Coping with the Limitations of Algorithm Power

Tackling Difficult Combinatorial Problems

- There are two principal approaches to tackling difficult combinatorial problems (NP-hard problems):
- Use a strategy that guarantees solving the problem exactly but doesn't guarantee to find a solution in polynomial time
- Use an approximation algorithm that can find an approximate (sub-optimal) solution in polynomial time

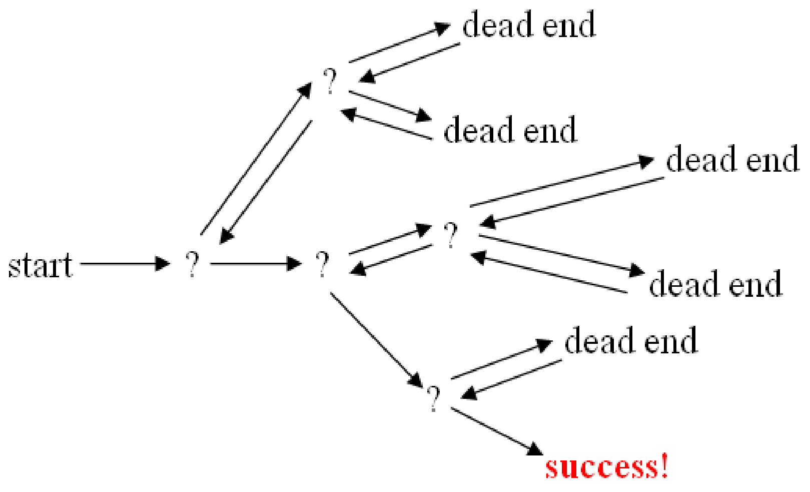
Exact Solution Strategies

- *exhaustive search* (brute force)
  - useful only for small instances
- *dynamic programming*
  - applicable to some problems (e.g., the knapsack problem)
- *backtracking*
  - eliminates some unnecessary cases from consideration
  - yields solutions in reasonable time for many instances but worst case is still exponential
- *branch-and-bound*
  - further refines the backtracking idea for optimization problems

Backtracking

- Suppose you have to make a series of *decisions,* among various *choices,* where
  - You don't have enough information to know what to choose
  - Each decision leads to a new set of choices
  - Some sequence of choices (possibly more than one) may be a solution to your problem
- Backtracking is a methodical way of trying out various sequences of decisions, until you find one that "works"

Backtracking : A Scenario



A tree is composed of nodes

Backtracking can be thought of as searching a tree for a particular "goal" leaf node

- Each non-leaf node in a tree is a parent of one or more other nodes (its children)
- Each node in the tree, other than the root, has exactly one parent

The backtracking algorithm

- Backtracking is really quite simple--we "explore" each node, as follows:
- To "explore" node N:

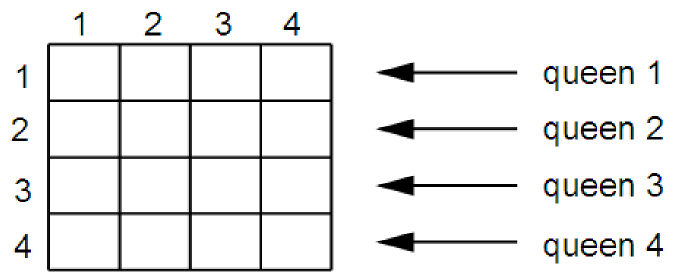
```
1. If N is a goal node, return "success"
2. If N is a leaf node, return "failure"
3. For each child C of N,
    3.1. Explore C
        3.1.1. If C was successful, return "success"
4. Return "failure"
```

- Construct the *state-space tree*
  - nodes: partial solutions
  - edges: choices in extending partial solutions

- Explore the state space tree using depth-first search

- "Prune" *nonpromising nodes*
  - dfs stops exploring subtrees rooted at nodes that cannot lead to a solution and backtracks to such a node's parent to continue the search

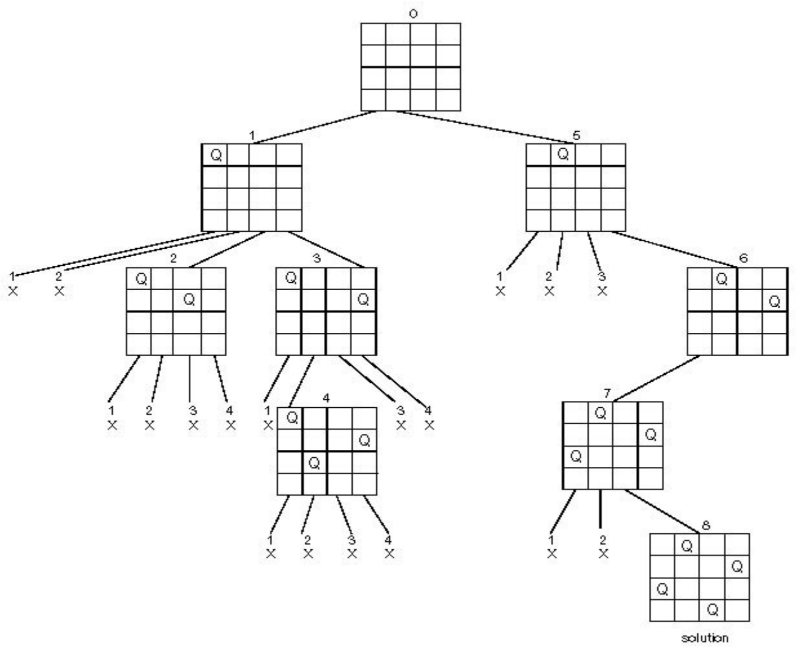
Example: *n*-Queens Problem
Place *n* queens on an *n*-by-*n* chess board so that no two of them are in the same row, column, or diagonal



State-Space Tree of the 4-Queens Problem

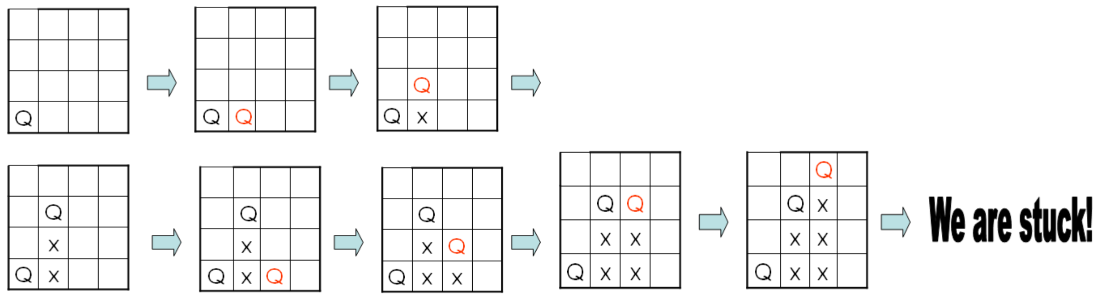
N-Queens Problem:

- The object is to place queens on a chess board in such as way as no queen can capture another one in a single move
  - Recall that a queen can move horz, vert, or diagonally an infinite distance
    - This implies that no two queens can be on the same row, col, or diagonal
  - We usually want to know how many different placements there are



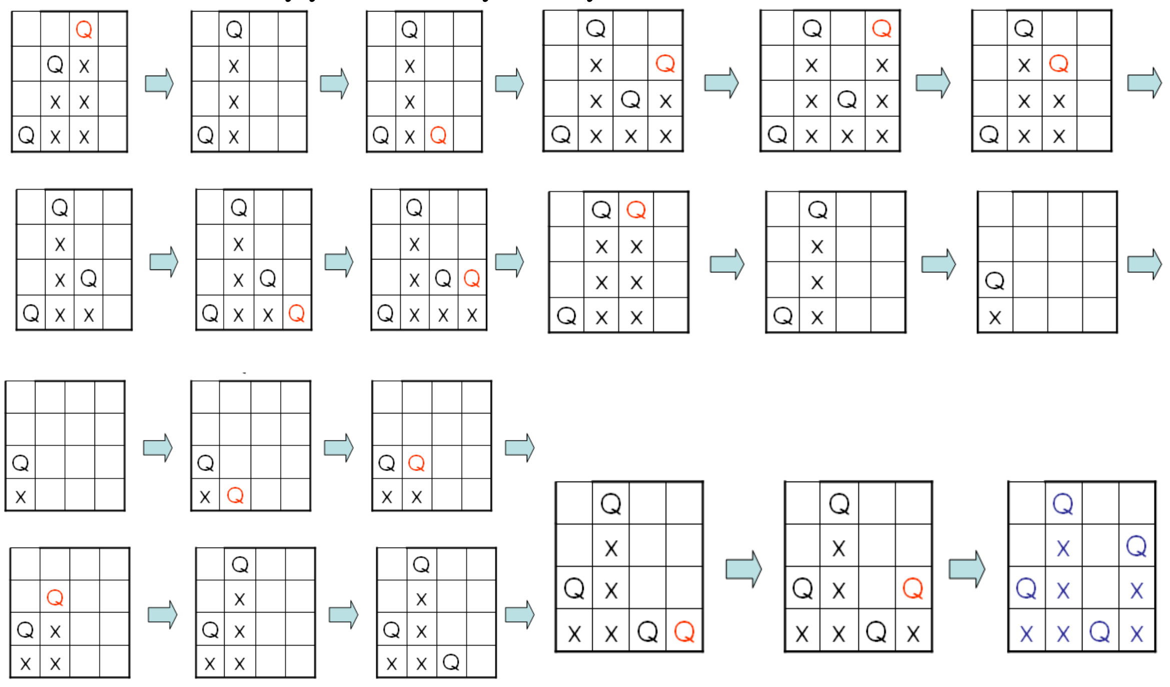
4-Queens

- Lets take a look at the simple problem of placing queens 4 queens on a 4x4 board
- The brute-force solution is to place the first queen, then the second, third, and forth
    - After all are placed we determine if they are placed legally
- There are 16 spots for the first queen, 15 for the second, etc.
    - Leading to 16*15*14*13 = 43,680 different combinations
- Obviously this isn't a good way to solve the problem
- First lets use the fact that no two queens can be in the same col to help us
    - That means we get to place a queen in each col
- So we can place the first queen into the first col, the second into the second, etc.
- This cuts down on the amount of work
    - Now there are 4 spots for the first queen, 4 spots for the second, etc.
        - 4*4*4*4 = 256 different combinations
- However, we can still do better because as we place each queen we can look at the previous queens we have placed to make sure our new queen is not in the same row or diagonal as a previously place queen
- Then we could use a Greedy-like strategy to select the next valid position for each col

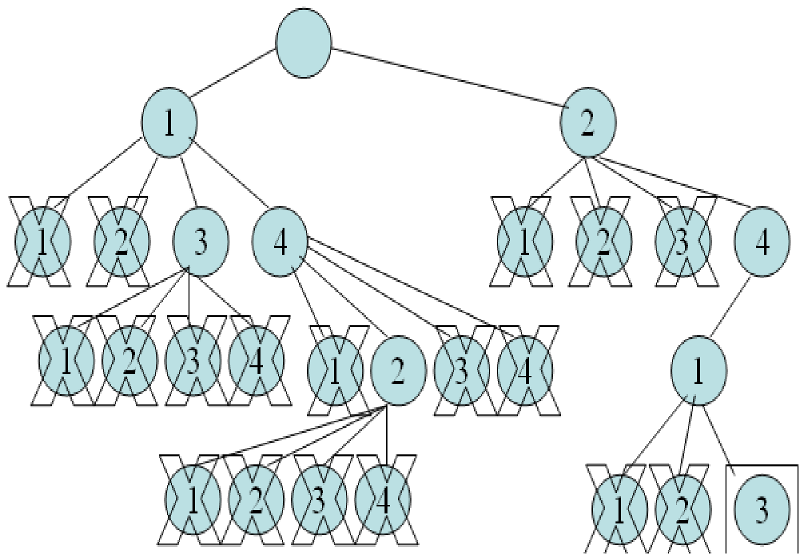


- So now what do we do?
- Well, this is very much like solving a maze

- As you walk though the maze you have to make a series of choices
- If one of your choices leads to a dead end, you need to back up to the last choice you made and take a different route
    - That is, you need to change one of your earlier selections
- Eventually you will find your way out of the maze



- **This type of problem is often viewed as a state-space tree**
    - A tree of all the states that the problem can be in
- **We start with an empty board state at the root and try to work our way down to a leaf node**
    - Leaf nodes are completed boards



Eight Queen Problem

- The solution is a vector of length 8 (a(1), a(2), a(3), ..., a(8)).
  a(i) corresponds to the column where we should place the i-th queen.
- The solution is to build a partial solution element by element until it is complete.
- We should backtrack in case we reach to a partial solution of length k, that we couldn't expand any more.

Eight Queen Problem: Algorithm

```
putQueen(row) {
  for every position col on the same row
    if position col is available
      place the next queen in position col
    if (row<8)
      putQueen(row+1);
    else  success;
    remove the queen from position col
}
```

```
putQueen(row) {
  for every position col on the same row
    if position col is available
      place the next queen in position col
    if (row<8)
      putQueen(row+1);
    else  success;
    remove the queen from position col
}
```

Eight Queen Problem: Implementation

- Define an 8 by 8 array of 1s and 0s to represent the chessboard
- The array is initialized to 1s, and when a queen is put in a position (c,r), board[r][c] is set to zero
- Note that the search space is very huge: 16,772, 216 possibilities.
- Is there a way to reduce search space? Yes Search Pruning.
- We know that for queens:
  each row will have exactly one queen
  each column will have exactly one queen
  each diagonal will have at most one queen
- This will help us to model the chessboard not as a 2-D array, but as a set of rows, columns and diagonals.

Hamiltonian Cycle

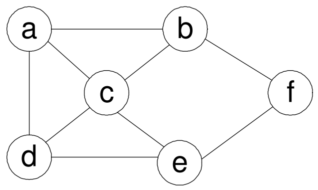
- Hamiltonian Cycle:
  - a cycle that contains every node exactly once
- Problem:
  - Given a graph, does it have a Hamiltonian cycle?

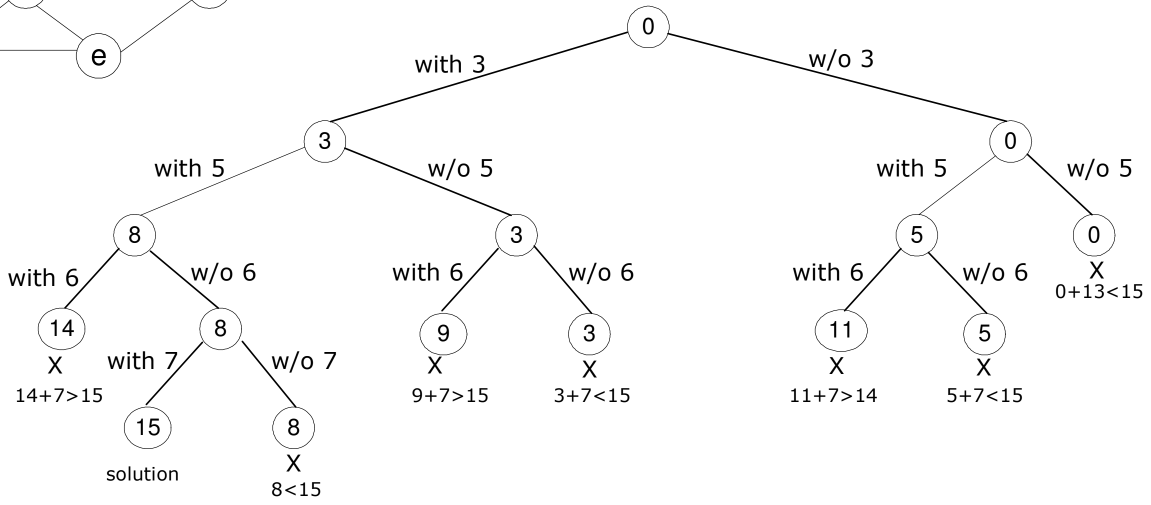
Background

- **NP-complete** problem:
  - Most difficult problems in NP (non- deterministic polynomial time)
- A decision problem *D* is NP-complete if it is complete for NP, meaning that:
  - it is in NP
  - it is NP-hard (every other problem in NP is reducible to it.)
- As they grow large, we are not able to solve them in a reasonable time (polynomial time)

Alternative Definition

- . **NP Problem such as Hamiltonian Cycle** :
  - Cannot be solved in Poly-time
  - Given a solution, easy to verify in poly-time

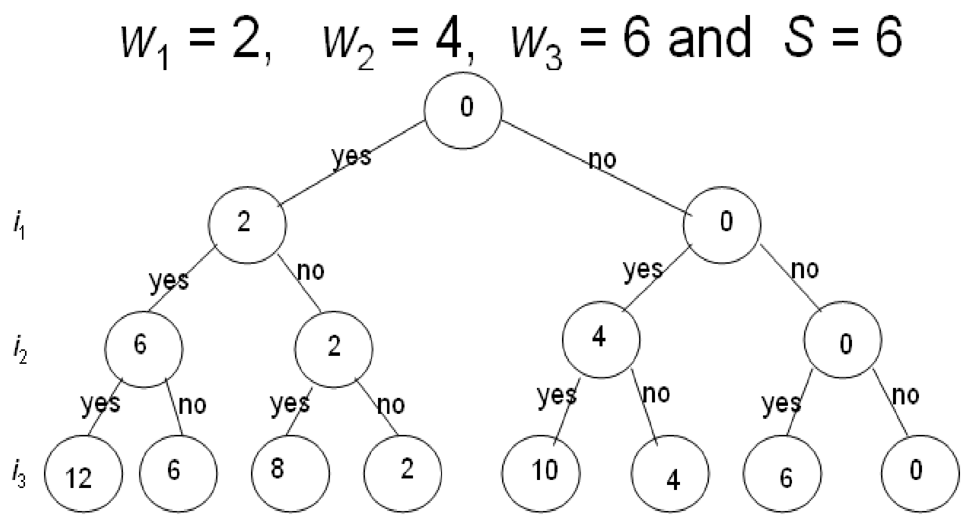




Sum of subsets

- Problem**: Given *n* positive integers *w*1, ... *wn* and a positive integer S. Find all subsets of *w*1, ... *wn* that sum to S.**
- Example**:**
  n=3, S=6, and w1=2, w2=4, w3=6
- Solutions**:**
  {2,4} and {6}
- **We will assume a binary state space tree.**
- **The nodes at depth 1 are for including (yes, no) item 1, the nodes at depth 2 are for item 2, etc.**
- **The left branch includes *wi*, and the right branch excludes *wi.***
- **The nodes contain the sum of the weights included so far**

Sum of subset Problem: State SpaceTree for 3 items



The sum of the included integers is stored at the node.

A Depth First Search solution

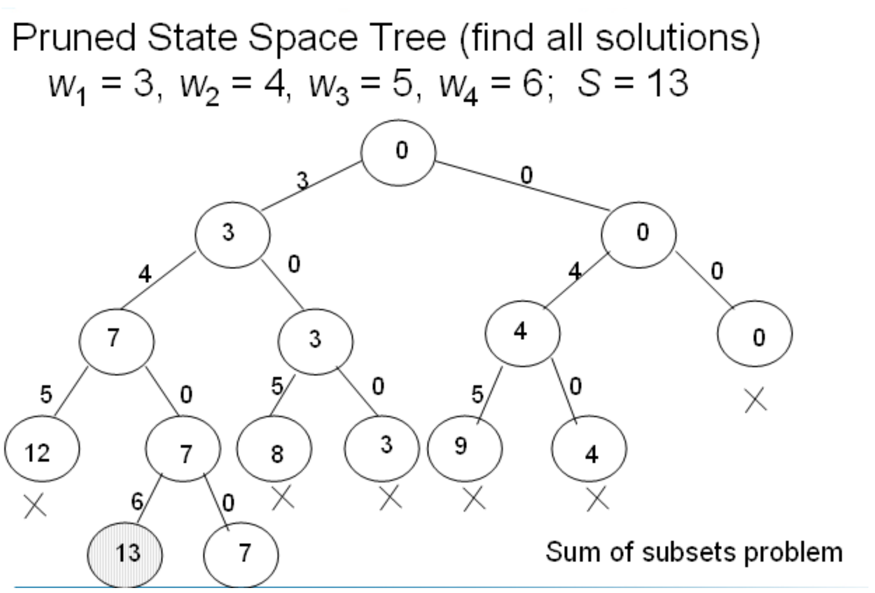
- Problems can be solved using depth first search of the (implicit) state space tree.
- Each node will save its depth and its (possibly partial) current solution
- DFS can check whether node v is a leaf.
  - If it is a leaf then check if the current solution satisfies the constraints
  - Code can be added to find the optimal solution

A DFS solution

- Such a DFS algorithm will be very slow.
- It does not check for every solution state (node) whether a solution has been reached, or whether a *partial* solution can lead to a *feasible* solution
- Is there a more efficient solution?

Backtracking solution to Sum of Subsets

- Definition: We call a node *nonpromising* if it cannot lead to a feasible (or optimal) solution, otherwise it is *promising*
- Main idea: Backtracking consists of doing a DFS of the state space tree, checking whether each node is promising and if the node is nonpromising backtracking to the node's parent
- The state space tree consisting of expanded nodes only is called the *pruned state space tree*
- The following slide shows the pruned state space tree for the sum of subsets example
- There are only 15 nodes in the pruned state space tree
- The full state space tree has 31 nodes


Pruned State Space Tree (find all solutions)
$w_1 = 3,\ w_2 = 4,\ w_3 = 5,\ w_4 = 6;\ S = 13$
Sum of subsets problem

Backtracking algorithm

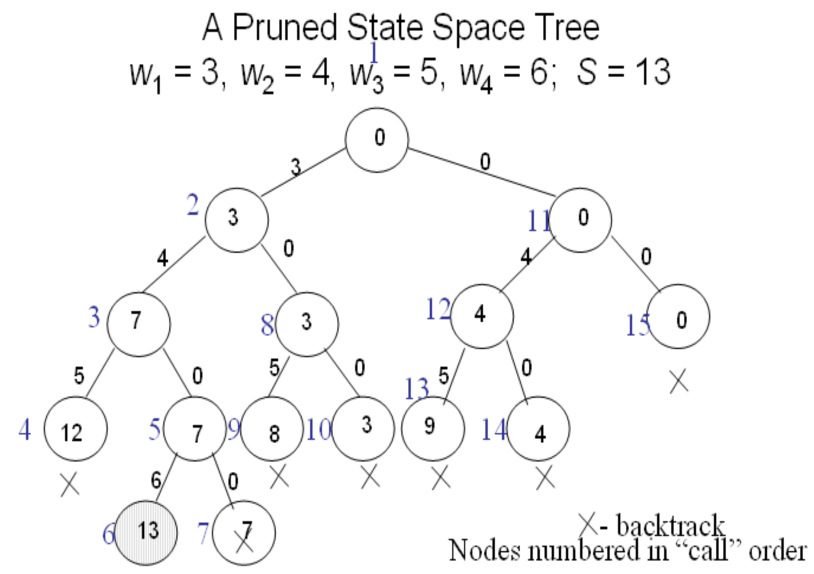
```
void checknode (node v) {
node u

if (promising ( v ))
        if (aSolutionAt( v ))
                write the solution
        else //expand the node
                for ( each child u of v )
                        checknode ( u )
```

Checknode

- Checknode uses the functions:
  - *promising*(v) which checks that the partial solution represented by v can lead to the required solution
  - *aSolutionAt*(v) which checks whether the partial solution represented by node v solves the problem.

Sum of subsets – when is a node "promising"?

- Consider a node at depth i
- *weightSoFar* = weight of node, i.e., sum of numbers included in partial solution node represents
- *totalPossibleLeft* = weight of the remaining items i+1 to n (for a node at depth i)
- A node at depth i is non-promising
  if (*weightSoFar* + *totalPossibleLeft* < S)
  or (*weightSoFar* + w[i+1] > S)
- To be able to use this "promising function" the w_i must be sorted in non-decreasing order


A Pruned State Space Tree
$w_1 = 3,\ w_2 = 4,\ w_3 = 5,\ w_4 = 6;\ S = 13$
X- backtrack
Nodes numbered in "call" order

```
sumOfSubsets ( i, weightSoFar, totalPossibleLeft )
1) if (promising ( i ))                          //may lead to solution
2)     then if ( weightSoFar == S )
3)         then print include[ 1 ] to include[ i ]     //found solution
4)     else      //expand the node when weightSoFar < S
5)         include [ i + 1 ] = "yes"                //try including
6)         sumOfSubsets ( i + 1,
                          weightSoFar + w[i + 1],
                      totalPossibleLeft - w[i + 1] )
7)         include [ i + 1 ] = "no"                  //try excluding
8)         sumOfSubsets ( i + 1,   weightSoFar ,
                          totalPossibleLeft - w[i + 1] )
boolean promising (i )
1) return ( weightSoFar + totalPossibleLeft ≥ S)  &&
          ( weightSoFar == S  ||  weightSoFar +  w[i + 1] ≤ S )
Prints all solutions!
```

Initial call sum Of Subsets (0, 0, $\sum_{i=1}^{n} w_i$)

Branch and Bound Searching Strategies

Feasible Solution vs. Optimal Solution

- DFS, BFS, hill climbing and best-first search can be used to solve some searching problem for searching a feasible solution.
- However, they cannot be used to solve the optimization problems for searching an (the) optimal solution.

The Branch-and-bound strategy

- This strategy can be used to solve optimization problems without an exhaustive search in the average case.
- 2 mechanisms:
  - A mechanism to generate branches when searching the solution space
  - A mechanism to generate a bound so that many braches can be terminated
- It is efficient in the average case because many branches can be terminated very early.
- Although it is usually very efficient, a very large tree may be generated in the worst case.
- Many NP-hard problem can be solved by B&B efficiently in the average case; however, the worst case time complexity is still exponential.

Bounding

- A bound on a node is a guarantee that any solution obtained from expanding the node will be:
  - Greater than some number (lower bound)
  - Or less than some number (upper bound)
- If we are looking for a minimal optimal, as we are in weighted graph coloring, then we need a lower bound
  - For example, if the best solution we have found so far has a cost of 12 and the lower bound on a node is 15 then there is no point in expanding the node
    - The node cannot lead to anything better than a 15
- We can compute a lower bound for weighted graph color in the following way:
  - The actual cost of getting to the node
  - Plus a bound on the future cost
    - Min weight color * number of nodes still to color
      - That is, the future cost cannot be any better than this
- Recall that we could either perform a depth-first or a breadth-first search
  - Without bounding, it didn’t matter which one we used because we had to expand the entire tree to find the optimal solution
  - Does it matter with bounding?
    - Hint: think about when you can prune via bounding
- We prune (via bounding) when:
  (currentBestSolutionCost <= nodeBound)
- This tells us that we get more pruning if:
  - The currentBestSolution is low
  - And the nodeBound is high
- So we want to find a low solution quickly and we want the highest possible lower bound
  - One has to factor in the extra computation cost of computing higher lower bounds vs. the expected pruning savings

The Assignment Problem

- In many business situations, management needs to assign - personnel to jobs, - jobs to machines, - machines to job locations, or - salespersons to territories.
- Consider the situation of assigning n jobs to n machines.
- When a job i (=1,2,....,n) is assigned to machine j (=1,2, ...n) that incurs a cost Cij.
- The objective is to assign the jobs to machines at the least possible total cost.

- This situation is a special case of the Transportation Model And it is known as the *assignment problem.*
- Here, jobs represent "sources" and machines represent "destinations."
- The supply available at each source is 1 unit And demand at each destination is 1 unit.

		Machine				
		1	2	…….	n	Source
Job	1	C11	C12	…….	C1n	1
	2	C21	C22	…….	C2n	1
	.	.	.		.	.
	.	.	.		.	.
	.	.	.		.	.
	n	Cn1	Cn2	…….	Cnn	1
Destination		1	1	……	1	

The assignment model can be expressed mathematically as follows:

Xij= 0, if the job j is not assigned to machine i
1, if the job j is assigned to machine i

$$\text{Min} \sum_{i=1}^{n} \sum_{j=1}^{n} C_{ij} X_{ij}$$

(Sum of assignments from a source should be exactly equal to 1):

$$\sum_{j=1}^{n} X_{ij} = 1 \qquad \text{For } i=1,2,\ldots,n$$

(Sum of assignments to a destination should be equal to the demanded quantity by that destination):

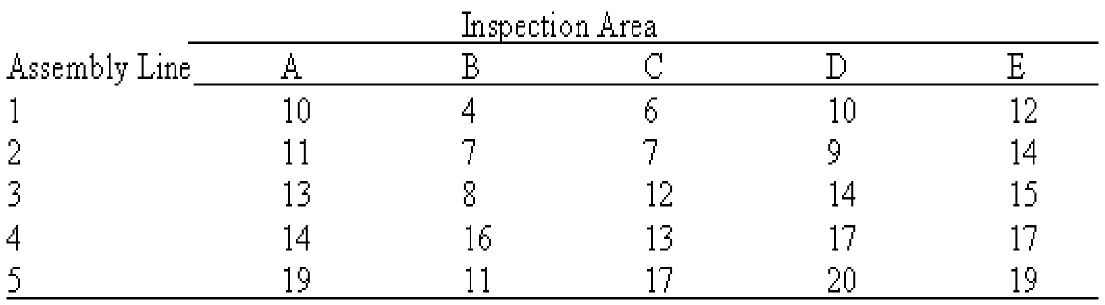
$$\sum_{i=1}^{n} X_{ij} = 1 \qquad \text{For } j=1,2,\ldots,n$$

(Quantities to be assigned can be either 0 or 1):

$$X_{ij} = 0 \text{ or } 1 \qquad \text{For all } i \text{ and } j.$$

The Assignment Problem Example

- Ballston Electronics manufactures small electrical devices.
- Products are manufactured on five different assembly lines (1,2,3,4,5).
- When manufacturing is finished, products are transported from the assembly lines to one of the five different inspection areas (A,B,C,D,E).
- Transporting products from five assembly lines to five inspection areas requires different times (in minutes)



	Inspection Area				
Assembly Line	A	B	C	D	E
1	10	4	6	10	12
2	11	7	7	9	14
3	13	8	12	14	15
4	14	16	13	17	17
5	19	11	17	20	19

Under current arrangement, assignment of inspection areas to the assembly lines are 1 to A, 2 to B, 3 to C, 4 to D, and 5 to E. This arrangement requires 10+7+12+17+19 = 65 man minutes.

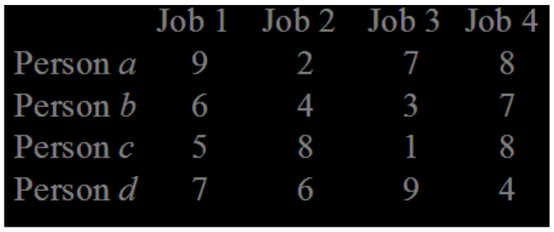
- Management would like to determine whether some other assignment of production lines to inspection areas may result in less cost.
- This is a typical assignment problem. n = 5 And each assembly line is assigned to each inspection area.
- It would be easy to solve such a problem when n is 5, but when n is large all possible alternative solutions are n!, this becomes a hard problem.
- Assignment problem can be either formulated as a linear programming model, or it can be formulated as a transportation model.
- However, An algorithm known as *Hungarian Method* has proven to be a quick and efficient way to solve such problems.

Assignment Problem Revisited:

Select one element in each row of the cost matrix *C* so that:

- **no two selected elements are in the same column**
- **the sum is minimized**

Example



	Job 1	Job 2	Job 3	Job 4
Person *a*	9	2	7	8
Person *b*	6	4	3	7
Person *c*	5	8	1	8
Person *d*	7	6	9	4

Lower bound: Any solution to this problem will have total cost at least: 2 + 3 + 1 + 4 (or 5 + 2 + 1 + 4)

Example: First two levels of the state-space tree

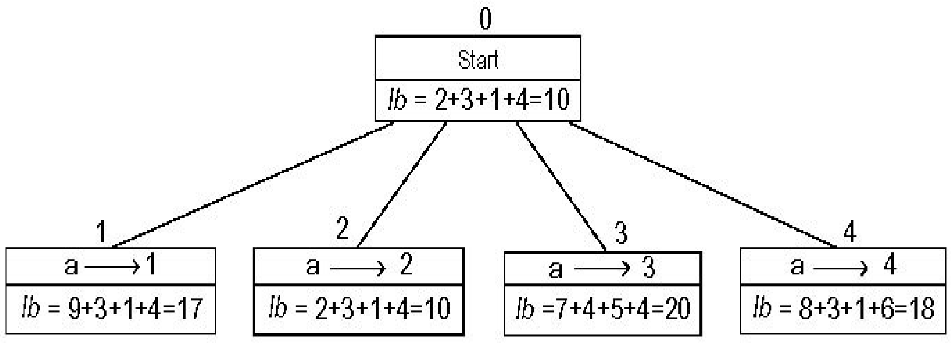


Figure 11.5 Levels 0 and 1 of the state-space tree for the instance of the assignment problem being solved with the best-first branch-and-bound algorithm. The number above a node shows the order in which the node was generated. A node's fields indicate the job number assigned to person a and the lower bound value, lb, for this node.

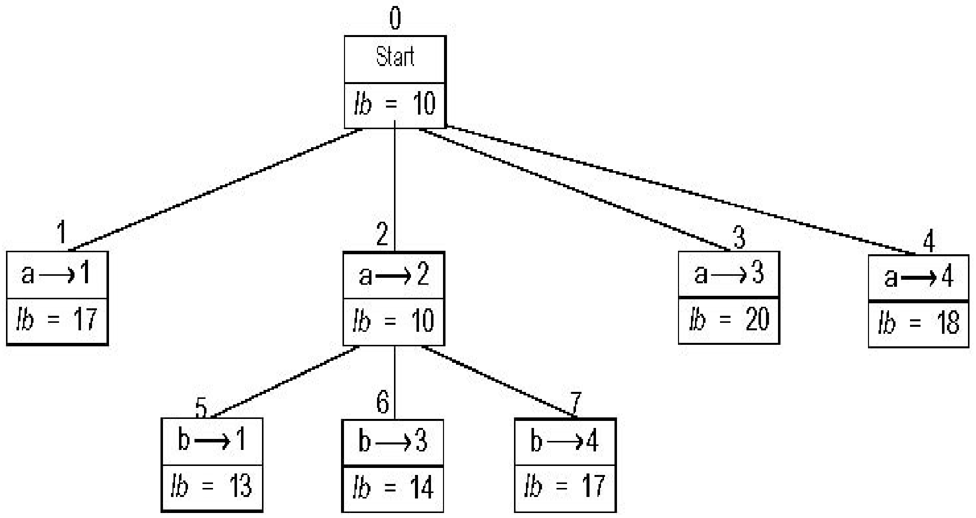


Figure 11.6 Levels 0, 1, and 2 of the state-space tree for the instance of the assignment problem being solved with the best-first branch-and-bound algorithm

Example: Complete state-space tree

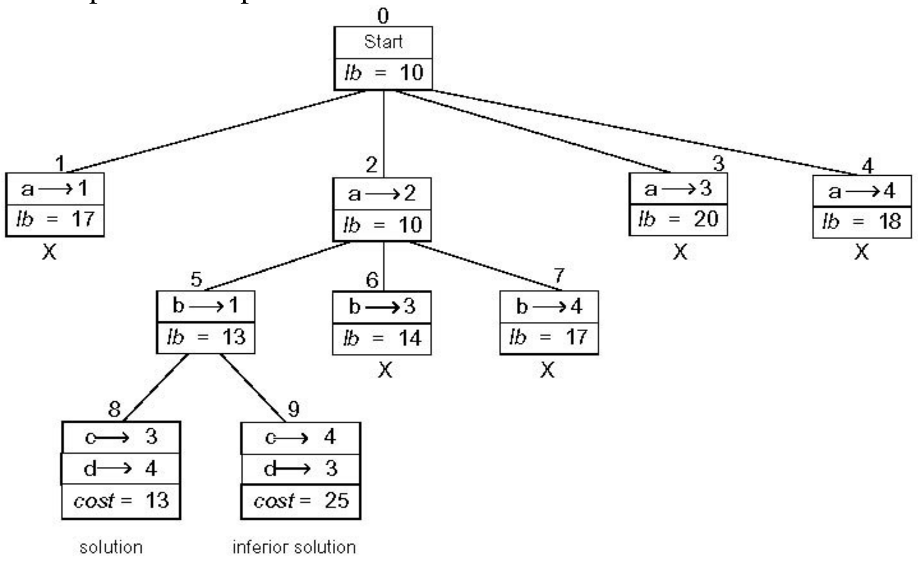
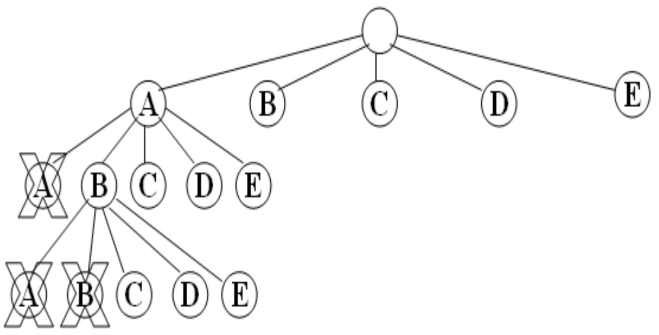


Figure 11.7 Complete state-space tree for the instance of the assignment problem solved with the best-first branch-and-bound algorithm

Traveling Salesperson Problem

- This is a classic CS problem
- Given a graph (cities), and weights on the edges (distances) find a minimum weight tour of the cities
  - Start in a particular city
  - Visit all other cities (*exactly* once each)
  - Return to the starting city
- Cannot be done by brute-force as this is worst-case exponential or worse running time
  - So we will look to backtracking with pruning to make it run in a reasonable amount of time in most cases
- We will build our state space by:
  - Having our children be all the potential cities we can go to next
  - Having the depth of the tree be equal to the number of cities in the graph
    - we need to visit each city exactly once
- So given a fully connected set of 5 nodes we have the following state space
  - only partially completed

Traveling Salesperson Problem



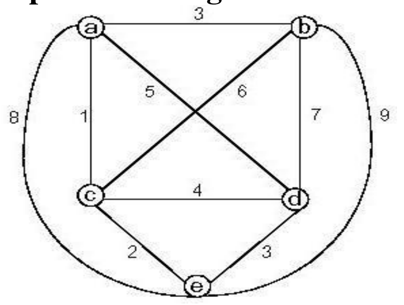
- Now we need to add bounding to this problem
  - It is a minimization problem so we need to find a lower bound
- We can use:
  - The current cost of getting to the node plus
  - An underestimate of the future cost of going through the rest of the cities
    - The obvious choice is to find the minimum weight edge in the graph and multiply that edge weight by the number of remaining nodes to travel through
- As an example assume we have the given adjacency matrix
- If we started at node A and have just traveled to node B then we need to compute the bound for node B
  - Cost 14 to get from A to B
  - Minimum weight in matrix is 2 times 4 more legs to go to get back to node A = 8
  - For a grand total of 14 + 8 = 22

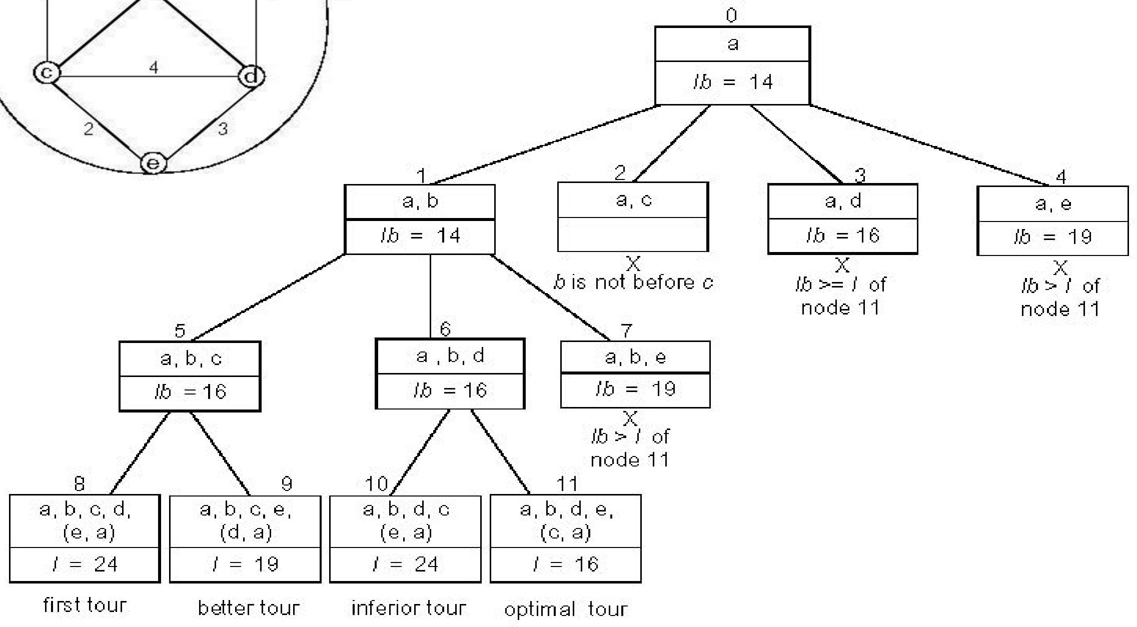
0	14	4	10	20
14	0	7	8	7
4	5	0	7	16
11	7	9	0	2
18	7	17	4	0

- Recall that if we can make the lower bound higher then we will get more pruning
- Note that in order to complete the tour we need to leave node B, C, D, and E
  - The min edge we can take leaving B is min(14, 7, 8, 7) = 7
  - Similarly, C=4, D=2, E=4
- This implies that at best the future underestimate can be 7+4+2+4=17
- 17 + current cost of 14 = 31
  - This is much higher than 8 + 14 = 22

0	14	4	10	20
14	0	7	8	7
4	5	0	7	16
11	7	9	0	2
18	7	17	4	0

Example: Traveling Salesman Problem





The 0/1 knapsack problem

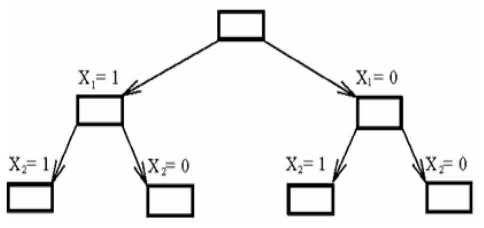
- Positive integer $P_1, P_2, \ldots, P_n$ (profit)
  $W_1, W_2, \ldots, W_n$ (weight)
  M (capacity)

maximize $\sum_{i=1}^{n} P_i X_i$

subject to $\sum_{i=1}^{n} W_i X_i \le M \quad X_i = 0 \text{ or } 1, i = 1, \ldots, n.$

The problem is modified:

minimize $-\sum_{i=1}^{n} P_i X_i$



The Branching Mechanism in the Branch-and-Bound Strategy to Solve 0/1 Knapsack Problem.

How to find the upper bound?

- Ans: by quickly finding a feasible solution in a greedy manner: starting from the smallest available i, scanning towards the largest i's until M is exceeded. The upper bound can be calculated.

The 0/1 knapsack problem

- E.g. n = 6, M = 34

i	1	2	3	4	5	6
P_i	6	10	4	5	6	4
W_i	10	19	8	10	12	8

$$(P_i/W_i \geq P_{i+1}/W_{i+1})$$

- A feasible solution: X1 = 1, X2 = 1, X3 = 0, X4 = 0, X5 = 0, X6 = 0
  -(P1+P2) = -16 (upper bound)
  Any solution higher than -16 can not be an optimal solution.

How to find the lower bound?

- Ans: by relaxing our restriction from Xi = 0 or 1 to 0 ≤ Xi ≤ 1 (knapsack problem)

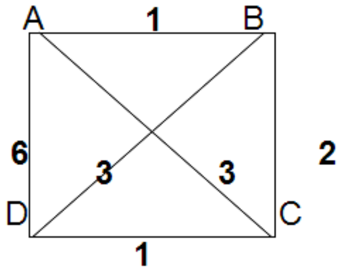
Let $-\sum_{i=1}^{n} P_i X_i$ be an optimal solution for 0/1 knapsack problem and $-\sum_{i=1}^{n} P_i X_i'$ be an optimal solution for fractional knapsack problem. Let $Y = -\sum_{i=1}^{n} P_i X_i$, $Y' = -\sum_{i=1}^{n} P_i X_i'$.

$\Rightarrow Y' \leq Y$

Approximation Approach
Apply a fast (i.e., a polynomial-time) approximation algorithm to get a solution that is not necessarily optimal but hopefully close to it
Accuracy measures:
accuracy ratio of an approximate solution *sa*
$r(sa) = f(sa) / f(s^*)$ for minimization problems
$r(sa) = f(s^*) / f(sa)$ for maximization problems
where f(*sa*) and f(*s**) are values of the objective function f for the approximate solution *sa* and actual optimal solution *s**
performance ratio of the algorithm A the lowest upper bound of *r*(*sa*) on all instances

Nearest-Neighbor Algorithm for TSP
Starting at some city, always go to the nearest unvisited city, and, after visiting all the cities, return to the starting one

Note: Nearest-neighbor tour may depend on the starting city



s_a : A – B – C – D – A of length 10

s^* : A – B – D – C – A of length 8

Accuracy: RA = ∞ (unbounded above) – make the length of AD arbitrarily large in the above example

Multifragment-Heuristic Algorithm
Stage 1: Sort the edges in nondecreasing order of weights. Initialize the set of tour edges to be constructed to empty set
Stage 2: Add next edge on the sorted list to the tour, skipping those whose addition would've created a vertex of degree 3 or a cycle of length less than *n*. Repeat this step until a tour of length *n* is obtained

Note: RA = ∞, but this algorithm tends to produce better tours than the nearest-neighbor algorithm

Twice-Around-the-Tree Algorithm
Stage 1: Construct a minimum spanning tree of the graph(e.g., by Prim's or Kruskal's algorithm)
Stage 2: Starting at an arbitrary vertex, create a path that goes twice around the tree and returns to the same vertex
Stage 3: Create a tour from the circuit constructed in Stage 2 by making shortcuts to avoid visiting intermediate vertices more than once

Note: RA = ∞ for general instances, but this algorithm tends to produce better tours than the nearest-neighbor algorithm

Christofides Algorithm
Stage 1: Construct a minimum spanning tree of the graph
Stage 2: Add edges of a minimum-weight matching of all the odd vertices in the minimum spanning tree
Stage 3: Find an Eulerian circuit of the multigraph obtained in Stage 2
Stage 3: Create a tour from the path constructed in Stage 2 by making shortcuts to avoid visiting intermediate vertices more than once

RA = ∞ for general instances, but it tends to produce better tours than the twice-around-the-minimum-tree alg.

Euclidean Instances
Theorem If *P ≠ NP,* there exists no approximation algorithm for TSP with a finite performance ratio.

Definition An instance of TSP is called *Euclidean*, if its distances satisfy two conditions:
1. *symmetry* $d[i, j] = d[j, i]$ for any pair of cities i and j
2. *triangle inequality* $d[i, j] \leq d[i, k] + d[k, j]$ for any cities i, j, k

For Euclidean instances:

approx. tour length / optimal tour length $\leq 0.5(\lceil \log2\ n \rceil + 1)$ for nearest neighbor and multifragment heuristic;
approx. tour length / optimal tour length $\leq$ 2 for twice-around-the-tree;
approx. tour length / optimal tour length $\leq$ 1.5 for Christofides

Local Search Heuristics for TSP
Start with some initial tour (e.g., nearest neighbor). On each iteration, explore the current tour's neighborhood by exchanging a few edges in it. If the new tour is shorter, make it the current tour; otherwise consider another edge change. If no change yields a shorter tour, the current tour is returned as the output.

Greedy Algorithm for Knapsack Problem
Step 1: Order the items in decreasing order of relative values:
$v_1/w_1 \geq \ldots \geq v_n/w_n$
Step 2: Select the items in this order skipping those that don't fit into the knapsack

Example: The knapsack's capacity is 16

item	weight	value	v/w
1	2	\$40	20
2	5	\$30	6
3	10	\$50	5
4	5	\$10	2

Accuracy
- R_A is unbounded (e.g., $n = 2$, $C = m$, $w_1=1$, $v_1=2$, $w_2=m$, $v_2=m$)
- yields exact solutions for the continuous version

Approximation Scheme for Knapsack Problem
Step 1: Order the items in decreasing order of relative values:
$v_1/w_1 \geq \ldots \geq v_n/w_n$
Step 2: For a given integer parameter k, $0 \leq k \leq n$, generate all subsets of k items or less and for each of those that fit the knapsack, add the remaining items in decreasing order of their value to weight ratios
Step 3: Find the most valuable subset among the subsets generated in Step 2 and return it as the algorithm's output

Bin Packing Problem: First-Fit Algorithm
First-Fit (*FF*) Algorithm: Consider the items in the order given and place each item in the first available bin with enough room for it; if there are no such bins, start a new one
Example: $n = 4$, $s_1 = 0.4$, $s_2 = 0.2$, $s_3 = 0.6$, $s_4 = 0.7$

Accuracy

- Number of extra bins never exceeds optimal by more than 70% (i.e., $R_A \leq 1.7$)
- Empirical average-case behavior is much better. (In one experiment with 128,000 bins, the relative error was found to be no more than 2%.)

Bin Packing: First-Fit Decreasing Algorithm

First-Fit Decreasing (FFD) Algorithm: Sort the items in decreasing order (i.e., from the largest to the smallest). Then proceed as above by placing an item in the first bin in which it fits and starting a new bin if there are no such bins

Example: $n = 4$, $s_1 = 0.4$, $s_2 = 0.2$, $s_3 = 0.6$, $s_4 = 0.7$

Accuracy

- Number of extra bins never exceeds optimal by more than 50% (i.e., $R_A \leq 1.5$)
- Empirical average-case behavior is much better, too

The End