Efficient Algorithms

Ekkapot Charoenwanit

Electrical and Computer Engineering (ECE)
TGGS
KMUTNB

Lecture 14: State Space Search

- Brute Force
- Backtracking
- Branch & Bound

Subset Sum

Definition: Given a multi-set $D = \{d_1, d_2, \ldots, d_n\}$, where $\forall d_i \geq 0$, and a target value k, find $S \subseteq D$ such that the sum of the elements in S equals k.

Example: $D = \{1,4,1,9,7\}$ and $k = 11$

Then, we have $S = \{1,1,9\}$, $\{4,7\}$.

Subset Sum: NP-Hard

We can show the ***NP-completeness*** of the decision version of $SubsetSum$ by reducing from $3SAT$:

$$3SAT \leq_p SubsetSum$$

Thus, the search version of $SubsetSum$ must be at least as hard as the decision version.

Subset Sum: Subset Enumeration

To enable us to search for solution to $SubsetSum$, we must decide on a ***format*** in which solutions S are represented.

We can ***enumerate*** all the possible subsets S using a ***bit vector***.

Suppose $x[1 \dots n]$ is a bit vector of length n.

$$x[i] = 1 \quad \text{iff} \quad d_i \in S$$

$$x[i] = 0 \quad \text{iff} \quad d_i \notin S$$

Subset Enumeration

Suppose $x[1 \dots n]$ is a bit vector of length n.

$$x[i] = 1 \quad \text{iff} \quad d_i \in S$$
$$x[i] = 0 \quad \text{iff} \quad d_i \notin S$$

Example: $D = \{1,4,1,9,7\}$ and $k = 11$.

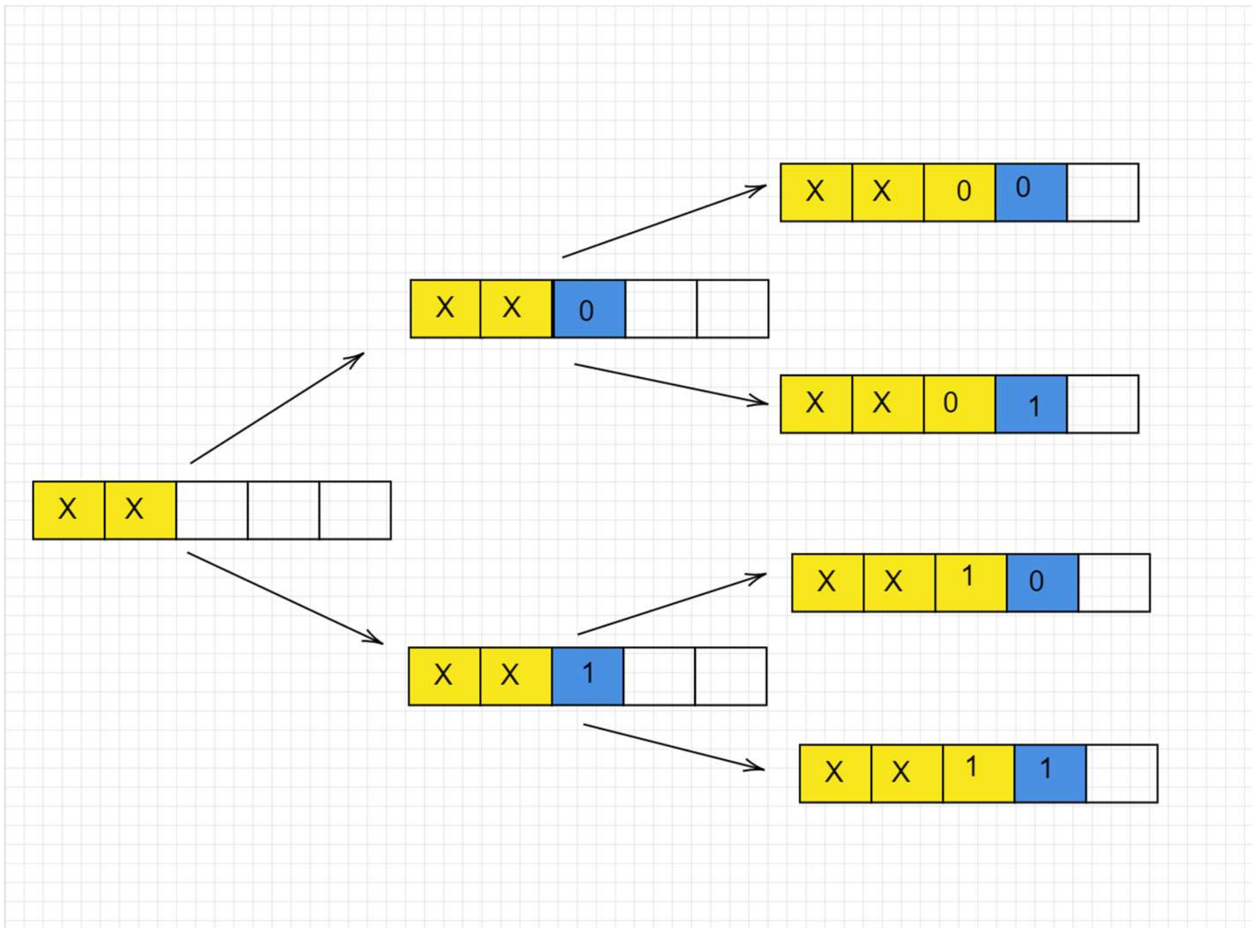
$S = \{1,1,9\}$ can be represented with $x = \{1,0,1,1,0\}$.

$S = \{4, 7\}$ can be represented with $x = \{0,1,0,0,1\}$.

Subset Enumeration: Binary Counter

```
procedure BINARY-COUNT(x[1...n], m)
    if m == n then
        PRINT(x)
    else
        x[m + 1] = 0
        BINARY-COUNT(x, m + 1)
        x[m + 1] = 1
        BINARY-COUNT(x, m + 1)
```

Subset Enumeration: Binary Counter



DFS and BFS

We can ***enumerate*** and ***evaluate*** solutions using

- Depth-First Search (***DFS***)
- Breadth-First Search (***BFS***)

DFS or BFS visits graphs or trees generated ***at runtime*** by the algorithm in question:

- Each state is represented by a vertex
- A decision is represented by an edge connecting one state to another adjacent state

If DFS is implemented via ***recursion***, ***stack frames*** can represent ***states***, generated by corresponding ***recursive calls***.

Subset Sum: Recursive DFS

procedure SUBSET-SUM$(d[1...n], k, x[1...n], m)$

    if $m == n$ **then**

        if $\mathrm{SUM}(d, x) == k$ **then**

            PRINT(x)

    else

        $x[m+1] = 0$

        SUBSET-SUM$(d, k, x, m+1)$

        $x[m+1] = 1$

        SUBSET-SUM$(d, k, x, m+1)$

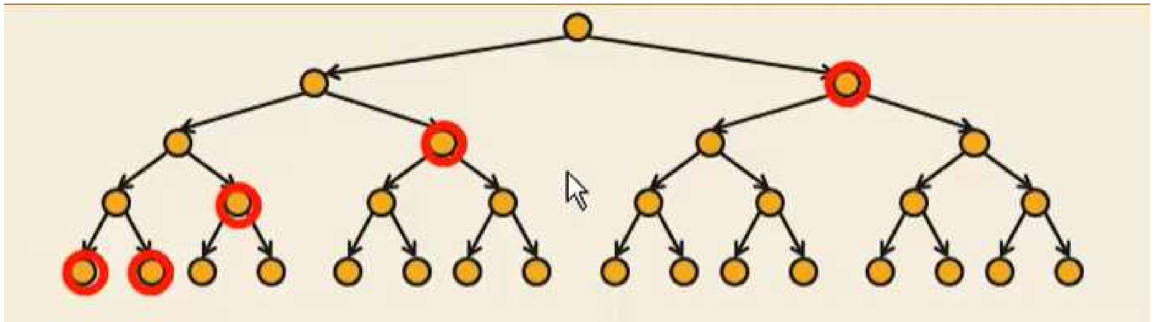
Subset Sum: Iterative DFS

```
procedure SUBSET-SUM(d[1...n], k)
    S = new Stack
    S.push([])
    while S ≠ ∅ do
        x[1...m] = S.pop()
        if m == n then
            if SUM(d, x) == k then
                PRINT(x)
        else
            x_0 = COPY(x, m + 1)
            x_0[m + 1] = 0
            x_1 = COPY(x, m + 1)
            x_1[m + 1] = 1
            S.push(x_1)
            S.push(x_0)
```

Subset Sum: BFS

```
procedure SUBSET-SUM(d[1...n], k)
    Q = new Queue
    Q.enqueue([])
    while Q ≠ ∅ do
        x[1...m] = Q.dequeue()
        if m == n then
            if SUM(d, x) == k then
                PRINT(x)
        else
            x0 = COPY(x, m + 1)
            x0[m + 1] = 0
            x1 = COPY(x, m + 1)
            x1[m + 1] = 1
            Q.enqueue(x0)
            Q.enqueue(x1)
```

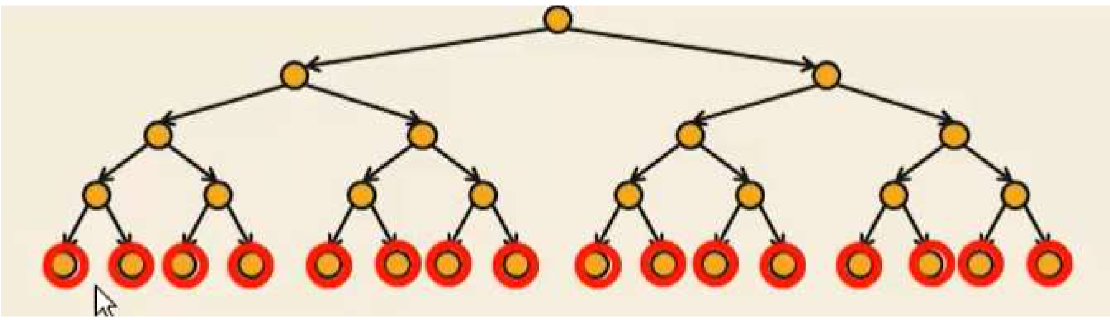
DFS: Memory Consumption



Suppose a search tree has a ***branching factor*** b and ***height*** h.

The ***space complexity*** is determined by the maximum number of states kept on the stack at any given point in time, which is $O(bh)$.

BFS: Memory Consumption



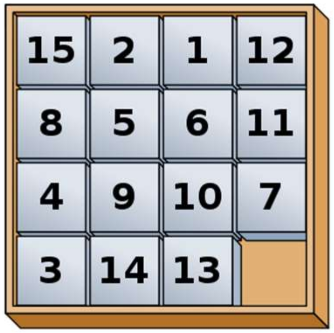
Suppose a search tree has a ***branching factor*** b and ***height*** h.

The ***space complexity*** is determined by the maximum number of states kept on the stack at any given point in time, which is $O(b^h)$.

15-Puzzle

Given the following instance of the 15-Puzzle, we would like to solve the puzzle in the ***fewest number of moves*** possible.

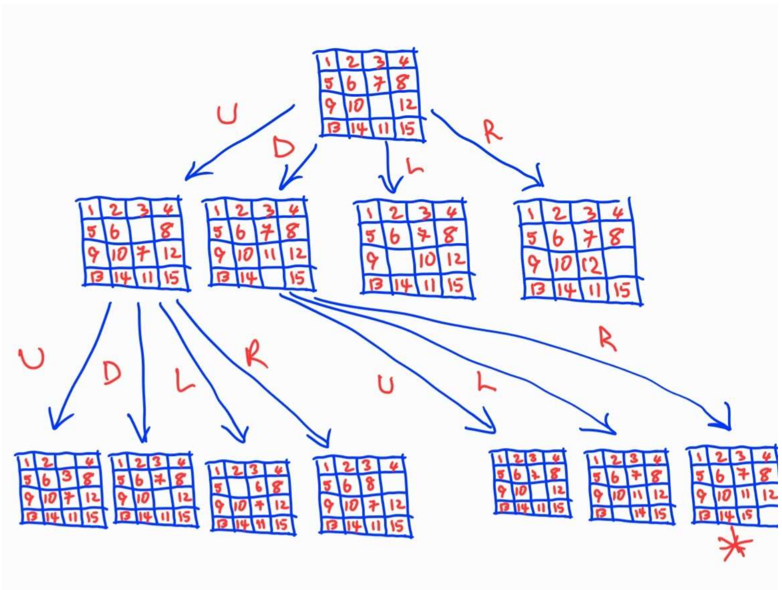
Which graph traversal technique would you choose? BFS or DFS?



15-Puzzle

Recall that BFS always finds a shortest path from a starting vertex to each reachable vertex in a graph.

Therefore, BFS will solve the 15-Puzzle in the ***fewest*** number of moves.



M3D2

Given a target value k, find a way from the initial value of 1 to get to k by performing only the following ***two operations*** successively:

- ***Multiplication by 3*** $(M3)$
- ***Division by 2*** $(D2)$

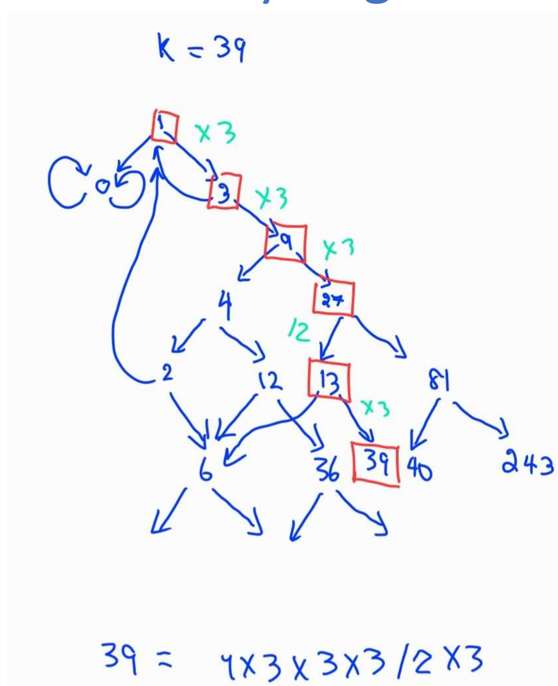
Example: $k = 41$

41 =

1x3x3/2/2x3x3x3x3/2/2 /2/2/2x3/2x3x3/2x3/2 /2/2x3/2/2x3x3/2x3x3 x3x3/2/2/2/2/2x3/2/2 /2/2x3/2x3/2x3/2x3/2/2

M3D2: BFS

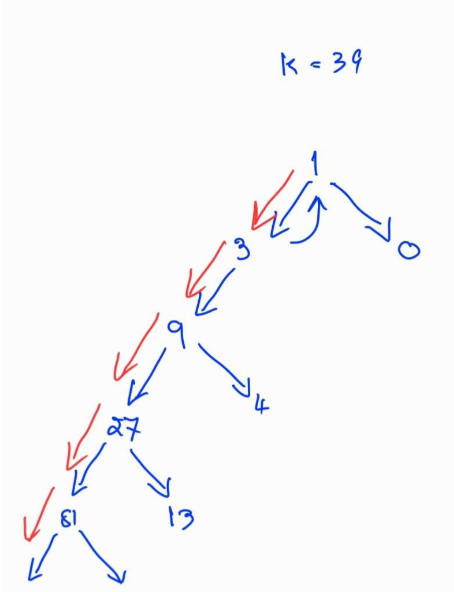
BFS yields the ***minimum*** number of operations possible to get to the target value k if there exists a way to get to k.



M3D2: DFS

Caveat: With the ***wrong*** state space design, DFS may not terminate on some inputs.

Example: Given $k = 39$, we choose to perform $M3$ before $D2$.



DFS vs BFS: Pros & Cons

DFS

- may not terminate on some inputs for infinite state-space graphs
- uses only $O(bh)$ stack space

BFS

- always yields a solution closest to the initial state
- uses only $O(b^h)$ queue space

Both approaches are bound to run ***extremely slowly*** for ***large*** state space trees/graphs.

- Memory can potentially run out before reaching a solution if any exists !!!

Backtracking

Exploring every solution state can be ***exponentially slow*** for ***large*** state-space graphs.

By ***intelligently*** considering each ***partial solution state***, we can sometimes avoid the need to go further down some subtrees that cannot contain ***answer states***.

Upon detecting such situations, we immediately ***backtrack*** and move on to look elsewhere.

Subset Sum

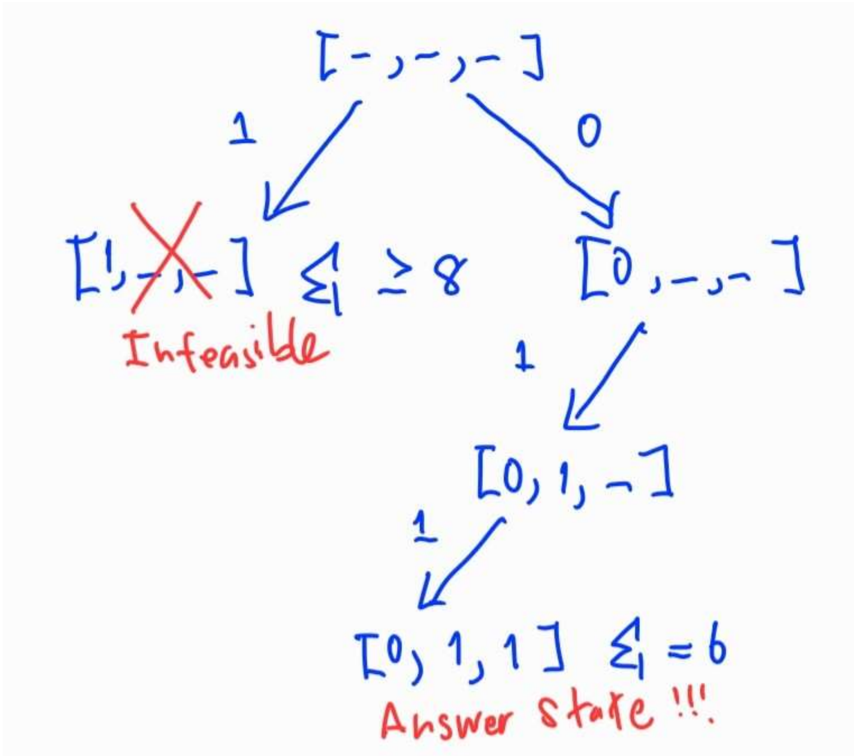
Let us revisit $SubsetSum$.

Example: $D = \{8, 2, 4\}$ and $k = 6$

Suppose we fill the bitvector x with 1 before 0 and explore the state-space graph using DFS.

Upon exploring the partial solution state $[1, \_, \_]$, we can be sure that the sum can never be 6 if 8 is included, since 8 is already ***strictly greater than*** 6.

Subset Sum: Backtracking with DFS



Subset Sum: Backtracking with DFS

Backtracking occurs in ***lines 3-4***.

```
1:  procedure Subset-Sum(d[1...n], k, x[1...n], m)
2:      sum = sum(d, x, m)
3:      if sum > k then
4:          return
5:      if m == n then
6:          if sum == k then
7:              Print(x)
8:      else
9:          x[m + 1] = 1
10:         Subset-Sum(d, k, x, m + 1)
11:         x[m + 1] = 0
12:         Subset-Sum(d, k, x, m + 1)
```

Subset Sum: Backtracking with BFS

Backtracking occurs when the if-statement evaluates to ***false*** in ***line 7***.

```
1:  procedure SUBSET-SUM(d[1...n], k)
2:      Q = new Queue
3:      Q.enqueue([])
4:      while Q ≠ ∅ do
5:          x[1...m] = Q.dequeue()
6:          sum = SUM(d, x, m)
7:          if sum ≤ k then
8:              if m == n then
9:                  if SUM(d, x) == k then
10:                     PRINT(x)
11:             else
12:                 x_0 = COPY(x, m + 1)
13:                 x_0[m + 1] = 0
14:                 x_1 = COPY(x, m + 1)
15:                 x_1[m + 1] = 1
16:                 Q.enqueue(x_0)
17:                 Q.enqueue(x_1)
```

Backtracking: Lesson Learned

Backtracking can help ***prune*** a potentially large state-space graph, thereby cutting down on those ***exponentially many*** vertices corresponding to ***infeasible solutions***.

Backtracking can be combined with ***DFS*** or ***BFS*** to improve the ***running time*** as well as ***memory consumption*** in a number of ***NP-Hard search problems*** whose objectives are to find solutions that satisfy the given constraints.

Branch & Bound

Branch & Bound can help improve both the running time and the memory consumption in algorithms involving ***NP-Hard optimization problems*** in a similar manner to ***Backtracking***, which helps improve the running time and memory consumption in algorithms involving ***search problems***.

We can incorporate both ***Branch & Bound*** and ***Backtracking*** to solve optimization problems.

0/1 Knapsack Problem

Given a set S of n different items, each item is associated with a weight w_i and a value v_i, the goal is to pick items in such a way that maximizes the total value V and satisfies the capacity constraint W:

Maximize $V = \sum_{k=1}^{n} x_k v_k$

subject to $\sum_{k=1}^{n} x_k w_k \leq W$

$x_k \in \{0,1\}$

0/1 Knapsack Problem

0/1 Knapsack can be solved using ***dynamic programming***.

The running time is $O(nW)$, which is pseudo-polynomial.

In fact, ***0/1 Knapsack*** is an ***NP-Hard*** optimization problem.

Thus, an algorithm that can solve ***0/1 Knapsack*** in polynomial time has not been discovered so far.

0/1 Knapsack Problem

To apply ***Branch & Bound***, we need to associate a ***bound*** to each ***partial solution state***.

There can be a number of schemes that can be used to compute bounds. Some schemes might provide ***tighter bounds*** than others.

Ones that provide tighter bounds can potentially ***prune more subtrees*** and hence be more efficient both in terms of running time and memory consumption.

0/1 Knapsack Problem: Example

	weight	value	Value per unit weight
Item 1	3	21	7
Item 2	6	30	5
Item 3	5	15	3
Item 4	10	20	2
Item 5	8	8	1

0/1 Knapsack Problem: Example

Scheme: Assume that we can take all the ***undecided*** items.

Suppose that we have decided on the first $k-1$ items.

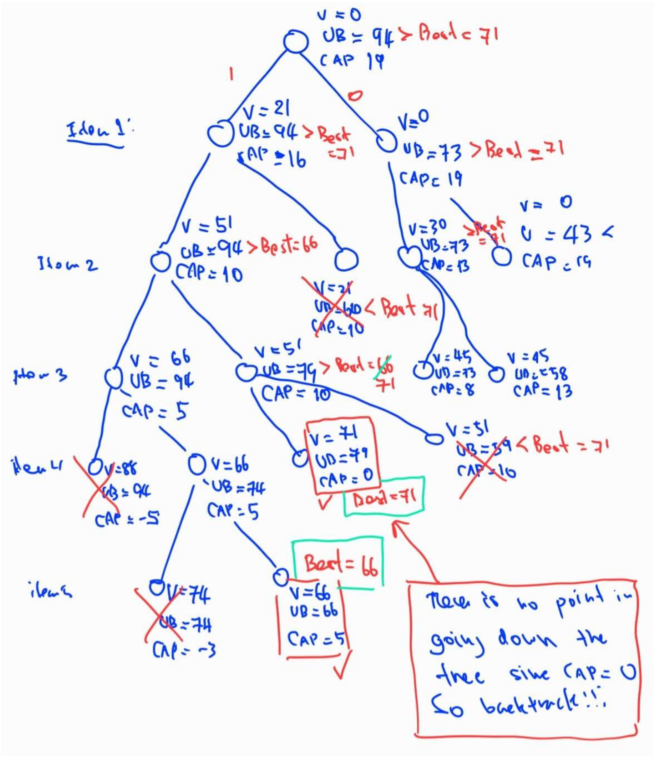
If item k is included, the upper bound UB remains the same whereas the remaining capacity is updated:

$$\mathrm{CAP} \coloneqq CAP - w_k$$

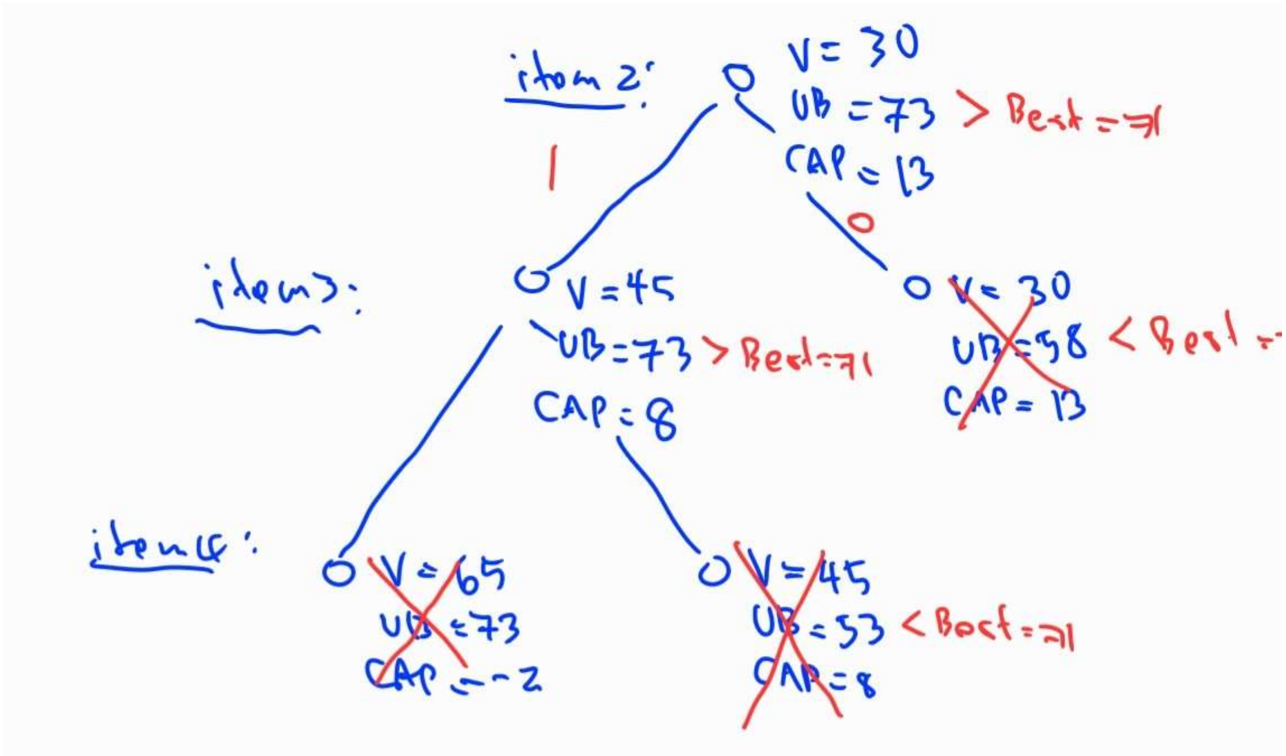
If item k is ***not*** included, the remaining capacity remains the same whereas the upper bound is updated:

$$UB := UB - v_k$$

0/1 Knapsack Problem: Example



0/1 Knapsack Problem: Example



0/1 Knapsack Problem:

A better scheme that can provide tighter upper bounds is to use the ***optimal values*** from the ***Fractional Knapsack Problem*** as ***upper bounds***.

Since ***Fractional Knapsack*** exhibits ***optimal substructure*** and ***greedy choice****,* we can find the optimal value using the greedy strategy of choosing one with the maximum value per unit, and the algorithm runs in polynomial time in the number of items n.

Branch & Bound : Lessons Learned

Branch & Bound can help prune subtrees that contain only solutions that cannot be better than the best solution found.

We decide whether to go down any given subtree by comparing the ***current best solution*** with the upper bound of the root vertex of that subtree.

Although there exists some computational overhead in computing an upper bound at each vertex, if that overhead is ***reasonably small*** relative to the number of solution states that can be potentially pruned, such an ***upper-bound scheme*** should be worth trying.

Summary

In this lecture, we covered the following state-space search techniques for solving NP-Hard and NP-complete problems:

- Brute-Force Search
- Backtracking
- Branch & Bound

In the next lecture, we will cover ***Randomized Algorithms***.