CSIT113
Problem Solving

UNIT 10
COPING WITH COMPLEXITY USING BRANCH AND BOUND

UNIVERSITY OF WOLLONGONG AUSTRALIA

SIM GLOBAL EDUCATION

Overview

- Semi-Formal Introduction of Complexity
- Branch-and-Bound – another means to cope with time complexity, more specifically, it helps to design efficient combinatorial algorithms.

Complexity

- Is there a sensible way to discuss the complexity of a problem?
- Can we say how hard it is to solve?
- This is not as easy a question as it may appear.

Complexity

- Let us take sorting as an example.
- The first time we looked at this we saw a number of different algorithms:
  - selection sort;
  - insertion sort;
  - bubble sort.
- All of these took around n^2 steps to sort a sequence of length n.

Sorting

- So, does this mean that it always takes n^2 steps to sort a sequence?
- No. Consider quicksort.
- It takes $n\log n$ steps which is considerably less.
- Is it possible that there are even faster ways to sort a sequence?

Theory vs. Practice

- Clearly, there is a difference between:
  1. how complex the problem is;
  2. how complex a particular algorithm to solve it is.
- If we can solve a problem of size n in n log n steps what can we say about the actual complexity of the problem?
- The problem's actual complexity is *no worse than* n log n.
- This lets us set an *upper limit* on the complexity of the problem.
- Can we find some sensible way to establish a *lower limit*?

Lower Bounds

- In general, this is a much more difficult thing to establish.
- Often the best we can do is to find a relative rather than an absolute answer.
- We can often only say that one problem is *at most as hard* as another problem.
- How can we do this?

Problem Transformation

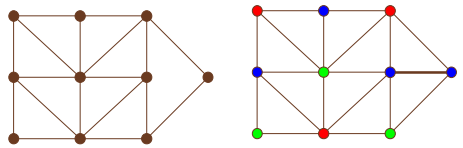
- If we can transform one problem into another problem we can say that the first problem is *no harder* than then the second problem.
- This assumes that the transformation process is relatively simple.
- Let us look at two problems:
  - graph colouring;
  - satisfiability.

Graph Colouring

- The problem here is to answer the question:
  - Can we colour the vertices of a given graph with three colours so that no two adjacent nodes are of the same colour?
- This is best seen with an example.

Graph Colouring

- Given the graph:-
- We can colour it as follows:-



- But what if we add an extra edge?

Satisfiability

- The problem here is to answer the question:
  - Can we find values for a set of logical variables which satisfies a given set of logical equations?
- Again, let us look at an example.

Satisfiability

- Given the expression:-
  - $(a \vee b \vee c) \wedge (\sim a \vee b) \wedge (\sim b \vee \sim c) \wedge (a \vee \sim c)$
- Where a, b and c are either true or false, $\vee$ represents "or", $\wedge$ represents "and" and $\sim$ represents "not".
- Can we assign values to a, b and c so that the expression has the value true? Yes, for example,
  - a = true, b = true, c = false

So What?

- What, if anything do these problems have in common?
- It turns out we can transform the graph colouring problem into the satisfiability problem.
- How?

Transforming the Problem

- Let us label each of the nodes of the graph with an integer value.
- Every node must have a colour
- Let:
  - R_i mean "node *i* is red";
  - G_i mean "node *i* is green";
  - B_i mean "node *i* is blue".
- Then:
  - $(R_i \vee G_i \vee B_i)$ = true mean node *i* is either red, green or blue.

Transforming the Problem

- But every node must have a single colour
- So:
  - $(R_i \wedge \sim G_i \wedge \sim B_i)$ = true – node *i* has the colour red (single);
  - $(\sim R_i \wedge G_i \wedge \sim B_i)$ = true – node *i* has the colour green (single);
  - $(\sim R_i \wedge \sim G_i \wedge B_i)$ = true – the node *i* has the colour blue (single).
- We combine these:
  - $(R_i \wedge \sim G_i \wedge \sim B_i) \vee (\sim R_i \wedge G_i \wedge \sim B_i) \vee (\sim R_i \wedge \sim G_i \wedge B_i)$ = true mean node *i*. has a single colour

Transforming the Problem

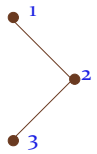
- Finally no two connected nodes may have the same colour.
- So, if node *i* is connected to node *j*:
  - $(\sim R_i \vee \sim R_j)$ = true – nodes *i* and *j* are not both red;
  - $(\sim G_i \vee \sim G_j)$ = true – nodes *i* and *j* are not both green;
  - $(\sim B_i \vee \sim B_j)$ = true – nodes *i* and *j* are not both blue.
- Thus:
  - $(\sim R_i \vee \sim R_j) \wedge (\sim G_i \vee \sim G_j) \wedge (\sim B_i \vee \sim B_j)$ = true mean node *i* and *j* are not both red, green or blue.

Transforming the Problem

- For a graph with n nodes and e edges we get $3*n + 3*e$ clauses in $3*n$ logical variables.
- Thus, we can transform any graph colouring problem into an equivalent satisfiability problem.
- So what?

A Simple Specific Example on the Transformation

- For a graph with 3 nodes and 2 edges we get $3*3 + 3*2 = 15$ clauses in $3*3 = 9$ logical variables:



- The logical variable are as follows:

$R_1, G_1, B_1, R_2, G_2, B_2, R_3, G_3, B_3$

A Sample Specific Example on the Transformation

- The logical equations for specifying each node has one colour are:

$(R_1 \wedge \sim G_i \wedge \sim B_1) \vee (\sim R_1 \wedge G_1 \wedge \sim B_1) \vee (\sim R_1 \wedge \sim G_1 \wedge B_i) = \text{true}$
$(R_2 \wedge \sim G_2 \wedge \sim B_2) \vee (\sim R_2 \wedge G_2 \wedge \sim B_2) \vee (\sim R_2 \wedge \sim G_2 \wedge B_2) = \text{true}$
$(R_3 \wedge \sim G_3 \wedge \sim B_3) \vee (\sim R_3 \wedge G_3 \wedge \sim B_3) \vee (\sim R_3 \wedge \sim G_3 \wedge B_3 = \text{true}$

- The logical equations for specifying the nodes connected by an edge do not have the same colour are:

$(\sim R_1 \vee \sim R_2) \wedge (\sim G_1 \vee \sim G_2) \wedge (\sim B_1 \vee \sim B_2) = \text{true}$
$(\sim R_2 \vee \sim R_3) \wedge (\sim G_2 \vee \sim G_3) \wedge (\sim B_2 \vee \sim B_3) = \text{true}$

- Note that each bracket is a clause, e.g., $(R_1 \wedge \sim G_i \wedge \sim B_1)$ is a clause.
- The colouring of the graph is to find a solution satisfying the above six logical equations: One solution is R_1, B_2 and R_3. Another solution is R_1, B_2 and G_3. Certainly, there are more solutions.

Comparative Complexity

- What this means is that, in the worst case, we can solve a graph colouring problem by transforming it and solving the equivalent satisfiability problem.
- This means that the graph colouring problem is *no harder* than the satisfiability problem.
- This does not mean, of itself, that we cannot find a better way of solving the colouring problem, only that it is never worse than this.

P and NP

- In fact, both the colouring and satisfiability problems are as hard as a problem can be and still be of some interest to us.
- Computer scientists define two sets of problems which are of particular interest.
  - P (Polynomial Time), the set of problems for which a polynomial complexity solution exists.
  - NP (Non-Deterministic Polynomial Time), the set of problems for which a polynomial complexity verification procedure exists.
- Clearly, P is a subset of NP.

Problems and P

- You should note that all problems fall into one of the following categories.
  - Known to be in P.
  - Known not to be in P.
  - Unknown
    - Possibly in P
    - Possibly not in P

Problems and NP

- In a similar way we can categorise problems with respect to NP.
- Of particular interest are the problems which are known to be in NP (we have already found a polynomial verification procedure) but for which we have no current proof of there membership in P (no polynomial solution strategy).

NP-Complete

- Of these problems (NP problems), there is a subset – known as NP-complete problems – which are the hardest NP problems and still have a polynomial verification process.
- This set contains, among others, both the satisfiability and colouring problems.

So What?

- So, why is this of interest to computer scientists?
- It has been conjectured that rather than being a subset of NP, P may be equal to NP.
- The consequences of this, if it were shown to be true, are of enormous importance.

If P = NP

- We could find polynomial solutions to all NP problems (including NP-complete problems).
- Any problem in NP would be as easy to solve as it is to verify.

Important Point!

- The identity P = NP is only conjecture.
- Although it would be very nice if it were true, there is currently no basis for assuming that it is.
- It is just that there is equally no basis for concluding that it is not.

More So What!

- Ok, so some problems are basically too hard to solve: NP-Complete problems.
- Some problems are difficult to solve algorithmically: NP problems.
- Does that mean we don't try to solve them? No
- We follow one of two approaches:
  - We use techniques that, hopefully, will bring us to a solution faster than brute-force.
  - We use techniques that get us close to a solution within reasonable time.
- We have already learnt the backtracking problem solving technique to solve some of such problems following the first approach.
- The remaining part of this unit will look at another technique ***branch-and-bound***, that also follows the second approach to solve some of such problems.
- Branch-and-Bound is applicable only to optimization problems. Backtracking is usually used for solving non-optimization problems. Both can be used to solve some large instances of such difficult problems.

Branch-and-Bound

- A further approach to problem solving is often adopted when the simpler methods (greedy, divide and conquer) fail is that of *Branch-and-Bound*.
- In this method, we attempt to 'prune' the solution tree by eliminating branches where we can safely assert that the (optimal) solution cannot be located.

Branch-and-Bound

- In general terms the algorithm starts with the following steps:
  - Find an initial possible solution – which, generally, is not the best solution. This establishes a *bound* on the final result.
  - Find the best solution – which, generally, is not possible. This establishes a second bound on the final result.
- We can now say that the actual solution lies between these two bounds.

Branch-and-Bound

- We now proceed as follows:
  - For each branch of the solution tree establish an estimate of the best expected solution that we can get by following this branch.
  - Eliminate all branches that are outside the bounds we have established.
  - Pick the branch with the best expectation as the next one to expand.
  - If we get a complete solution and we use it to update the appropriate bound.

Branch-and-Bound

- This technique is best seen with an example problem.
- We will use the assignment problem as an example.
- Simply stated the problem is to assign *n* jobs to *n* employees – one job to each employee.
- The problem is that each combination has a different cost and we must find the cheapest solution.

Using Branch-and-Bound for the assignment problem

- The assignment problem
  - A set of n agents are assigned n tasks.
  - Each agent performs exactly one task.
  - If agent i is assigned task j then a cost c_{ij} is associated with this combination.
  - The problem is to minimise the total cost C.

The assignment problem

- For example suppose agents a, b and c are to be assigned to tasks 1, 2 and 3 with the following cost matrix:

	1	2	3
a	4	7	3
b	2	6	1
c	3	9	4

- If we allocate task 1 to agent c, task 2 to agent b and task 3 to agent a, the total cost is 3 + 6 + 3 = 12. This provides an upper bound for assigning one task to one agent with minimum total cost.

The assignment problem

- The assignment problem has many applications:
  - Buildings and sites – the cost of erecting building i on site j.
  - Trucks and destinations – the cost of sending truck i to destination j.
  - Etc.
- In general, with n agents and n jobs there are $n!$ possible assignments.
- This is too many to consider, even for relatively small values of n.

The assignment problem

- Suppose we have to solve the following instance:

	1	2	3	4
a	11	12	18	40
b	14	15	13	22
c	11	17	19	23
d	17	14	20	28

- Note the assignment a ⇒ 1, b ⇒ 2, c ⇒ 3, d ⇒ 4 has cost 73.
- This represents a feasible solution
- This cost of 73, therefore, provides an *upper bound* on the solution.
- The minimal cost must be ≤ 73
- Can we get a *lower bound*?

The assignment problem

- We get a lower bound in the following way by choose according to the lowest cost for each task without considering the constraints – each assign performs exactly one task:
  - Task 1 must cost at least 11
  - Task 2 Must cost at least 12
  - Task 3 Must cost at least 13
  - Task 4 Must cost at least 22
- The minimal cost must be ≥ 58
- Hence, 58 ≤ Total Cost (C) ≤ 73

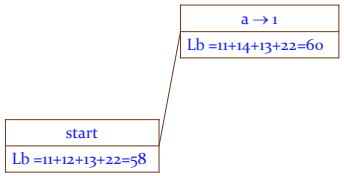
	1	2	3	4
a	11	12	18	40
b	14	15	13	22
c	11	17	19	23
d	17	14	20	28

The Solution Strategy

- The solution proceeds as follows:
  - We explore a tree whose nodes correspond to partial assignments.
  - At the root of the tree, no assignments have been made.
  - At each level we fix the assignment of one more agent.
  - At each node we calculate a bound on the costs that can be achieved by completing this sub tree.
  - We prune any sub tree with too high a minimum cost bound.

The assignment problem: 58 ≤ C ≤ 73

- First alternative: assign agent a to job 1, calculate lower bounds as follows:



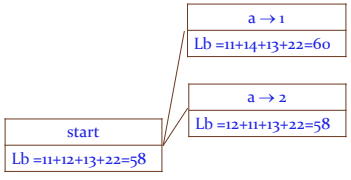
Ub = 73

	1	2	3	4
a	11	12	18	40
b	14	15	13	22
c	11	17	19	23
d	17	14	20	28

- If a ⇒ 1, the lower bound cost is 11 + 14 + 13 + 22 = 60 (choose according to the lowest cost for each unassigned task without considering the constraints – each agent performs exactly one task).

The assignment problem: 58 ≤ C ≤ 73

- Second alternative: assign agent a to job 2, calculate lower bound as follows:



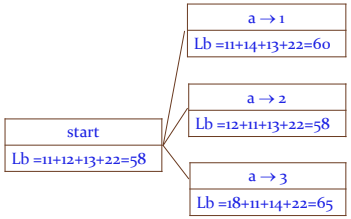
Ub = 73

	1	2	3	4
a	11	12	18	40
b	14	15	13	22
c	11	17	19	23
d	17	14	20	28

If a ⇒ 2, the lower bound cost is 12 + 11 + 13 + 22 = 58

The assignment problem: 58 ≤ C ≤ 73

- Third alternative: assign agent a to job 3, calculate lower bounds as follows:



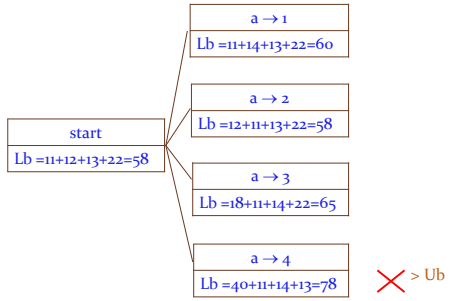
Ub = 73

	1	2	3	4
a	11	12	18	40
b	14	15	13	22
c	11	17	19	23
d	17	14	20	28

- If a ⇒ 3, the lower bound cost is 18 + 11 + 14 + 22 = 65

The assignment problem: 58 ≤ C ≤ 73

- Last alternative: assign agent a to job 4, calculate lower bounds as follows:



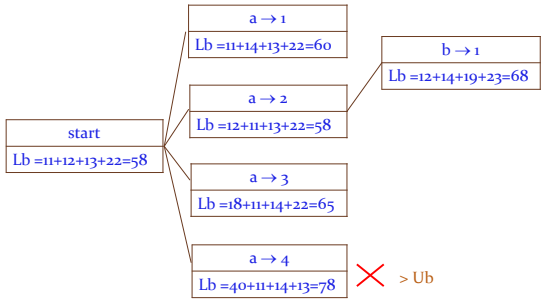
Ub = 73

	1	2	3	4
a	11	12	18	40
b	14	15	13	22
c	11	17	19	23
d	17	14	20	28

- If a ⇒ 4, the lower bound cost is 40 + 11 + 14 + 13 =78
- The minimum cost for this path is bigger than the upper bound, we can eliminate this branch

The assignment problem: 58 ≤ C ≤ 73

- Since a ⇒ 2 has the lowest lower bound, we expand a ⇒ 2 to the second level first.
- First alternative: assign agent b to job 1, calculate lower bounds as follows:



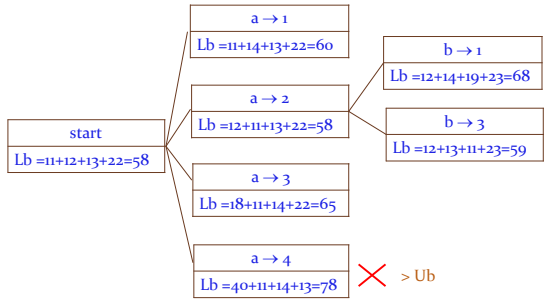
Ub = 73

	1	2	3	4
a	11	12	18	40
b	14	15	13	22
c	11	17	19	23
d	17	14	20	28

- If b ⇒ 1, the lower bound cost is 12 + 14 + 19 + 23 = 68

The assignment problem: 58 ≤ C ≤ 73

- Second alternative: assign agent b to job 3, calculate lower bounds as follows:



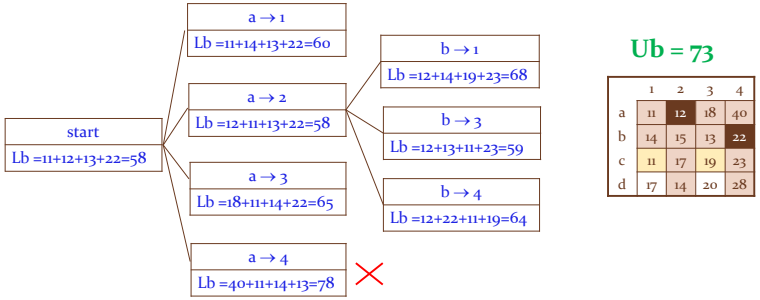
Ub = 73

	1	2	3	4
a	11	12	18	40
b	14	15	13	22
c	11	17	19	23
d	17	14	20	28

- If b ⇒ 3, the lower bound cost is 12 + 13 + 11 + 23 = 59

The assignment problem: 58 ≤ C ≤ 73

- Last alternative: assign agent b to job 4, calculate lower bounds as follows:



Ub = 73

	1	2	3	4
a	11	12	18	40
b	14	15	13	22
c	11	17	19	23
d	17	14	20	28

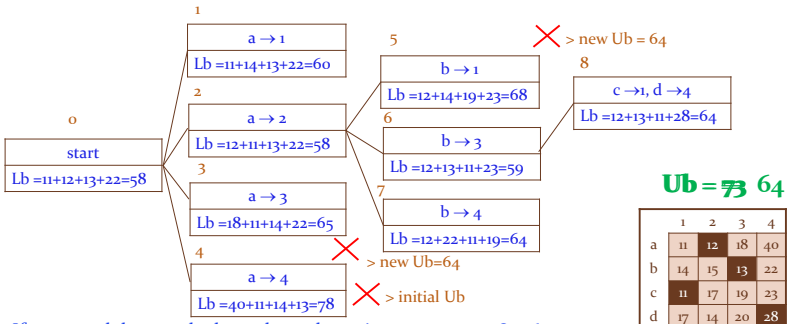
- If b ⇒ 4, the lower bound cost is 12 + 22 + 11 + 19 = 64
- Our best choice seems to be to assign b to task 3.

The assignment problem: 58 ≤ C ≤ 73

- Since b ⇒ 3 has the lowest lower bound, we next expand b ⇒ 3 to the third level.
- Since after assigning the next agent c, we left with one agent and one job, we should assign it together.
- In such a case, the lower bound cost is always for a feasible solution – each agent performs exactly one task
- If the cost for the feasible solution is lower than the current upper bound, the upper bound must be updated to the cost.
- We should eliminate sub trees with lower bound higher than the updated upper bound.

The assignment problem: 58 ≤ C ≤ 73

- First alternative: assign c and d to job 1 and 4 respectively, calculate the lower bound as follows:



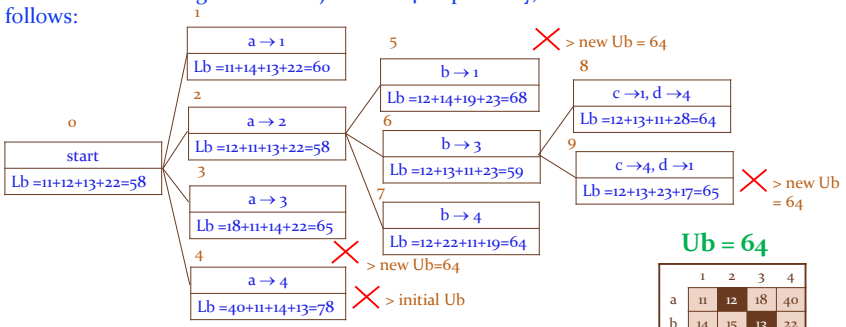
Ub = ~~73~~ 64

	1	2	3	4
a	11	12	18	40
b	14	15	13	22
c	11	17	19	23
d	17	14	20	28

- If c ⇒ 1 and d ⇒ 4 , the lower bound cost is 12 + 13 + 11 + 28 = 64
- Since the cost is for a feasible solution and has a value 64 < 73, 64 is a new upper bound.
- And, we eliminate several branches.

The assignment problem: 58 ≤ C ≤ 64

- First alternative: assign c and d to job 1 and 4 respectively, calculate the lower bound as follows:



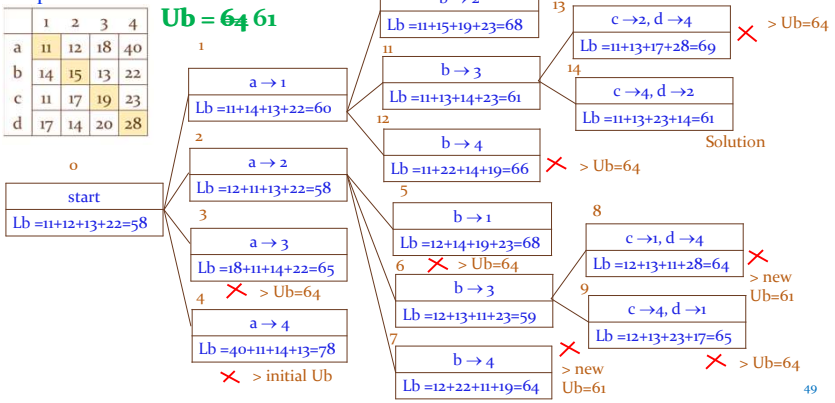
Ub = 64

	1	2	3	4
a	11	12	18	40
b	14	15	13	22
c	11	17	19	23
d	17	14	20	28

- If c ⇒ 1 and d ⇒ 4 , the lower bound cost is 12 + 13 + 23 + 17 = 65.
- We eliminate this new branch as Lb > Ub.

The assignment problem: $58 \le C \le 64$

- From the state-space tree, node 1 is the live node has the lowest lower bound, we next expand from node 1.



The assignment problem: final conclusion

- From previous slide, except the branch that has node 14, all the branches are eliminated.
- Hence, the state-space tree is completed now.
- So, the assignment in node 14 is a solution.
- Note that each step expands the node that has the lowest lower bound.
- When we arrive at a feasible solution, if its lower bound is lower than the upper bound, we immediately update the upper bound.

The assignment problem: final conclusion (cont'd)

- In this solution, the bounds are computed as follows:
  - Upper bound: the sum of the cost values in top-to-bottom diagonal (\ )
  - Lower bound: the sum of the lowest cost values in each column
- However, this is not the only method to compute the bounds. For example, we can also solve the problem by computing the bounds as follows:
  - Upper bound: the sum of the cost values in bottom-to-top diagonal (/)
  - Lower bound: the sum of the lowest cost values in each row
- Once we have decided an option for lower and upper bound

Using Branch-and-Bound to solve the knapsack problem

- **The knapsack problems**: Given n items of known weights w_i and values v_i, i = 1, 2, …., n, and a knapsack of capacity W, find the most valuable subset of the items that fit
- It is convenient to order the items in non-increasing order by their value-to-weight ratios.
- Each node in the state-space tree represents the result after i selection made: It records the total weight w and total value v so far along with the upper bound ub on the value that can be calculated s follows:

$$ub = v + (W - w)(\frac{v_{i+1}}{w_{i+1}})$$

Using Branch-and-Bound to solve the knapsack problem

- Let apply the branch-and-bound technique to the following instance of the knapsack problem:

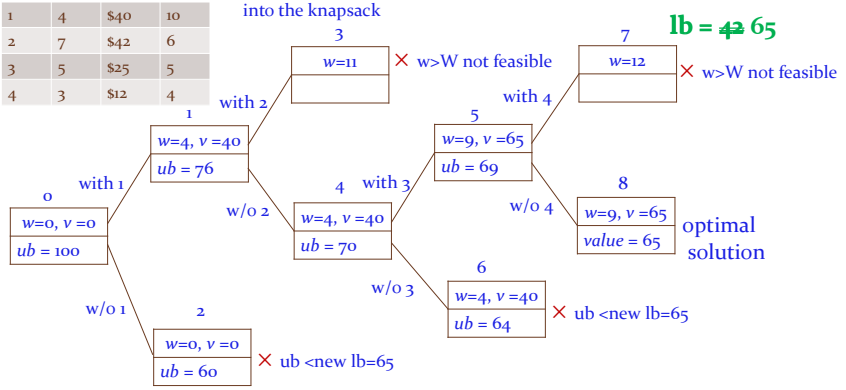
Item	Weight	Value	Value/Weight
1	4	$40	10
2	7	$42	6
3	5	$25	5
4	3	$12	4

- The knapsack capacity *W* is 10.

Using Branch-and-Bound to solve the knapsack problem

Item	w_i	v_i	v_i/w_i
1	4	$40	10
2	7	$42	6
3	5	$25	5
4	3	$12	4

initial lb = 42 (the highest value of the item (one item) that can fit into the knapsack



Branch and bound

- The basic idea behind branch-and-bound algorithms is shown in the preceding examples:
  - Explore the sample space of possible solutions.
  - At each stage, prune off any part of the sample space that cannot lead to a solution better than the best one found so far.
  - If we have not found any solution so far we can still prune off any part of the sample space for which the best possible solution is worse than the worst possible solution (initial bound).
- In branch-and-bound we look at all of the branches and select the most promising at each step, jumping from branch to branch as needed.
- Hence, it adopts the combination of both *breadth-first* and *depth-first* strategies.

Depth-First vs. Breadth-First

- Several strategies that we have already seen are depth first.
- Greedy:
  - Follow a single branch from root to leaf and then stop.
  - Note that we are also pruning all other branches at each level.
- Backtracking:
  - Follow a single branch as far as possible.
  - Go back up the tree only until you find an unexplored branch.
- Generally, the breadth-first strategy will find a solution more quickly (it will require less of the solution space to be explored) but will involve more housekeeping for the program.

Branch-and-Bound

- Many difficult problems can be attacked by using a branch-and-bound technique.
- Ways to estimate the bounds are required for the use of branch-and-bound technique. Hence, it is can only be used for optimization problems.
- The closer the estimate is to the real value, the better the strategy will work.
- It should be noted that, although branch-and-bound usually dramatically reduces the part of the state tree that needs to be examined, the strategy does not guarantee that a given problem will be any easier to solve than pure brute-force.
- Generally, however, branch-and-bound will massively reduce the work to be done.