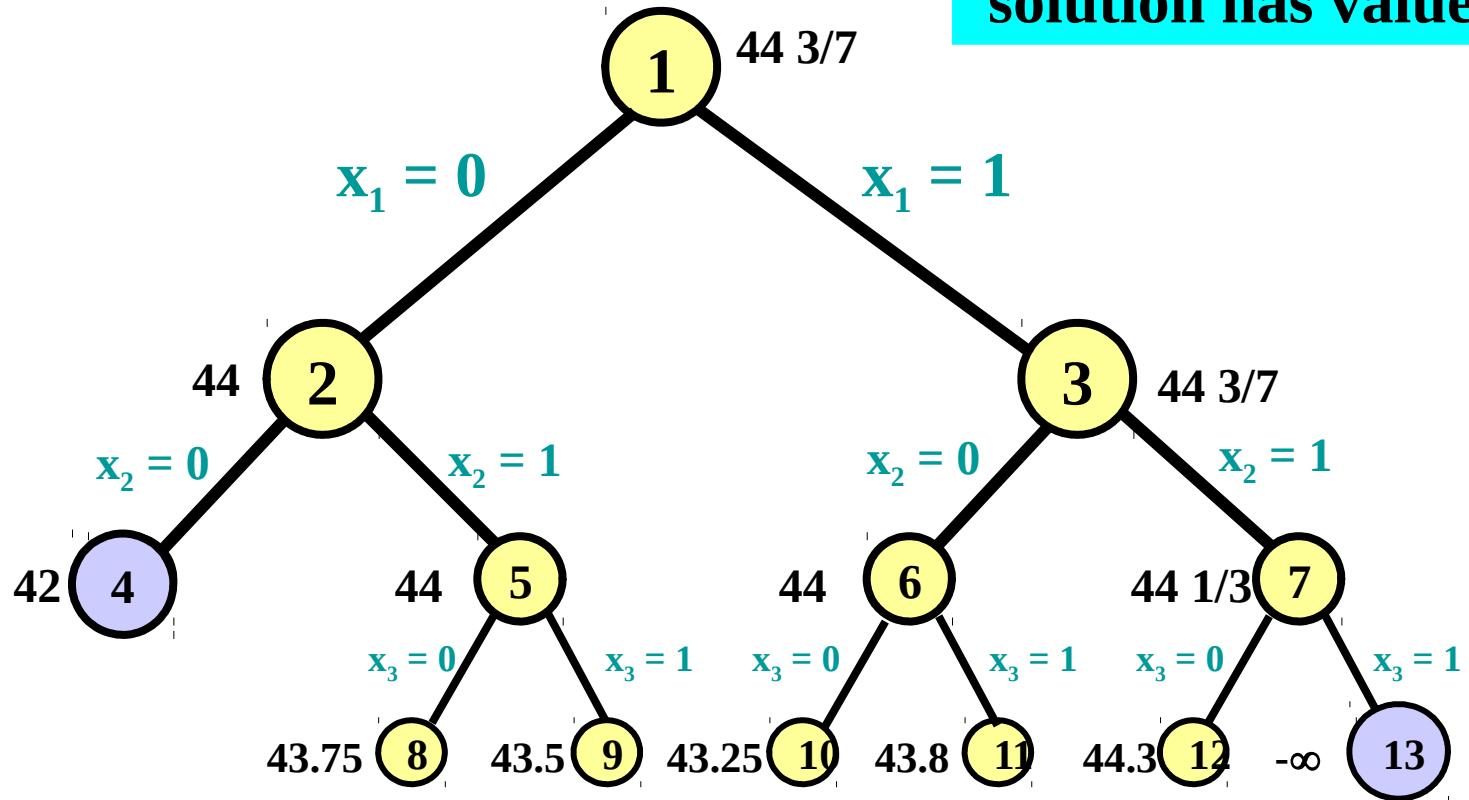
We next solved the LP's associated with nodes 5, 6, and 7.

No new integer solutions were found.

We would eliminate (prune) a subtree if we were guaranteed that no solution in the subtree were better than the incumbent.

Branch and Bound

The incumbent solution has value 42

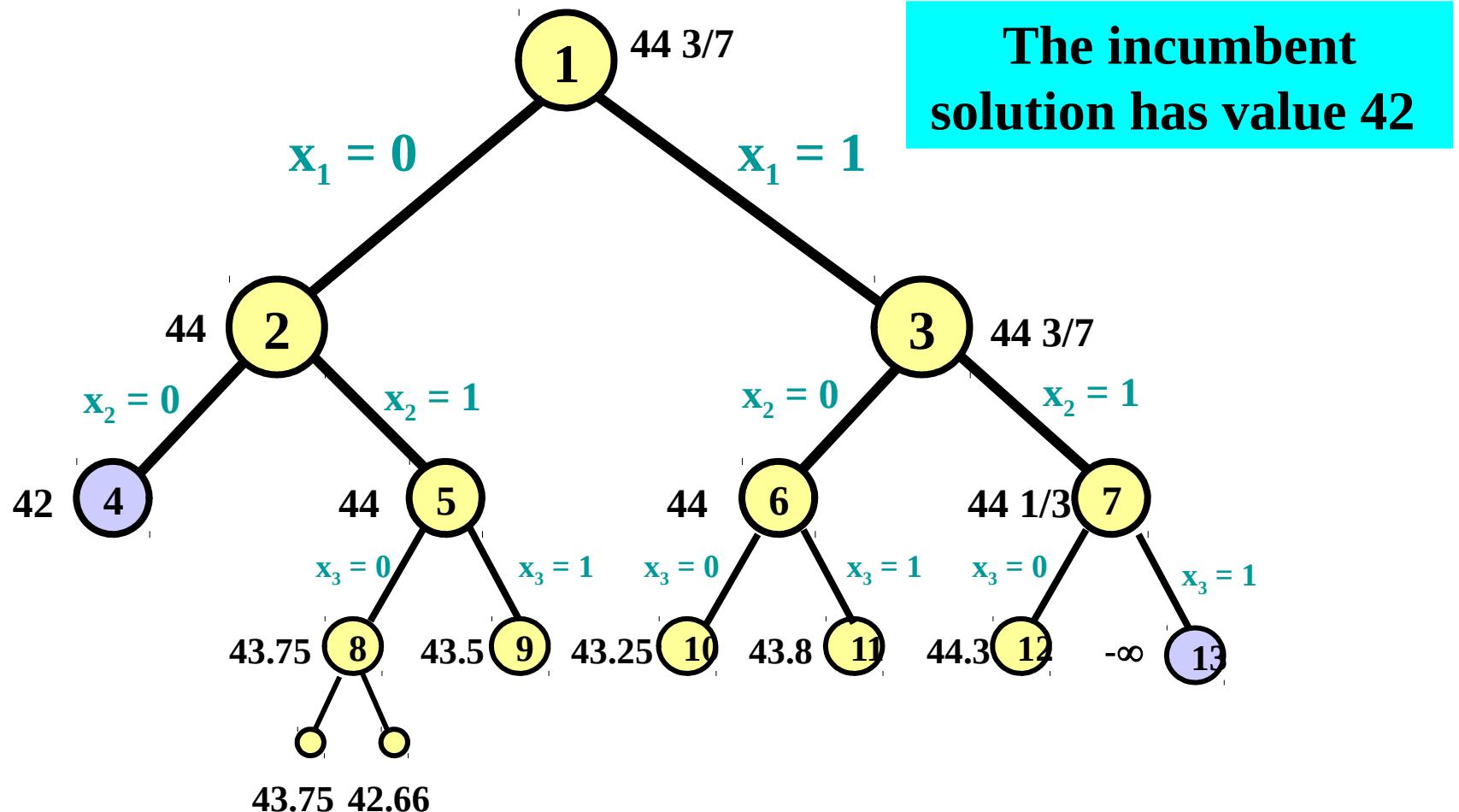


We next solved the LPs associated with nodes 8-13

Summary so far

- We have solved 13 different linear programs so far.
 - One integer solution was found
 - One subtree pruned because the solution was integer (node 4)
 - One subtree pruned because the solution was infeasible (node 13)
 - No subtrees pruned because of the bound

Branch and Bound



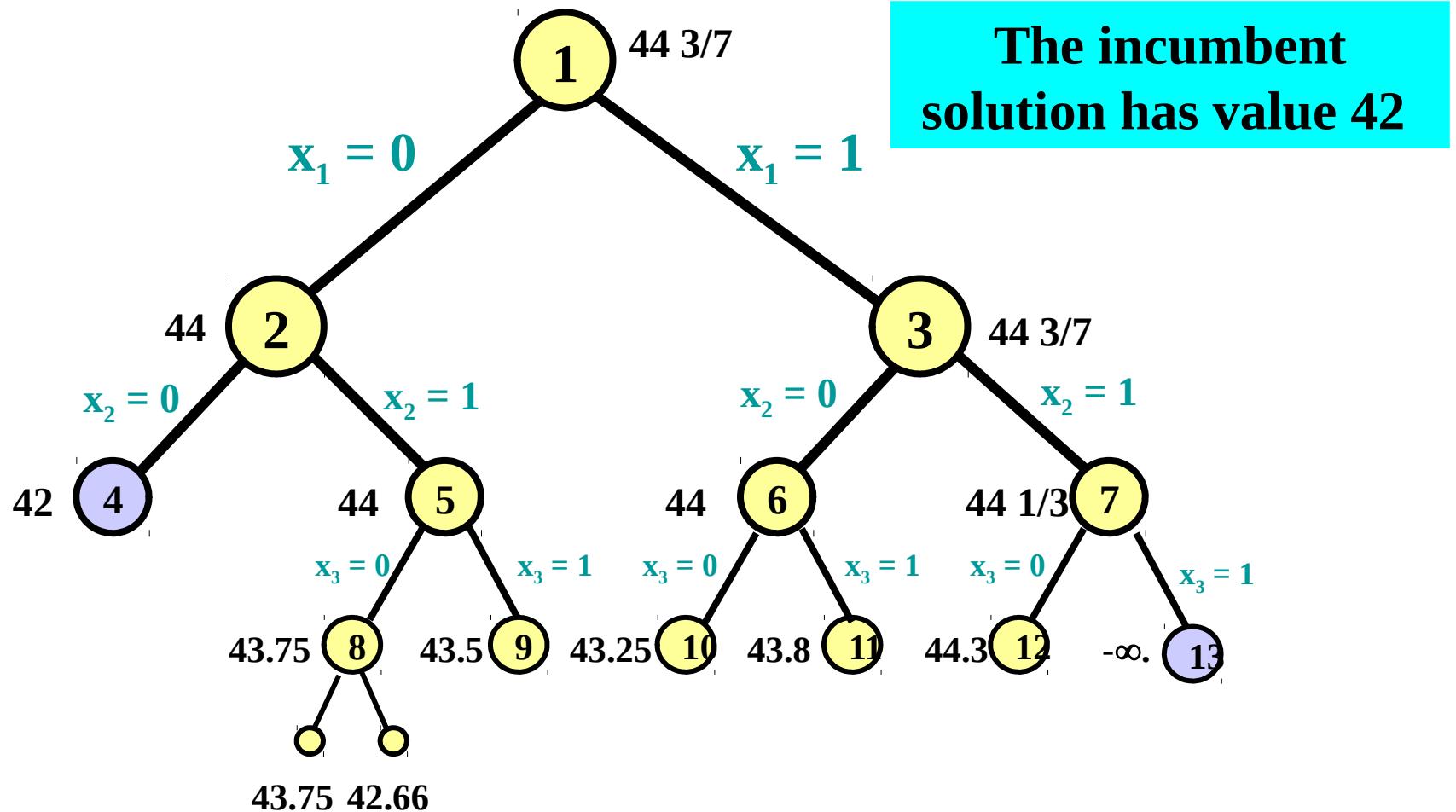
We next solved the LPs associated with the next nodes.

We can prune the node with $z = 42.66$. Why?

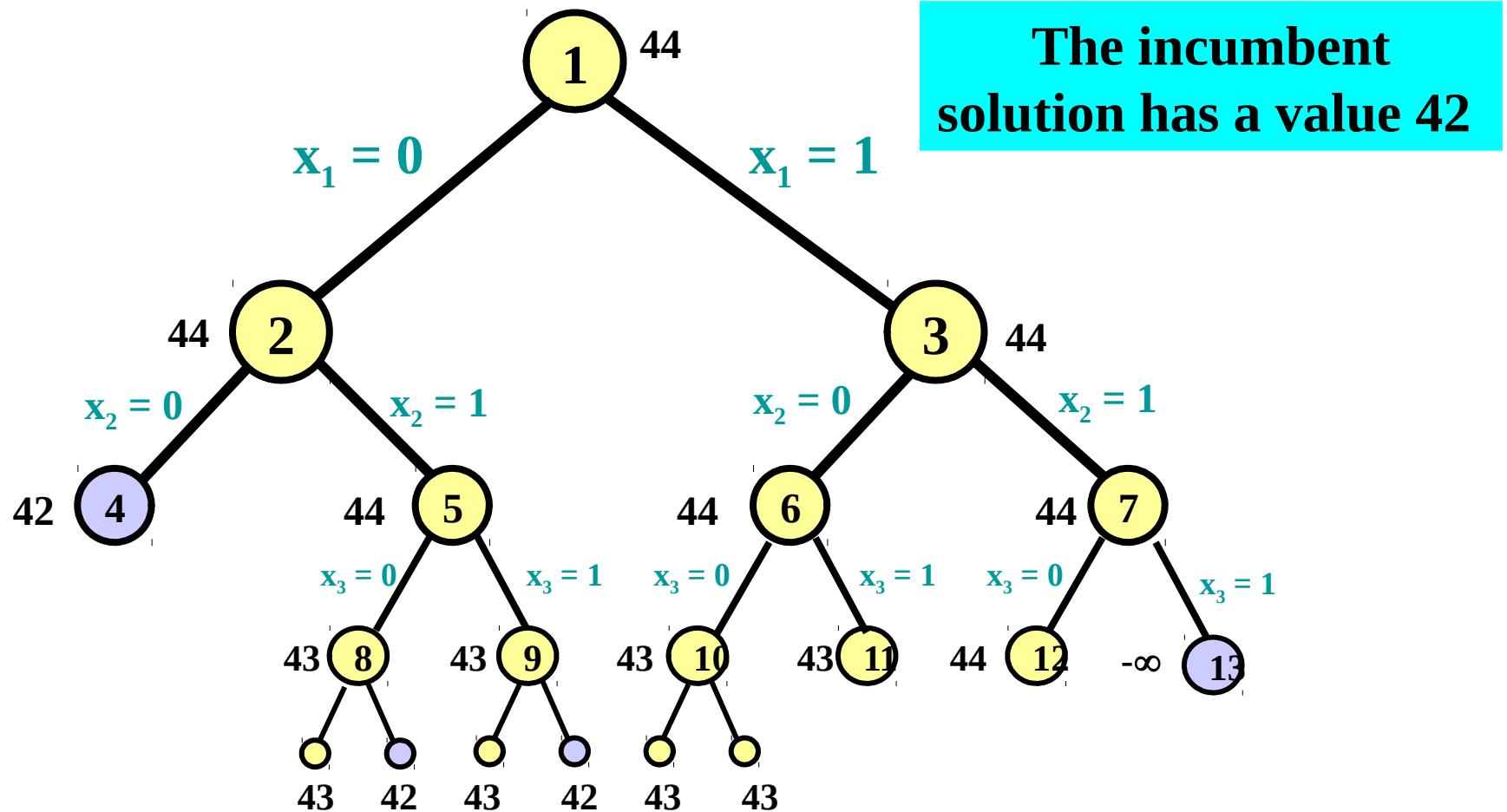
Getting a Better Bound

- The bound at each node is obtained by solving an LP.
- But all costs are integer, and so the objective value of each integer solution is integer. So, the best integer solution has an integer objective value.
- If the best integer valued solution for a node is at most 42.66, then we know the best bound is at most 42.
- Other bounds can also be rounded down.

Branch and Bound



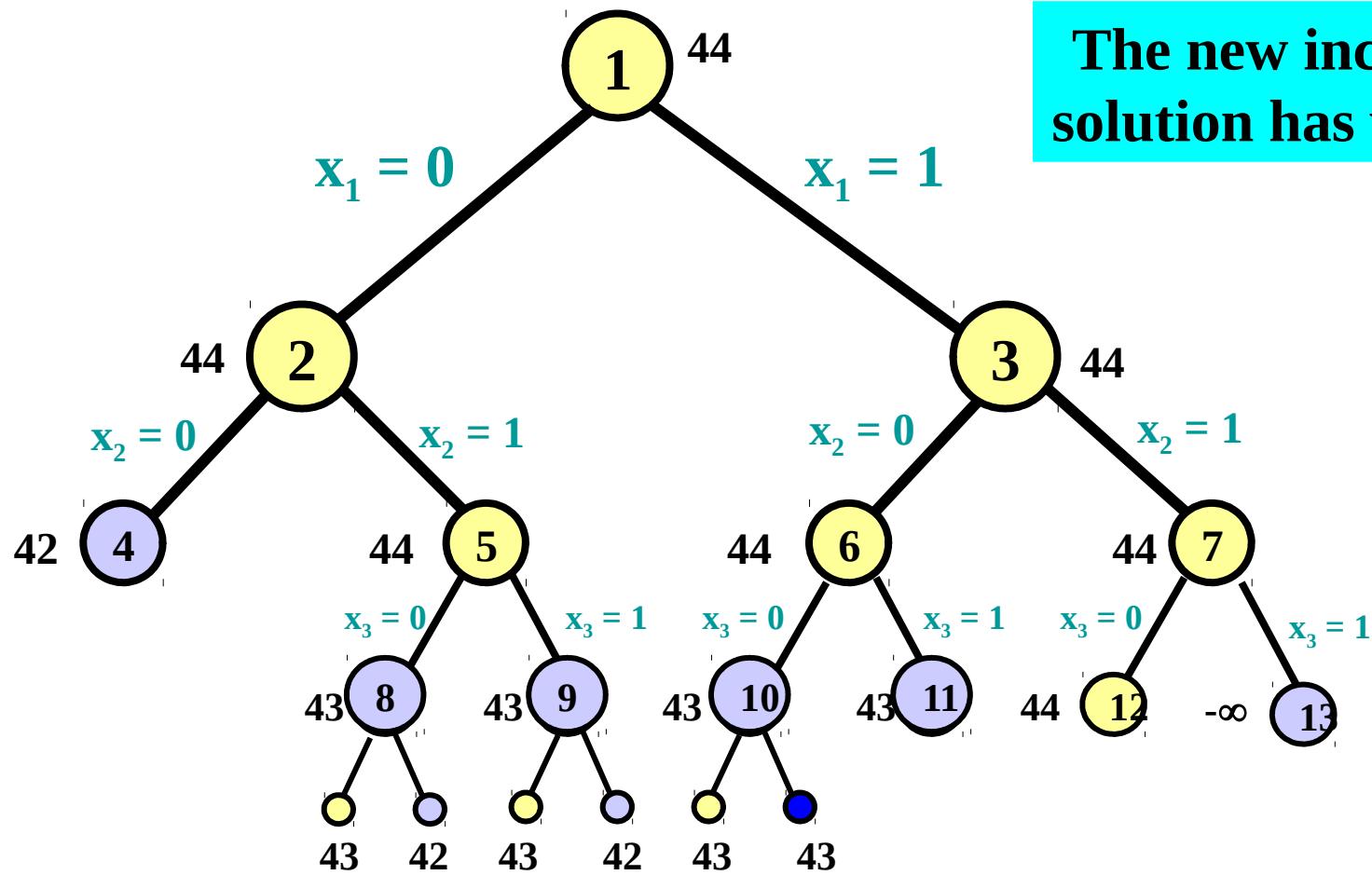
Branch and Bound



We found a new incumbent solution!

$$x_1 = 1, x_2 = x_3 = 0, x_4 = 1, x_5 = 0, x_6 = 1 \quad z = 43$$

Branch and Bound

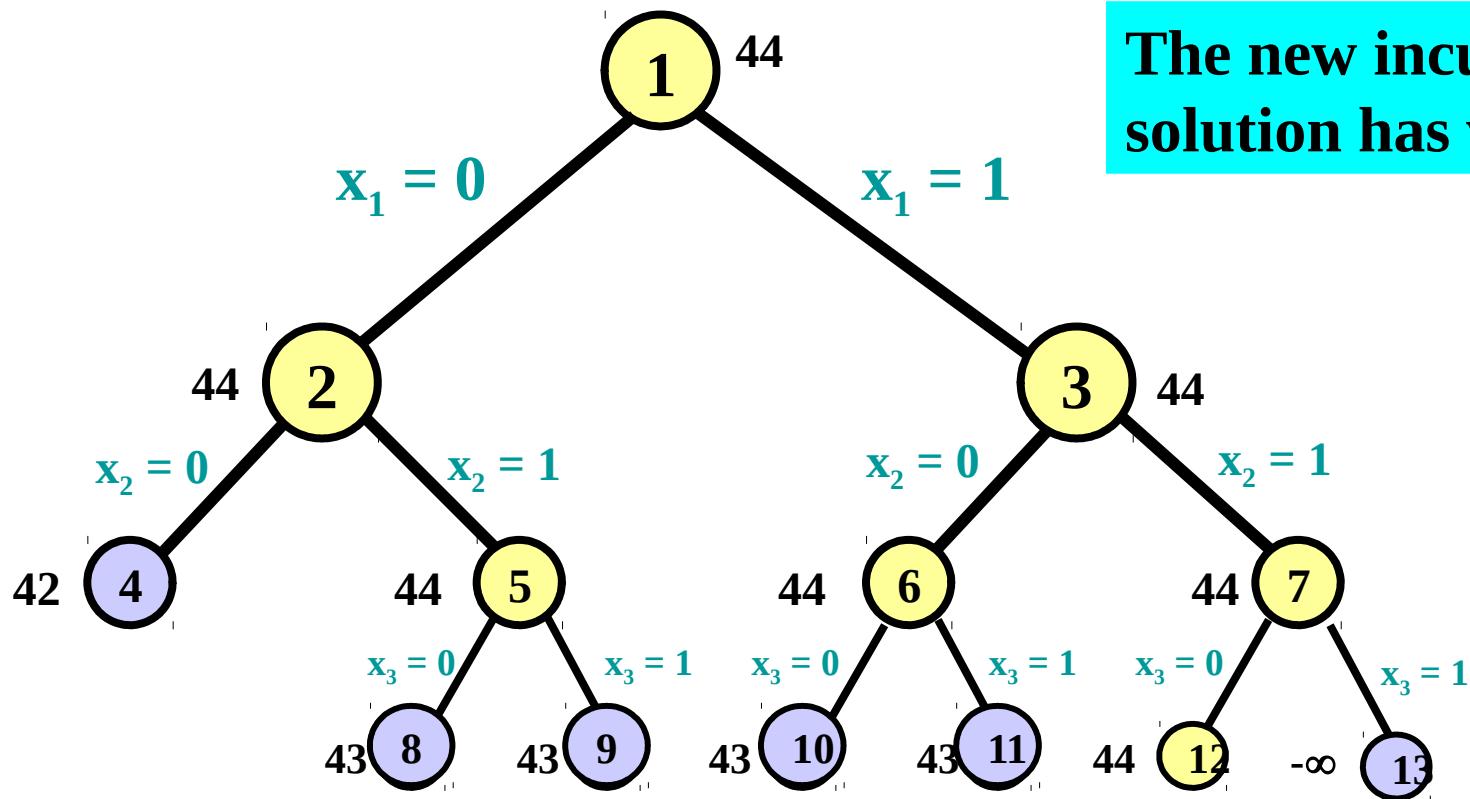


The new incumbent solution has value 43

We found a new incumbent solution!

$$x_1 = 1, x_2 = x_3 = 0, x_4 = 1, x_5 = 0, x_6 = 1 \quad z = 43$$

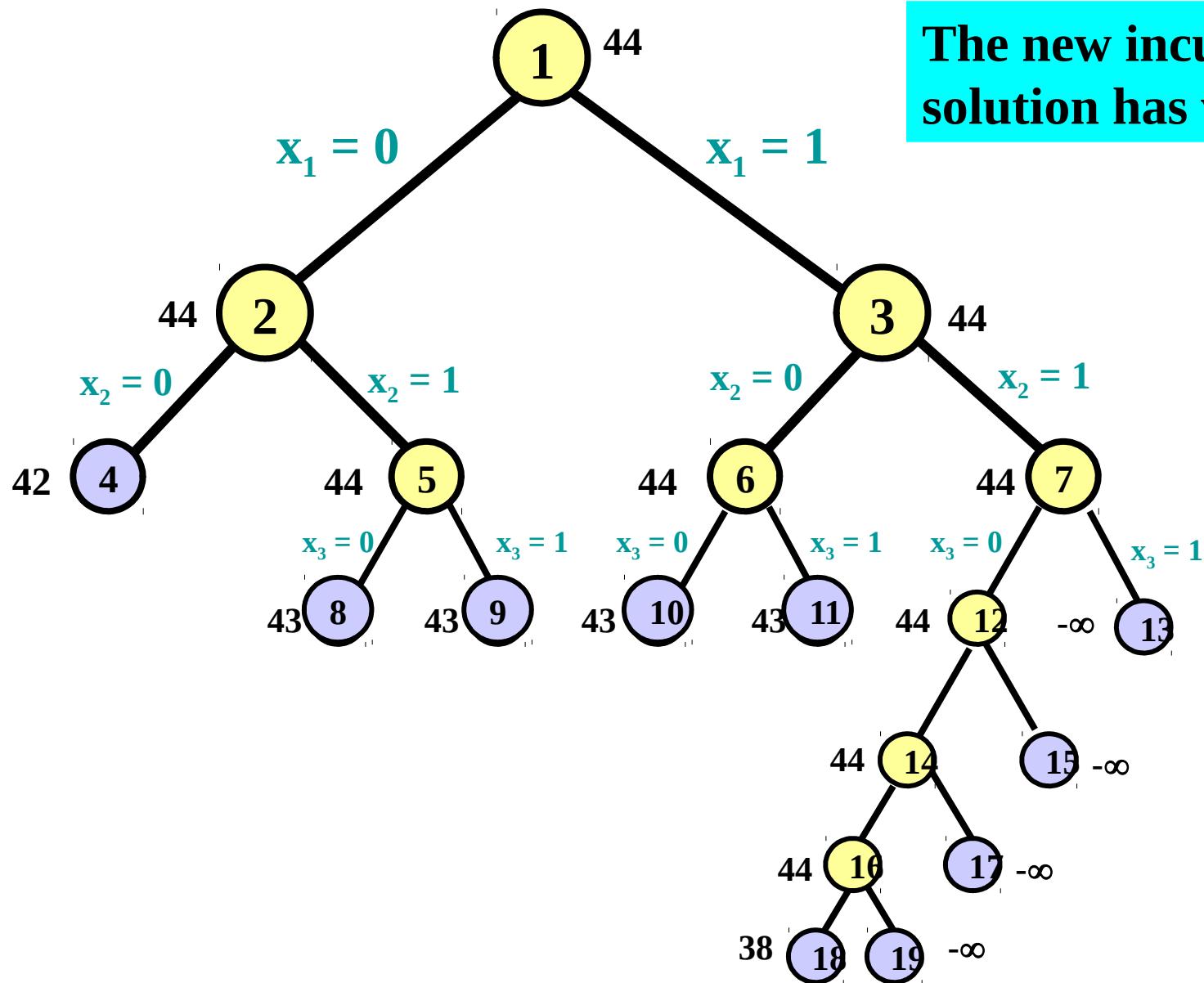
Branch and Bound



The new incumbent solution has value 43

If we had found this incumbent solution earlier,
we could have saved some searching.

Finishing Up



The new incumbent solution has value 43

Key Observations

- Branch and Bound can speed up the search process
 - Only 25 nodes (linear programs) were evaluated
 - Other nodes were pruned
- Obtaining a good incumbent earlier can be valuable
 - only 19 nodes would have been evaluated.
- Solve linear programs faster, because we start with an excellent or optimal solution
- Obtaining better bounds can be valuable.
 - We sometimes use properties that are obvious to us, such as the fact that integer solutions have integer solution values

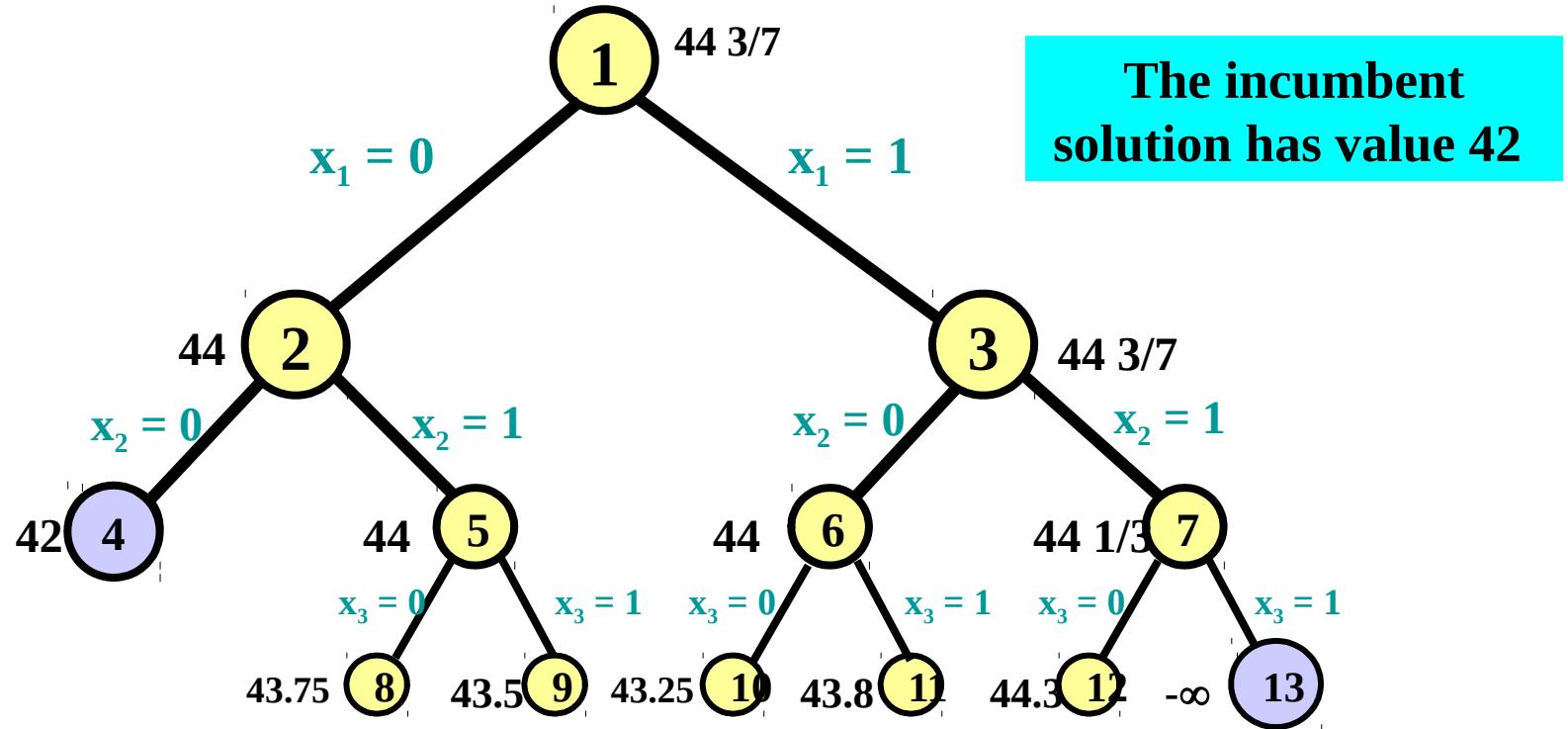
Branch and Bound

Notations:

- z^* = optimal integer solution value
- Subdivision: a node of the B&B Tree
- Incumbent: the best solution on hand
- z^I : value of the incumbent
- z^{LP} : value of the LP relaxation of the current node
- Children of a node: the two problems created for a node, e.g., by saying $x_j = 1$ or $x_j = 0$.
- LIST: the collection of active (not fathomed) nodes, with no active children.

NOTE: $z^I \leq z^*$

Illustrating the Definitions



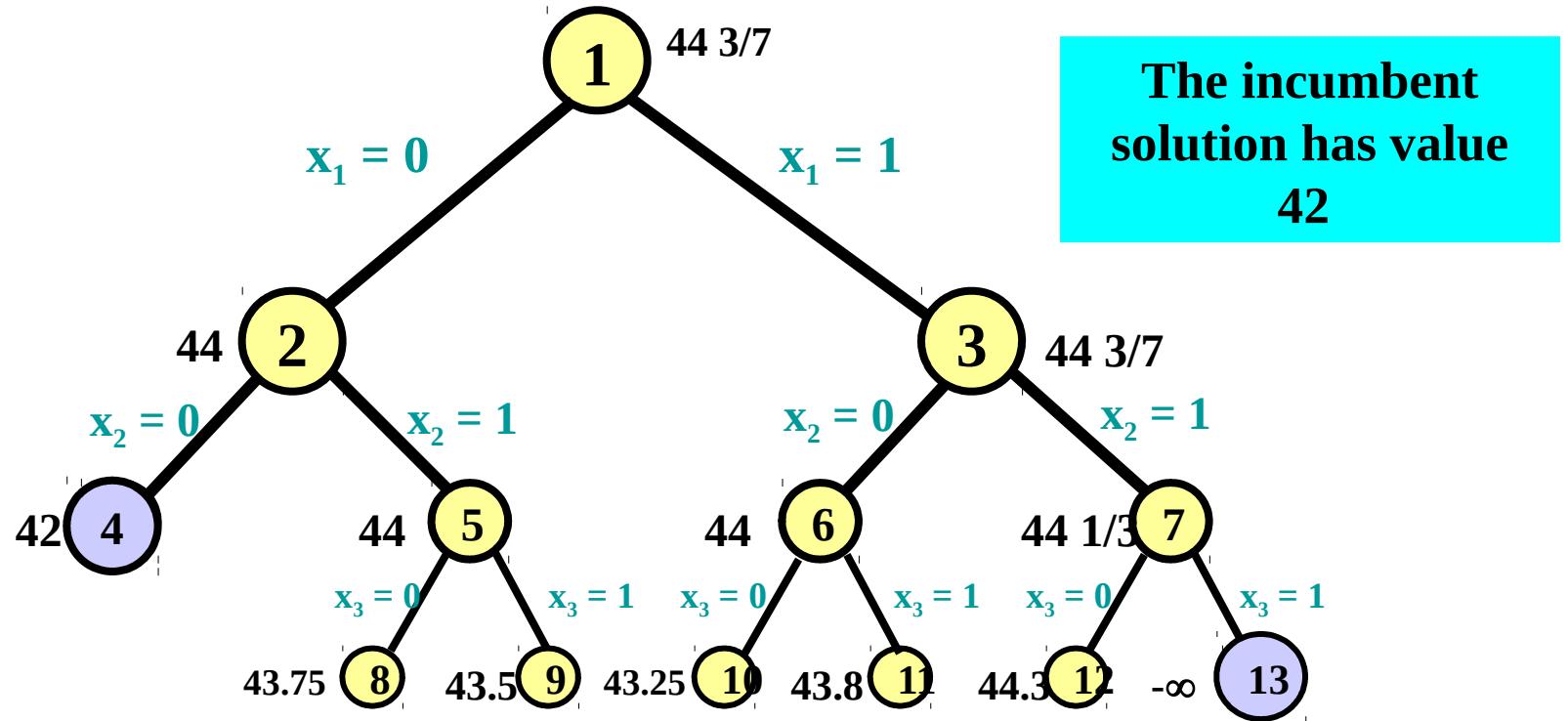
$z^* = 43$ = optimal integer solution value. (We found it later in the search)

Incumbent is $0 \quad 0 \quad 1 \quad 0 \quad 1 \quad 1$ $z^I = 42$.

It is the optimal solution for the subdivision 4.

The z^{LP} values for each subdivision are next to the nodes.

Illustrating the Definitions



The incumbent solution has value
42

The children of node (subdivision) 1 are nodes 2 and 3.
The children of node 3 are nodes 6 and 7.
LIST = { 8, 9, 10, 11, 12 } = unpruned nodes with no active children

Branch and Bound Algorithm

INITIALIZE LIST = {original problem}

Incumbent: = \emptyset

$z^I = -\infty$

SELECT:

If LIST = \emptyset , then the Incumbent is optimal if it exists,
and the problem is infeasible if no incumbent exists;
else, let S be a node (subdivision) from LIST.

Let x^{LP} be the optimal solution to S

Let z^{LP} = its objective value

e.g., S = {1}

44 3/7 1

e.g., S = {13}

$-\infty$ 13

CASE 1. $z^{LP} = -\infty$ (the LP is infeasible)

Remove S from LIST (prune it)

Return to SELECT

Branch and Bound Algorithm

INITIALIZE

SELECT:

If $\text{LIST} = \emptyset$, then the Incumbent is optimal (if it exists),
and the problem is infeasible if no incumbent exists;
else, let S be a node from LIST .

Let x^{LP} be the optimal solution to S

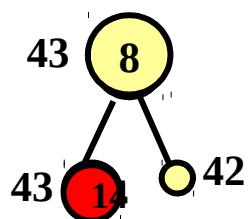
Let $z^{\text{LP}} =$ its objective value

CASE 2. $-\infty < z^{\text{LP}} \leq z^I$.

That is, the LP is dominated by the incumbent.

Then remove S from LIST (fathom it)

Return to **SELECT**



e.g., the incumbent has value 43, and node 14 is selected. $z^{\text{LP}} = 43$.

Branch and Bound Algorithm

INITIALIZE

SELECT:

If $\text{LIST} = \emptyset$, then the Incumbent is optimal (if it exists),
and the problem is infeasible if no incumbent exists;
else, let S be a subdivision from LIST .

Let x^{LP} be the optimal solution to S

Let z^{LP} = its objective value

CASE 3. $z^I < z^{\text{LP}}$ and x^{LP} is integral.

**That is, the LP solution is integral and
dominates the incumbent.**

e.g., node 4 was selected,
and the solution to the
LP was integer-valued.

Then Incumbent := x^{LP} ;
 $z^I := z^{\text{LP}}$

Remove S from LIST (fathomed by integrality)
Return to SELECT



Branch and Bound Algorithm

INITIALIZE

SELECT:

If $\text{LIST} = \emptyset$, then the Incumbent is optimal (if it exists),
and the problem is infeasible if no incumbent exists;
else, let S be a subdivision from LIST .

Let x^{LP} be the optimal solution to S

Let z^{LP} = its objective value

e.g., select node 3.

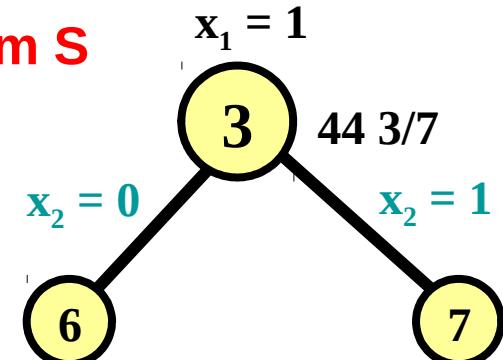
CASE 4. $z^I < z^{\text{LP}}$ and x^{LP} is not integral.

There is not enough information to fathom S

Remove S from LIST

Add the children of S to LIST

Return to SELECT



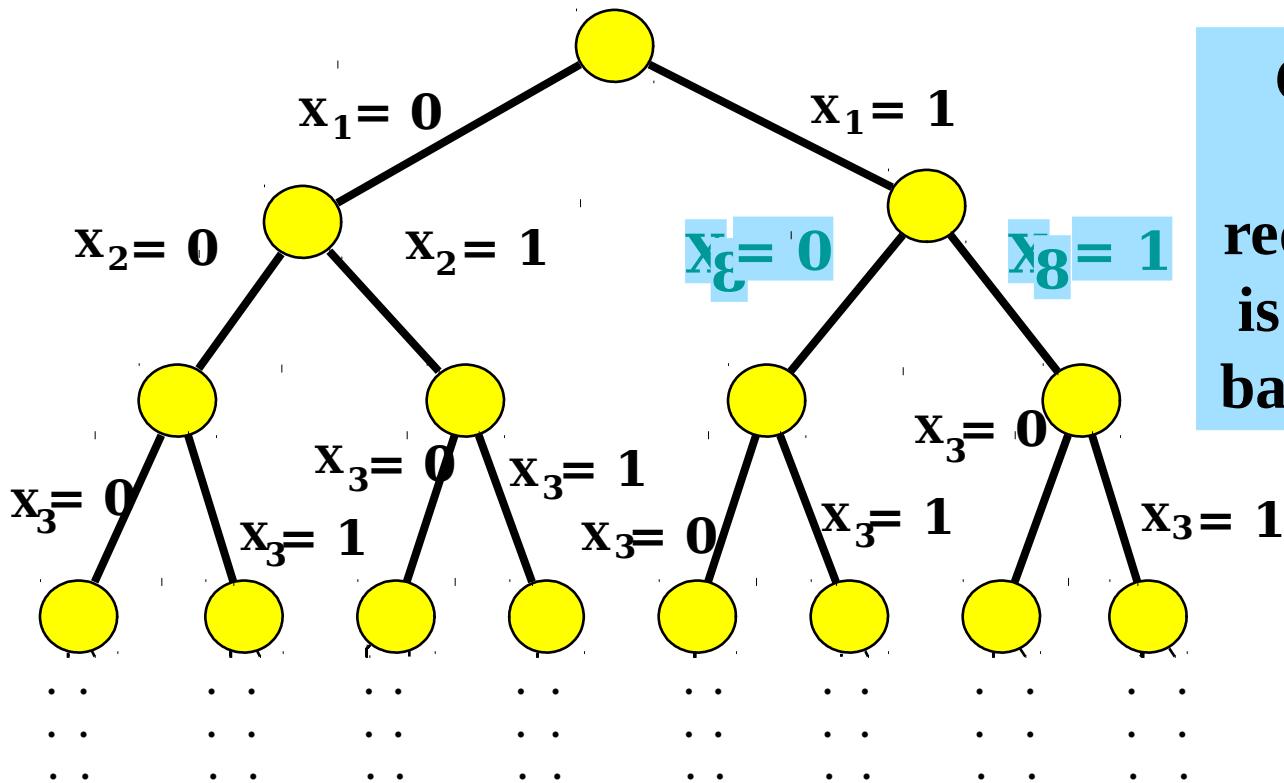
List := List – 3 + {6,7}

Possible Selection Rules

- **Rule of Thumb 1:** Don't let LIST get too big (the solutions must be stored). So, prefer nodes that are further down in the tree.
- **Rule of Thumb 2:** Pick a node of LIST that is likely to lead to an improved incumbent. Sometimes special heuristics are used to come up with a good incumbent.

Branching

One does not have to have the B&B tree be symmetric, and one does not select subtrees by considering variables in order.



Choosing how to branch so as to reduce running time is largely “art” and based on experience.

Possible Branching Rules

- Branching: determining children for a node. There are many choices (Rules of Thumb).
- Rule 1: if it appears clear that $x_j = 1$ in an optimal solution, it is often good to branch on $x_j = 0$ vs $x_j = 1$.
 - The hope is that a subdivision with $x_j = 0$ can be pruned.
- Rule 2: branching on important variables is worthwhile

Possible Bounding Techniques

- We use the bound obtained by dropping the integrality constraints (LP relaxation). There are other choices.
- Key tradeoff for bounds: time to obtain a bound vs quality of the bound.
- If one can obtain a bound much quicker, sometimes we would be willing to get a bound that is worse
- It usually is worthwhile to get a bound that is better, so long as it does not take too long.

What if the Variables are General Integer Variables?

- One can choose children as follows:
 - child 1: $x_1 \leq 3$ (or $x_j \leq k$)
 - child 2 $x_1 \geq 4$ (or $x_j \geq k+1$)
- How would one choose the variable j and the value k
 - A common choice would be to take a fractional value from x^{LP} . e.g., if $x_7 = 5.62$, then we may branch on $x_7 \leq 5$ and $x_7 \geq 6$.
 - Other choices are also possible.

Summary

- Branch and Bound is the standard way of solving IPs to optimality.
- There is art to making it work well in practice.
- Much of the art is built into state-of-the-art solvers such as CPLEX.

Dynamic Programming

Algorithmic Paradigms

- **Greedy.** Build up a solution incrementally, myopically optimizing some local criterion.
- **Divide-and-Conquer.** Break up a problem into non-overlapping (independent) sub-problems, solve each sub-problem, and then combine these solutions of sub-problems to form the solution to original problem.
- **Dynamic Programming.** Break up a problem into a series of **overlapping** sub-problems, and then combine solutions of smaller sub-problems to form the solution to a larger sub-problem.

Dynamic Programming History

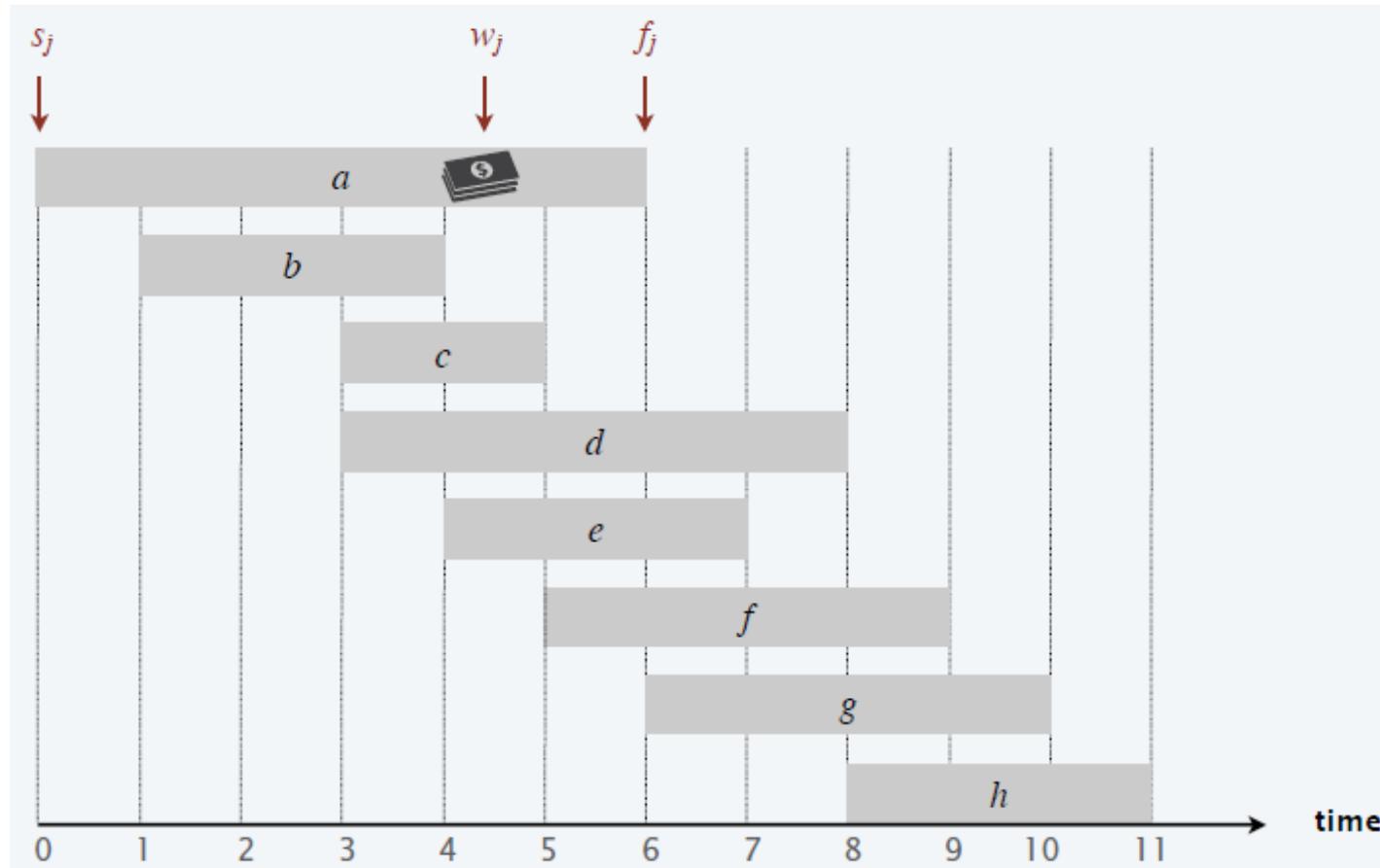
- **Bellman.** Pioneered the systematic study of dynamic programming in the 1950s.
- **Etymology:** It is the history of words, their origins, and how their form and *meaning* have changed over time. The *etymology* of [a word]" means the origin of the particular word.
 - Dynamic Programming = Planning over time.
 - Secretary of Defense was hostile to mathematics research.
 - Bellman sought an impressive name to avoid confrontation.
 - "it's impossible to use dynamic in a pejorative sense"
 - "something not even a Congressman could object to"

Dynamic Programming Applications

- Application Areas:
 - Bioinformatics.
 - Control Theory.
 - Information Theory.
 - Operations Research.
 - **Computer Science: AI, Compilers, Graphics, Systems, Theory,**
- Some Famous Dynamic Programming Algorithms:
 - Viterbi for Hidden Markov Models.
 - Unix diff for Comparing Two Files.
 - Knuth–Plass for word wrapping text in
 - Needleman-Wunsch/Smith-Waterman \TeX sequence Alignment.
 - **Bellman-Ford-Moore for Shortest Path Routing in Networks.**
 - Cocke-Kasami-Younger for Parsing Context Free Grammars.

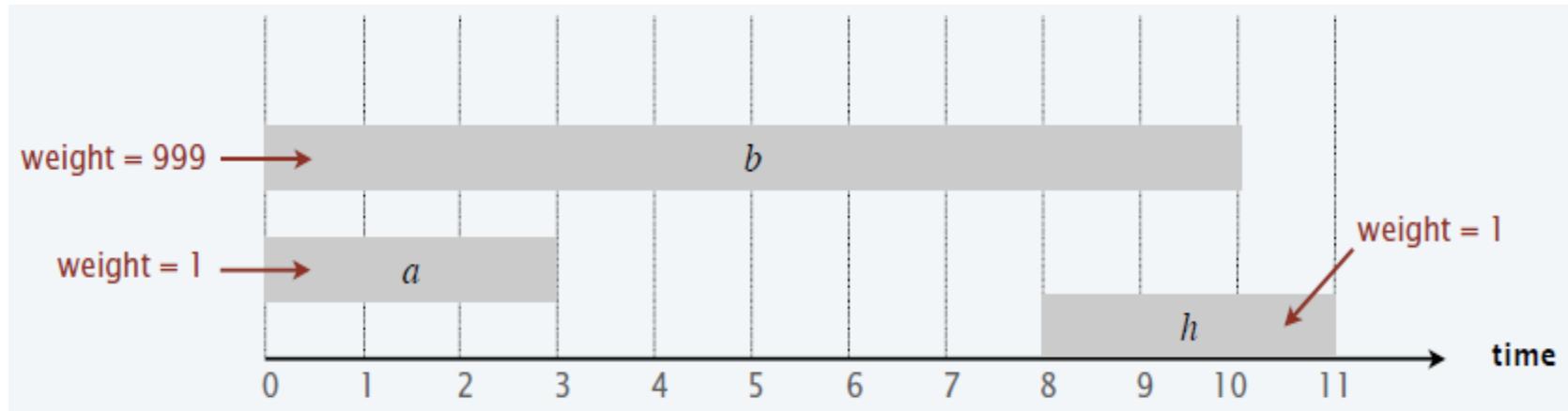
Weighted Interval Scheduling

- Weighted Interval Scheduling Problem.
 - Job j starts at s_j , finishes at f_j , and has weight or value $w_j > 0$.
 - Two jobs are **Compatible** if they don't overlap.
 - Goal: To find the **Maximum Weight** subset of Mutually Compatible Jobs.



Unweighted Interval Scheduling (Earliest-Finish-Time First Algorithm)

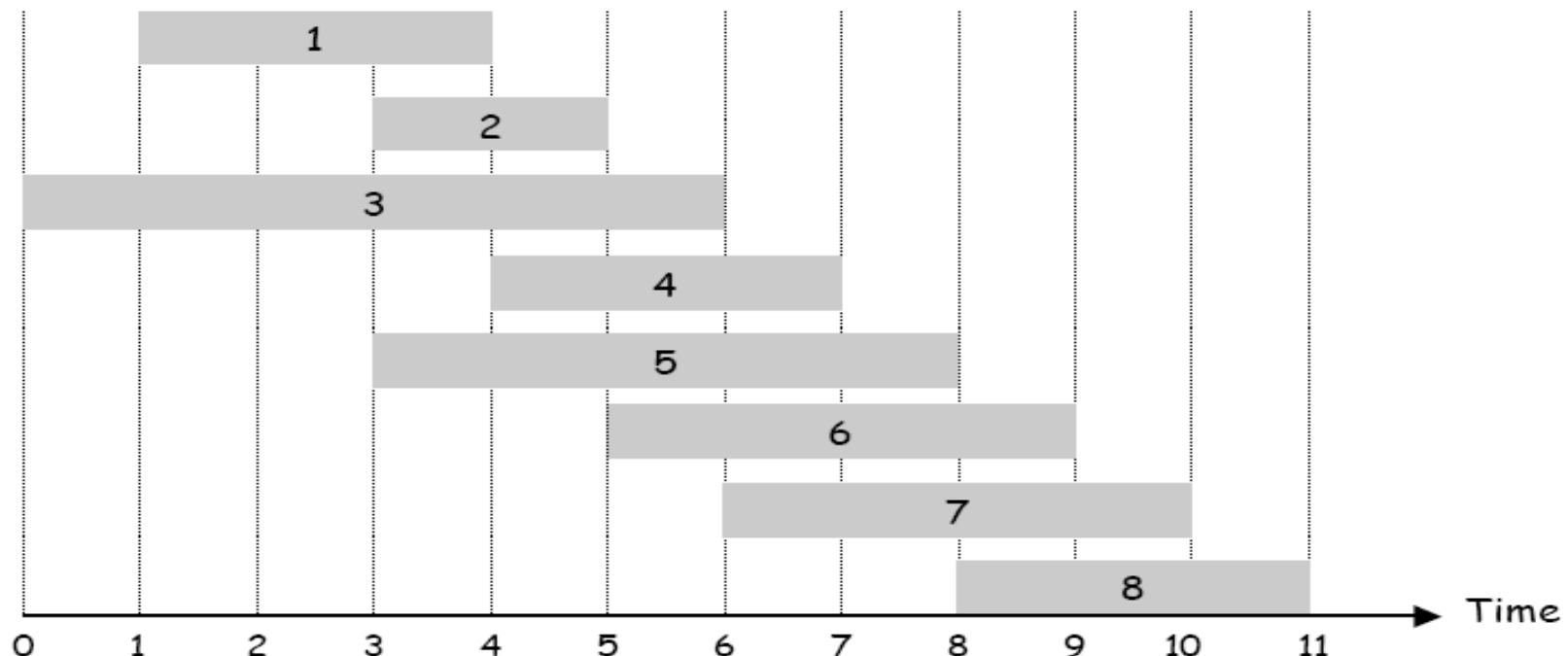
- Earliest-Finish-Time First.
 - Consider jobs in ascending order of finish time.
 - Add job to subset if it is compatible with previously chosen jobs
- Recall. Greedy Algorithm works if all weights are 1.
- Observation. Greedy Algorithm fails spectacularly for weighted version i.e. if arbitrary weights are allowed.



Weighted Interval Scheduling

- Notation. Label jobs by finishing time: $f_1 \leq f_2 \leq \dots \leq f_n$.
- Def. $p(j)$ = Largest index $i < j$ such that job i is compatible with j .
- Ex: $p(8) = 5$, $p(7) = 3$, $p(2) = 0$.

i is rightmost interval that ends before j begins



Dynamic Programming: Binary Choice

Def. $OPT(j)$ = max weight of any subset of mutually compatible jobs for subproblem consisting only of jobs $1, 2, \dots, j$.

Goal. $OPT(n)$ = max weight of any subset of mutually compatible jobs.

Case 1. $OPT(j)$ does not select job j .

- Must be an optimal solution to problem consisting of remaining jobs $1, 2, \dots, j - 1$.

Case 2. $OPT(j)$ selects job j .

- Collect profit w_j .
- Can't use incompatible jobs $\{ p(j) + 1, p(j) + 2, \dots, j - 1 \}$.
- Must include optimal solution to problem consisting of remaining compatible jobs $1, 2, \dots, p(j)$.

optimal substructure property
(proof via exchange argument)

Bellman equation. $OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max \{ OPT(j - 1), w_j + OPT(p(j)) \} & \text{if } j > 0 \end{cases}$

Weighted Interval Scheduling: Brute Force

BRUTE-FORCE ($n, s_1, \dots, s_n, f_1, \dots, f_n, w_1, \dots, w_n$)

Sort jobs by finish time and renumber so that $f_1 \leq f_2 \leq \dots \leq f_n$.

Compute $p[1], p[2], \dots, p[n]$ via binary search.

RETURN COMPUTE-OPT(n).

COMPUTE-OPT(j)

IF ($j = 0$)

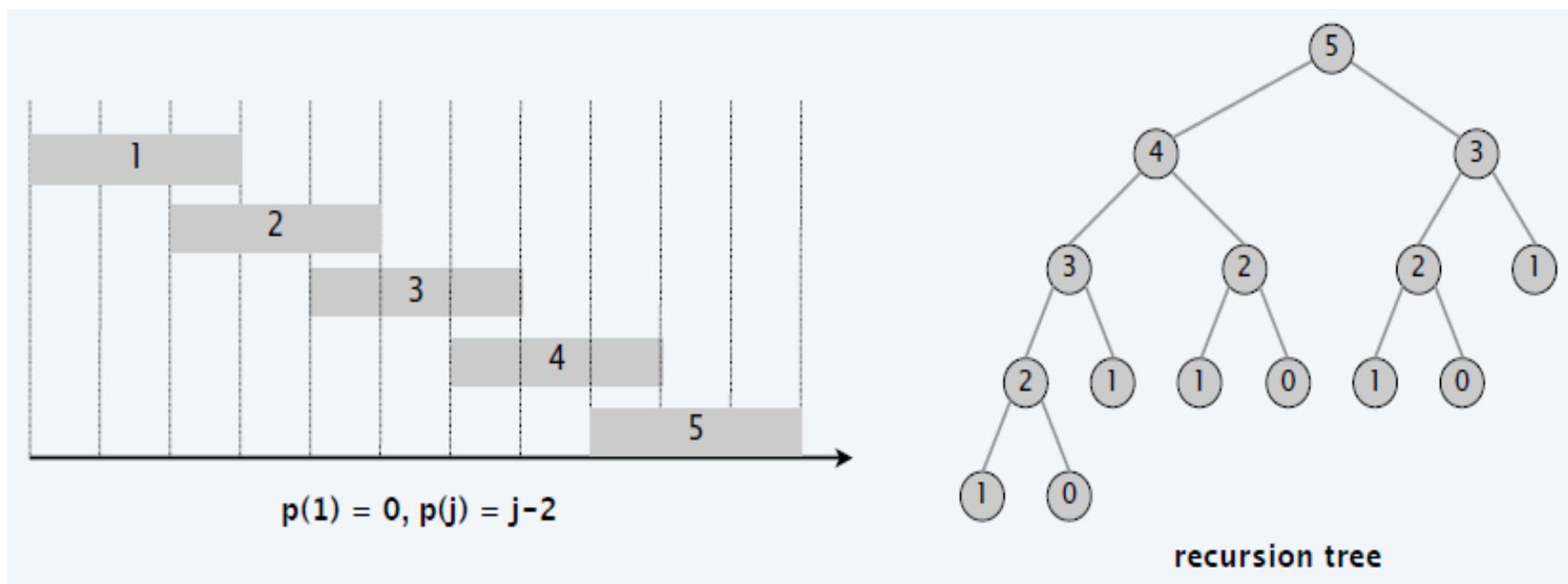
 RETURN 0.

ELSE

 RETURN $\max \{ \text{COMPUTE-OPT}(j-1), w_j + \text{COMPUTE-OPT}(p[j]) \}$.

Weighted Interval Scheduling: Brute Force

- **Observation.** Recursive algorithm is spectacularly slow because of overlapping sub-problems => exponential-time algorithm.
- **Ex.** Number of recursive calls for family of "layered" instances grows like Fibonacci sequence.



Weighted Interval Scheduling: Memoization

- Top-Down Dynamic Programming (Memoization): It is an Optimization Technique used mainly to speed up programs by storing the results of expensive function calls and returning the cached result ($M[j]$) when the same inputs occur again, thus avoids solving the sub-problem j more than once. Store results of each sub-problem j in a cache ($M[j]$); lookup table as needed.

TOP-DOWN($n, s_1, \dots, s_n, f_1, \dots, f_n, w_1, \dots, w_n$)

Sort jobs by finish time and renumber so that $f_1 \leq f_2 \leq \dots \leq f_n$.

Compute $p[1], p[2], \dots, p[n]$ via binary search.

$M[0] \leftarrow 0$. \longleftarrow global array

RETURN M-COMPUTE-OPT(n).

M-COMPUTE-OPT(j)

IF ($M[j]$ is uninitialized)

$M[j] \leftarrow \max \{ \text{M-COMPUTE-OPT}(j-1), w_j + \text{M-COMPUTE-OPT}(p[j]) \}$.

RETURN $M[j]$.

Weighted Interval Scheduling: Running Time

- **Claim.** Memoized version of algorithm takes $O(n \log n)$ time.
- **Proof.**
 - Sort by finish time: $O(n \log n)$ via Merge-Sort.
 - Computing $p(j)$ for each j : $O(n \log n)$ via Binary Search
 - M-Compute-Opt(j): each invocation takes $O(1)$ time and either
 - (i) returns an initialized value $M[j]$
 - (ii) initializes $M[j]$ and makes two recursive calls
 - Progress measure $\Phi = \#$ initialized entries among $M[1, \dots, n]$.
 - initially $\Phi = 0$, throughout $\Phi \leq n$.
 - (ii) increases Φ by 1 \Rightarrow $\leq 2n$ recursive calls (at most $2n$ recursive calls).
 - Overall running time of M-Compute-Opt(n) is $O(n)$.
- **Remark.** $O(n)$ if jobs are pre-sorted by start and finish times.

Automated Memoization

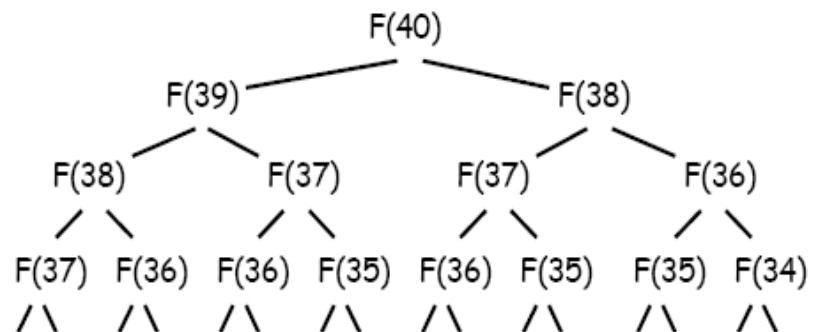
- **Automated Memoization.** Many functional programming languages (e.g., Lisp) have built-in support for memoization.
- **Question:** Why not in imperative languages (e.g., Java)?

```
(defun F (n)
  (if
    (<= n 1)
    n
    (+ (F (- n 1)) (F (- n 2))))))
```

Lisp (efficient)

```
static int F(int n) {
    if (n <= 1) return n;
    else return F(n-1) + F(n-2);
}
```

Java (exponential)



Weighted Interval Scheduling: Finding Solution

- **Question:** Dynamic programming algorithms computes optimal value. What if we want the solution itself?
- **Answer:** Do some Post-processing i.e. Make a second pass by calling Find-Solution(n).

FIND-SOLUTION(j)

IF ($j = 0$)

RETURN \emptyset .

ELSE IF ($w_j + M[p[j]] > M[j - 1]$)

RETURN $\{ j \} \cup$ FIND-SOLUTION($p[j]$).

ELSE

RETURN FIND-SOLUTION($j - 1$).

$$M[j] = \max \{ M[j - 1], w_j + M[p[j]] \}.$$

- # of recursive calls $\leq n \Rightarrow O(n)$.

Weighted Interval Scheduling: Bottom-Up

- Bottom-up Dynamic Programming: Unwind Recursion.

BOTTOM-UP($n, s_1, \dots, s_n, f_1, \dots, f_n, w_1, \dots, w_n$)

Sort jobs by finish time and renumber so that $f_1 \leq f_2 \leq \dots \leq f_n$.

Compute $p[1], p[2], \dots, p[n]$.

$M[0] \leftarrow 0$. previously computed values

FOR $j = 1$ TO n



$M[j] \leftarrow \max \{ M[j-1], w_j + M[p[j]] \}$.

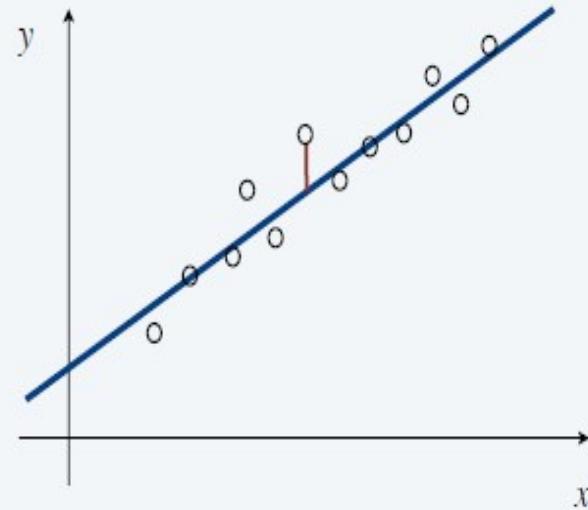
Running time. The bottom-up version takes $O(n \log n)$ time.

Least Squares

- **Least Squares.**

- Foundational problem in statistics and numerical analysis.
- Given n points in the plane: $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$.
- Find a line $y = ax + b$ that minimizes the sum of the squared error (SSE):

$$SSE = \sum_{i=1}^n (y_i - ax_i - b)^2$$



Solution. Calculus => minimum error is achieved when

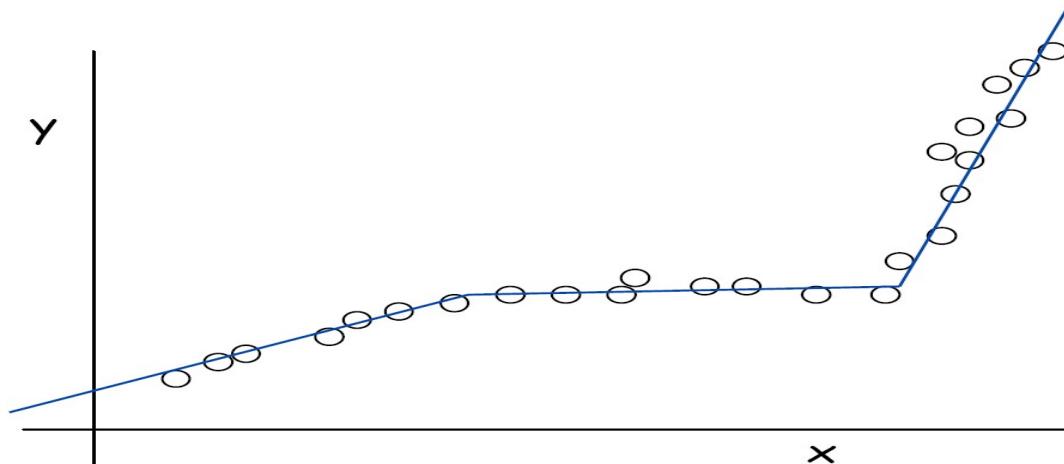
$$a = \frac{n \sum_i x_i y_i - (\sum_i x_i)(\sum_i y_i)}{n \sum_i x_i^2 - (\sum_i x_i)^2}, \quad b = \frac{\sum_i y_i - a \sum_i x_i}{n}$$

Segmented Least Squares

- Segmented Least Squares.
 - Points lie roughly on a sequence of several line segments.
 - Given n points in the plane $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$ with
 $x_1 < x_2 < \dots < x_n$, find a sequence of lines that minimizes $f(x)$.
- Q. What's a reasonable choice for $f(x)$ to balance accuracy (goodness of fit) and parsimony (number of lines)?

Goal. Minimize $f(x) = E + c L$ for some constant $c > 0$, where

- E = sum of the sums of the squared errors in each segment.
- L = number of lines.



Dynamic Programming: Multiway Choice

Notation.

- $OPT(j)$ = minimum cost for points p_1, p_2, \dots, p_j .
- e_{ij} = SSE for points p_i, p_{i+1}, \dots, p_j .

To compute $OPT(j)$:

- Last segment uses points p_i, p_{i+1}, \dots, p_j for some $i \leq j$.
- Cost = $e_{ij} + c + OPT(i - 1)$. ← optimal substructure property
(proof via exchange argument)

Bellman equation.

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \min_{1 \leq i \leq j} \{ e_{ij} + c + OPT(i - 1) \} & \text{if } j > 0 \end{cases}$$

Segmented Least Squares: Algorithm

SEGMENTED-LEAST-SQUARES(n, p_1, \dots, p_n, c)

FOR $j = 1$ TO n

FOR $i = 1$ TO j

Compute the SSE e_{ij} for the points p_i, p_{i+1}, \dots, p_j .

$$M[0] \leftarrow 0.$$

FOR $j = 1$ TO n

$$M[j] \leftarrow \min_{1 \leq i \leq j} \{ e_{ij} + c + M[i-1] \}.$$

RETURN $M[n]$.

Segmented least squares analysis

Theorem. [Bellman 1961] DP algorithm solves the segmented least squares problem in $O(n^3)$ time and $O(n^2)$ space.

Pf.

- Bottleneck = computing SSE e_{ij} for each i and j .

$$a_{ij} = \frac{n \sum_k x_k y_k - (\sum_k x_k)(\sum_k y_k)}{n \sum_k x_k^2 - (\sum_k x_k)^2}, \quad b_{ij} = \frac{\sum_k y_k - a_{ij} \sum_k x_k}{n}$$

- $O(n)$ to compute e_{ij} .
 - There are $O(n^2)$ pairs of ij for which we need to compute e_{ij} ; each e_{ij} computation takes $O(n)$ time. Thus all e_{ij} values can be computed in $O(n^3)$ time. For storing all e_{ij} values, OPT array can be filled in $O(n^2)$ time.

Remark. Can be improved to $O(n^2)$ time.

- For each i : precompute cumulative sums $\sum_{k=1}^i x_k, \sum_{k=1}^i y_k, \sum_{k=1}^i x_k^2, \sum_{k=1}^i x_k y_k$.
- Using cumulative sums, can compute e_{ij} in $O(1)$ time.

Dynamic programming: two variables

Def. $OPT(i, w)$ = optimal value of knapsack problem with items $1, \dots, i$, subject to weight limit w .

Goal. $OPT(n, W)$.

Case 1. $OPT(i, w)$ does not select item i .

- $OPT(i, w)$ selects best of $\{1, 2, \dots, i-1\}$ subject to weight limit w .

possibly because $w_i > w$

Case 2. $OPT(i, w)$ selects item i .

- Collect value v_i .
- New weight limit = $w - w_i$.
- $OPT(i, w)$ selects best of $\{1, 2, \dots, i-1\}$ subject to new weight limit.

optimal substructure property
(proof via exchange argument)

Bellman equation.

$$OPT(i, w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1, w) & \text{if } w_i > w \\ \max \{ OPT(i-1, w), v_i + OPT(i-1, w - w_i) \} & \text{otherwise} \end{cases}$$

Knapsack Problem

Goal. Pack knapsack so as to maximize total value of items taken.

- There are n items: item i provides value $v_i > 0$ and weighs $w_i > 0$.
- Value of a subset of items = sum of values of individual items.
- Knapsack has weight limit of W .

Ex. The subset { 1, 2, 5 } has value \$35 (and weight 10).

Ex. The subset { 3, 4 } has value \$40 (and weight 11).

Assumption. All values and weights are integral.



i	v_i	w_i
1	\$1	1 kg
2	\$6	2 kg
3	\$18	5 kg
4	\$22	6 kg
5	\$28	7 kg

weights and values
can be arbitrary
positive integers

knapsack instance
(weight limit $W = 11$)

Knapsack problem: bottom-up dynamic programming

KNAPSACK($n, W, w_1, \dots, w_n, v_1, \dots, v_n$)

FOR $w = 0$ TO W

$M[0, w] \leftarrow 0.$

FOR $i = 1$ TO n

FOR $w = 0$ TO W

IF ($w_i > w$) $M[i, w] \leftarrow M[i - 1, w].$

previously computed values



ELSE $M[i, w] \leftarrow \max \{ M[i - 1, w], v_i + M[i - 1, w - w_i] \}.$

RETURN $M[n, W].$

$$OPT(i, w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i - 1, w) & \text{if } w_i > w \\ \max \{ OPT(i - 1, w), v_i + OPT(i - 1, w - w_i) \} & \text{otherwise} \end{cases}$$

Knapsack problem: bottom-up dynamic programming demo

i	v_i	w_i
1	\$1	1 kg
2	\$6	2 kg
3	\$18	5 kg
4	\$22	6 kg
5	\$28	7 kg

$$OPT(i, w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i - 1, w) & \text{if } w_i > w \\ \max \{OPT(i - 1, w), v_i + OPT(i - 1, w - w_i)\} & \text{otherwise} \end{cases}$$

		weight limit w											
		0	1	2	3	4	5	6	7	8	9	10	11
subset of items $1, \dots, i$	{ }	0	0	0	0	0	0	0	0	0	0	0	0
	{ 1 }	0	1	1	1	1	1	1	1	1	1	1	1
	{ 1, 2 }	0	1	6	7	7	7	7	7	7	7	7	7
	{ 1, 2, 3 }	0	1	6	7	7	18	19	24	25	25	25	25
	{ 1, 2, 3, 4 }	0	1	6	7	7	18	22	24	28	29	29	40
	{ 1, 2, 3, 4, 5 }	0	1	6	7	7	18	22	28	29	34	35	40

$OPT(i, w)$ = optimal value of knapsack problem with items $1, \dots, i$, subject to weight limit w

Knapsack problem: running time

Theorem. The DP algorithm solves the knapsack problem with n items and maximum weight W in $\Theta(n W)$ time and $\Theta(n W)$ space.

Pf.

- Takes $O(1)$ time per table entry.
- There are $\Theta(n W)$ table entries.
- After computing optimal values, can trace back to find solution:
 $OPT(i, w)$ takes item i iff $M[i, w] > M[i - 1, w]$. ▀

weights are integers
between 1 and W

Remarks.

- Algorithm depends critically on assumption that weights are integral.
- Assumption that values are integral was not used.
- Running time. $\Theta(n W)$.
 - Not polynomial in input size!
 - "Pseudo-polynomial."
 - Decision version of Knapsack Problem is NP-complete Problem.
- Knapsack Approximation Algorithm. There exists a polynomial algorithm that produces a feasible solution that has value within **0.01% of Optimum Solution**.

Sequence Alignment Problem

String similarity

Q. How similar are two strings?

Ex. occurrence and occurrence.

o	c	u	r	r	a	n	c	e	-
o	c	c	u	r	r	e	n	c	e

6 mismatches, 1 gap

o	c	-	u	r	r	a	n	c	e
o	c	c	u	r	r	e	n	c	e

1 mismatch, 1 gap

o	c	-	u	r	r	-	a	n	c	e
o	c	c	u	r	r	e	-	n	c	e

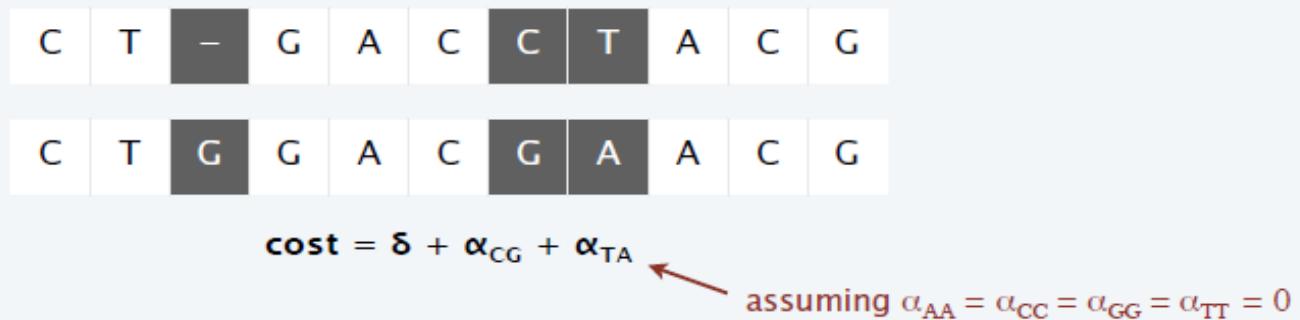
0 mismatches, 3 gaps

Sequence Alignment Problem

Edit distance

Edit distance. [Levenshtein 1966, Needleman–Wunsch 1970]

- Gap penalty δ ; mismatch penalty α_{pq} .
- Cost = sum of gap and mismatch penalties.



Applications. Bioinformatics, spell correction, machine translation, speech recognition, information extraction, ...

Spokesperson confirms senior government adviser was found
Spokesperson said the senior adviser was found

Sequence Alignment Problem

BLOSUM matrix for proteins

In bioinformatics, the BLOSUM (BLOcks SUbstitution Matrix) matrix is a substitution matrix used for sequence alignment of proteins. BLOSUM matrices are used to score alignments between evolutionarily divergent protein sequences using local alignments.

	A	R	N	D	C	Q	E	G	H	I	L	K	M	F	P	S	T	W	Y	V
A	7	-3	-3	-3	-1	-2	-2	0	-3	-3	-3	-1	-2	-4	-1	2	0	-5	-4	-1
R	-3	9	-1	-3	-6	1	-1	-4	0	-5	-4	3	-3	-5	-3	-2	-2	-5	-4	-4
N	-3	-1	9	2	-5	0	-1	-1	1	-6	-6	0	-4	-6	-4	1	0	-7	-4	-5
D	-3	-3	2	10	-7	-1	2	-3	-2	-7	-7	-2	-6	-6	-3	-1	-2	-8	-6	-6
C	-1	-6	-5	-7	13	-5	-7	-6	-7	-2	-3	-6	-3	-4	-6	-2	-2	-5	-5	-2
Q	-2	1	0	-1	-5	9	3	-4	1	-5	-4	2	-1	-5	-3	-1	-1	-4	-3	-4
E	-2	-1	-1	2	-7	3	8	-4	0	-6	-6	1	-4	-6	-2	-1	-2	-6	-5	-4
G	0	-4	-1	-3	-6	-4	-4	9	-4	-7	-7	-3	-5	-6	-5	-1	-3	-6	-6	-6
H	-3	0	1	-2	-7	1	0	-4	12	-6	-5	-1	-4	-2	-4	-2	-3	-4	3	-5
I	-3	-5	-6	-7	-2	-5	-6	-7	-6	7	2	-5	2	-1	-5	-4	-2	-5	-3	4
L	-3	-4	-6	-7	-3	-4	-6	-7	-5	2	6	-4	3	0	-5	-4	-3	-4	-2	1
K	-1	3	0	-2	-6	2	1	-3	-1	-5	-4	8	-3	-5	-2	-1	-1	-6	-4	-4
M	-2	-3	-4	-6	-3	-1	-4	-5	-4	2	3	-3	9	0	-4	-3	-1	-3	-3	1
F	-4	-5	-6	-6	-4	-5	-6	-6	-2	-1	0	-5	0	10	-6	-4	-4	0	4	-2
P	-1	-3	-4	-3	-6	-3	-2	-5	-4	-5	-5	-2	-4	-6	12	-2	-3	-7	-6	-4
S	2	-2	1	-1	-2	-1	-1	-1	-2	-4	-4	-1	-3	-4	-2	7	2	-6	-3	-3
T	0	-2	0	-2	-2	-1	-2	-3	-3	-2	-3	-1	-1	-4	-3	2	8	-5	-3	0
W	-5	-5	-7	-8	-5	-4	-6	-6	-4	-5	-4	-6	-3	0	-7	-6	-5	16	3	-5
Y	-4	-4	-4	-6	-5	-3	-5	-6	3	-3	-2	-4	-3	4	-6	-3	-3	3	11	-3
V	-1	-4	-5	-6	-2	-4	-4	-6	-5	4	1	-4	1	-2	-4	-3	0	-5	-3	7

Sequence Alignment Problem

Dynamic programming: quiz 1

What is edit distance between these two strings?

P A L E T T E

P A L A T E

Assume gap penalty = 2 and mismatch penalty = 1.

- A. 1
- B. 2
- C. 3
- D. 4
- E. 5

Sequence Alignment Problem

Sequence alignment

Goal. Given two strings $x_1 x_2 \dots x_m$ and $y_1 y_2 \dots y_n$, find a min-cost alignment.

Def. An **alignment** M is a set of ordered pairs $x_i - y_j$ such that each character appears in at most one pair and no crossings.

$x_i - y_j$ and $x_{i'} - y_{j'}$ cross if $i < i'$, but $j > j'$

Def. The **cost** of an alignment M is:

$$\text{cost}(M) = \underbrace{\sum_{(x_i, y_j) \in M} \alpha_{x_i y_j}}_{\text{mismatch}} + \underbrace{\sum_{i : x_i \text{ unmatched}} \delta + \sum_{j : y_j \text{ unmatched}} \delta}_{\text{gap}}$$

x_1	x_2	x_3	x_4	x_5	x_6
C	T	A	C	C	-
y_1	y_2	y_3	y_4	y_5	y_6
-	T	A	C	A	T

an alignment of CTACCG and TACATG

$$M = \{ x_2 - y_1, x_3 - y_2, x_4 - y_3, x_5 - y_4, x_6 - y_6 \}$$

Sequence Alignment Problem

Sequence alignment: problem structure

Def. $OPT(i, j) = \min$ cost of aligning prefix strings $x_1 x_2 \dots x_i$ and $y_1 y_2 \dots y_j$.

Goal. $OPT(m, n)$.

Case 1. $OPT(i, j)$ matches $x_i - y_j$.

Pay mismatch for $x_i - y_j$ + min cost of aligning $x_1 x_2 \dots x_{i-1}$ and $y_1 y_2 \dots y_{j-1}$.

Case 2a. $OPT(i, j)$ leaves x_i unmatched.

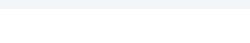
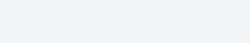
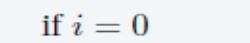
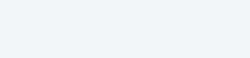
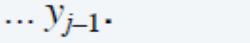
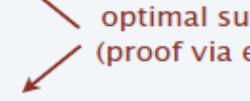
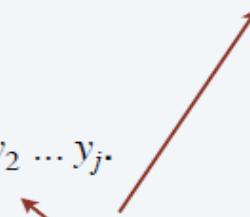
Pay gap for x_i + min cost of aligning $x_1 x_2 \dots x_{i-1}$ and $y_1 y_2 \dots y_j$.

Case 2b. $OPT(i, j)$ leaves y_j unmatched.

Pay gap for y_j + min cost of aligning $x_1 x_2 \dots x_i$ and $y_1 y_2 \dots y_{j-1}$.

Bellman equation.

$$OPT(i, j) = \begin{cases} j\delta & \text{if } i = 0 \\ i\delta & \text{if } j = 0 \\ \min \begin{cases} \alpha_{x_i y_j} + OPT(i-1, j-1) \\ \delta + OPT(i-1, j) \\ \delta + OPT(i, j-1) \end{cases} & \text{otherwise} \end{cases}$$



Sequence Alignment Problem

Sequence alignment: bottom-up algorithm

SEQUENCE-ALIGNMENT($m, n, x_1, \dots, x_m, y_1, \dots, y_n, \delta, \alpha$)

```
FOR  $i = 0$  TO  $m$ 
     $M[i, 0] \leftarrow i \delta.$ 
FOR  $j = 0$  TO  $n$ 
     $M[0, j] \leftarrow j \delta.$ 
```

```
FOR  $i = 1$  TO  $m$ 
    FOR  $j = 1$  TO  $n$ 
         $M[i, j] \leftarrow \min \{ \alpha_{x_i y_j} + M[i - 1, j - 1],$ 
         $\delta + M[i - 1, j],$ 
         $\delta + M[i, j - 1] \}.$ 
```

already
computed

RETURN $M[m, n]$.

Sequence Alignment Problem

Sequence alignment: traceback



1 mismatch, 1 gap

Sequence Alignment Problem

Sequence alignment: analysis

Theorem. The DP algorithm computes the edit distance (and an optimal alignment) of two strings of lengths m and n in $\Theta(mn)$ time and space.

Pf.

- Algorithm computes edit distance.
- Can trace back to extract optimal alignment itself. ▀

Theorem. [Backurs–Indyk 2015] If can compute edit distance of two strings of length n in $O(n^{2-\varepsilon})$ time for some constant $\varepsilon > 0$, then can solve SAT with n variables and m clauses in $\text{poly}(m) 2^{(1-\delta)n}$ time for some constant $\delta > 0$.

Edit Distance Cannot Be Computed
in Strongly Subquadratic Time
(unless SETH is false)*

which would disprove SETH
(strong exponential time hypothesis)

Sequence Alignment Problem

Dynamic programming: quiz 2

It is easy to modify the DP algorithm for edit distance to...

- A. Compute edit distance in $O(mn)$ time and $O(m + n)$ space.
- B. Compute an optimal alignment in $O(mn)$ time and $O(m + n)$ space.
- C. Both A and B.
- D. Neither A nor B.

$$OPT(i, j) = \begin{cases} j\delta & \text{if } i = 0 \\ i\delta & \text{if } j = 0 \\ \min \left\{ \begin{array}{ll} \alpha_{x_i y_j} + OPT(i - 1, j - 1) & \\ \delta + OPT(i - 1, j) & \text{otherwise} \\ \delta + OPT(i, j - 1) & \end{array} \right. & \end{cases}$$

Sequence Alignment Problem

Sequence alignment in linear space

Theorem. [Hirschberg] There exists an algorithm to find an optimal alignment in $O(mn)$ time and $O(m + n)$ space.

- Clever combination of divide-and-conquer and dynamic programming.
- Inspired by idea of Savitch from complexity theory.

Programming
Techniques

G. Manacher
Editor

A Linear Space Algorithm for Computing Maximal Common Subsequences

D.S. Hirschberg
Princeton University

The problem of finding a longest common subsequence of two strings has been solved in quadratic time and space. An algorithm is presented which will solve this problem in quadratic time and in linear space.

Key Words and Phrases: subsequence, longest common subsequence, string correction, editing

CR Categories: 3.63, 3.73, 3.79, 4.22, 5.25

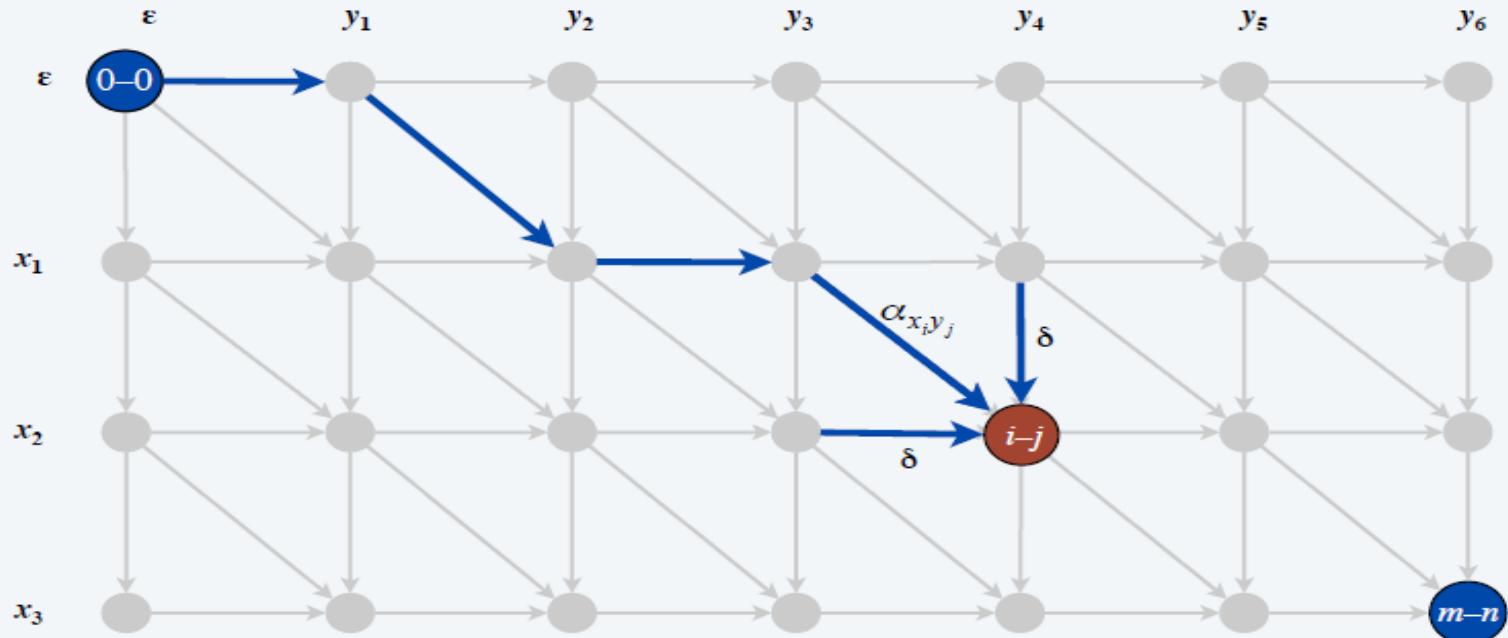


Sequence Alignment Problem

Hirschberg's algorithm

Edit distance graph.

- Let $f(i, j)$ denote length of shortest path from $(0, 0)$ to (i, j) .
- Lemma: $f(i, j) = OPT(i, j)$ for all i and j .



Sequence Alignment Problem

Hirschberg's algorithm

Edit distance graph.

- Let $f(i, j)$ denote length of shortest path from $(0,0)$ to (i,j) .
- Lemma: $f(i, j) = OPT(i, j)$ for all i and j .

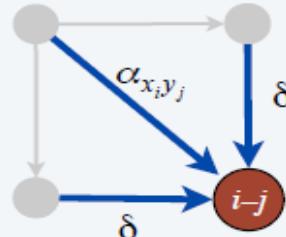
Pf of Lemma. [by strong induction on $i + j$]

- Base case: $f(0, 0) = OPT(0, 0) = 0$.
- Inductive hypothesis: assume true for all (i', j') with $i' + j' < i + j$.
- Last edge on shortest path to (i, j) is from $(i - 1, j - 1)$, $(i - 1, j)$, or $(i, j - 1)$.
- Thus,

$$f(i, j) = \min\{\alpha_{x_i y_j} + f(i - 1, j - 1), \delta + f(i - 1, j), \delta + f(i, j - 1)\}$$

$$\begin{aligned} &= \min\{\alpha_{x_i y_j} + OPT(i - 1, j - 1), \delta + OPT(i - 1, j), \delta + OPT(i, j - 1)\} \\ &= OPT(i, j) \quad \blacksquare \end{aligned}$$

inductive hypothesis
Bellman equation

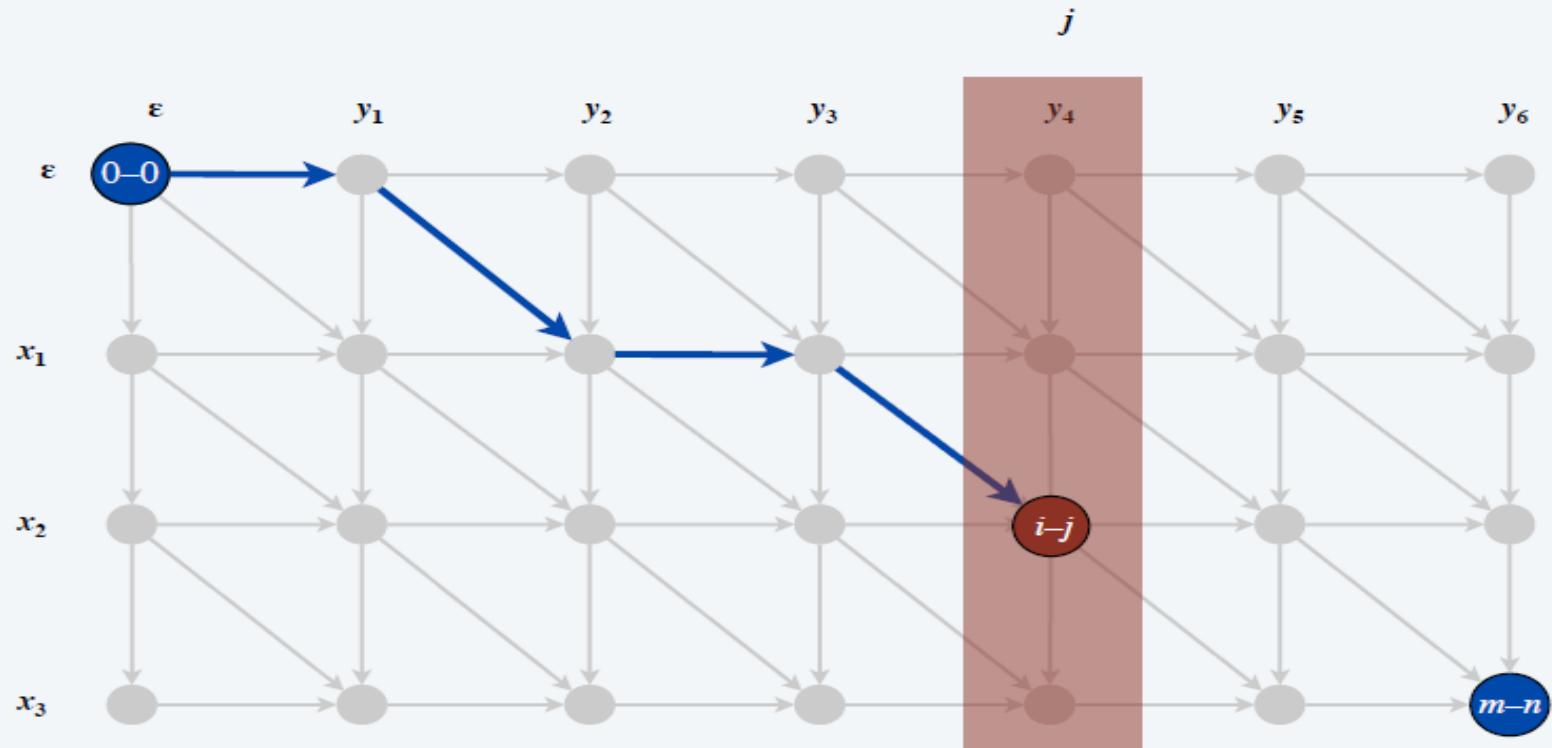


Sequence Alignment Problem

Hirschberg's algorithm

Edit distance graph.

- Let $f(i, j)$ denote length of shortest path from $(0,0)$ to (i, j) .
- Lemma: $f(i, j) = OPT(i, j)$ for all i and j .
- Can compute $f(\cdot, j)$ for any j in $O(mn)$ time and $O(m + n)$ space.

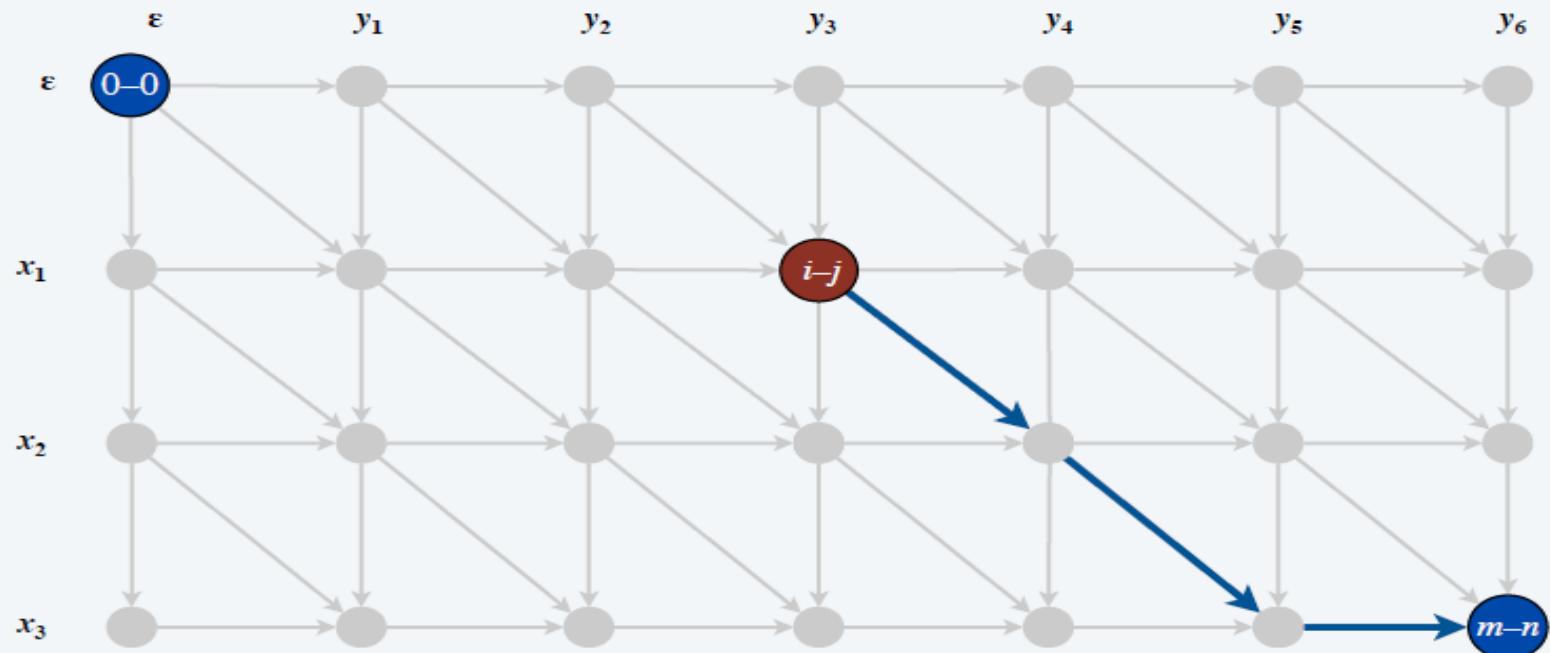


Sequence Alignment Problem

Hirschberg's algorithm

Edit distance graph.

- Let $g(i, j)$ denote length of shortest path from (i, j) to (m, n) .

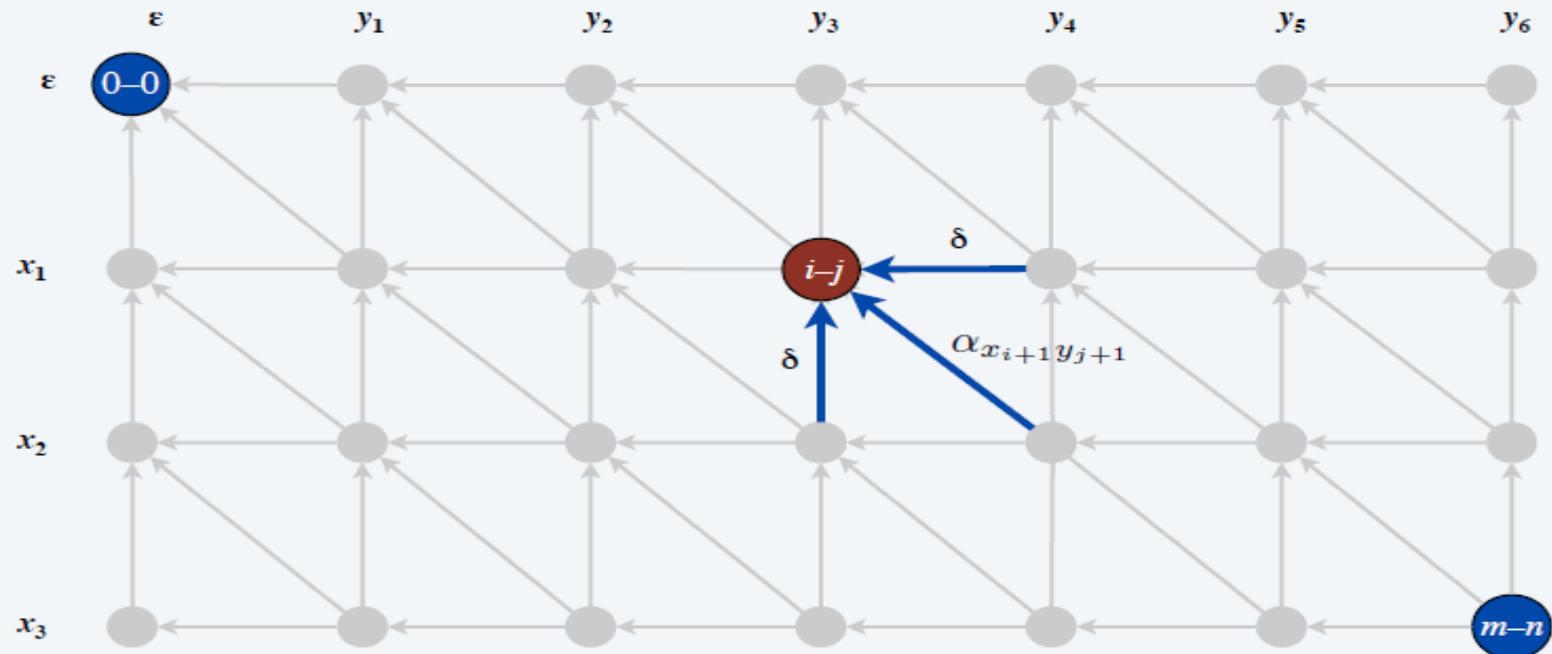


Sequence Alignment Problem

Hirschberg's algorithm

Edit distance graph.

- Let $g(i, j)$ denote length of shortest path from (i, j) to (m, n) .
- Can compute $g(i, j)$ by reversing the edge orientations and inverting the roles of $(0, 0)$ and (m, n) .

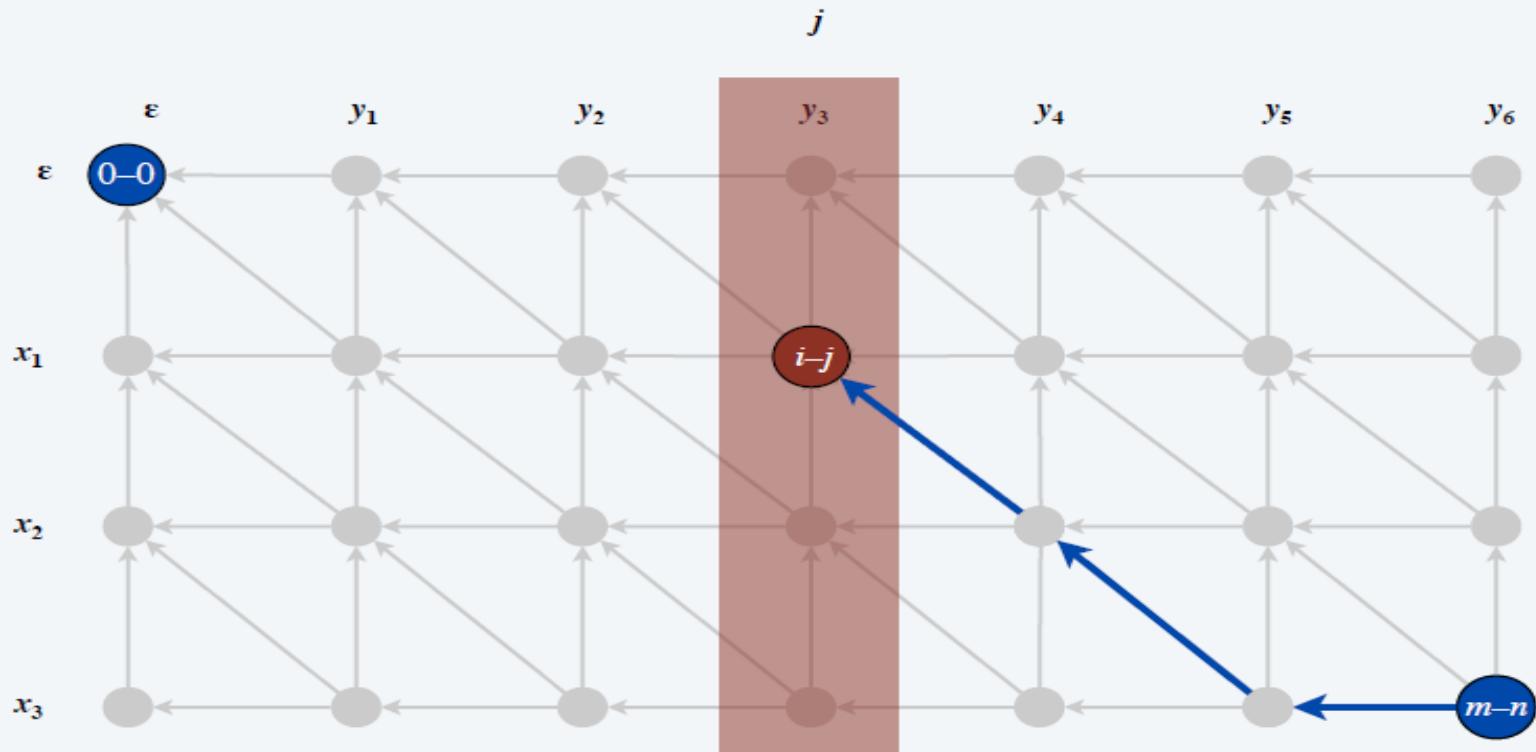


Sequence Alignment Problem

Hirschberg's algorithm

Edit distance graph.

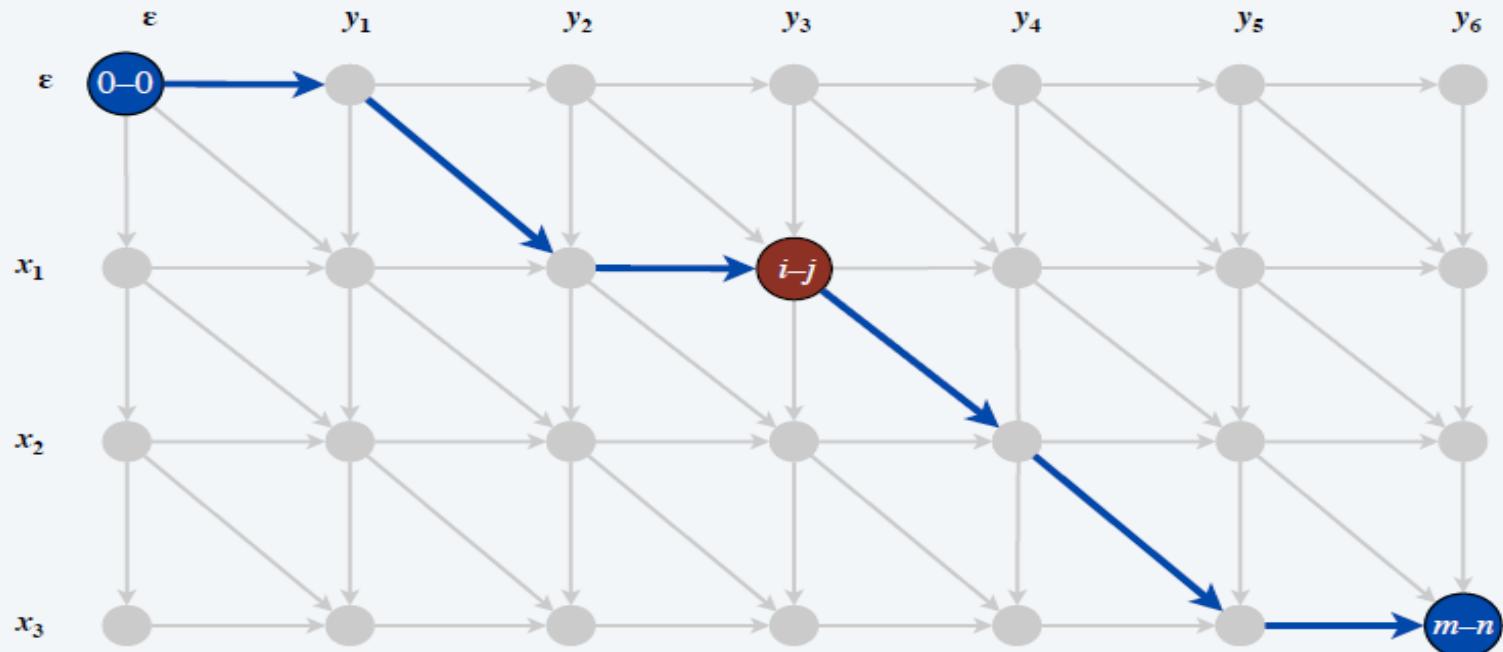
- Let $g(i, j)$ denote length of shortest path from (i, j) to (m, n) .
- Can compute $g(\cdot, j)$ for any j in $O(mn)$ time and $O(m + n)$ space.



Sequence Alignment Problem

Hirschberg's algorithm

Observation 1. The length of a shortest path that uses (i, j) is $f(i, j) + g(i, j)$.

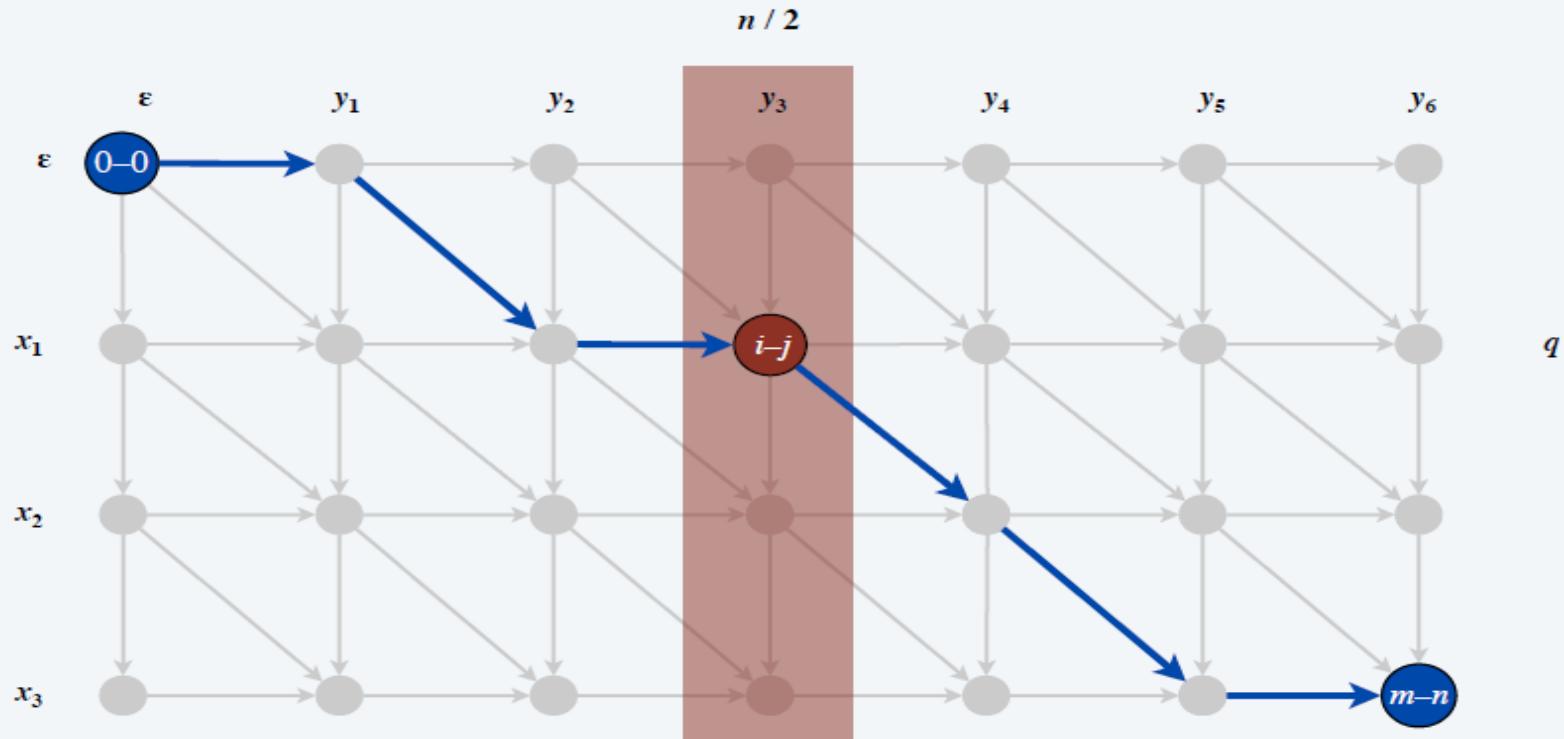


Sequence Alignment Problem

Hirschberg's algorithm

Observation 2. let q be an index that minimizes $f(q, n/2) + g(q, n/2)$.

Then, there exists a shortest path from $(0, 0)$ to (m, n) that uses $(q, n/2)$.

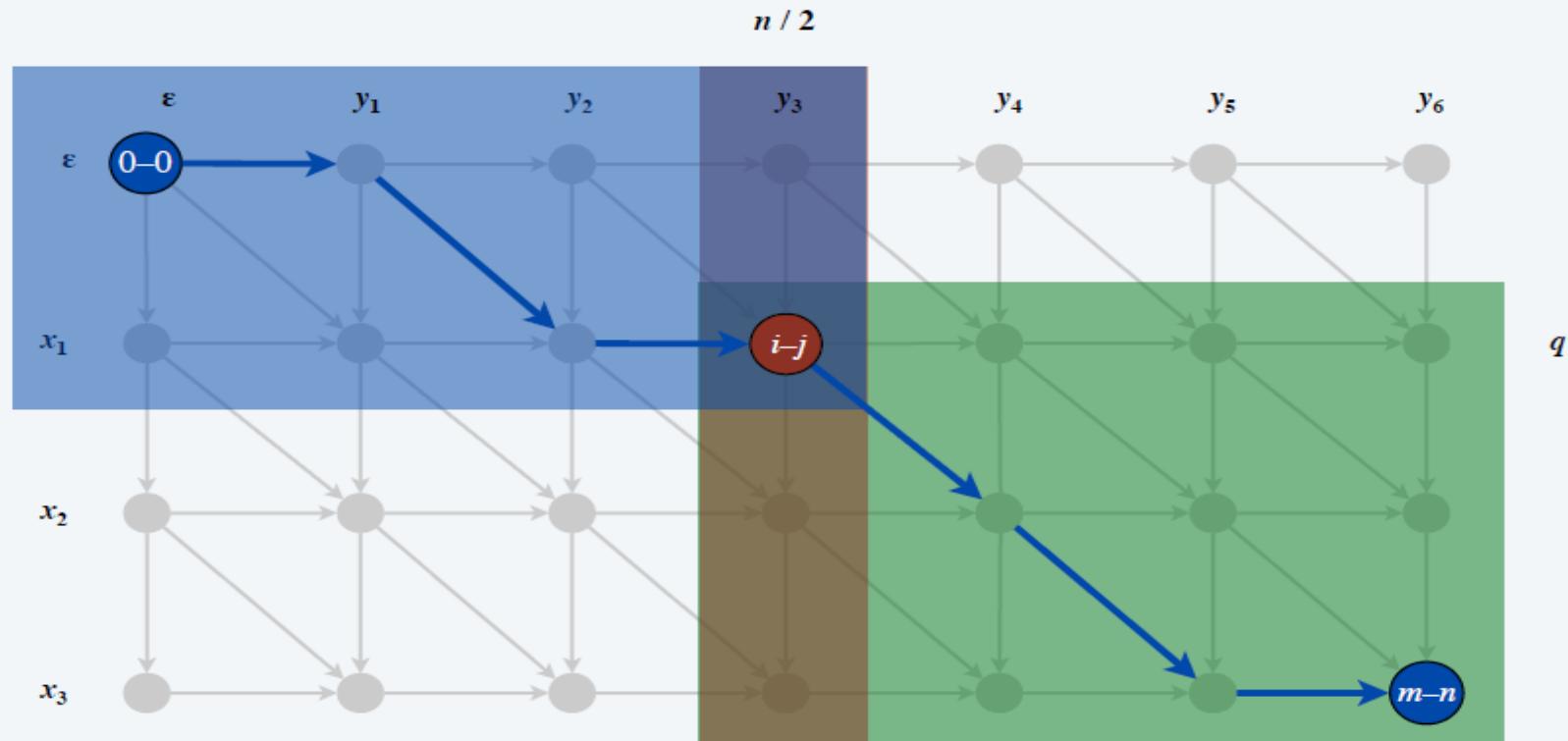


Sequence Alignment Problem

Hirschberg's algorithm

Divide. Find index q that minimizes $f(q, n / 2) + g(q, n / 2)$; save node $i-j$ as part of solution.

Conquer. Recursively compute optimal alignment in each piece.



Sequence Alignment Problem

Hirschberg's algorithm: space analysis

Theorem. Hirschberg's algorithm uses $\Theta(m + n)$ space.

Pf.

- Each recursive call uses $\Theta(m)$ space to compute $f(\cdot, n/2)$ and $g(\cdot, n/2)$.
- Only $\Theta(1)$ space needs to be maintained per recursive call.
- Number of recursive calls $\leq n$. ■

Hirschberg's algorithm: running time analysis warmup

Theorem. Let $T(m, n) = \max$ running time of Hirschberg's algorithm on strings of lengths at most m and n . Then, $T(m, n) = O(m n \log n)$.

Pf.

- $T(m, n)$ is monotone nondecreasing in both m and n .
- $T(m, n) \leq 2 T(m, n/2) + O(m n)$ Remark. Analysis is not tight because two subproblems are of size $(q, n/2)$ and $(m-q, n/2)$. Next, we prove $T(m, n) = O(m n)$.
 $\Rightarrow T(m, n) = O(m n \log n)$.

Sequence Alignment Problem

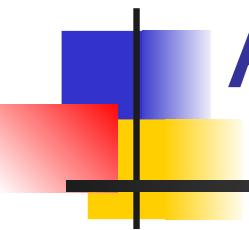
Hirschberg's algorithm: running time analysis

Theorem. Let $T(m, n) = \max$ running time of Hirschberg's algorithm on strings of lengths at most m and n . Then, $T(m, n) = O(mn)$.

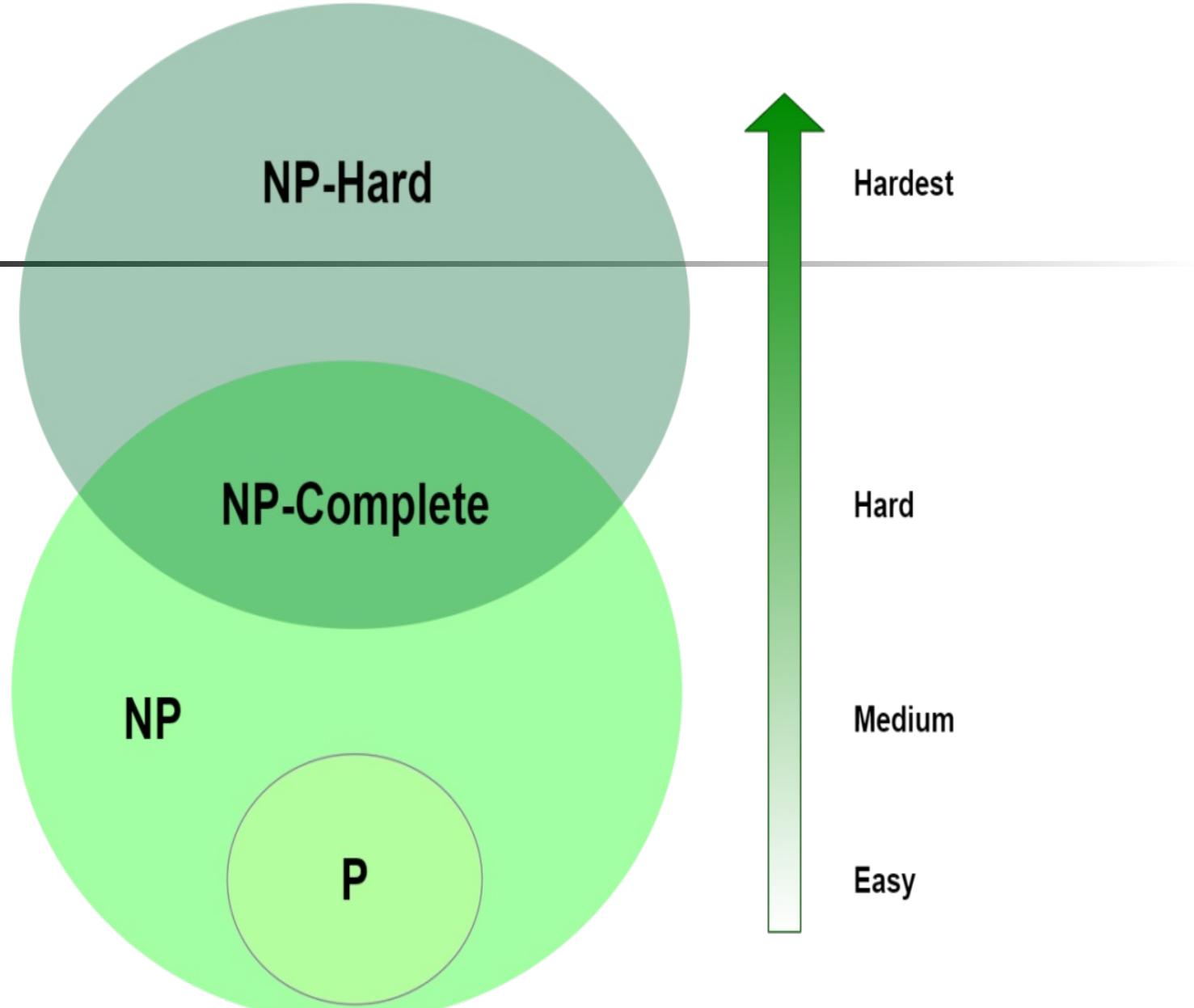
Pf. [by strong induction on $m + n$]

- $O(mn)$ time to compute $f(\cdot, n/2)$ and $g(\cdot, n/2)$ and find index q .
- $T(q, n/2) + T(m - q, n/2)$ time for two recursive calls.
- Choose constant c so that:
$$T(m, 2) \leq cm$$
$$T(2, n) \leq cn$$
$$T(m, n) \leq cmn + T(q, n/2) + T(m - q, n/2)$$
- Claim. $T(m, n) \leq 2cmn$.
- Base cases: $m = 2$ and $n = 2$.
- Inductive hypothesis: $T(m, n) \leq 2cmn$ for all (m', n') with $m' + n' < m + n$.

$$\begin{aligned} T(m, n) &\leq T(q, n/2) + T(m - q, n/2) + cmn \\ &\leq 2cq n/2 + 2c(m - q)n/2 + cmn \\ &\stackrel{\text{inductive hypothesis}}{=} cq n + cmn - cq n + cmn \\ &= 2cmn \blacksquare \end{aligned}$$



Approximation Algorithms



NP-completeness



“I can’t find an efficient algorithm, but neither can all these famous people.”

Coping With NP-Hardness

Brute-force Algorithms.

- Develop clever enumeration strategies.
- Guaranteed to find optimal solution.
- No guarantees on running time.

Heuristics.

Develop intuitive algorithms.

Guaranteed to run in polynomial time.

No guarantees on quality of solution.

Approximation Algorithms.

- Guaranteed to run in polynomial time.
- Guaranteed to find "high quality" solution, say within 1% of optimum.

Obstacle: need to prove a solution's value is close to optimum, without even knowing what optimum value is!

Coping with NP-completeness

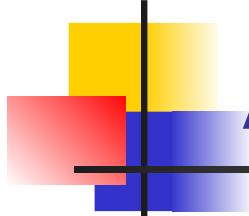
Q. Suppose I need to solve an NP-hard optimization problem.
What should I do?

- A. Sacrifice one of three desired features.
- Runs in polynomial time.
 - Solves arbitrary instances of the problem.
 - Finds optimal solution to problem.

ρ -approximation algorithm.

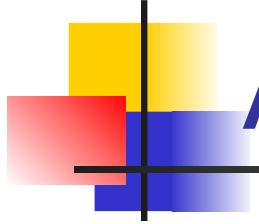
- Runs in polynomial time.
- Solves arbitrary instances of the problem
- Finds solution that is within ratio ρ of optimum.

Challenge. Need to prove a solution's value is close to optimum,
without even knowing what is optimum value.



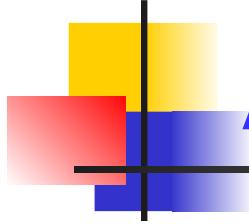
Approximation Algorithms

- Up to now, the best algorithm for solving an NP-complete problem requires exponential time in the worst case. It is too time-consuming.
- To reduce the time required for solving a problem, we can relax the problem, and obtain a feasible solution “close” to an optimal solution



Approximation Algorithms

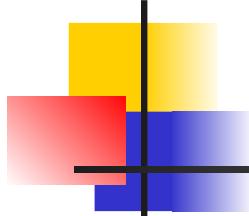
- One compromise is to use **heuristic** solutions.
- The word “heuristic” may be interpreted as “educated guess.”



Approximation Algorithms

An algorithm that returns near-optimal solutions is called an ***Approximation Algorithm.***

We need to find an ***Approximation Ratio Bound*** for an approximation algorithm.



Approximation Ratio Bound

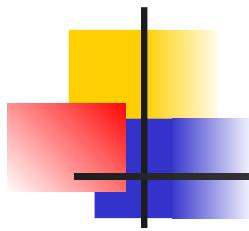
We say an approximation algorithm for the problem has a ratio bound of $\rho(n)$ if for any input size n , the cost C of the solution produced by the approximation algorithm is within a factor of $\rho(n)$ of the C^* of the optimal solution:

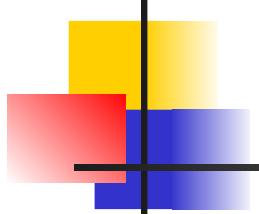
$$\max\left\{\frac{C}{C^*}, \frac{C^*}{C}\right\} = \rho(n)$$

This applies for both minimization and maximization problems.

Performance

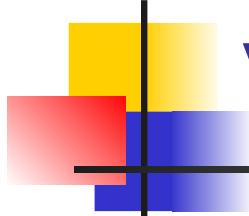
Guarantees

- 
- An approximation algorithm is bounded by $\rho(n)$ if, for all input of size n , the cost c of the solution obtained by the algorithm is within a factor $\rho(n)$ of the c^* of an optimal solution



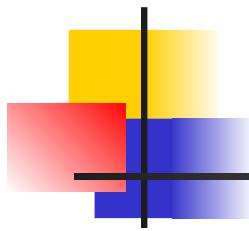
ρ -approximation algorithm

- An approximation algorithm with an approximation ratio bound of ρ is called a ρ -approximation algorithm or a $(1+\varepsilon)$ -approximation algorithm.
- Note that ρ is always larger than 1 and $\varepsilon = \rho - 1$.

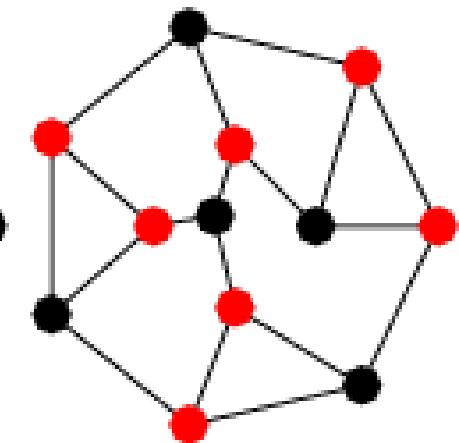
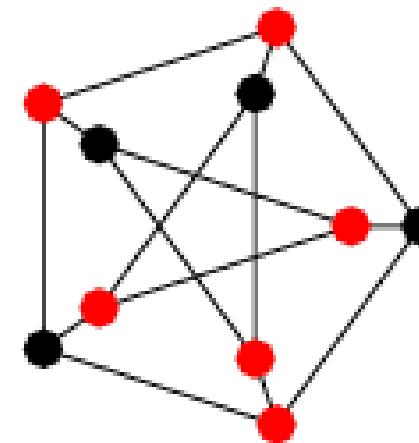
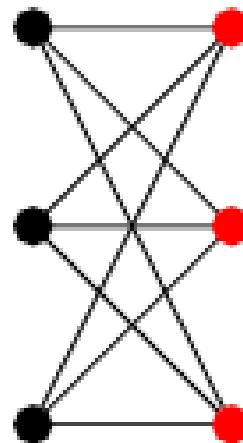
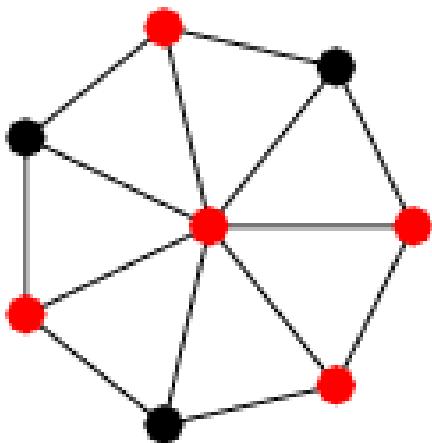


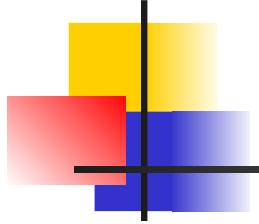
Vertex Cover Problem

- Let $G=(V, E)$. The subset S of V that meets every edge of E is referred to as the **Vertex Cover**.
- The Vertex Cover Problem is solved for finding a vertex cover of the **Minimum** size. It is NP-hard Computational Problem or the Optimization Version of an NP-Complete Decision Problem.



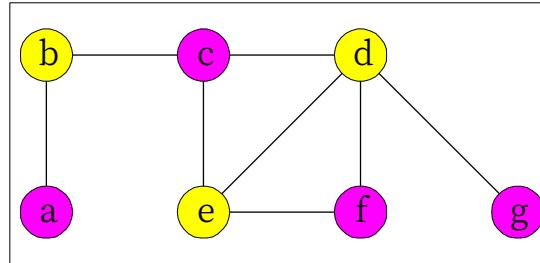
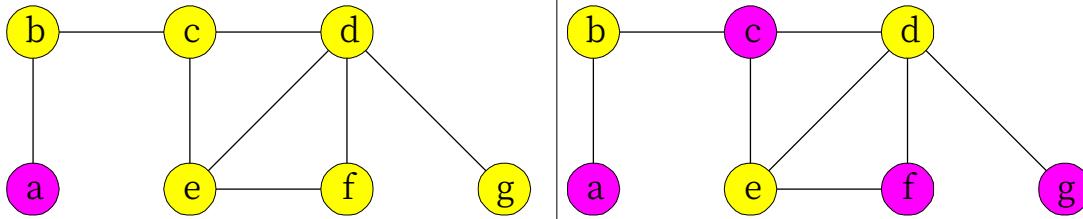
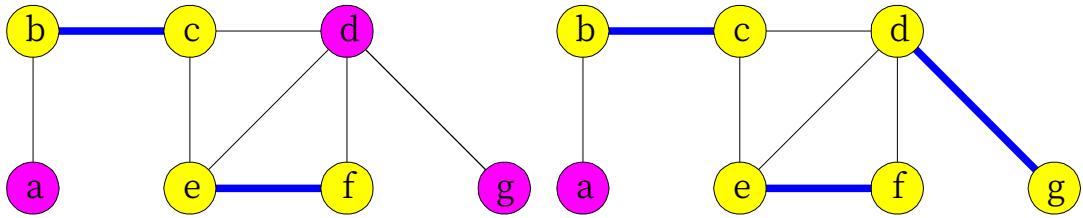
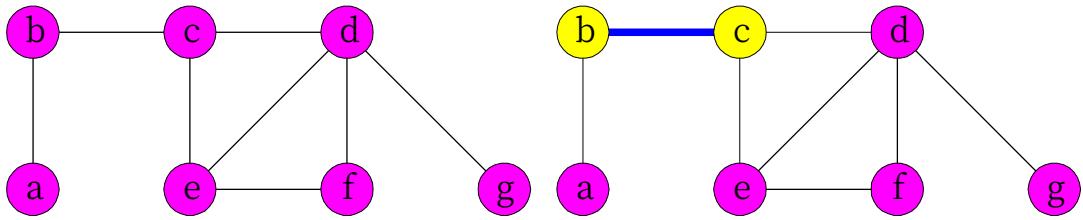
Examples of Vertex Cover



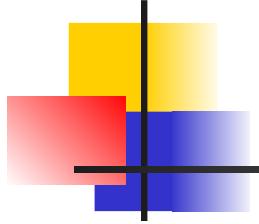


APPROX_VERTEX_COVER(G)

- 1 $C \leftarrow \phi$
- 2 $E' \leftarrow E(G)$
- 3 **while** $E' \neq \phi$
- 4 **do** let (u, v) be an arbitrary edge of E'
- 5 $C \leftarrow C \cup \{u, v\}$
- 6 remove from E' every edge incident on either u or v
- 7 **return** C



Complexity: $O(E)$



Theorem: APPROX_VERTEX_COVER has ratio bound of 2.

Proof.

C^* : optimal solution

C : approximate solution

A : the set of edges selected in

Let A be the set of selected edges.

step 4

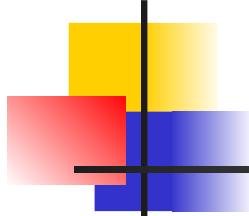
$$|C|=2|A|$$

$$|A| \leq |C^*|$$

$$\Rightarrow |C| \leq 2|C^*|$$

When one edge is selected, 2 vertices are added into C .

No two edges in A share a common endpoint.

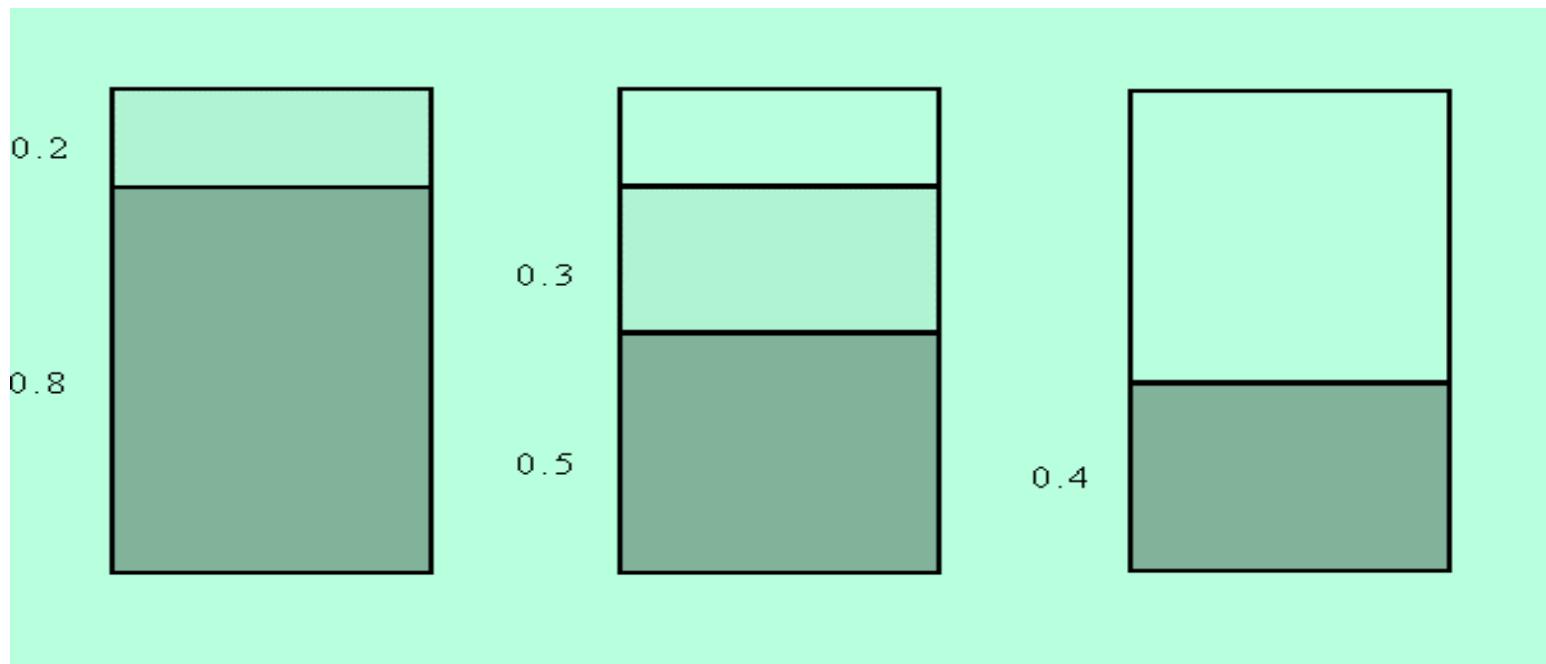


Bin Packing Problem

- Given n items of sizes a_1, a_2, \dots, a_n , $0 < a_i \leq 1$ for $1 \leq i \leq n$, which have to be placed in bins of unit capability, the bin packing problem is solved for determining the minimum number of bins to accommodate all items.
- If we consider the items of different sizes to be the lengths of time of executing different jobs on a standard processor, then the problem becomes to use minimum number of processors which can finish all of the jobs within a fixed time. // We can assume the longest job takes one unit time, which equals 1 .

Example of Bin Packing Problem

- Ex. Given $n = 5$ items with sizes 0.3, 0.5, 0.8, 0.2, 0.4, the optimal solution is 3 bins.



The bin packing problem is NP-hard optimization problem.

An Approximation Algorithm for the Bin Packing Problem

- An Approximation Algorithm: (First-Fit (FF)) place the item i into the lowest-indexed bin which can accommodate the item i .
- OPT: The number of bins of the Optimal Solution
- FF: The number of bins in the First-Fit Algorithm
- $C(B_i)$: The sum of the sizes of items packed in bin B_i in the First-Fit Algorithm
- Let $FF = m$.

An Approximation Algorithm for the Bin Packing Problem

- $\text{OPT} \geq \left\lceil \sum_{i=1}^n a_i \right\rceil$, ceiling of sum of sizes of all items
- $C(B_i) + C(B_{i+1}) > C(B_i)$: C(B_i): the sum of sizes of items packed in bin B_i
B_{i+1} will be put in B_i).
- $C(B_1) + C(B_m) > 1$ (b)(Otherwise, the items in B_m will be put in B₁.)
- For m nonempty bins,
$$C(B_1) + C(B_2) + \dots + C(B_m) > \frac{\sum_{i=1}^n a_i}{m} > \frac{m}{2}$$
 (a)+(b) for i=1,...,m
$$\rightarrow FF = m < 2 \times \frac{m}{2} = 2 \leq 2 \text{ OPT}$$

Load balancing

Input. m identical machines; $n \geq m$ jobs, job j has processing time t_j .

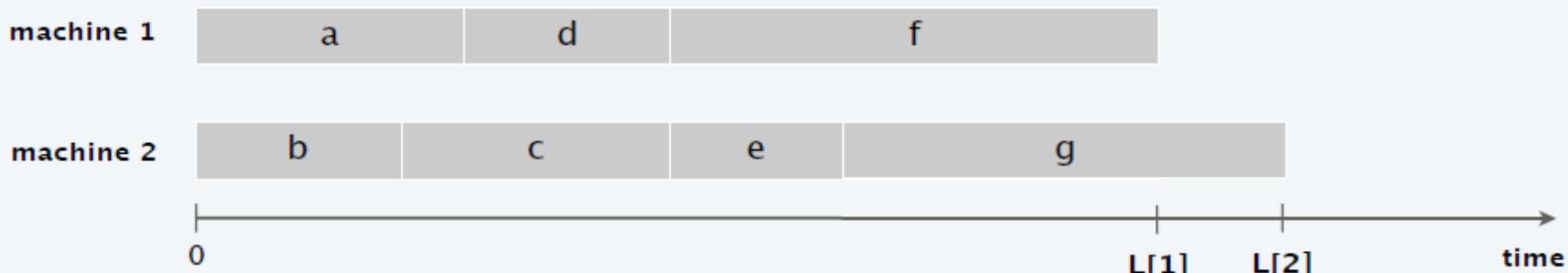
- Job j must run contiguously on one machine.
- A machine can process at most one job at a time.

Def. Let $S[i]$ be the subset of jobs assigned to machine i .

The **load** of machine i is $L[i] = \sum_{j \in S[i]} t_j$.

Def. The **makespan** is the maximum load on any machine $L = \max_i L[i]$.

Load balancing. Assign each job to a machine to minimize makespan.

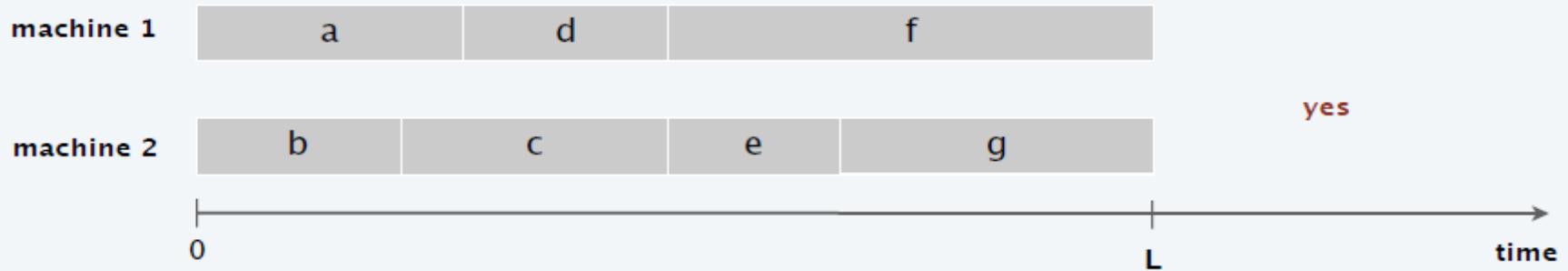
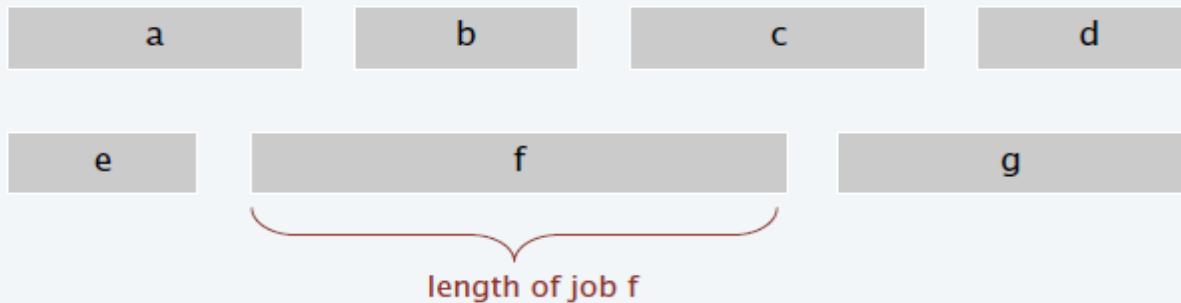


Load balancing on 2 machines is NP-hard

Claim. Load balancing is hard even if $m = 2$ machines.

Pf. $\text{PARTITION} \leq_p \text{LOAD-BALANCE}$.

NP-complete by Exercise 8.26



Load balancing: list scheduling

List-scheduling algorithm.

- Consider n jobs in some fixed order.
- Assign job j to machine i whose load is smallest so far.

LIST-SCHEDULING ($m, n, t_1, t_2, \dots, t_n$)

FOR $i = 1$ TO m

$L[i] \leftarrow 0$. ← load on machine i

$S[i] \leftarrow \emptyset$. ← jobs assigned to machine i

FOR $j = 1$ TO n

$i \leftarrow \operatorname{argmin}_k L[k]$. ← machine i has smallest load

$S[i] \leftarrow S[i] \cup \{j\}$. ← assign job j to machine i

$L[i] \leftarrow L[i] + t_j$. ← update load of machine i

RETURN $S[1], S[2], \dots, S[m]$.

Implementation. $O(n \log m)$ using a priority queue for loads $L[k]$.

Load balancing: list scheduling analysis

Theorem. [Graham 1966] Greedy algorithm is a 2-approximation.

- First worst-case analysis of an approximation algorithm.
- Need to compare resulting solution with optimal makespan L^* .

Lemma 1. For all k : the optimal makespan $L^* \geq t_k$.

Pf. Some machine must process the most time-consuming job. ▀

Lemma 2. The optimal makespan $L^* \geq \frac{1}{m} \sum_k t_k$.

Pf.

- The total processing time is $\sum_k t_k$.
- One of m machines must do at least a $1/m$ fraction of total work. ▀

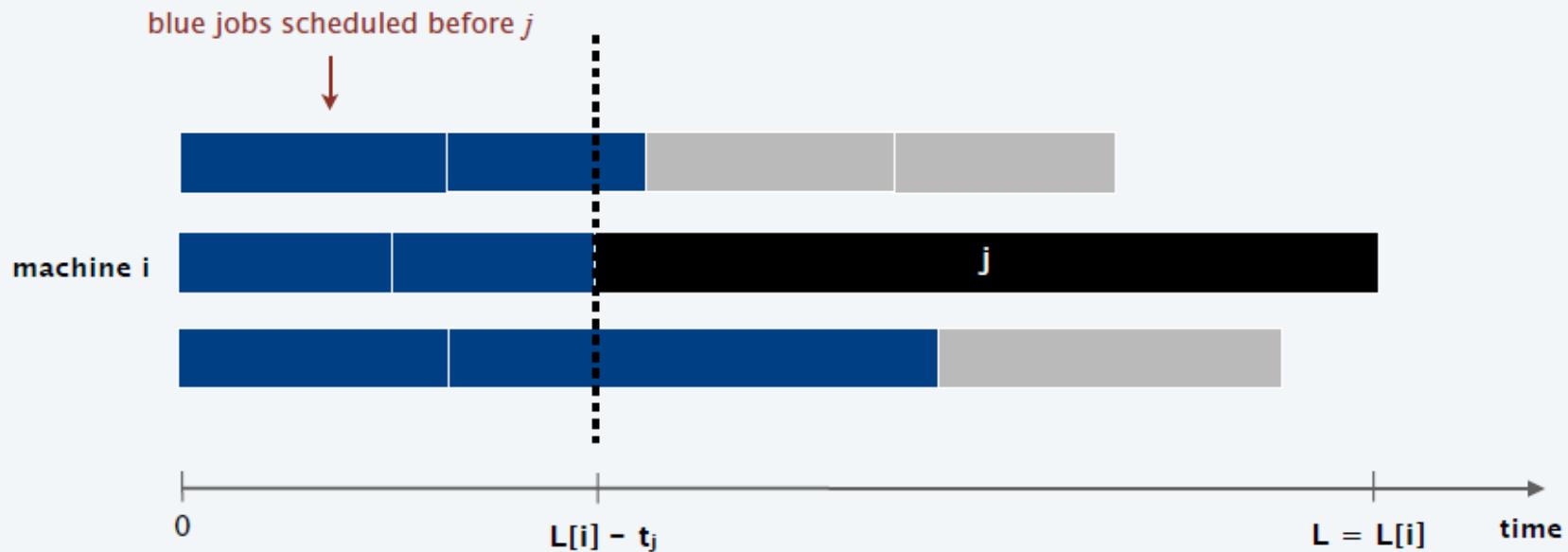
Load balancing: list scheduling analysis

Theorem. Greedy algorithm is a 2-approximation.

Pf. Consider load $L[i]$ of bottleneck machine i . \leftarrow machine that ends up with highest load

- Let j be last job scheduled on machine i .
- When job j assigned to machine i , i had smallest load.

Its load before assignment is $L[i] - t_j$; hence $L[i] - t_j \leq L[k]$ for all $1 \leq k \leq m$.



Load balancing: list scheduling analysis

Theorem. Greedy algorithm is a 2-approximation.

Pf. Consider load $L[i]$ of bottleneck machine i . \leftarrow machine that ends up with highest load

- Let j be last job scheduled on machine i .
- When job j assigned to machine i , i had smallest load.
Its load before assignment is $L[i] - t_j$; hence $L[i] - t_j \leq L[k]$ for all $1 \leq k \leq m$.
- Sum inequalities over all k and divide by m :

$$\begin{aligned} L[i] - t_j &\leq \frac{1}{m} \sum_k L[k] \\ &= \frac{1}{m} \sum_k t_k \\ \text{Lemma 2} \longrightarrow &\leq L^*. \end{aligned}$$

- Now, $L = L[i] = \underbrace{(L[i] - t_j)}_{\leq L^*} + \underbrace{t_j}_{\leq L^*} \leq 2L^*$.

$$\begin{array}{ccc} \overbrace{} & & \overbrace{} \\ \leq L^* & & \leq L^* \\ \uparrow & & \uparrow \\ \text{above inequality} & & \text{Lemma 1} \end{array}$$

Load balancing: list scheduling analysis

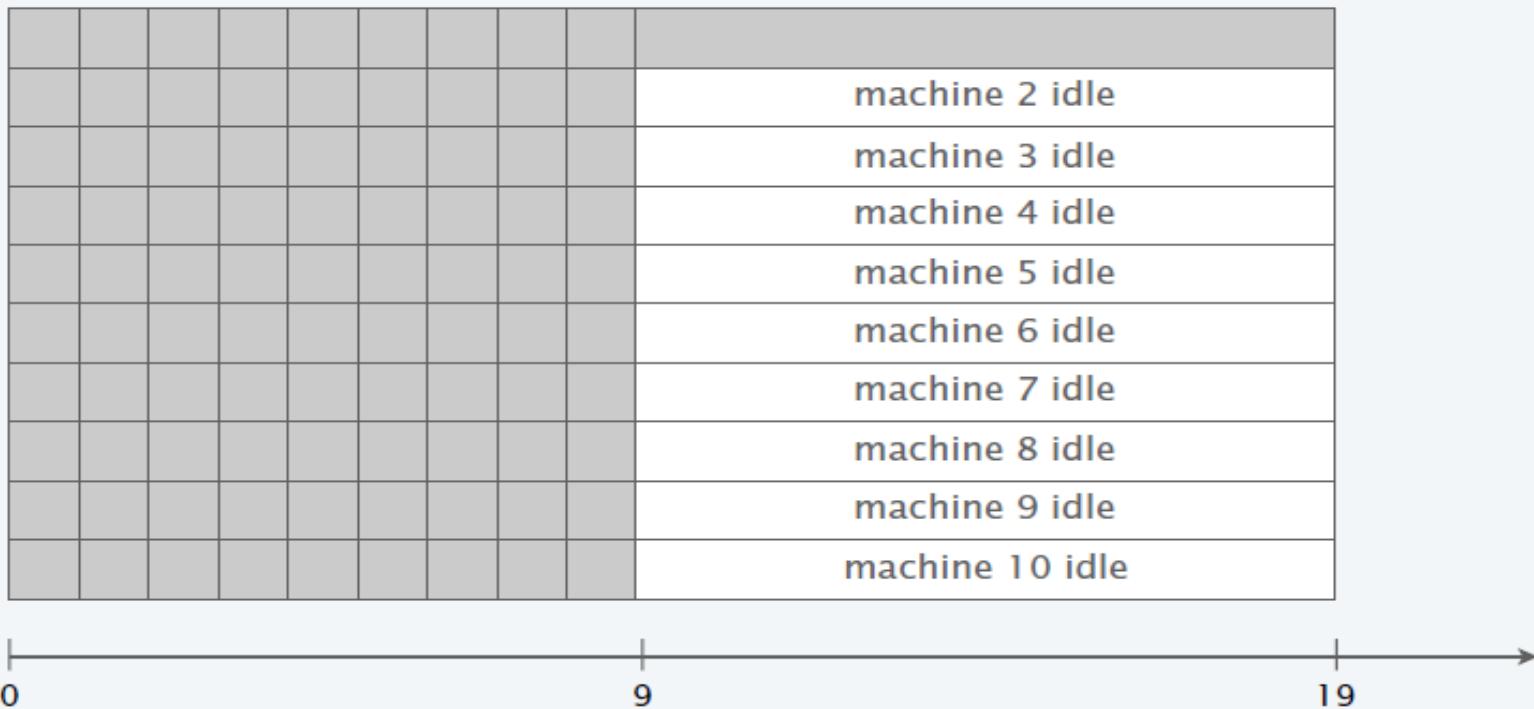
Q. Is our analysis tight?

A. Essentially yes.

Ex: m machines, first $m(m - 1)$ jobs have length 1, last job has length m .

list scheduling makespan = $19 = 2m - 1$

$m = 10$

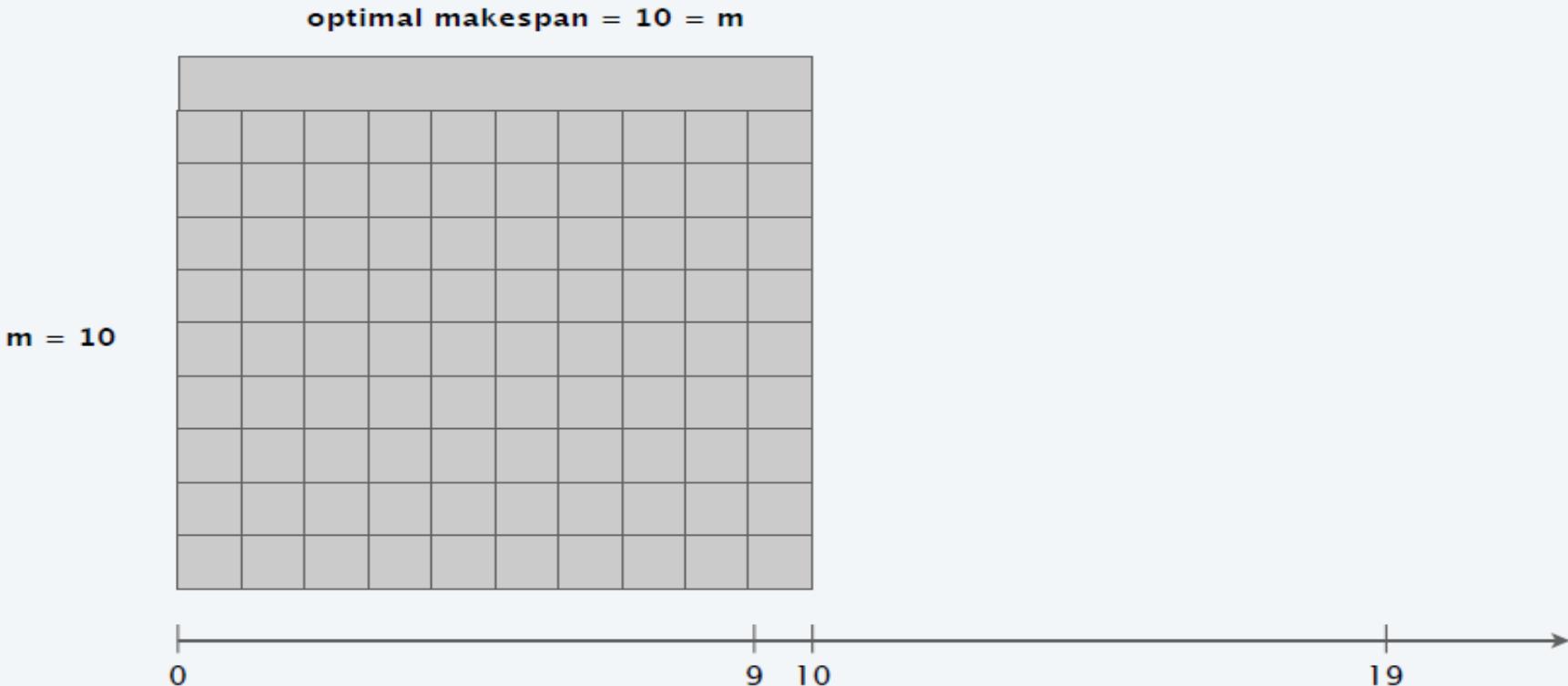


Load balancing: list scheduling analysis

Q. Is our analysis tight?

A. Essentially yes.

Ex: m machines, first $m(m - 1)$ jobs have length 1, last job has length m .



Load balancing: LPT rule

Longest processing time (LPT). Sort n jobs in decreasing order of processing times; then run list scheduling algorithm.

LPT-LIST-SCHEDULING ($m, n, t_1, t_2, \dots, t_n$)

SORT jobs and renumber so that $t_1 \geq t_2 \geq \dots \geq t_n$.

FOR $i = 1$ TO m

$L[i] \leftarrow 0$. \leftarrow load on machine i

$S[i] \leftarrow \emptyset$. \leftarrow jobs assigned to machine i

FOR $j = 1$ TO n

$i \leftarrow \operatorname{argmin}_k L[k]$. \leftarrow machine i has smallest load

$S[i] \leftarrow S[i] \cup \{j\}$. \leftarrow assign job j to machine i

$L[i] \leftarrow L[i] + t_j$. \leftarrow update load of machine i

RETURN $S[1], S[2], \dots, S[m]$.

Load balancing: LPT rule

Observation. If bottleneck machine i has only 1 job, then optimal.

Pf. Any solution must schedule that job. ■

Lemma 3. If there are more than m jobs, $L^* \geq 2t_{m+1}$.

Pf.

- Consider processing times of first $m+1$ jobs $t_1 \geq t_2 \geq \dots \geq t_{m+1}$.
- Each takes at least t_{m+1} time.
- There are $m+1$ jobs and m machines, so by pigeonhole principle, at least one machine gets two jobs. ■

Theorem. LPT rule is a $3/2$ -approximation algorithm.

Pf. [similar to proof for list scheduling]

- Consider load $L[i]$ of bottleneck machine i .
- Let j be last job scheduled on machine i . assuming machine i has at least 2 jobs,
we have $j \geq m + 1$

$$L = L[i] = (\underbrace{L[i] - t_j}_{\text{as before}}) + \underbrace{t_j}_{\leq L^*} \leq \frac{3}{2} L^* .$$

$\leq \frac{1}{2} L^*$ ← Lemma 3 (since $t_{m+1} \geq t_j$)

Load balancing: LPT rule

Q. Is our $3/2$ analysis tight?

A. No.

Theorem. [Graham 1969] LPT rule is a $4/3$ -approximation.

Pf. More sophisticated analysis of same algorithm.

Q. Is Graham's $4/3$ analysis tight?

A. Essentially yes.

Ex.

- m machines
- $n = 2m + 1$ jobs
- 2 jobs of length $m, m+1, \dots, 2m-1$ and one more job of length m .
- Then, $L / L^* = (4m - 1) / (3m)$

Generalized load balancing

Input. Set of m machines M ; set of n jobs J .

- Job $j \in J$ must run contiguously on an authorized machine in $M_j \subseteq M$.
- Job $j \in J$ has processing time t_j .
- Each machine can process at most one job at a time.

Def. Let J_i be the subset of jobs assigned to machine i .

The load of machine i is $L_i = \sum_{j \in J_i} t_j$.

Def. The makespan is the maximum load on any machine = $\max_i L_i$.

Generalized load balancing. Assign each job to an authorized machine to minimize makespan.

Generalized load balancing: integer linear program and relaxation

ILP formulation. x_{ij} = time machine i spends processing job j .

$$(IP) \min L$$

$$\text{s. t. } \sum_i x_{ij} = t_j \quad \text{for all } j \in J$$

$$\sum_j x_{ij} \leq L \quad \text{for all } i \in M$$

$$x_{ij} \in \{0, t_j\} \quad \text{for all } j \in J \text{ and } i \in M_j$$

$$x_{ij} = 0 \quad \text{for all } j \in J \text{ and } i \notin M_j$$

LP relaxation.

$$(LP) \min L$$

$$\text{s. t. } \sum_i x_{ij} = t_j \quad \text{for all } j \in J$$

$$\sum_j x_{ij} \leq L \quad \text{for all } i \in M$$

$$x_{ij} \geq 0 \quad \text{for all } j \in J \text{ and } i \in M_j$$

$$x_{ij} = 0 \quad \text{for all } j \in J \text{ and } i \notin M_j$$

Generalized load balancing: lower bounds

Lemma 1. The optimal makespan $L^* \geq \max_j t_j$.

Pf. Some machine must process the most time-consuming job. ■

Lemma 2. Let L be optimal value to the LP . Then, optimal makespan $L^* \geq L$.

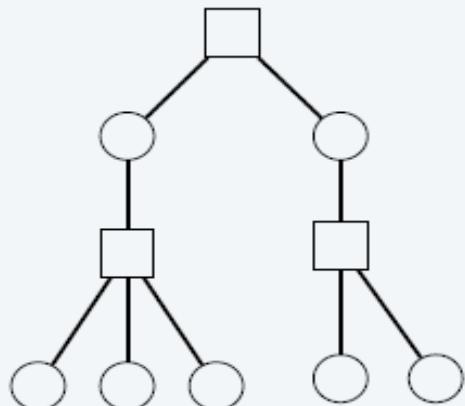
Pf. LP has fewer constraints than ILP formulation. ■

Generalized load balancing: structure of LP solution

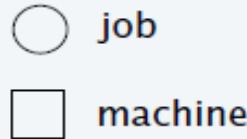
Lemma 3. Let x be solution to LP . Let $G(x)$ be the graph with an edge between machine i and job j if $x_{ij} > 0$. Then $G(x)$ is acyclic.

Pf. (deferred)

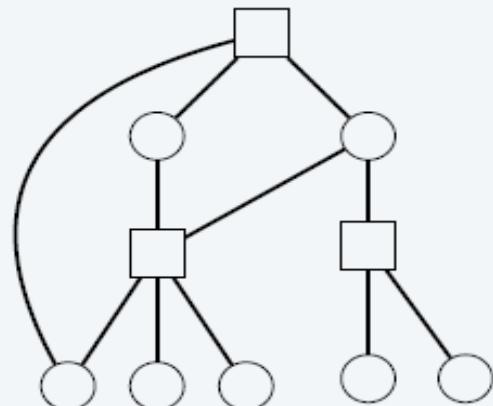
can transform x into another LP solution where $G(x)$ is acyclic if LP solver doesn't return such an x



$G(x)$ acyclic



$$x_{ij} > 0$$



$G(x)$ cyclic

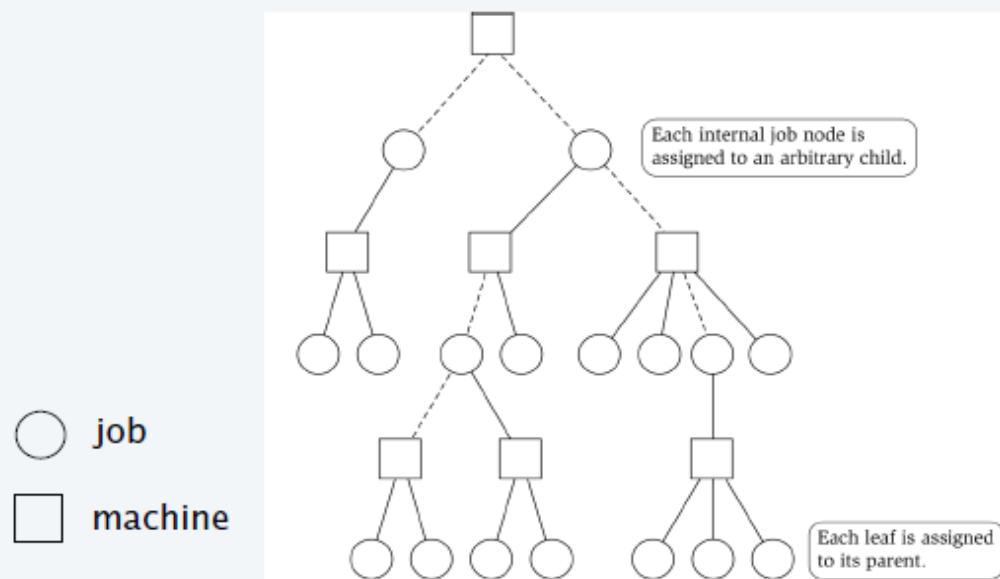
Generalized load balancing: rounding

Rounded solution. Find LP solution x where $G(x)$ is a forest. Root forest $G(x)$ at some arbitrary machine node r .

- If job j is a leaf node, assign j to its parent machine i .
- If job j is not a leaf node, assign j to any one of its children.

Lemma 4. Rounded solution only assigns jobs to authorized machines.

Pf. If job j is assigned to machine i , then $x_{ij} > 0$. LP solution can only assign positive value to authorized machines. ■



Generalized load balancing: analysis

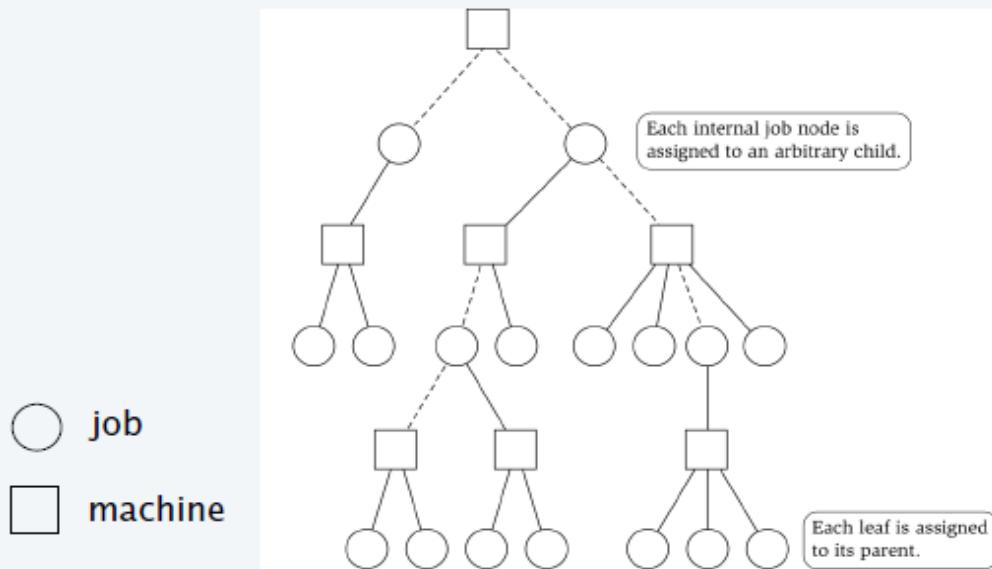
Lemma 5. If job j is a leaf node and machine $i = \text{parent}(j)$, then $x_{ij} = t_j$.

Pf.

- Since i is a leaf, $x_{ij} = 0$ for all $j \neq \text{parent}(i)$.
- LP constraint guarantees $\sum_i x_{ij} = t_j$. ■

Lemma 6. At most one non-leaf job is assigned to a machine.

Pf. The only possible non-leaf job assigned to machine i is $\text{parent}(i)$. ■



Generalized load balancing: analysis

Theorem. Rounded solution is a 2-approximation.

Pf.

- Let $J(i)$ be the jobs assigned to machine i .
- By LEMMA 6, the load L_i on machine i has two components:

- leaf nodes:

$$\sum_{\substack{j \in J(i) \\ j \text{ is a leaf}}} t_j \xrightarrow{\text{Lemma 5}} \sum_{\substack{j \in J(i) \\ j \text{ is a leaf}}} x_{ij} \leq \sum_{j \in J} x_{ij} \xrightarrow[\text{optimal value of LP}]{\substack{\text{LP} \\ \text{Lemma 2 (LP is a relaxation)}}} L \leq L^*$$

- parent: $t_{\text{parent}(i)} \leq L^*$

- Thus, the overall load $L_i \leq 2L^*$. ■

Generalized load balancing: flow formulation

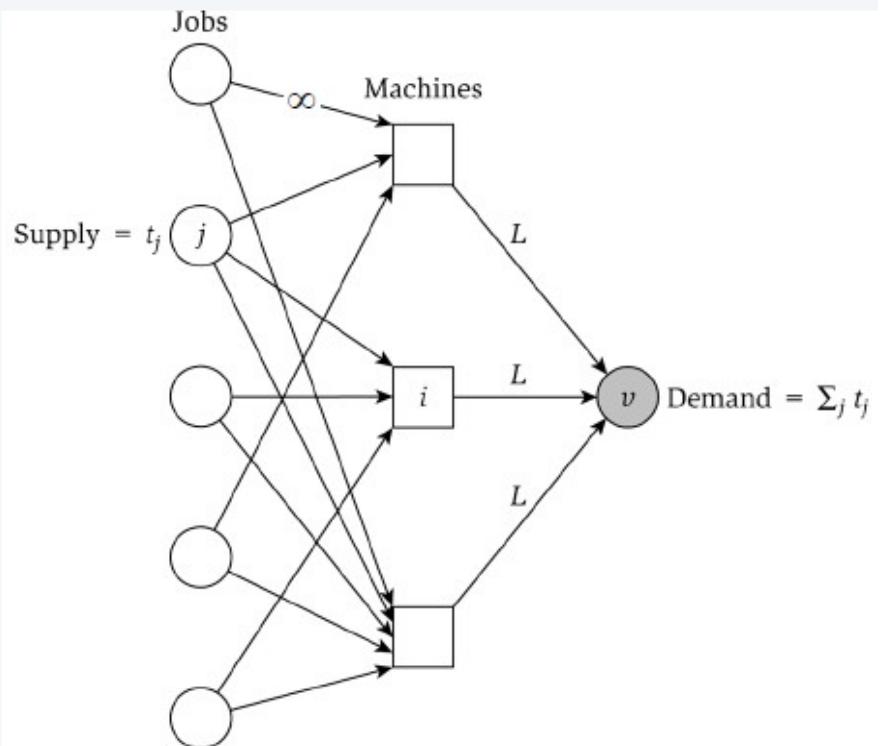
Flow formulation of *LP*.

$$\sum_i x_{ij} = t_j \quad \text{for all } j \in J$$

$$\sum_j x_{ij} \leq L \quad \text{for all } i \in M$$

$$x_{ij} \geq 0 \quad \text{for all } j \in J \text{ and } i \in M_j$$

$$x_{ij} = 0 \quad \text{for all } j \in J \text{ and } i \notin M_j$$



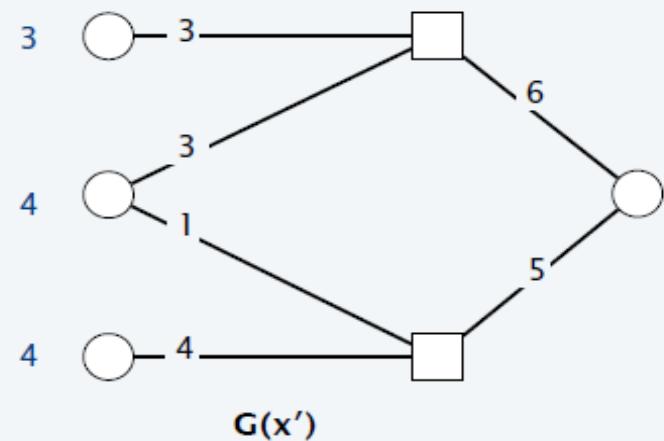
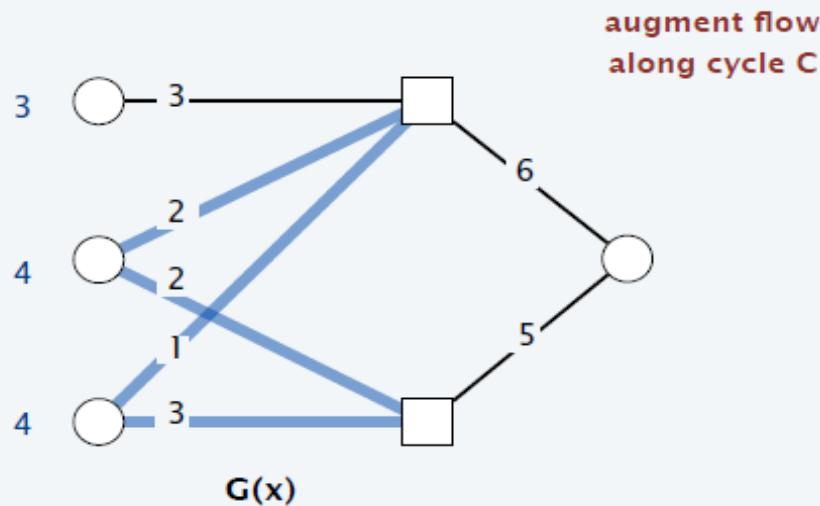
Observation. Solution to feasible flow problem with value L are in 1-to-1 correspondence with *LP* solutions of value L .

Generalized load balancing: structure of solution

Lemma 3. Let (x, L) be solution to LP . Let $G(x)$ be the graph with an edge from machine i to job j if $x_{ij} > 0$. We can find another solution (x', L) such that $G(x')$ is acyclic.

Pf. Let C be a cycle in $G(x)$.

- Augment flow along the cycle C . \leftarrow flow conservation maintained
 - At least one edge from C is removed (and none are added).
 - Repeat until $G(x')$ is acyclic. ■



Conclusions

Running time. The bottleneck operation in our 2-approximation is solving one LP with $mn + 1$ variables.

Remark. Can solve LP using flow techniques on a graph with $m+n+1$ nodes: given L , find feasible flow if it exists. Binary search to find L^* .

Extensions: unrelated parallel machines. [Lenstra–Shmoys–Tardos 1990]

- Job j takes t_{ij} time if processed on machine i .
- 2-approximation algorithm via LP rounding.
- If $P \neq NP$, then no ρ -approximation exists for any $\rho < 3/2$.

Local Search and Optimization

Outline

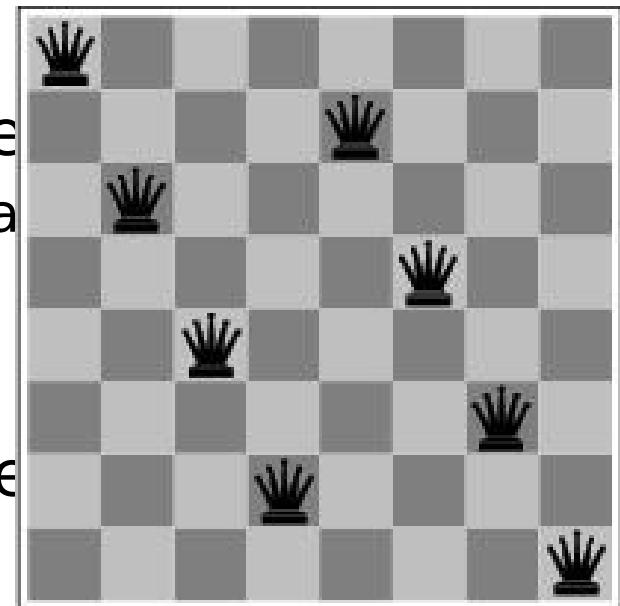
- Local Search Algorithms and Optimization
 - Hill-Climbing
 - Gradient Methods
 - Simulated Annealing
 - Genetic Algorithms
 - Issues with Local Search

Local Search Algorithms

- In many optimization problems, the **path** to the goal is irrelevant; the goal state itself is the solution
 - Local Search: It is widely used for *very big* problems
 - Returns Good Solution but *not Optimal* Solutions
- State Space = Set of "complete" configurations
- Find configuration satisfying constraints
 - Examples: n-Queens, VLSI Layout, Airline Flight Schedules
- **Local Search Algorithms**
 - Keep a single "current" state, or small set of states

Local Search and Optimization

- Path to goal is a solution to the problem
 - Systematic exploration of search space.
- State is a solution to the problem
 - for some problems path is irrelevant
 - E.g., 8-queens
- Different algorithms can be used
 - Depth First Branch and Bound
 - Local Search



Goal Satisfaction

Reach the Goal Node
Constraint Satisfaction

Optimization

Optimize(Objective Fn)
Constraint Optimization

We can go back and forth between the two problems
Typically in the same complexity class

Local Search and Optimization

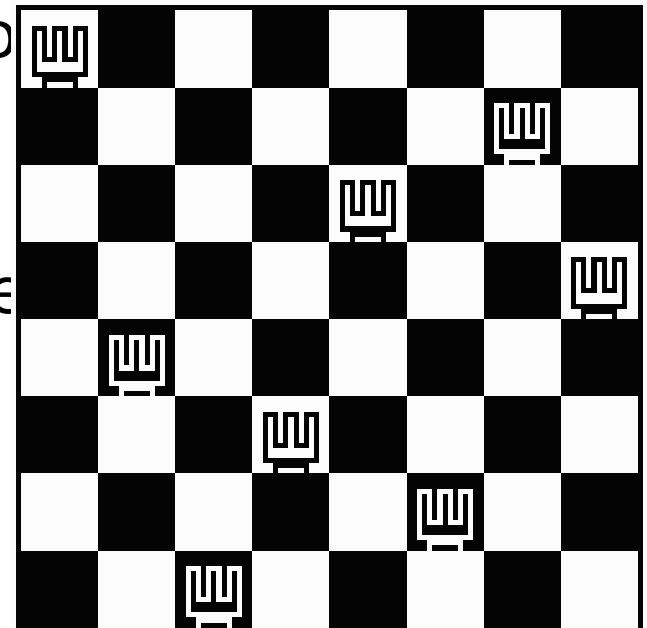
- Local Search
 - Keep track of single current state
 - Move only to neighboring states
 - Ignore paths
- Advantages:
 - Use **very little memory** (space)
 - Can often find **Reasonable Solutions** in large or infinite (continuous) state spaces for which the **other algorithms are not suitable**.
- “Pure Optimization” Problems
 - All states have an Objective Function
 - Goal is to find state with max (or min) objective value
 - Does not quite fit into path-cost/goal-state formulation
 - Local search can do quite well on these problems.

Optimization

- Local search is often suitable for optimization problems. Search for best state by optimizing an **Objective Function**.
- $F(x)$ where x is a vector of continuous or discrete values
- Begin with a complete configuration
- A successor of state S is S with a single element changed
- Move from the current state to a successor state
- Low memory requirements, because the search tree or graph is not maintained

Examples

- 8 queens: Find an arrangement of 8 queens on a chess board such that no two queens are attacking each other
- Start with some arrangement of queen per column
- $X[j]$: row of queen in column j
- Successors of a state: move one queen
- F : # pairs attacking each other



Examples

- Traveling Salesperson Problem: Visit each city exactly once
- Start with some ordering of the cities
- State Representation – Order of the cities visited
- Successor State: A change to the current ordering
- F : Length of the route

Examples

- Flight Travel Problem
- Flight Schedule. The general problem is for all members of the family to travel to the same place.
- A state consists of flights for each member of the family.
- Successor States: All schedules that have one person on the next later or the earlier departing or returning flight.
- F : Sum of different types of costs (\$\$, time, etc)

Examples

- Cryptosystems: Resistance to types of attacks is often discussed in terms of Boolean functions used in them
- Much work on constructing the Boolean functions with desired cryptographic properties (balancedness, high nonlinearity, etc.)
- One Approach: Local search, where states are represented with truth tables (**that's a simplification**); a successor results from a change to the truth table; objective functions have been devised to assess (estimate) relevant qualities of the Boolean functions

Examples

- Racing yacht hull design
- Design representation has multiple components, including a vector of B-Spline surfaces.
- Successors: Modification of a B-Spline surface.
- Objective Function estimates the time the yacht would take to traverse a course given certain wind conditions

Visualization

- States are laid out in a landscape
- Height corresponds to the objective function value
- Move around the landscape to find the highest
(or lowest) peak
- Only keep track of the current states and immediate neighbors

Algorithm Design Considerations

- How do we represent our problem?
- What is a “complete state”?
- What is our objective function?
 - How do we measure cost or value of a state?
- What is a “neighbor” of a state?
 - Or, what is a “step” from one state to another?
 - How can we compute a neighbor or a step?
- Are there any constraints we can exploit?

Trivial Algorithms

- Random Sampling
 - Generate a state randomly
- Random Walk
 - Randomly pick a neighbor of the current state
- Both algorithms asymptotically complete.

Random Restart Wrapper

- We'll use Stochastic Local Search methods
 - Return different solution for each trial & initial state
- Almost every trial hits difficulties (see sequel)
 - Most trials will not yield a good result (sad!)
- Using many random restarts improves your chances
 - Many “shots at goal” may finally get a good one
- Restart a random initial state, *many times*
 - Report the best result found across *many* trials

Random Restart Wrapper

```
best_found ← RandomState() //initialize to something  
  
while not (tired of doing it): //now do repeated local search  
    result ← LocalSearch( RandomState() )  
    if (Cost(result) < Cost(best_found)):  
        best_found = result //keep best result found so far  
  
return best_found
```

Typically, “you are tired of doing it” means that some resource limit is exceeded, e.g., number of iterations, wall clock time, CPU time, etc. It may also mean that Result improvements are small and infrequent, e.g., less than 0.1% Result improvement in the last week of run time.

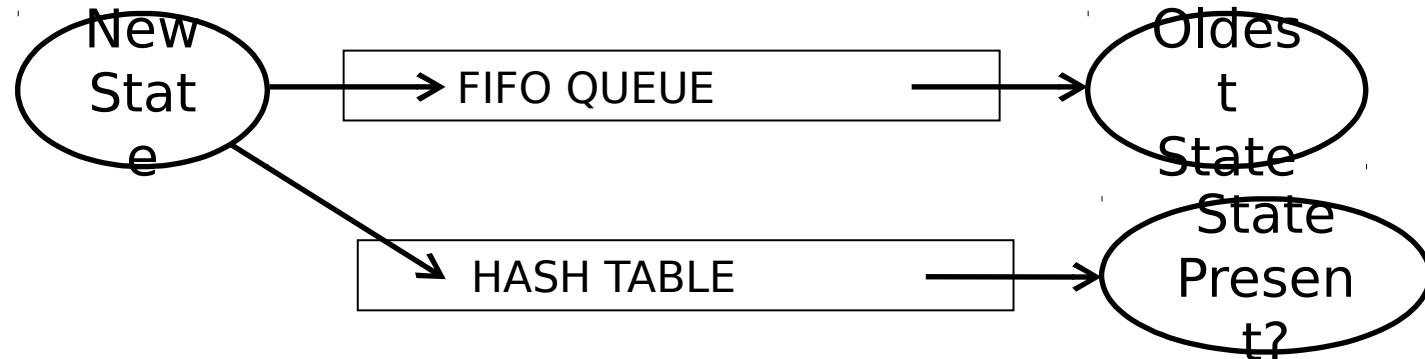
Tabu Search

- Prevent returning quickly to the same state
- Keep fixed length queue (“Tabu list”)
- add most recent state to queue; drop oldest
- Never make the step that is currently Tabu’ed
- Properties:
 - As the size of the Tabu list grows, hill-climbing will asymptotically become “non-redundant” (won’t look at the same state twice)
 - In practice, a reasonable sized Tabu list (say 100 or so) improves the performance of hill climbing in many problems

Tabu Search Wrapper

- Add recently visited states to a tabu-list
 - Temporarily excluded from being visited again
 - Forces solver away from explored regions
 - Avoid getting stuck in local minima (in principle)
- Implemented as a hash table + FIFO queue
 - Unit time cost per step; constant memory cost
 - You control how much memory is used

Tabu Search Wrapper



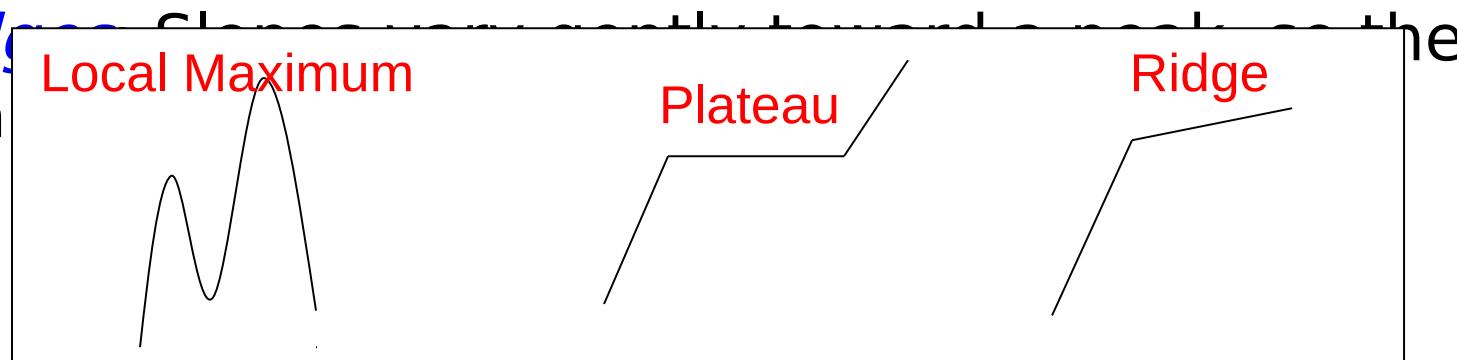
```
UNTIL ( you are tired of doing it ) DO {  
    set Neighbor to makeNeighbor( CurrentState );  
    IF ( Neighbor is in HASH ) THEN ( discard Neighbor );  
    ELSE { push Neighbor onto FIFO, pop OldestState;  
          remove OldestState from HASH, insert  
          Neighbor;  
          set CurrentState to Neighbor;  
          run yourFavoriteLocalSearch on  
          CurrentState; } }
```

Local Search Algorithms

- ❑ Two strategies for choosing the state to visit next
 - Hill-Climbing Search
 - Gradient Descent in continuous state spaces
 - Can use e.g. Newton's method to find roots
 - Simulated Annealing Search
- ❑ Then, an extension to multiple current states:
 - Local Beam Search
 - Genetic Algorithm

Hill Climbing (Greedy Local Search)

- Generate nearby successor states to the current state
- Pick the best and replace the current state with that one.
- Loop
- *Local Maximum*: A peak that is lower than the highest peak, so a **Suboptimal Solution** is returned
- *Plateau*: The evaluation function is flat, resulting in a random walk
- *Ridge*: Stuck in a local maximum



Hill-Climbing (Greedy Local Search)

max version

function HILL-CLIMBING(*problem*) **return** a state that is a local maximum

input: *problem*, a problem

local variables: *current*, a node.
neighbor, a node.

current \leftarrow MAKE-NODE(INITIAL-STATE[*problem*])

loop do

neighbor \leftarrow the highest valued successor of *current*

if VALUE [*neighbor*] \leq VALUE[*current*] **then return**
 STATE[*current*]

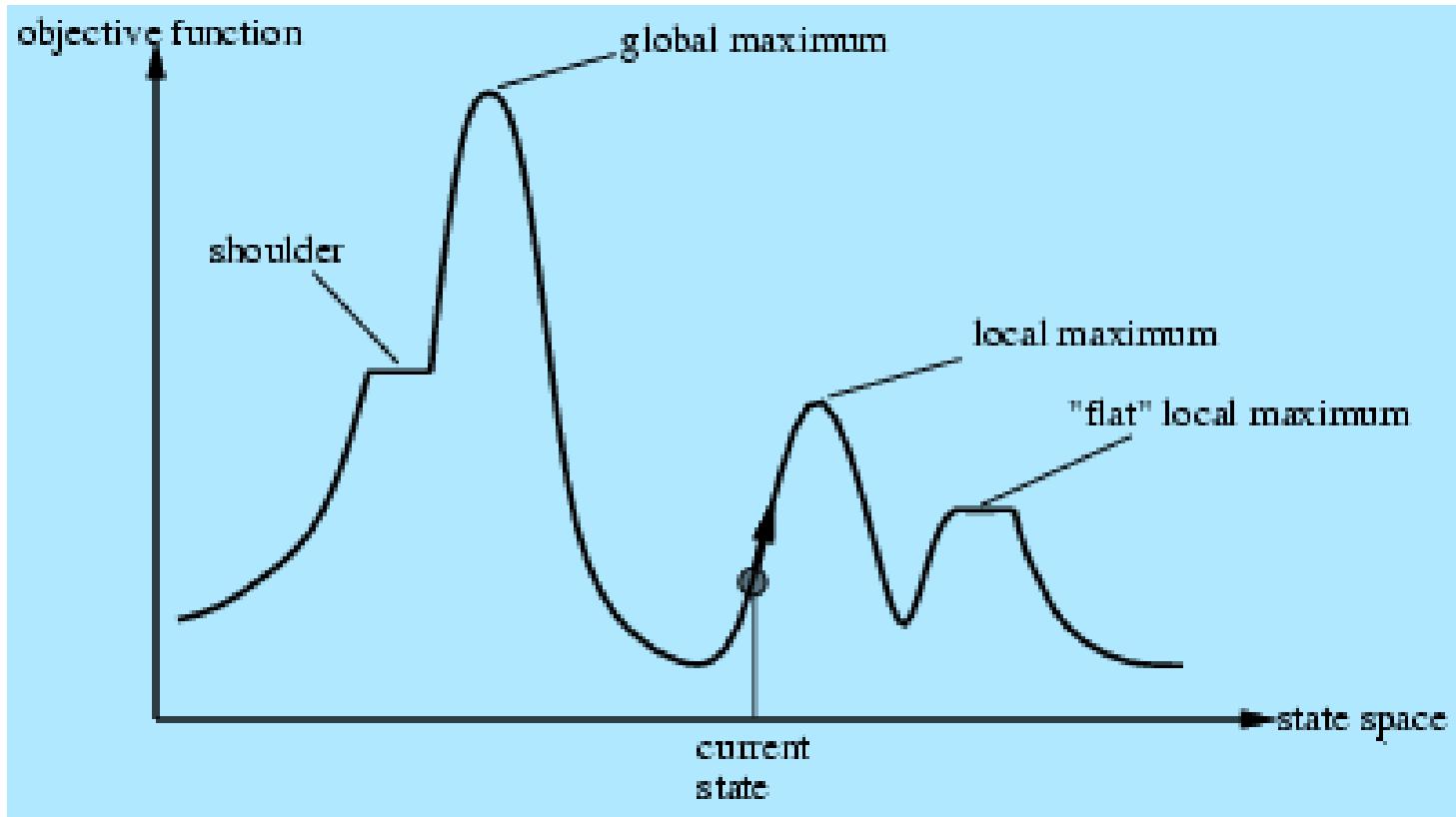
current \leftarrow *neighbor*

min version will reverse inequalities
and look for the lowest valued
successor

Hill-Climbing Search

- “A loop that continuously moves towards increasing value”
 - terminates when a peak is reached
 - Aka **Greedy Local Search**
- Value can be either
 - Objective Function Value
 - Heuristic Function Value (Minimized)
- Hill Climbing does not look ahead of the immediate neighbors
- Can randomly choose among the set of best successors
 - **NOTE: Amnesia refers to the Loss of Memories, such as Facts, Information and Experiences.**
- **“Climbing Mount Everest in a Thick Fog with Amnesia”**

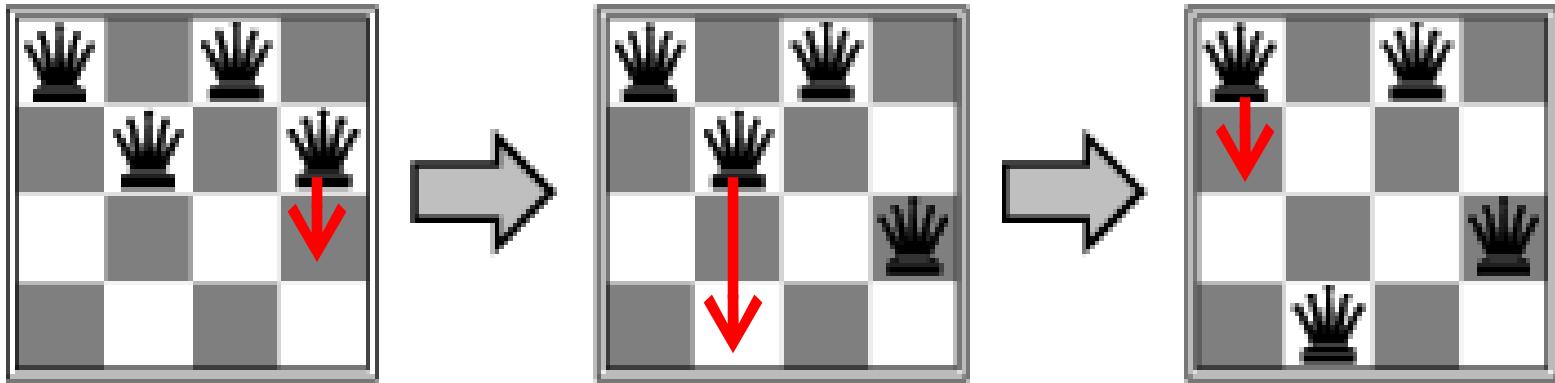
“Landscape” of Search



Hill Climbing gets stuck in Local Minima
depending on?

Example: n -queens

- Goal: Put n queens on an $n \times n$ board with no two queens on the same row, column, or diagonal
- Neighbor: Move one queen to another row
- Search: Go from one neighbor to the next...



- Is it a Satisfaction or Optimization Problem?

Ex: Hill-Climbing Search, 8-queens Problem

Need to convert this to an Optimization Problem

$h = \#$ of pairs of queens that are attacking each other, either directly or indirectly

$h = 17$ for this state

18	12	14	13	13	12	14	14
14	16	13	15	12	14	12	16
14	12	18	13	15	12	14	14
15	14	14	14	13	16	13	16
14	14	17	15	14	16	16	16
17	15	16	18	15	14	15	15
18	14	15	15	15	14	14	16
14	14	13	17	12	14	12	18

Each number indicates h if we move a queen in its column to that square

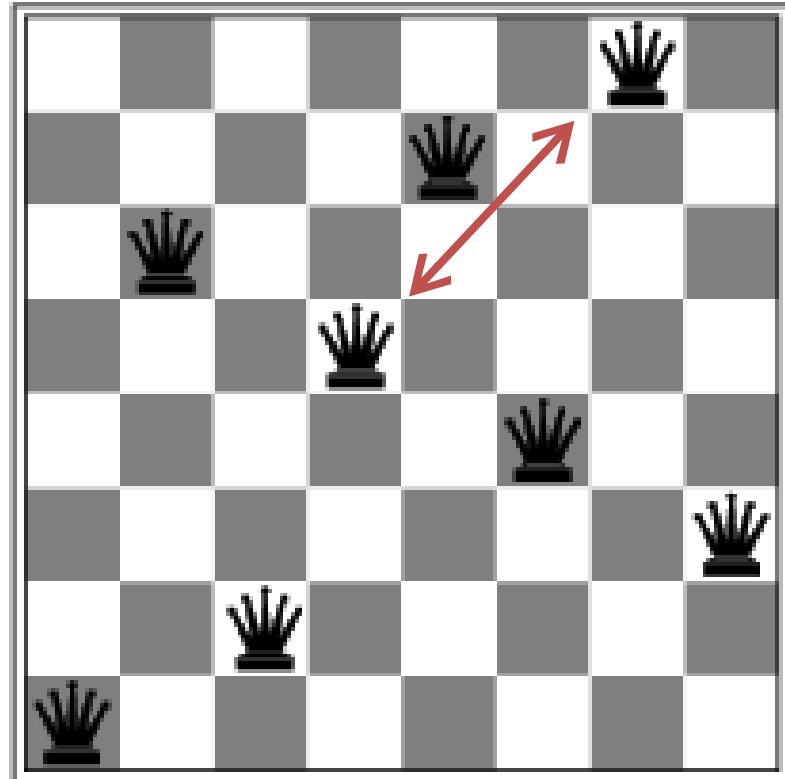
12 (boxed) = best h among all neighbors; select one randomly

Search Space

- State
 - All 8 queens on the chess board in some configuration
- Successor Function
 - move a queen to another square in the same column.
- Example of a Heuristic Function $h(n)$:
 - The # of pairs of queens that are attacking each other
 - (so we want to minimize this)

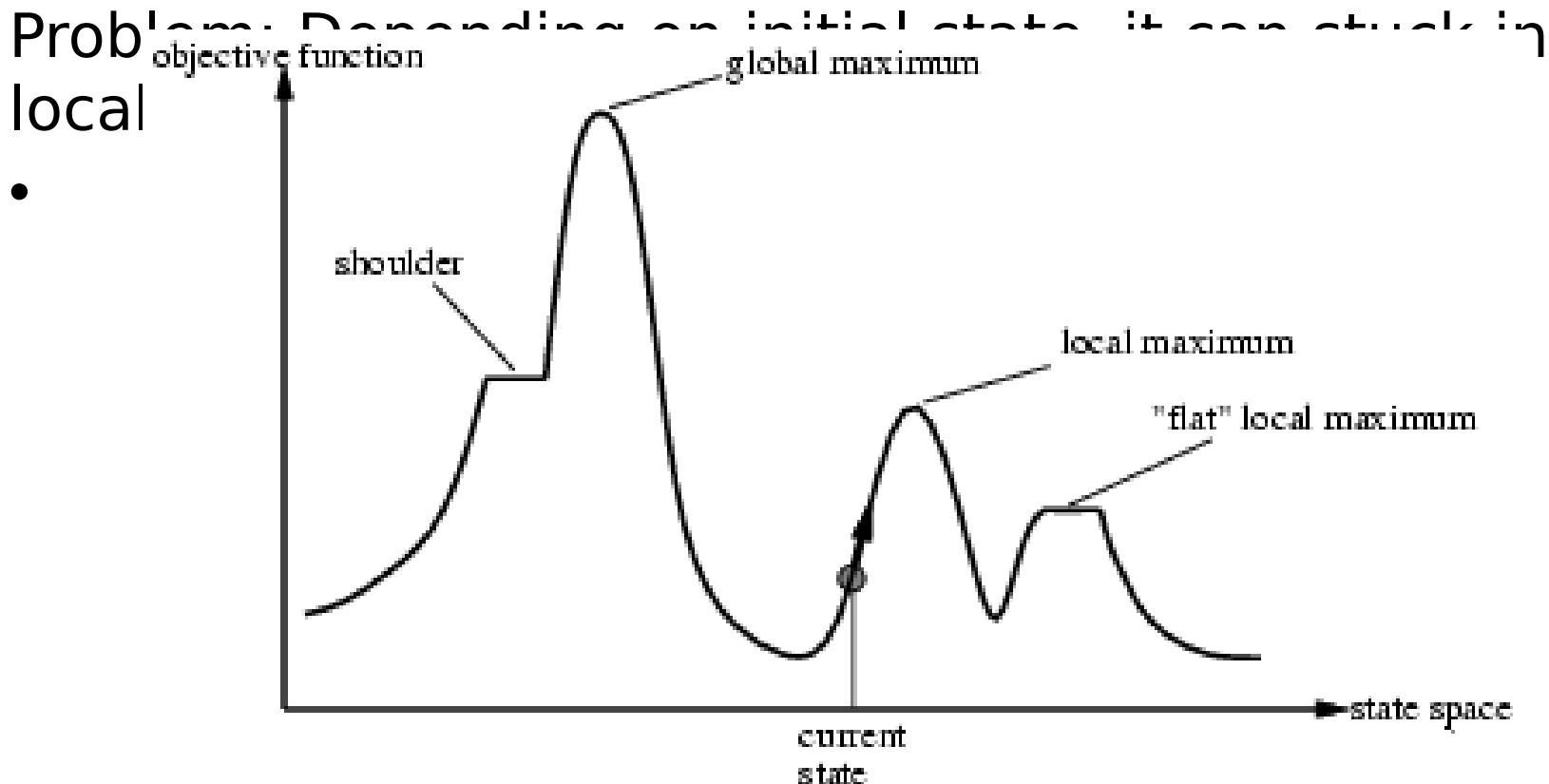
Hill-Climbing Search: 8-queens Problem

- Is this a solution?
- What is the value of h ?
- A local minimum with $h=1$
- All one-step neighbors have higher h values
- What can we do to get out of this local minimum?



Hill-Climbing Difficulties

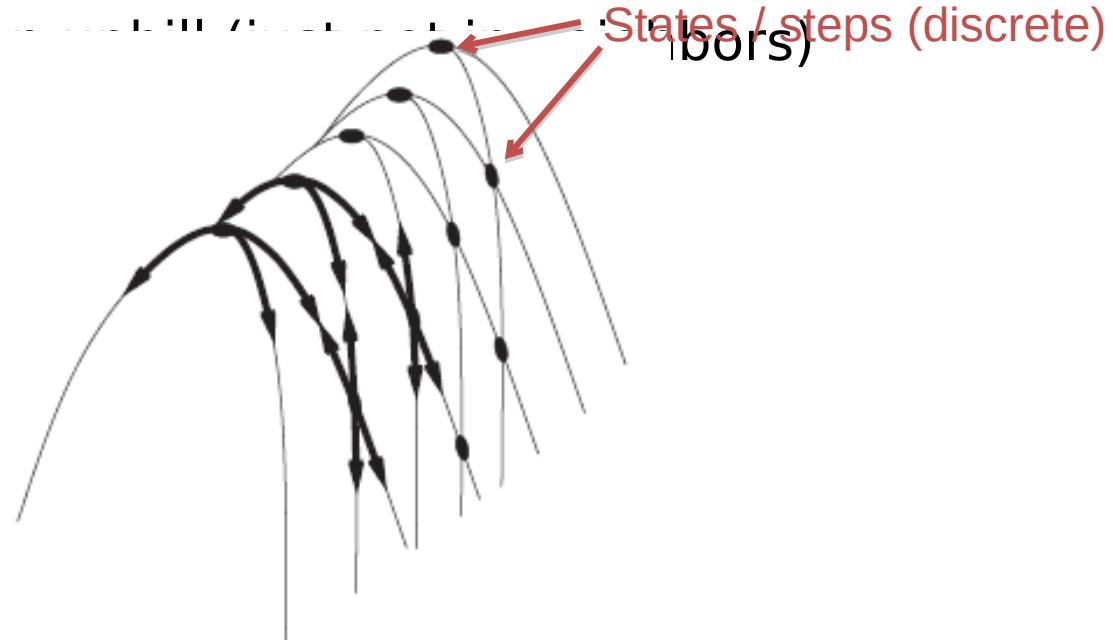
Note: These difficulties apply to all local search algorithms, and usually become much worse as the search space becomes higher dimensional



Hill-Climbing Difficulties

Note: These difficulties apply to all local search algorithms, and usually become much worse as the search space becomes higher dimensional

- Ridge Problem: Every neighbor appears to be downhill
- But: search space has Ridge:
Fold a piece of paper and hold it tilted up at an unfavorable angle to every possible search space step. Every step leads downhill; but the ridge leads uphill.

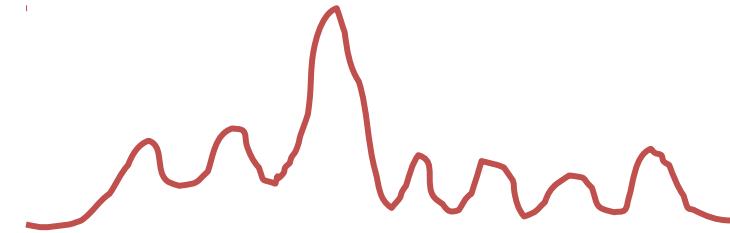


Hill-Climbing on 8-queens

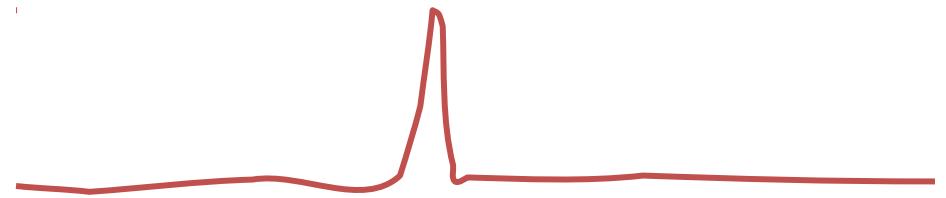
- Randomly generated 8-queens starting states...
- 14% the time it solves the problem
- 86% of the time it get stuck at a Local Minimum
- However...
 - Takes only 4 steps on average when it succeeds
 - And 3 on average when it gets stuck
(for a state space with $8^8 = \sim 17$ million)

Hill Climbing Drawbacks

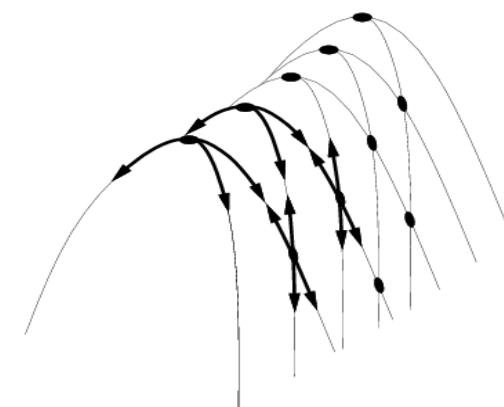
- Local Maxima



- Plateaus



- Diagonal
Ridges



Escaping Shoulders: Sideways Move

- If no downhill (uphill) moves, allow sideways moves in hope that algorithm can escape
 - Need to place a limit on the possible number of sideways moves to avoid infinite loops
- For 8-queens
 - Now allow sideways moves with a limit of 100
 - Raises percentage of problem instances solved from 14 to 94%
 - However....
 - 21 steps for every successful solution
 - 64 for each failure

Escaping Shoulders/Local Optima Enforced Hill Climbing

- Perform Breadth First Search from a Local Optima
 - to find the next state with better h function
- Typically,
 - prolonged periods of exhaustive search
 - bridged by relatively quick periods of hill-climbing
- Middle ground b/w local and systematic

Hill-Climbing: Stochastic Variations

- Stochastic Hill-Climbing
 - Random selection among the uphill moves.
 - The selection probability can vary with the steepness of the uphill move.
- To avoid getting stuck in Local Minima
 - Random-walk Hill-Climbing
 - Random-restart Hill-Climbing
 - Hill-Climbing with both

Hill Climbing: Stochastic Variations

- When the state-space landscape has local minima, any search that moves only in the greedy direction cannot be complete
- Random walk, on the other hand, is asymptotically complete

Idea: Put random walk into Greedy Hill-Climbing

Hill-Climbing with Random Restarts

- If at first we don't succeed, try, try again!
- Different variations
 - For each restart: run until termination vs. run for a fixed time
 - Run a fixed number of restarts or run indefinitely
- Analysis
 - Say each search has probability p of success
 - E.g., for 8-queens, $p = 0.14$ with no sideways moves
 - Expected number of restarts?
 - Expected number of steps taken?
- If we want to pick one local search algorithm, learn this one!!

Hill-Climbing with Random Walk

- At each step do one of the two
 - Greedy: With prob. p move to the neighbor with largest value
 - Random: With prob. $1-p$ move to a random neighbor

Hill-Climbing with both

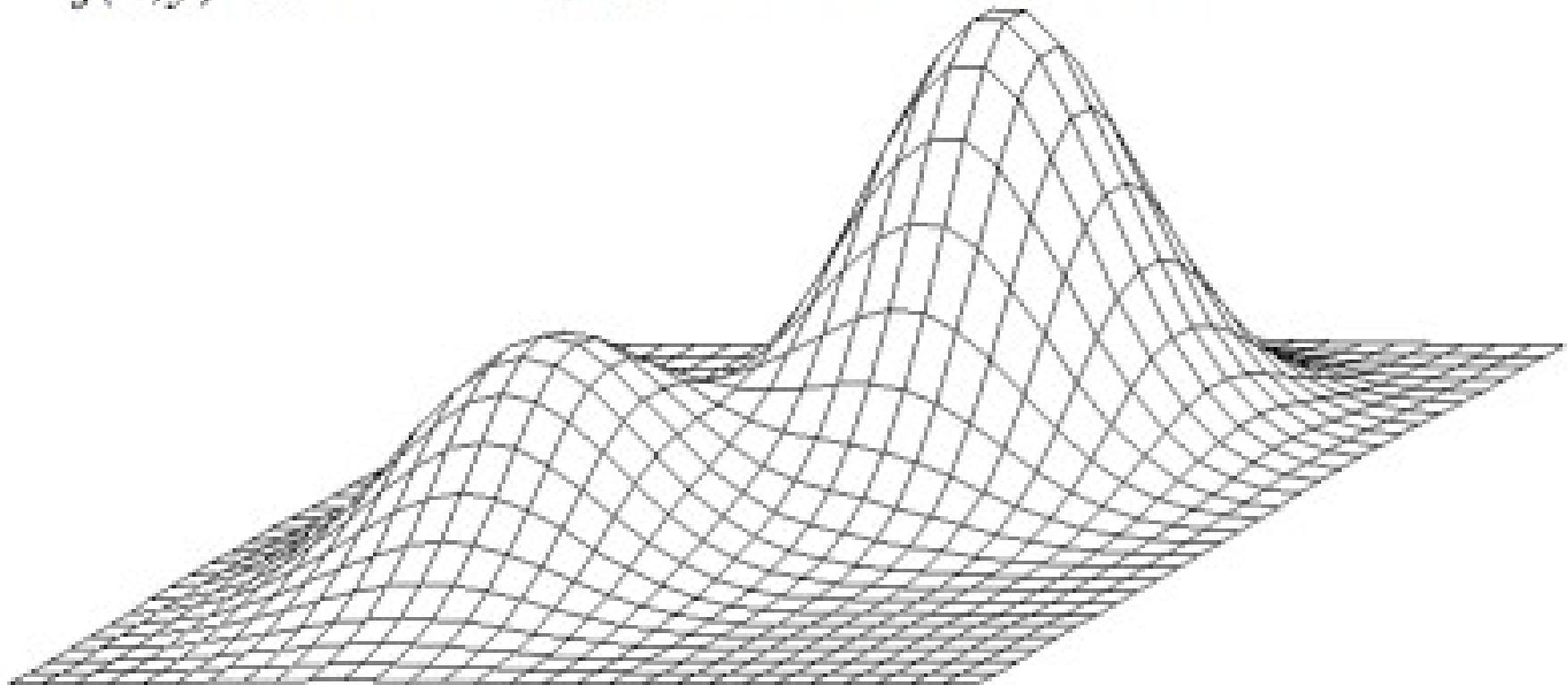
- At each step do one of the three
 - Greedy: move to the neighbor with largest value
 - Random Walk: move to a random neighbor
 - Random Restart: Resample a new current state

Optimization of Continuous Functions

- Discretization
 - Use Hill-Climbing
- Gradient Descent
 - Make a move in the direction of the gradient
 - Gradients: Closed Form or Empirical

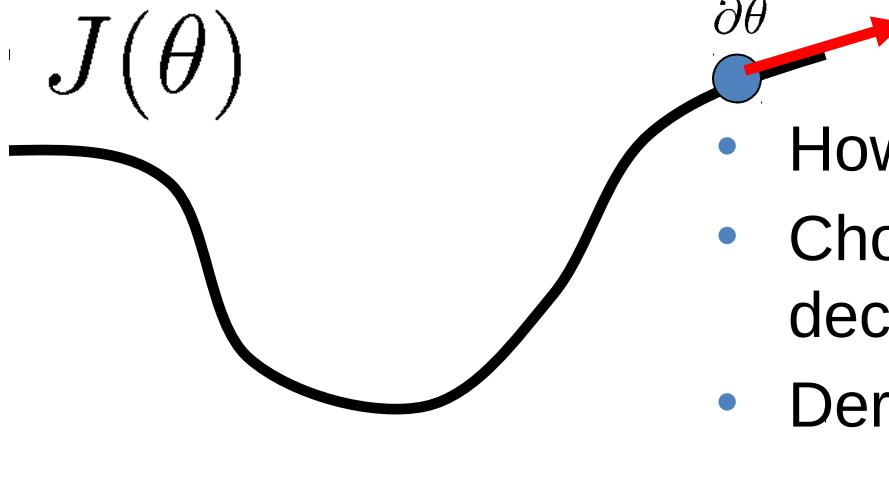
Gradient Descent

$$f(x,y) = e^{-(x^2+y^2)} + 2e^{-((x-1.7)^2+(y-1.7)^2)}$$



Gradient Descent

- Hill-Climbing in Continuous State Spaces
- Denote “State” $\frac{\partial J(\theta)}{\partial \theta}$; Cost as $J(\mu)$



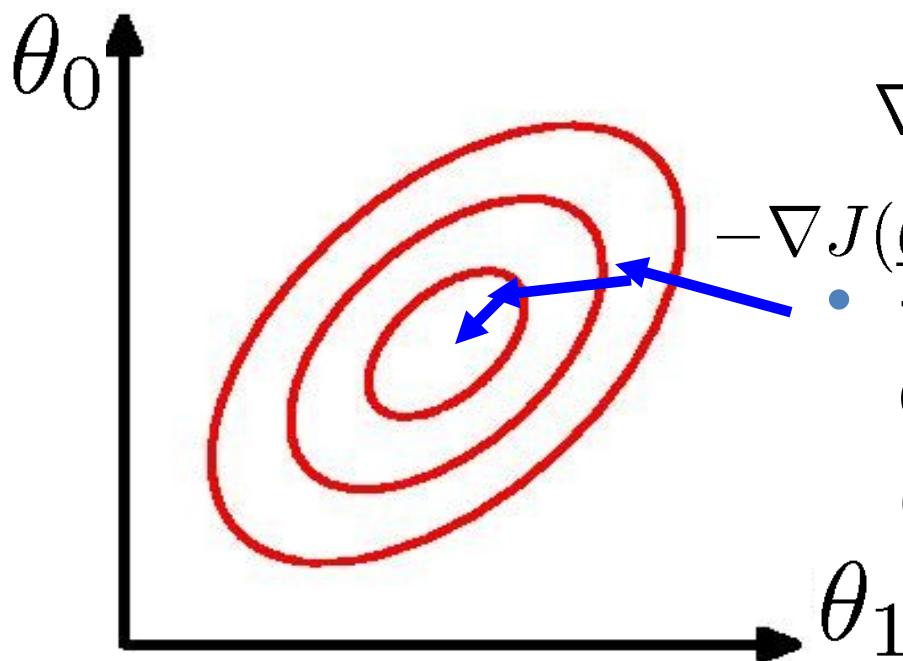
- How to change μ to improve $J(\mu)$?
- Choose a direction in which $J(\mu)$ is decreasing
- Derivative $\frac{\partial J(\theta)}{\partial \theta}$
 - Positive (+ve) => Increasing
 - Negative (-ve) => Decreasing

Gradient Descent

■ Hill-Climbing in Continuous Spaces

- Gradient Vector

$$\nabla J(\underline{\theta}) = \left[\frac{\partial J(\underline{\theta})}{\partial \theta_0} \quad \frac{\partial J(\underline{\theta})}{\partial \theta_1} \quad \dots \right]$$

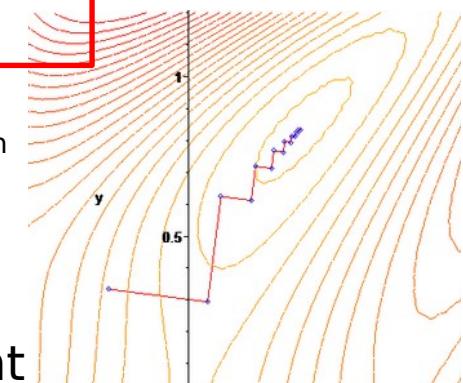
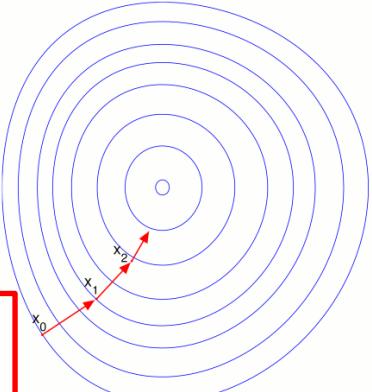


- +ve Indicates the direction of Steepest Ascent
(-ve = Steepest Descent)

Gradient Descent

■ Hill-Climbing in Continuous Spaces

Gradient = The most direct direction up-hill in the Objective (Cost) Function, so its negative direction minimizes the Cost Function.



* Assume we have some cost-function $J(x_1, x_2, \dots, x_n)$ and we want minimize over continuous variables x_1, x_2, \dots, x_n

1. Compute the *gradient* : $\frac{\partial}{\partial x_i} J(x_1, \dots, x_n) \quad \forall i$

2. Take a small step downhill in the direction of the gradient

$$x'_i = x_i - \lambda \frac{\partial}{\partial x_i} J(x_1, \dots, x_n)$$

3. Check if $J(x'_1, \dots, x'_n) < J(x_1, \dots, x_n)$

- How to select λ

- Line search: successively double

4. If true then accept move, if not “reject” (decrease step size)

- until f starts to increase again

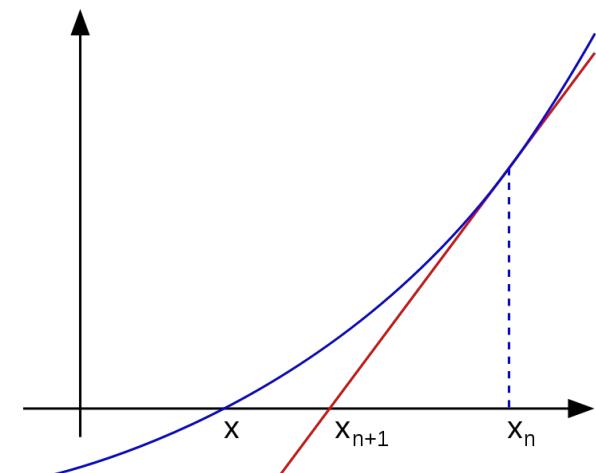
5. Repeat.

Gradient Descent

- Hill-Climbing in Continuous Spaces
 - How do we determine the gradient?
 - Derive formula using multivariate calculus.
 - Ask a mathematician or a domain expert.
 - Do a literature search.
 - Variations of gradient descent can improve performance for this or that special case.
 - See Numerical Recipes in C (and in other languages) by Press, Teukolsky, Vetterling, and Flannery.
 - Simulated Annealing, Linear Programming too
 - Works well in Smooth Spaces; poorly in rough

Newton-Rapison Method for Function Minimization

- Want to find the roots of a polynomial $f(x)$
 - “Root”: value of x for which $f(x)=0$
- Initialize to *some* point x
- To minimize a function $f(x)$, we need to find the roots of $f'(x) = \nabla f(x) = 0$
- Compute the step size $\frac{0 - f(x)}{\nabla f(x)} \Rightarrow x' = x - \frac{f(x)}{\nabla f(x)}$ crosses x-axis
- Opt $\nabla \nabla f(x) = \frac{0 - \nabla f(x)}{x' - x} \Rightarrow x' = x - \frac{\nabla f(x)}{\nabla \nabla f(x)}$
- Does not always converge; sometimes unstable
 - If converges, usually very fast
 - Works well for smooth, non-pathological functions, linearization
 - Equivalent to fitting a quadratic function for $f(x)$
 - Works poorly for wiggly, ill-behaved functions in the local neighborhood of x .



$H_f(x)$

Hessian is costly to compute
(n^2 double derivative entries for an n -dimensional vector)
□ approximations

(“Step size”, $= 1/\sqrt{H_f(x)}$, inverse curvature)

(Multivariate:

$\nabla f(x)$ = gradient vector

$\nabla \nabla f(x)$ = matrix of 2nd derivatives
 $a/b = a b^{-1}$, matrix inverse)

Simulated Annealing (SA)

- Since the first development of Simulated Annealing (SA) by Kirkpatrick et al., SA has been applied in almost every area of optimization. The metaphor of SA came from the annealing characteristics in metal/metallurgy processing; however, SA has, in essence, strong similarity to the classic **Metropolis Algorithm** developed by Metropolis et al.

Metropolis Algorithm

- Simulate behaviour of a physical system according to principles of statistical mechanics.
- Globally biased toward “downhill” steps, but occasionally makes “uphill” steps to break out of local minima.
- Gibbs-Boltzmann function. The probability of finding a physical system in a state with energy E is proportional to $e^{-E} / (kT)$, where $T > 0$ is temperature and k is a Boltzmann constant.
- For any temperature $T > 0$, function is monotone decreasing function of energy E . System is more likely to be in a lower energy state than higher one.
 - T large: high and low energy states have same probability

Metropolis Algorithm

Metropolis algorithm.

- Given a fixed temperature T , maintain current state S .
- Randomly perturb current state S to new state $S' \in N(S)$.
- If $E(S') \leq E(S)$, update current state to S' .
Otherwise, update current state to S' with probability $e^{-\Delta E / (kT)}$,
where $\Delta E = E(S') - E(S) > 0$.

Theorem. Let $f_S(t)$ be fraction of first t steps in which simulation is in state S .
Then, assuming some technical conditions, with probability 1:

$$\lim_{t \rightarrow \infty} f_S(t) = \frac{1}{Z} e^{-E(S)/(kT)},$$

$$\text{where } Z = \sum_{S \in N(S)} e^{-E(S)/(kT)}.$$

Intuition. Simulation spends roughly the right amount of time in each state,
according to Gibbs-Boltzmann equation.

Simulated Annealing (SA)

- ❑ SA is based on a Metallurgical Metaphor (Physics inspired twist on Random Walk)
 - Start with a temperature set very high and slowly reduce it.
 - Run Hill-Climbing with the twist that we can occasionally replace the Current State with a Worse State based on the current temperature and how much worse the new state is.
- Basic Ideas:
 - Like Hill-Climbing, SA identifies the quality of the local improvements
 - Pick one randomly instead of picking the best move
 - Let δ be the change in the objective function
 - If δ is positive, then move to that state
 - Otherwise:
 - Move to this state with probability proportional to δ
 - Thus: Worse moves (very large negative δ) are executed less often
 - However, there is always a chance of escaping from local maxima
 - Over time, make it less likely to accept locally bad moves
 - (Can also make the size of the move random as well)

Physical Interpretation of Simulated Annealing

- A Physical Analogy:
 - Imagine letting a ball roll downhill on the function surface
 - This is like Hill-Climbing (for Minimization)
 - Now imagine shaking the surface, while the ball rolls, gradually reducing the amount of shaking
 - This is like Simulated Annealing
- Annealing = Harden metals by heating them to a high temperature and then gradually cooling them. It is the Physical Process of Cooling a Metal until Particles Achieve a Certain Frozen Crystal State.
 - Simulated Annealing (SA):
 - Free Variables are like Particles
 - Seek “Low Energy” (High Quality) Configuration
 - Slowly Reducing Temp. T with Particles Moving around Randomly

Simulated Annealing (SA)

- More formally...
 - Generate a random new neighbor from current state.
 - If it's better take it.
 - If it is worse then take it with some probability proportional to the temperature T and ΔE
- Probability of a move decreases with the amount ΔE by which the evaluation is worsened
- A second parameter (Temperature T) is also used to determine the probability: high T allows more worse moves, T close to zero results in few or no bad moves
- *Temperature Schedule* input determines the value of T as a function of the completed cycles

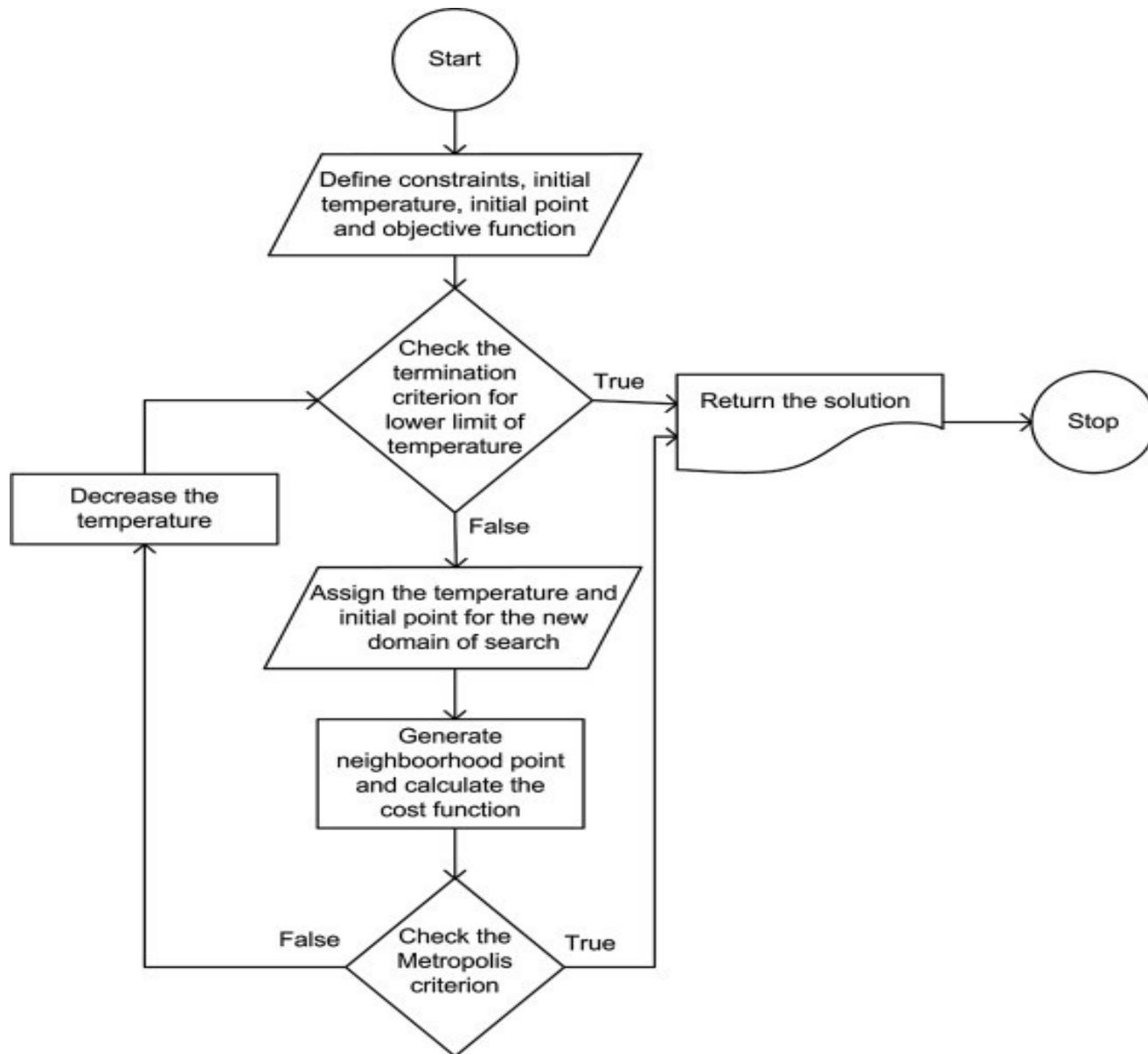
Simulated Annealing Search

- Idea: Escape local maxima by allowing some "bad" moves but **gradually decrease** their

```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
  inputs: problem, a problem
          schedule, a mapping from time to “temperature”
  local variables: current, a node
                    next, a node
                    T, a “temperature” controlling prob. of downward steps
  current  $\leftarrow$  MAKE-NODE(INITIAL-STATE[problem])
  for t  $\leftarrow$  1 to  $\infty$  do
    T  $\leftarrow$  schedule[t]
    if T = 0 then return current
    next  $\leftarrow$  a randomly selected successor of current
     $\Delta E \leftarrow$  VALUE[next] – VALUE[current]
    if  $\Delta E > 0$  then current  $\leftarrow$  next
    else current  $\leftarrow$  next only with probability  $e^{-\Delta E / \kappa T}$ 
```

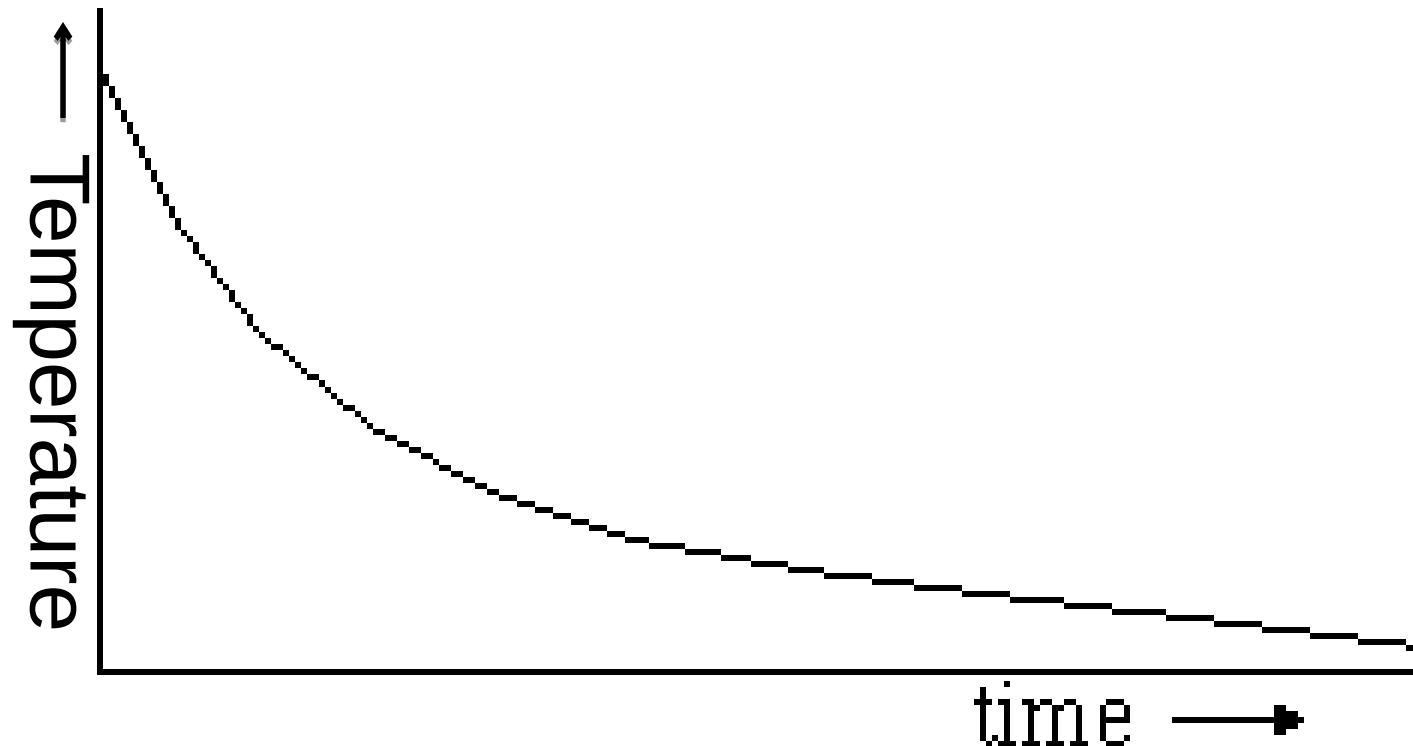
Improvement: Track the Best Result Found So far

Simulated Annealing Search



Typical Annealing Schedule

- Usually use a decaying exponential
- Axis values scaled to fit problem characteristics

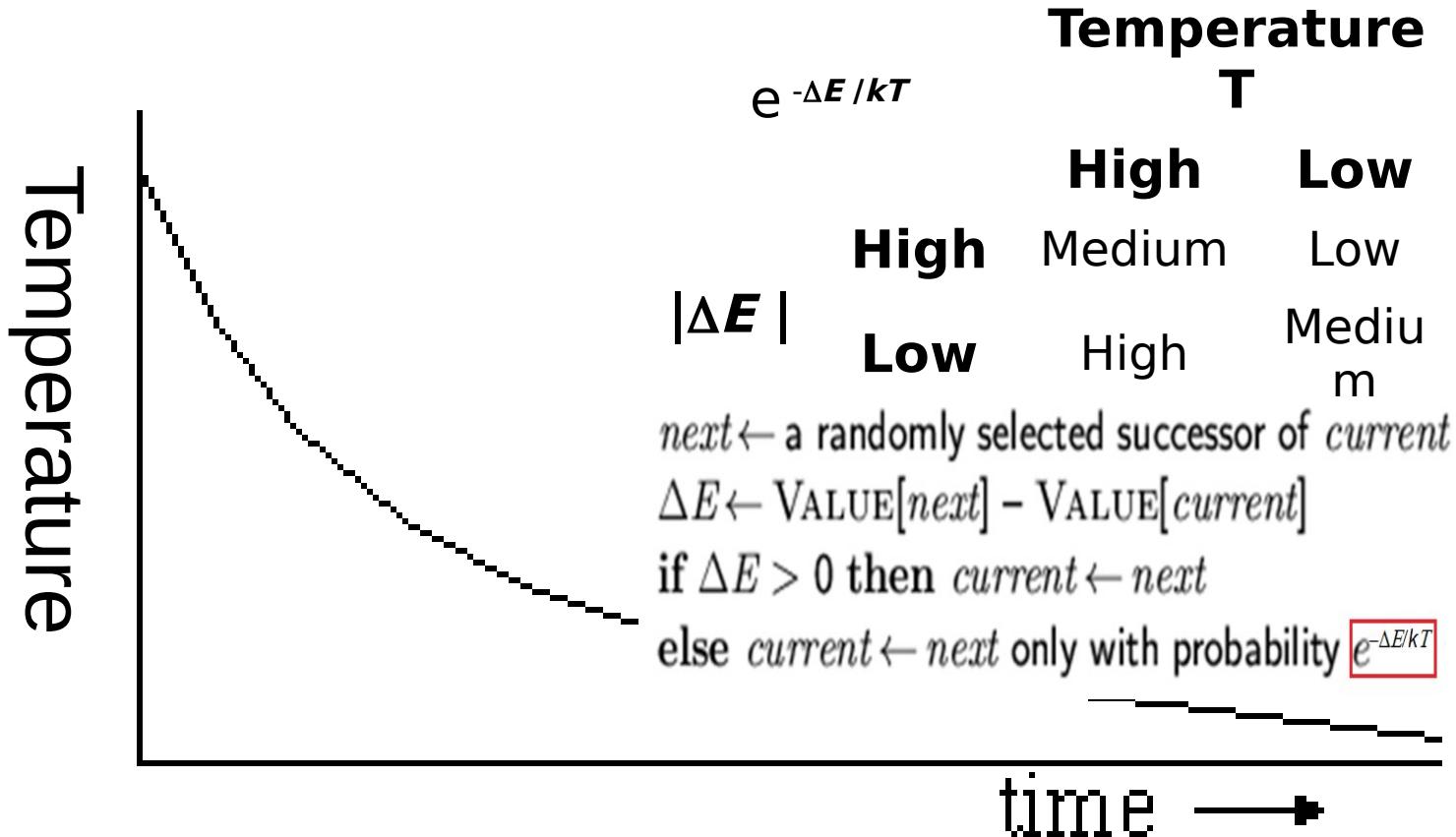


Temperature T

- High T: probability of “locally bad” move is higher
- Low T: probability of “locally bad” move is lower
- Typically, T is decreased as the algorithm runs longer i.e., there is a “temperature schedule”
- Update the current state S to the new state S' with the probability $p = e^{-\Delta E / kT}$
- Where, k is the Boltzmann's constant, and for simplicity, we can set k=1. Then T is the temperature for controlling the annealing process

Pr(Accept Worse Successor)

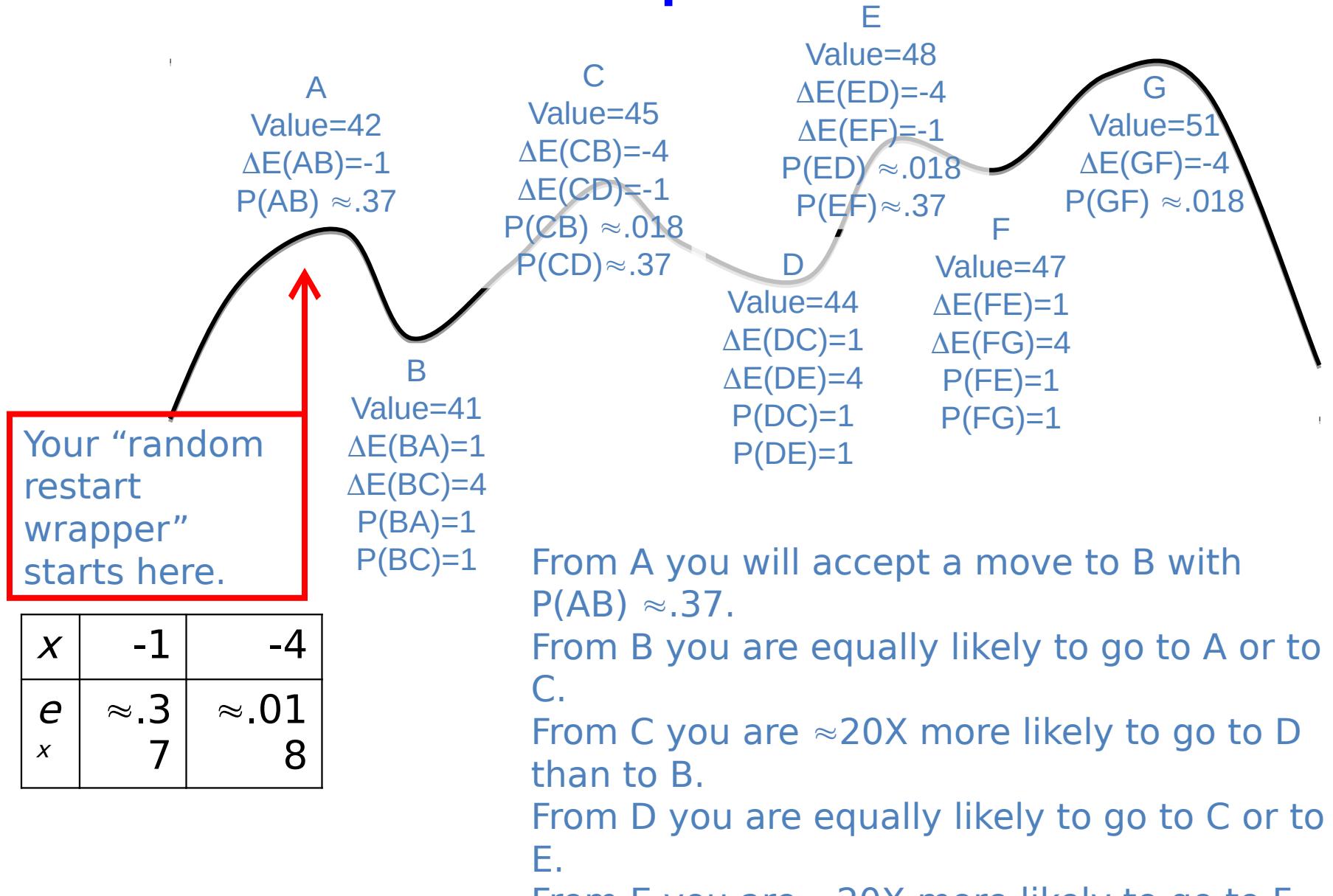
- Decreases as temperature T decreases (accept bad moves early on)
- Increases as $|\Delta E|$ decreases (accept not “much” worse)
- Sometimes, step size also decreases with T



Goal: “ratchet up” a jagged slope



Goal: “ratchet up” a jagged slope



Properties of Simulated Annealing

- One can prove:
 - If T decreases slowly enough, then simulated annealing search will find a global optimum with probability approaching 1
 - Unfortunately this can take a VERY VERY long time
 - Note: in any finite search space, random guessing also will find a global optimum with probability approaching 1
 - So, ultimately this is a very weak claim
- Often works very well in practice
 - But usually VERY VERY slow
- Widely used in VLSI layout, Airline

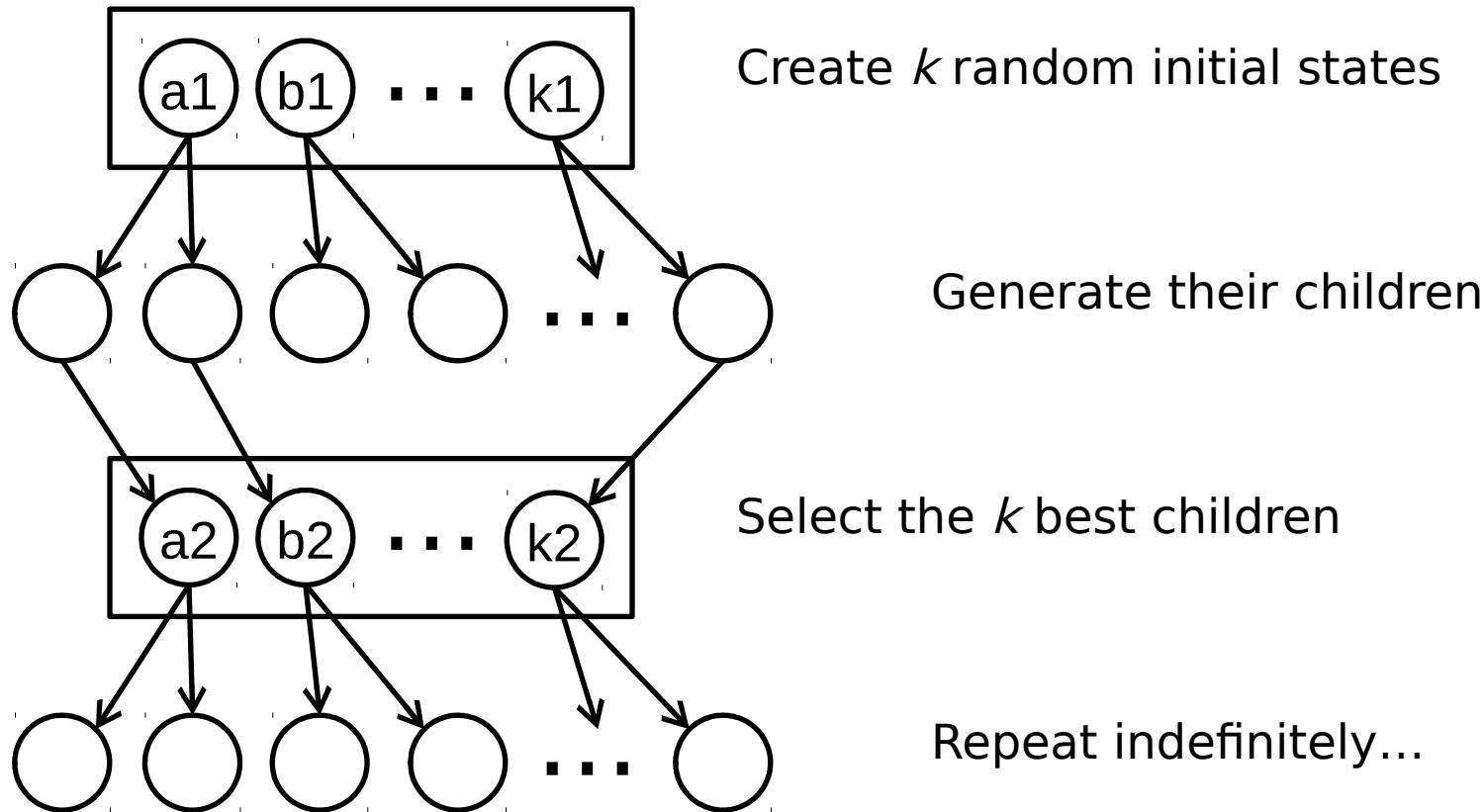
Simulated Annealing in Practice

- Method proposed in 1983 by IBM researchers for solving VLSI Layout Problems (Kirkpatrick et al., *Science*, 220:671-680, 1983).
 - Theoretically will always find the global optimum
- Other Applications: Traveling Salesperson Problem, Graph Partitioning, Graph Coloring, Scheduling, Facility Layout, Image Processing, ...
- Useful for some problems, but can be very slow

Local Beam Search

- Idea: Keeping only one node in memory is an extreme reaction to memory problems.
- Keep track of k states rather than just one in Hill-Climbing
- Start with k randomly generated states
- At each iteration, all the successors of all k states are generated
- If any one is a goal state, stop; else select the k best successors from the complete list and repeat.
- Concentrates search effort in areas believed to be fruitful
 - May lose diversity as search progresses, resulting in wasted effort

Local Beam Search (Contd...)



Is it better than simply running k searches?
Maybe...??

Local Beam Search (Contd...)

- Not the same as *k random-start searches run in parallel!*
- Searches that find good states recruit other searches to join it. Successors can become concentrated in a part of state space
- Problem: Quite often, all *k states end up on same Local Hill*
- Idea: Stochastic Beam Search
 - Choose *k successors randomly, biased towards good ones, with probability of choosing a given successor increasing with value*
- Observe the close analogy to natural selection!
- Natural Selection: Successors (*Offspring*) of⁶⁴ a state (*Organism*) populate the next generation

Genetic Algorithm (GA)

- GA is a variant of Stochastic Beam Search
- Twist on Local Search: Successor is generated by combining two parent states
- State is represented as a string over a finite alphabet (**Individual**)
 - A successor state is generated by combining two parent states
 - 8-queens Problem
 - State = Position of 8 queens each in a column
- Start with k randomly generated states (**Initial Population**)
- Evaluation Function (**Fitness Function**):
 - Higher values for better states.
 - Opposite to Heuristic Function, e.g., # non-attacking pairs in 8-queens
- Produce the next generation of states by “Simulated Evolution”
- **Selection:** Select individuals for next generation based on fitness
 - $P(\text{indiv. in next gen}) = \text{indiv. fitness} / \text{total population fitness}$
- **Crossover:** Fit parents to yield next generation

Genetic Algorithm (GA)

- Representation of Individuals
 - *Classic Approach*: individual is a string over a finite alphabet with each element in the string called a *Gene*
 - Usually binary instead of AGTC as in real DNA
- Selection Strategy
 - Random
 - Selection probability proportional to fitness
 - Selection is done with replacement to make a very fit individual reproduce several times
- Reproduction
 - Random pairing of *Selected Individuals*
 - Random selection of *Crossover* points
 - Each gene can be altered by a random *Mutation*

Genetic Algorithm (GA)

```
function GA (pop, fitness-fn)
```

Repeat

```
    new-pop = {}
```

```
    for i from 1 to size(pop):
```

```
        x = rand-sel(pop,fitness-fn)
```

```
        y = rand-sel(pop,fitness-fn)
```

```
        child = reproduce(x,y)
```

```
        if (small rand prob): child
```

```
            mutate(child)
```

```
            add child to new-pop
```

```
    pop = new-pop
```

Until an indiv is fit enough, or out of time

Return best indiv in pop, according to

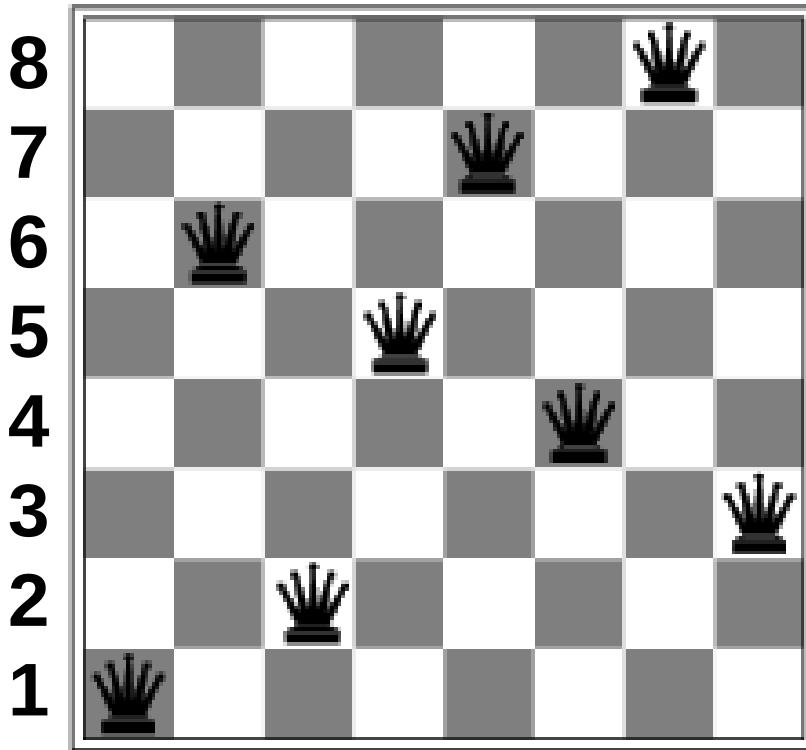
```
fitness-fn
```

```
n = len(x)
```

```
c = random num from 1 to n
```

```
return:
```

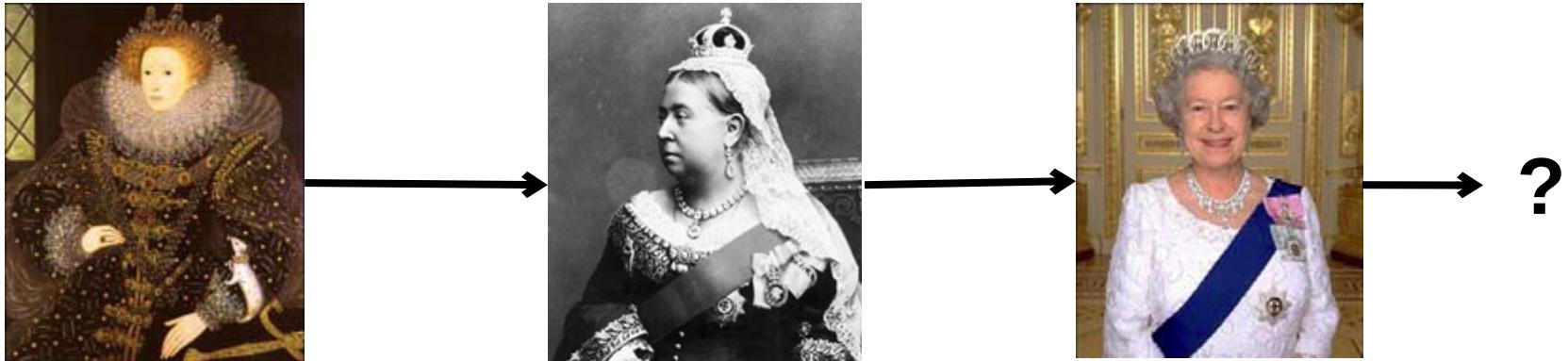
```
append(substr(x,1,c),substr(y,c+1,  
n))
```



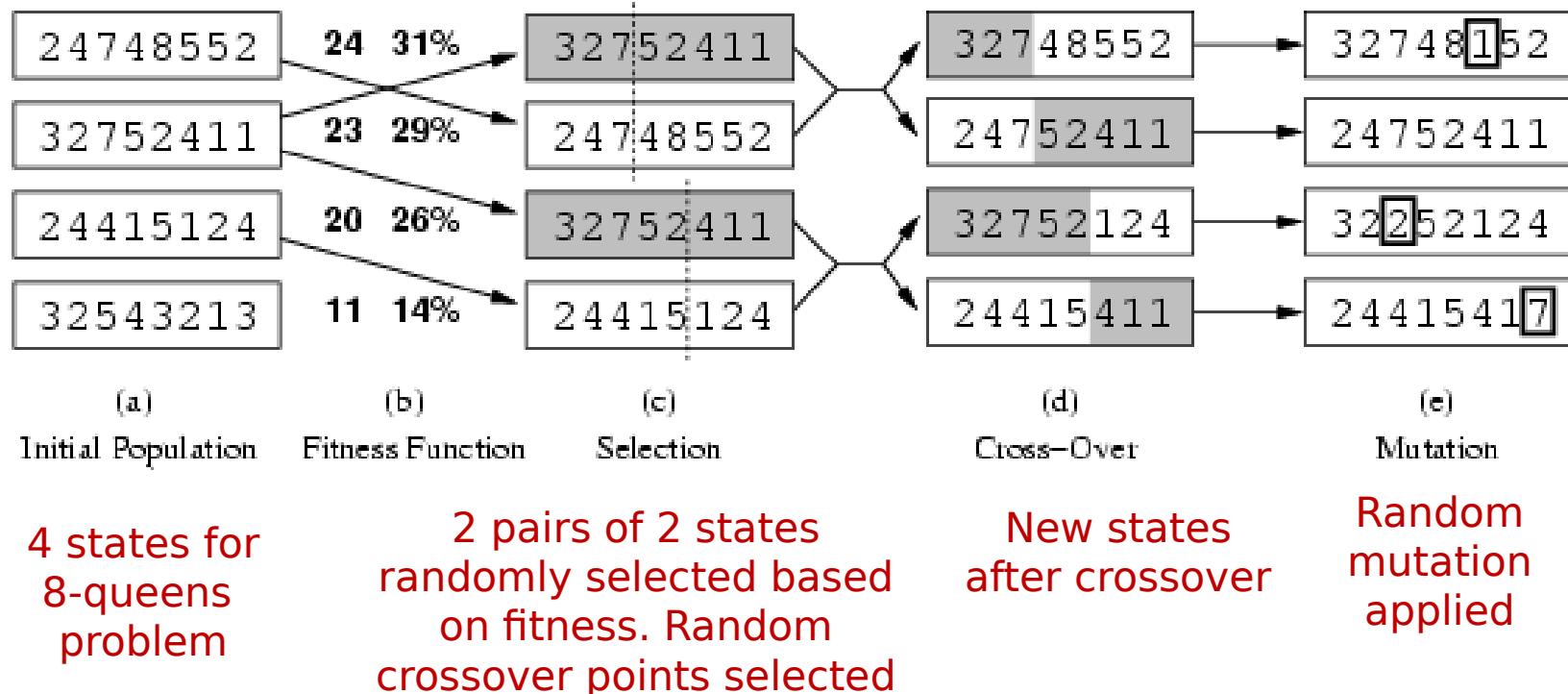
String Representation
16257483

Can we evolve 8-queens through Genetic Algorithm?

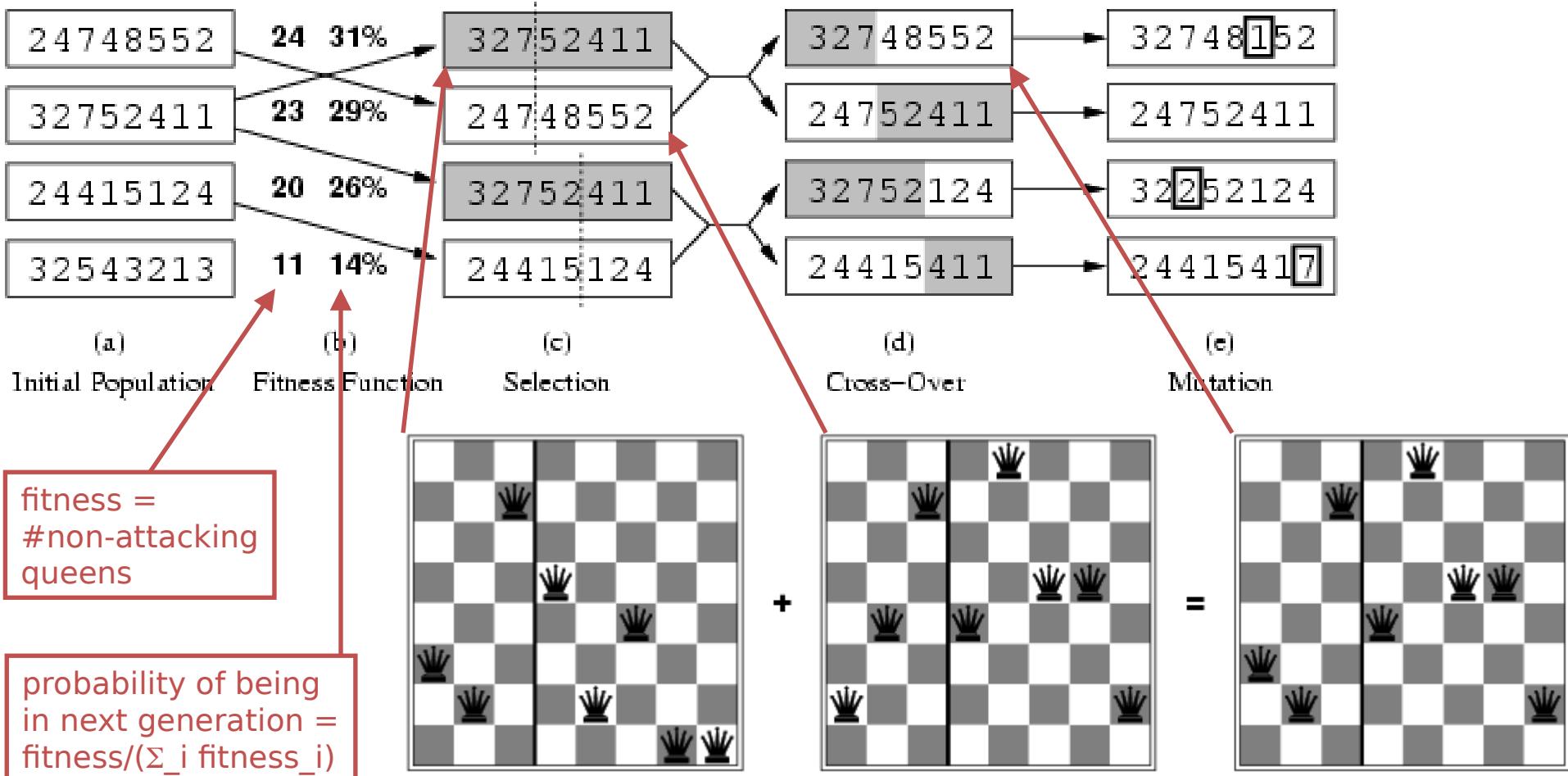
Evolving 8-queens



Genetic Algorithm



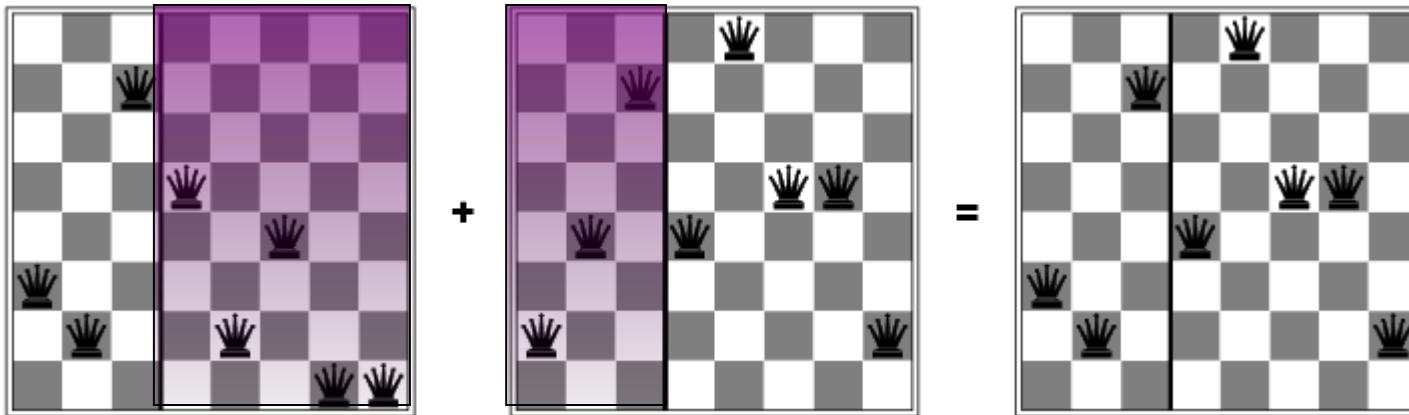
- Fitness Function: Number of non-attacking pairs of queens
(min = 0, max = $8 \times 7/2 = 28$)
- $24/(24+23+20+11) = 31\%$
- $23/(24+23+20+11) = 29\%$ etc.



- Fitness function: #non-attacking queen pairs
 - min = 0, max = $8 \times 7/2 = 28$
- $\sum_i \text{fitness}_i = 24+23+20+11 = 78$
- $P(\text{pick child}_1 \text{ for next gen.}) = \text{fitness}_1/(\sum_i \text{fitness}_i) = 24/78 = 31\%$
- $P(\text{pick child}_2 \text{ for next gen.}) = \text{fitness}_2/(\sum_i \text{fitness}_i) = 23/78 = 29\%$

How to convert a **fitness** value into a probability of being in the next generation.

Genetic Algorithm



Has the effect of “jumping” to a completely different new part of the search space (quite non-local)

Comments on Genetic Algorithm

- Genetic algorithm is a variant of “stochastic beam search”
- Genetic Algorithm is easy to apply and the Results can be good on some problems, but bad on other problems
- Positive points
 - Random exploration can find solutions that local search can’t
 - (via crossover primarily)
 - Appealing connection to human evolution
 - “neural” networks, and “genetic” algorithms are **metaphors!**
- Negative points
 - Large number of “tunable” parameters
 - Difficult to replicate performance from one problem to another

Summary

- Local search maintains a complete solution
 - Seeks consistent (also complete) solution
 - vs: path search maintains a consistent solution; seeks complete
 - Goal of both: consistent & complete solution
- Types:
 - Hill Climbing, Gradient Ascent
 - Simulated Annealing, Monte Carlo Methods
 - Population Methods: Beam Search; Genetic / Evolutionary Algorithms
 - Wrappers: Random Restart; Tabu Search
- Local search often works well on large problems
 - Abandons optimality
 - Always has some answer available (best found so far)

P, NP, NP-Complete and NP-Hard Problems

Polynomial Time

- Most (but not all) of the algorithms we have studied so far are easy, in that they can be solved in polynomial time, be it linear, quadratic, cubic, etc.
- Cubic may not sound very fast, and isn't when compared to linear, but compared to exponential time we have seen that it has a much better asymptotic behavior.
- There are many algorithms which are not polynomial. In general, if the space of possible solutions grows exponentially as n increases, then we should not hope for a polynomial time algorithm. These are the exception rather than the rule.

General Problems, Input Size and Time Complexity of Algorithms_

- Time Complexity of Algorithms:
Polynomial Time Algorithm ("Efficient Algorithm") vs.
Exponential Time Algorithm ("Inefficient Algorithm")

$f(n) \setminus n$	10	30	50
n	0.00001 sec	0.00003 sec	0.00005 sec
n^5	0.1 sec	24.3 sec	5.2 mins
2^n	0.001 sec	17.9 mins	35.7 yrs

“Hard” and “Easy’ Problems

- Sometimes the dividing line between “easy” and “hard” problems is a fine one. For example
 - Find the **shortest path** in a graph from X to Y. (**easy**)
 - Find the **longest path** in a graph from X to Y. (with no cycles) (**hard**)
- View another way – as “yes/no” problems
 - Is there a simple path from X to Y with weight $\leq M$? (**easy**)
 - Is there a simple path from X to Y with weight $\geq M$? (**hard**)
 - First problem can be solved in polynomial time.
 - All known algorithms for the second problem (could) take exponential time .

Hard Problems

- Problems with no known polynomial solution are called Hard problems.
- Another class of problems (NP) seem like they should be easy. They have the following property:
 1. A Solution can be **Verified** in Polynomial Time.
- Consider the following very common example called *Satisfiability*: (SAT)
 - **Input:** A Boolean expression, F , over n variables (x_1, \dots, x_n)
 - **Output:** 1 if the variables of F can be fixed to values which make F equal **true**; 0 otherwise.

Decision and Optimization Problems

- Decision Problem: It is a Computational Problem with an intended output of “Yes” or “No”, 1 or 0
- Optimization Problem: It is a Computational Problem where we try to maximize or minimize some value
- Introduce a parameter k and let us ask if the optimal value for the problem is at most or at least k . Turn optimization into decision

- Decision Problem: The solution to the problem is "yes" or "no". Most optimization problems can be phrased as decision problems (still have the same time complexity).

Example: :

Assume that we have a decision algorithm X for 0/1 Knapsack Problem with capacity M, i.e. Algorithm X returns “Yes” or “No” to the following question

*“Is there a solution with profit $\geq P$
subject to knapsack capacity $\leq M?$ ”*

We can repeatedly run algorithm X for various profits (P values) to find an optimal solution.

Example : Use binary search to get the optimal profit, maximum of $\log \sum p_i$ runs.

(where M is the capacity of the Knapsack optimization problem)

Min Bound Optimal Profit Max Bound

0

Search for the optimal solution
 $\sum p_i$



The Classes of P and NP Problems

- **P Class and Deterministic Turing Machine**
 - Given a decision problem X, if there is a Deterministic Turing Machine program that solves X in polynomial time, then X is belong to P Class.
 - *Informally, there is a Polynomial Time Algorithm to solve the problem.*

Complexity Class P

- Deterministic in nature
- Solved by conventional computers in polynomial time
 - $O(1)$ Constant
 - $O(\log n)$ Sub-linear
 - $O(n)$ Linear
 - $O(n \log n)$ Nearly Linear
 - $O(n^2)$ Quadratic
- Polynomial upper and lower bounds

NP Problems

- NP stands for non-deterministic polynomial time.
- The basic premise is we could guess at a solution and then test whether we guessed right or not easily. Guessing is of course non-deterministic!
- So, think of this as guessing in polynomial time. No guarantee that you will ever guess correctly though.

- NP Class and Non-deterministic Turing Machine

- Given a decision problem X .
If there is a Non-deterministic Turing machine program that solves X in polynomial time, then X belongs to NP.
- *Given a decision problem X .
For every instance I of X ,*
 - (a) *guess solution S for I , and*
 - (b) *check “is S a solution to I ? ”**If (a) and (b) can be done in polynomial time, then X belongs to NP.*

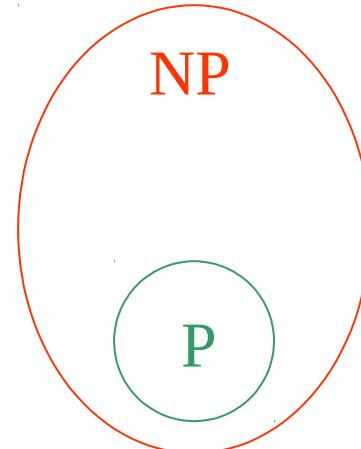
Complexity Class NP

- Non-deterministic part as well
- Choose(b): Choose a bit in a non-deterministic way and assign to b
- If someone tells us the solution to a problem, we can verify it in polynomial time
- Two Properties: Non-deterministic method to generate possible solutions, deterministic method to verify in polynomial time that the solution is correct.

Relation of P and NP

- P is a subset of NP
- “P = NP”?
- Language L is in NP, complement of L is in co-NP
- co-NP \neq NP
- P \neq co-NP

- Obvious : $P \subseteq NP$, i.e. A (decision) problem in P does not need “guess solution”. The correct solution can be computed in polynomial time.



- Some problems which are in NP , but may not in P :
 - 0/1 Knapsack Problem
 - PARTITION Problem : Given a finite set of positive integers Z .
Question : Is there a subset Z' of Z such that
Sum of all numbers in $Z' =$ Sum of all numbers in $Z-Z'$?
i.e. $\sum Z' = \sum (Z-Z')$

- One of the most important open problem in theoretical computer science:

Is P=NP ?

Most likely “No”.

Currently, there are many known (decision) problems in NP, and there is no solution to show anyone of them in P.

Polynomial Time Reducibility

- There are many mappings (called **reductions**) that have been discovered that map one problem to the other, so if we solve one we can solve the other.
- Moreover, these mappings are polynomial time mappings.

Polynomial-Time Reducibility

- Language L is polynomial-time reducible to language M if there is a function computable in polynomial time that takes an input x of L and transforms it to an input $f(x)$ of M, such that x is a member of L if and only if $f(x)$ is a member of M.
- Shorthand, $L \underset{\text{poly}}{\parallel} M$ means L is polynomial-time reducible to M

NP-Hard and NP-Complete

- Language M is NP-hard if every other language L in NP is polynomial-time reducible to M
- For every L that is a member of NP, $L \xrightarrow{\text{poly}} M$
- If Language M is NP-hard and also in the class of NP itself, then language M is NP-complete

NP-Hard and NP-Complete

- Restriction: A known NP-complete problem M is actually just a special case of L
- Local Replacement: reduce a known NP-Complete problem M to L by dividing instances of M and L into “basic units” then showing each unit of M can be converted to a unit of L
- Component Design: reduce a known NP-Complete problem M to L by building components for an instance of L that enforce important structural functions for instances of M .

NP-Complete Problems

- The set of NP problems that can be mapped to each other in polynomial time is called NP-Complete.
- It is difficult to prove that something can not be done.
- So, while we know how to solve many of these algorithms in exponential time, whose to say that one of you bright students won't come up with a clever polynomial time algorithm!
- NP-Complete is useful from that standpoint, if any of them turns out to have a polynomial time algorithm, they all do (hence $P=NP$).
- These problems are looked at from every angle and no such solution will ever be found (hence, $P \neq NP$).

NP-Complete Problems

- Stephen Cook introduced the notion of NP-Complete Problems.
 - This makes the problem “ $P = NP ?$ ” much more interesting to study.
- The following are several important things presented by Cook :

1. Polynomial Transformation (" \propto ")

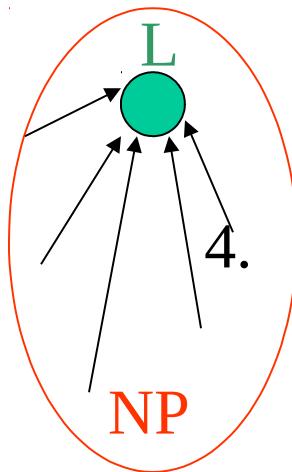
- **L1 \propto L2 :**

There is a polynomial time transformation that transforms arbitrary instance of L1 to some instance of L2.

- **If L1 \propto L2 then L2 is in P implies L1 is in P**
(or L1 is not in P implies L2 is not in P)
- **If L1 \propto L2 and L2 \propto L3 then L1 \propto L3**

2. Focus on the class of NP – **decision** problems only. Many intractable (unsolvable) problems, when phrased as decision problems, belong to this class.
3. L is **NP-Complete**
if (#1) $L \in NP$ & (#2) for all other $L' \in NP, L' \propto L$

- If an NP-complete problem can be solved in polynomial time then all problems in NP can be solved in polynomial time.
- If a problem in NP cannot be solved in polynomial time then all problems in NP-complete cannot be solved in polynomial time.
- Note that an NP-Complete problem is one of those hardest problems in NP.



4. L is **NP-Hard** if
(#2 of NP-Complete) for all other $L' \in NP, L' \propto L$

- Note that an NP-Hard problem is a problem which is as hard as an NP-Complete problem and it's not necessary a decision problem.
- So, if an NP-Complete problem is in P then P=NP
- if P \neq NP then all NP-Complete problems are in NP-P

Question: How can we obtain the first NP-Complete problem L?

Cook Theorem: SATISFIABILITY is NP-Complete.
(The first NP-Complete problem)

Instance : Given a set of variables, U , and a collection of clauses, C , over U .

Question : Is there a truth assignment for U that satisfies all clauses in C ?

Example :

$$U = \{x_1, x_2\}$$

$$C_1 = \{(x_1, \neg x_2), (\neg x_1, x_2)\}$$

$$= (x_1 \text{ OR } \neg x_2) \text{ AND } (\neg x_1 \text{ OR } x_2)$$

if $x_1 = x_2 = \text{True} \quad \square C_1 = \text{True}$

$$C_2 = (x_1, x_2) (x_1, \neg x_2) (\neg x_1) \quad \square \text{not satisfiable}$$

“ $\neg x_i$ ” = “not x_i ” “OR” = “logical or” “AND” = “logical and”

This problem is also called “**CNF-Satisfiability**” since the expression is in **CNF – Conjunctive Normal Form** (the product of sums).

- With the Cook Theorem, we have the following property :

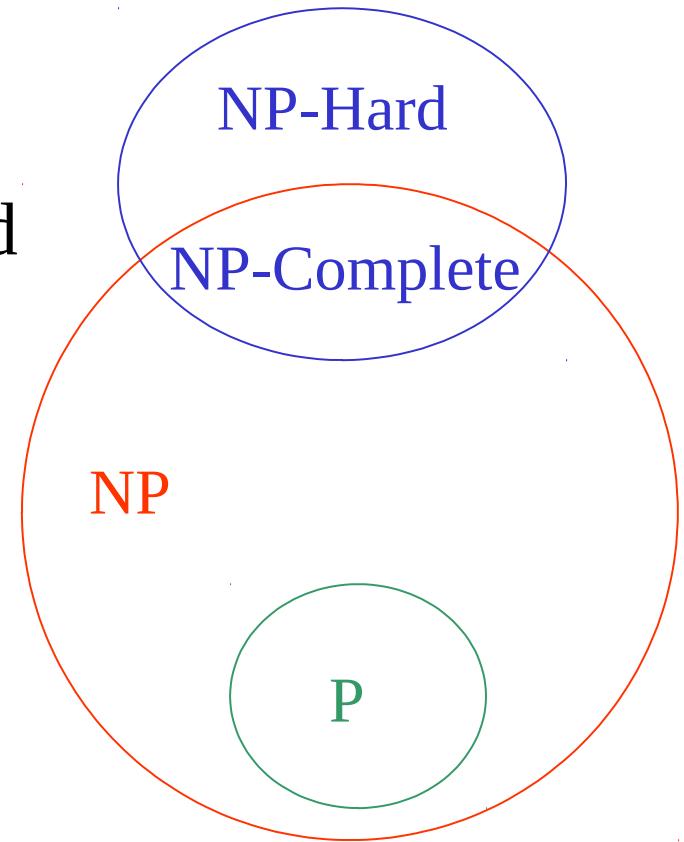
Lemma :

If L_1 and L_2 belong to NP,
 L_1 is NP-complete, and $L_1 \propto L_2$
then L_2 is NP-complete.

i.e. $L_1, L_2 \in NP$ and for all other $L' \in NP$, $L' \propto L_1$ and $L_1 \propto L_2 \sqcup L' \propto L_2$

- So now, to prove a (decision) problem L to be NP-complete, we need to
 - show L is in NP
 - select a known NP-complete problem L'
 - construct a polynomial time transformation f from L' to L
 - prove the correctness of f (i.e. L' has a solution if and only if L has a solution) and that f is a polynomial transformation

- P: (Decision) problems solvable by deterministic algorithms in polynomial time
- NP: (Decision) problems solved by non-deterministic algorithms in polynomial time
- A group of (decision) problems, including all the ones we discussed (**Satisfiability, 0/1 Knapsack, Longest Path, Partition**) have an additional important property:
If any of them can be solved in polynomial time, then they all **can!**
- These problems are referred to as **NP-Complete** problems.



NP-Complete Problems

- It is also useful to know these algorithms, as they occur frequently in real applications and tackling them in a brute force fashion may be disastrous.
 - SAT Problem
 - Traveling Salesperson Problem
 - Knapsack Problem
 - Longest Path Problem
 - Graph Clique Problem

NP-Complete

- It is also interesting to look at the relationship between some easy problems and hard ones:

Hard Problems (NP-Complete)	Easy Problems (in P)
SAT, 3SAT	2SAT
Traveling Salesman Problem	Minimum Spanning Tree
3D Matching	Bipartite Matching
Knapsack	Fractional Knapsack