Algorithms and Lab (CSE130)

More Problems

Muhammad Tariq Mahmood

tariq@koreatech.ac.kr
School of Computer Science and Engineering

Note: **These notes are prepared from the following resources.**

- **(main text)Foundations of Algorithms, by Richard Neapolitan and Kumarss Naimipour**
- **Python Algorithm (파이썬 알고리즘) by Y.K. Choi (2021) (Korean)**
- **Introduction to the Design and Analysis of Algorithms by Anany Levitin**
- **Introduction to Algorithms, by By Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein**
- `https://www.geeksforgeeks.org`

Contents

P Class and NP Class Problems

- Types (categories) of Problems
  - Tractable problems
  - Intractable problems
  - Decision problems
  - Optimization problems
  - Undecidable problems
- Tractable Problem : Problems that can be solved in polynomial time are called tractable
- Examples
  - Linear search: $\Theta(n)$
  - bubble sort: $\Theta(n^2)$
  - Matrix Multiplication : $\Theta(n^{2.38})$
  - Merge Sort: $\Theta(n \lg n)$
- Intractable Problem : problems that cannot be solved in polynomial time are called intractable

P Class and NP Class Problems (cont...)

- Examples

$$W(n) = \mathrm{O}(2^n)$$
$$W(n) = \mathrm{O}(n!)$$
$$W(n) = \mathrm{O}\left(2^{\sqrt{n}}\right)$$

- Decision problems : decision problems are problems with yes/no answers.
  Example: Does a graph G have a MST of $weight \leq W$?

- Optimization Problem : An optimization problem is one which asks, "What is the optimal solution to problem X?".
  Many problems will have decision and optimization versions
  Example: Traveling salesman problem
  optimization: find hamiltonian cycle of minimum weight
  decision: is there a hamiltonian cycle of $weight \leq k$

- Undecidable problems : Some decision problems cannot be solved at all by any algorithm. Such problems are called **undecidable**

- Class P problems Class P is a class of decision problems that can be solved in polynomial time by (deterministic) algorithms. That is, they are solvable in $O(p(n))$, where $p(n)$ is a polynomial on n This class of problems is called **polynomial**.

P Class and NP Class Problems (cont...)

- Deterministic algorithm: In deterministic algorithm, for a given particular input, the algorithm will always produce the same output going through the same states
- Nondeterministic algorithm : A non-deterministic algorithm can provide different outputs for the same input on different executions. A nondeterministic algorithm is a two-stage procedure that takes as its input an instance *I* of a decision problem and does the following.

  1. **Nondeterministic ("guessing") stage:** An arbitrary string *S* is generated that can be thought of as a candidate solution to the given instance *I*
  2. **Deterministic ("verification") stage:** A deterministic algorithm takes both I and S as its input and outputs yes if *S* represents a solution to instance *I*.

- Nondeterministic polynomial : A **nondeterministic algorithm** is said to be nondeterministic polynomial if the time efficiency of its verification stage is polynomial.
- Class NP Problems : Class NP is the class of decision problems that can be solved by nondeterministic polynomial algorithms. This class of problems is called nondeterministic polynomial.
- There are many important problems, however, for which no polynomial-time algorithm has been found, nor has the impossibility of such an algorithm been proved.
- Some of the best-known problems are listed below:

P Class and NP Class Problems (cont...)

4. **Knapsack problem:** Find the most valuable subset of n items of given positive integer weights and values that fit into a knapsack of a given positive integer capacity.

5. **Hamiltonian circuit problem:** Determine whether a given graph has a Hamiltonian circuit—a path that starts and ends at the same vertex and passes through all the other vertices exactly once.

6. **Traveling salesman problem:** Find the shortest tour through n cities with known positive integer distances between them (find the shortest Hamiltonian circuit in a complete graph with positive integer weights).

7. **Partition problem:** Given n positive integers, determine whether it is possible to partition them into two disjoint subsets with the same sum.

8. **Bin-packing problem** Given n items whose sizes are positive rational numbers not larger than 1, put them into the smallest number of bins of size 1.

9. **Graph-coloring problem** For a given graph, find its chromatic number, which is the smallest number of colors that need to be assigned to the graph's vertices so that no two adjacent vertices are assigned the same color.

10. **Integer linear programming problem:**Find the maximum (or minimum) value of a linear function of several integer-valued variables subject to a finite set of constraints in the form of linear equalities and inequalities.

Knapsack Problem

- Given a knapsack with maximum capacity W, and a set S consisting of n items.
- items are of known weights $\{w_1, ..., w_n\}$ and values $\{v_1, ..., v_n\}$. It means that each item i has some weight w_i and a value v_i (all w_i , v_i and W are integer values) .

Item i	Weight w_i	Value v_i
1	2	3
2	3	4
3	4	5
4	5	8
5	9	10

Knapsack can hold W = 20

- **Problem:** How to pack the knapsack to achieve maximum total value of packed items?
- Mathematically, the problem can be expressed as

$$\begin{aligned} &\text{maximize} && \sum_{i \in \text{subset of items}} v_i \\ &S.T. && \sum_{i \in \text{subset of items}} w_i \leq W \end{aligned}$$

- The knapsack problem is in combinatorial optimization problem. It appears as a subproblem in many, more complex mathematical models of real-world problems.

Dynamic Programming based Solution

Dynamic Programming based Algorithm for 0-1 Knapsack problem

- For each $i \leq n$ and each $w_i \leq W$ solve the knapsack problem for the first i objects when the capacity is w_i.
- Why will this work? Because solutions to larger sub problems can be built up easily from solutions to smaller ones.
- We construct a matrix $F[0...n][0...W]$ for storing maximum profit values.
- For $1 \leq i \leq n$, and $0 \leq j \leq W$, $F[i][j]$ will store the maximum value of any set of objects $\{1, 2, ..., \mathrm{i}\}$ that can fit into a knapsack of weight j.
- $F[n][W]$ will contain the maximum value of all n objects that can fit into the entire knapsack of weight W.
- Table F for solving the knapsack problem by dynamic programming

		0	$j-w_i$	j	W
	0	0	0	0	0
	$i-1$	0	$F(i-1, j-w_i)$	$F(i-1, j)$	
w_i, v_i	i	0		$F(i, j)$	
	n	0			goal

Figure 1: Table for solving the knapsack problem by dynamic programming

Dynamic Programming based Solution (cont...)

- To compute entries of $F[i][j]$, consider two cases:

  1. **Leave object** i: If we choose to not take object i, then the optimal value will come about by considering how to fill a knapsack of size j with the remaining objects $\{1, 2, \ldots, \mathrm{i}-1\}$. This is just $F[i-1][j]$.

  2. **Take object i:** If we take object i, then we gain a value of v_i. But we use up w_i of our capacity. With the remaining $(j - w_i)$ capacity in the knapsack, we can fill it in the best possible way with objects $\{1, 2, \ldots, \mathrm{i}\}$. This is $v_i + F[i-1][j-w_i]$. This is only possible if $w_i \leq j$.

- Recursive Property

$$\left\{\begin{array}{l} F[i][j] = -\infty, \quad \text{if } j < 0 \\ F[0][j] = 0, \quad \text{if } j \geq 0 \end{array}\right\} (\text{base cases}) \qquad \left\{\begin{array}{l} F[i][j] = F[i-1][j], \qquad \text{if } w_i > j, \quad \langle \text{Leave it} \rangle \\ F[i][j] = \max\left(F[i-1][j], \; F[i-1][j-w_i] + v_i\right), \quad \text{if } w_i \leq j \quad \langle \text{Take it} \rangle \end{array}\right\} (\text{Recursive Cases})$$

- Algorithm for knapsack problem by dynamic programming (Pseudo-code)

Dynamic Programming based Solution (cont...)

```
1: procedure dpKnapsack(Integer n, Integer W, Integer w[], Number v[])
2:     Integer i, j, Number F[][]                                   ▷ variables
3:     for (j = 1; i <= W; i++) do                                  ▷ base case-1
4:         F[0][j] = 0
5:     end for
6:     for (i = 1; i <= n; i++) do                                  ▷ base case-2
7:         F[i][0] = 0
8:     end for
9:     for (i = 1; i <= n; i++) do                                  ▷ Recursive Cases
10:        for (j = 1; j <= W; j++) do
11:            if (w_i <= j) then
12:                F[i][j] = maximum(F[i − 1][j] + v_i + F[i − 1][j − w_i])
13:            else
14:                F[i][j] = F[i − 1][j]
15:            end if
16:        end for
17:    end for
18:    return F[n][W]
19: end procedure
```

- **Complexity**

$$T(n) \in \Theta(nW)$$

Greedy Approach for Knapsack Problem

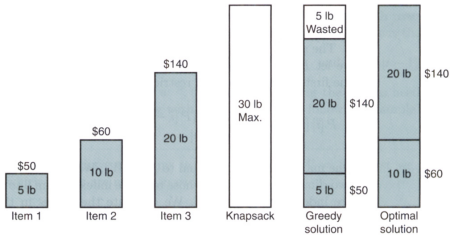
0/1 Knapsack Problem(Greedy Approach is failed to provide optimal solution)

- In the 0-1 Knapsack problem, we are not allowed to break items. We either take the whole item or don't take it.
- Possible strategies for **Greedy approach**
  1. An obvious greedy strategy is to steal the items with the **largest profit** first; that is, steal them in non-increasing order according to profit. This strategy, however, would not work very well if the most profitable item had a large weight in comparison to its profit.
  2. Another greedy strategy is to steal the **lightest** items first. This strategy fails badly when the light items have small profits compared with their weights.
  3. A more sophisticated greedy strategy is to steal the items with the largest profit per unit weight first. That is, we order the items in non-increasing order according to profit per unit weight, and select them in sequence

- Consider the problem instance

item	weight(lb)	profit($)	profit per unit
1	5	50	10
2	20	140	7
3	10	60	6

- Solution



Greedy Approach for Knapsack Problem (cont...)

Fractional Knapsack Problem

- In **Fractional Knapsack Problem :**, we can break items for maximizing the total value of knapsack. This problem in which we can break an item is also called the fractional knapsack problem.
- An efficient solution is to use **Greedy approach.** The basic idea of the greedy approach is to calculate the ratio value/weight for each item and sort the item on basis of this ratio. Then take the item with the highest ratio and add them until we can't add the next item as a whole and at the end add the next item as much as we can. Which will always be the optimal solution to this problem.
- Solution

$$50\,(5) + 140\,(20) + \frac{60}{10}\,(5) = 220$$

```
1:  procedure GreedyFractionalKnapsack(w[1..n], p[1..n], W)
2:      for (i = 0; i <= n; i++) do
3:          T[i]=0
4:          weight = 0
5:      end for
6:      for (i = 0; i <= n − m; i++) do
7:          if weight + w[i] ≤ W then
8:              T[i] = 1
9:              weight = weight + w[i]
10:         else
11:             T[i] = (W − weight)/w[i]
12:             break
13:         end if
14:     end for
15:     return T[]
16: end procedure
```

Backtracking based solution for 0-1 Knapsack problem

- In this problem we have a set of items, each of which has a weight and a profit
- This problem can be solved using a state space tree exactly like the one in the Sum-of-Subsets problem
- Each path from the root to a leaf is a candidate solution
- For optimization problems we always visit a promising node's children.
- The following is a general algorithm for backtracking in the case of optimization problems.
- The variable *best* has the value of the best solution found so far, and *value*(v) is the value of the solution at the node.

```
1:  procedure checknode(Node v)
2:      Node u
3:      if (value(v) is better than best) then
4:          best = value(v)
5:      end if
6:      if (promising(v)) then
7:          for each child u of v do
8:              checknode(u)
9:          end for
10:     end if
11: end procedure
```

Backtracking for 0-1 Knapsack problem (cont...)

- Suppose we have $n = 4$, $W = 16$ and the details given in the Table 1.

Table 1: input for an instance of 0-1 knapsack problem

i	p_i	w_i	$\frac{p_i}{w_i}$
1	40	2	20
2	30	5	6
3	50	10	5
4	10	5	2

- The pruned state space tree produced using backtracking. Stored at each node from top to bottom are the total profit of the items stolen up to the node, their total weight, and the bound on the total profit that could be obtain

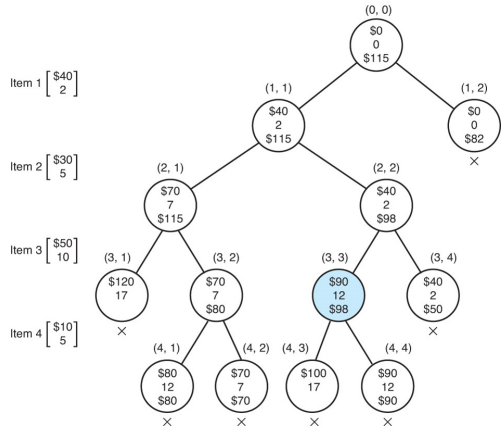


Figure 2:

Backtracking for 0-1 Knapsack problem (cont...)

- **Function (Promising) for Knapsack problem:**
  - It depends on profit that could be obtained by expanding beyond that node.
  - *profit* be the sum of the profits of the items included up to the node.
  - *weight* is the sum of the weights of those items
  - Suppose the node is at level i, and the node at level k is the one that would bring the sum of the weights above W. Then

$$totweight = weight + \sum_{j=i+1}^{k-1} w_j$$

$$bound = \underbrace{\left(profit + \sum_{j=i+1}^{k-1} p_j \right)}_{\text{profit from first } k-1 \text{ items}} + \underbrace{(W - totweight)}_{\text{Remaing capacity for } k^{th} \text{ item}} \times \underbrace{\frac{p_k}{w_k}}_{\text{ratio for } k^{th} \text{ item}}$$

  - The node is promising if ($bound > \text{maxprofit}$)
  - The node is not promising if ($bound \leq \text{maxprofit}$)

Backtracking for 0-1 Knapsack problem (cont...)

Computing Bounds

- Node $(0, 0)$: Compute its *profit* , *weight* and *bound*.

$$profit = 0$$
$$weight = 0$$

Because $2 + 5 + 10 = 17$, and $17 > 16$, the value of W, the third item would bring the sum of the weights above W. Therefore, $k = 3$, and we have bound

$$totweight = weight + \sum_{j=0+1}^{3-1} w_j = 0 + 2 + 5 = 7$$

$$bound = profit + \sum_{j=0+1}^{3-1} p_j + (W - totweight) \times \frac{p_3}{w_3}$$

$$= \$0 + \$40 + \$30 + (16 - 7) \times \frac{\$50}{10} = \$115.$$

- Node $(1, 1)$: Compute its *profit* , *weight* and *bound*.

$$profit = \$0 + \$40 = \$40$$
$$weight = 0 + 2 = 2$$

Because $2 + 5 + 10 = 17$, and $17 > 16$, the value of W, the third item would bring the sum of the weights above W. Therefore, $k = 3$, and we have bound

$$totweight = weight + \sum_{j=0+1}^{3-1} w_j = 0 + 2 + 5 = 7$$

$$bound = profit + \sum_{j=0+1}^{3-1} p_j + (W - totweight) \times \frac{p_3}{w_3}$$

$$= \$0 + \$40 + \$30 + (16 - 7) \times \frac{\$50}{10} = \$115.$$

Backtracking for 0-1 Knapsack problem (cont...)

Backtracking Algorithm for 0-1 Knapsack problem

- Problem: Let n items be given, where each item has a weight and a profit. The weights and profits are positive integers. Furthermore, let a positive integer W be given. Determine a set of items with maximum total profit, under the constraint that the sum of their weights cannot exceed W.
- Inputs: Positive integers n and W; arrays w and p, each indexed from 1 to n, and each containing positive integers sorted in nonincreasing order according to the values of $p[i]/w[i]$.
- Outputs: an array bestset indexed from 1 to n, where the values of bestset[i] is "yes" if the ith item is included in the optimal set and is "no" otherwise; an integer maxprofit that is the maximum profit.

,

Pseudo-code for Backtracking Algorithm for 0-1 Knapsack problem

```
1: procedure knapsack(index i,int profit, int weight)
2:     if (weight <= W    &&    profit > maxprofit) then
3:         maxprofit = profit
4:         numbest = i
5:         bestset = include
6:     end if
7:     if (promising(i)) then
8:         include [i + 1] = "yes"
9:         knapsack(i + 1, profit + p[i + 1], weight + w[i + 1])
10:        include [i + 1] = "no"
11:        knapsack (i + 1, profit, weight)
12:    end if
13: end procedure
```

Backtracking for 0-1 Knapsack problem (cont...)

Pseudo-code for promising function for 0-1 Knapsack problem

```
1:  procedure promising (i, maxProfit)
2:      int, j, , k, int, totweight, float, bound
3:      if (weight >= W) then
4:          return false
5:      else
6:          j = i + 1, bound = profit, totweight = weight
7:          while (j <= n  &&  totweight + w[j] <= W) do
8:              totweight = totweight + w[j]
9:              bound = bound + p[j]
10:             j++
11:         end while
12:         k = j
13:         if (k <= n) then
14:             bound = bound + (W − totweight) ∗ p[k]/w[k]
15:         end if
16:         return (bound > maxprofit)
17:     end if
18: end procedure
```

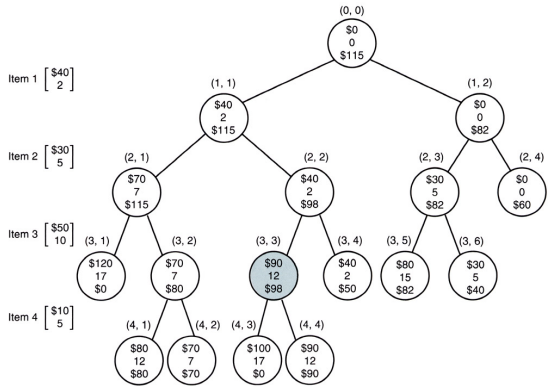
Backtracking for 0-1 Knapsack problem (cont...)

Time Complexity Analysis

- The state space tree in the 0–1 Knapsack problem is the same as that in the Sum-of-Subsets problem.
- the number of nodes in that tree is $2^{n+1} - 1$
- In the worst case, the backtracking algorithm checks $O(2^n)$ nodes.
- Monte Carlo technique can be used to estimate the efficiency of the algorithm for a particular instance
- the dynamic programming algorithm for the 0–1 Knapsack problem is in $O(minimum(2^n, nW))$.
- Horowitz and Sahni (1978) found that the backtracking algorithm is usually more efficient than the dynamic programming algorithm.
- Horowitz and Sahni (1974) coupled the divide-and-conquer approach with the dynamic programming approach to develop an algorithm for the 0–1 Knapsack problem that is $O(2^{n/2})$ in the worst case

Branch-and-Bound based solution for 0-1 Knapsack problem with BFS

- The backtracking algorithm for the 0–1 Knapsack problem is actually a branch-and-bound algorithm.
- In that algorithm, the promising function returns false if the value of bound is not greater than the current value of maxprofit.
- A backtracking algorithm, however, does not exploit the real advantage of using branch-and-bound.
- Besides using the bound to determine whether a node is promising, we can compare the bounds of promising nodes and visit the children of the one with the best bound.
- The pruned state space tree produced using breadth-first search with branch-and-bound pruning. Stored at each node from top to bottom are the total profit of the items stolen up to that node, their total weight, and the bound on the total profit that could be obtained by expanding beyond the node. The node shaded in color is the one at which an optimal solution is found.



Branch-and-Bound for 0-1 Knapsack (cont...)

Branch-and-Bound Algorithm for 0/1 Knapsack problem:

- **Problem:** Let n items be given, where each item has a weight and a profit. The weights and profits are positive integers. Furthermore, let a positive integer W be given. Determine a set of items with maximum total profit, under the constraint that the sum of their weights cannot exceed W.
- **Inputs:** positive integers n and W, arrays of positive integers w and p, each indexed from 1 to n, and each of which is sorted in nonincreasing order according to the values of $p[i]/w[i]$.
- **Outputs:** an integer *maxprofit* that is the sum of the profits in an optimal set.
- **Pseudo-code for Backtracking Algorithm for 0-1 Knapsack problem**

```
1:  procedure knapsack2(index i,int p[], int w[], int W)
2:      Queue Q, Node u, v
3:      v.level = 0; v.profit = 0; v.weight = 0, maxprofit = 0
4:      enqueue(Q, v)
5:      while (!empty(Q)) do
6:          dequeue(Q, v)
7:          u.level = v.level + 1
8:          u.weight = v.weight + w[u.level]
9:          u.profit = v.profit + p[u.level]
10:         if (u.weight <= W&& u.profit > maxprofit) then
11:             maxprofit = u.profit
12:         end if
13:         if (bound(u) > maxprofit) then
14:             enqueue(Q, u)
15:         end if
16:         (u.weight = v.weight)
17:         u.profit = v.profit
18:         if (bound(u) > maxprofit) then
19:             enqueue(Q, u)
20:         end if
21:     end while
22: end procedure
```

Branch-and-Bound for 0-1 Knapsack (cont...)

- Function **bound** is essentially the same as function **promising**

```
1:  procedure bound(int, i)
2:      int, j, , k, int, totweight, float, bound
3:      if (weight >= W) then
4:          return false
5:      else
6:          j = i + 1, bound = profit, totweight = weight
7:          while (j <= n  &&  totweight + w[j] <= W) do
8:              totweight = totweight + w[j]
9:              bound = bound + p[j]
10:             j++
11:         end while
12:         k = j
13:         if (k <= n) then
14:             bound = bound + (W − totweight) ∗ p[k]/w[k]
15:         end if
16:         return (bound)
17:     end if
18: end procedure
```

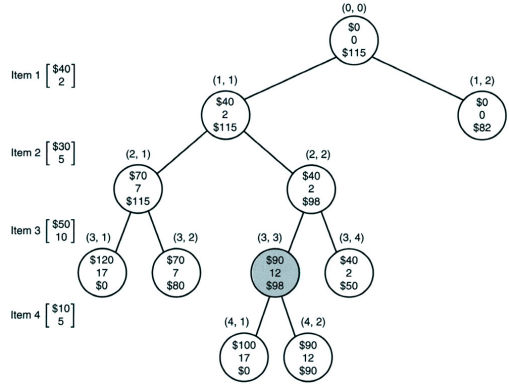
Branch-and-Bound for 0-1 Knapsack with Best First Search

Best-First Search with Branch-and-Bound Pruning Algorithm for the 0–1 Knapsack problem

- What makes branch and bound more efficient than backtracking is that, instead of traversing the tree in a predetermined order (depth-first, breadth-first, etc), we traverse it based on the optimization criteria for the problem.
- For example, in the 0-1 Knapsack Problem we want to maximize profit. So we pick a child to process based on which one has the highest maximum possible profit. This allows us to quickly determine a pretty good solution. That pretty good solution can be used to eliminate inferior solutions more quickly than in backtracking or breadth-first searching.
- It's important to realize that all the children yet to process are included in the set of possible next nodes. We pick the one with the highest maximum possible profit, so we may be jumping around the tree quite a bit. When we process a node, if the node is promising then we immediately add its children to the set of possible next nodes.
- So how do we write a best-first search? Recall that for a breadth-first search, we put nodes into a queue. For a best-first search, we put items into a data structure called a priority queue. In a priority queue, you can put items into the queue in any order, but you always remove the one with the maximum value first. In this way we put the children of a node into the priority queue, and when we pull them out we process the one with the maximum possible profit first.

Branch-and-Bound for 0-1 Knapsack with Best First Search (cont...)

- Thus, the implementation of best-first search consists of a simple modification to breadth-first search. Instead of using a **queue**, we use a **priority queue**.
- The pruned state space tree produced using best-first search with branch-and-bound pruning. Stored at each node from top to bottom are the total profit of the items stolen up to the node, their total weight, and the bound on the total profit that could be obtained by expanding beyond the node. The node shaded in color is the one at which an optimal solution is found.



Branch-and-Bound for 0-1 Knapsack with Best First Search (cont...)

- Best-First Search with Branch-and-Bound Pruning Algorithm for the 0–1 Knapsack problem

- **Problem:** Let n items be given, where each item has a weight and a profit. The weights and profits are positive integers. Furthermore, let a positive integer W be given. Determine a set of items with maximum total profit, under the constraint that the sum of their weights cannot exceed W.

- **Inputs:** positive integers n and W, arrays of positive integers w and p, each indexed from 1 to n, and each of which is sorted in nonincreasing order according to the values of $p[i]/w[i]$.

- **Outputs:** an integer *maxprofit* that is the sum of the profits of an optimal set.

- Branch-and-Bound Pruning Algorithm for the 0-1 Knapsack problem with Best-First Search

```
1:  procedure knapsack2(index i,int p[], int w[], int W)
2:      PriorityQueue PQ, Node u, v
3:      v.level = 0; v.profit = 0; v.weight = 0
4:      maxprofit = 0
5:      enqueue(PQ, v)
6:      while (!empty(PQ)) do
7:          dequeue(PQ, v)
8:          if (if(v.bound > maxprofit)) then
9:              u.level = v.level + 1
10:             u.weight = v.weight + w[u.level]
11:             u.profit = v.profit + p[u.level]
12:             if (u.weight <= W&&u.profit > maxprofit) then
13:                 maxprofit = u.profit
14:             end if
15:             if (bound(u) > maxprofit) then
16:                 enqueue(PQ, u)
17:             end if
18:             (u.weight = v.weight)
19:             u.profit = v.profit
20:             if (bound(u) > maxprofit) then
21:                 enqueue(PQ, u)
22:             end if
23:         end if
24:     end while
25: end procedure
```

Branch-and-Bound for 0-1 Knapsack with Best First Search (cont...)

- Function **bound** is essentially the same as function **promising**

```
1:  procedure bound(int, i)
2:      int, j, , k, int, totweight, float, bound
3:      if (weight >= W) then
4:          return false
5:      else
6:          j = i + 1, bound = profit, totweight = weight
7:          while (j <= n  &&  totweight + w[j] <= W) do
8:              totweight = totweight + w[j]
9:              bound = bound + p[j]
10:             j++
11:         end while
12:         k = j
13:         if (k <= n) then
14:             bound = bound + (W − totweight) ∗ p[k]/w[k]
15:         end if
16:         return (bound)
17:     end if
18: end procedure
```

Best-First Search with Branch-and-Bound Pruning Algorithm for the 0–1 Knapsack problem

- Suppose a salesperson is planning a sales trip that includes 20 cities.
- Each city is connected to some of the other cities by a road.
- To minimize travel time, we want to determine a shortest route that starts at the salesperson's home city, visits each of the cities once, and ends up at the home city.
- This problem of determining a shortest route is called the Traveling Salesperson problem.
- An instance of this problem can be represented by a weighted graph, in which each vertex represents a city.
- **Assumptions**
  - The weight (distance) going in one direction can be different from the weight going in another direction.
  - The weights are nonnegative numbers

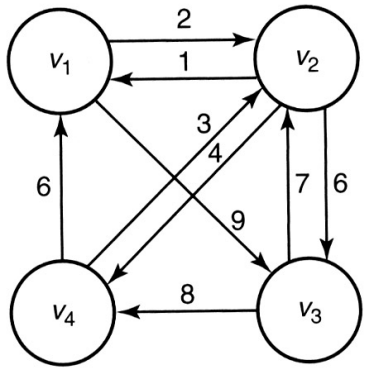


Figure 3: The optimal tour is $[v1, v3, v4, v2, v1]$

Traveling salesman problem (cont...)

- A **tour** (also called a **Hamiltonian circuit**) in a directed graph is a path from a vertex to itself that passes through each of the other vertices exactly once.
- An **optimal tour** in a weighted, directed graph is such a path of minimum length.
- The Traveling Salesperson problem is to find an optimal tour in a weighted, directed graph when at least one tour exists.
- The following are the three tours and lengths for the graph in Figure 3:are shown. The last tour is the optimal.

$$\begin{aligned} length\,[v_1, v_2, v_3, v_4, v_1] &= 22 \\ length\,[v_1, v_3, v_2, v_4, v_1] &= 26 \\ length\,[v_1, v_3, v_4, v_2, v_1] &= 21 \end{aligned}$$

- If we consider all possible tours, the second vertex on the tour can be any of $(n-1)$ vertices, the third vertex on the tour can be any of $(n-2)$ vertices, ..., the nth vertex on the tour can be only one vertex. Therefore, the total number of tours is

$$(n-1)\,(n-2)\cdots 1 = (n-1)!$$

Dynamic Programming based Solution

- If v_k is the first vertex after v_1 on an optimal tour, the subpath of that tour from v_k to v_1 must be a shortest path from v_k to v_1 that passes through each of the other vertices exactly once.
- The graph is presented by an adjacency matrix W
- Let

  $V = \textit{set of all vertices}$

  $A = \textit{a subset of } V$

  $D[v_i][v_1] = \textit{lengths of shortest path from} v_i \textit{ and } v_1$

- **Recursive Property**

$$\begin{cases} D[v_i][\phi] = W[i][1], \ A = \phi \\ D[v_i][A] = \underset{j: v_j \in A}{\text{minimum}} \left(W[i][j] + D\left[v_j\right]\left[A - \left\{v_j\right\}\right] \right), \ \textit{if } A \neq \phi \end{cases}$$

	1	2	3	4
1	0	2	9	∞
2	1	0	6	4
3	∞	7	0	8
4	6	3	∞	0

- **Example**
  - The vertex set $V = \{v_1, v_2, v_3, v_4\}$
  - Determine an optimal tour by considering A is empty set:

$$D[v_2][\{\}] = 1$$
$$D[v_3][\{\}] = \infty$$
$$D[v_4][\{\}] = 6$$

  - Next consider all sets containing one element:

$$D[v_3][\{v_2\}] = \underset{j:v_j \in \{v_2\}}{\text{minimum}} (W[3][j] + D[v_j][\{v_2\} - \{v_j\}])$$
$$= D[3][2] + D[v_2][\{\}]$$
$$= 7 + 1 = 8$$

  - Similarity

$$D[v_4][\{v_2\}] = 3 + 1 = 4$$
$$D[v_2][\{v_3\}] = 6 + \infty = \infty$$
$$D[v_4][\{v_3\}] = \infty + \infty = \infty$$
$$D[v_2][\{v_4\}] = 4 + 6 = 10$$
$$D[v_3][\{v_4\}] = 8 + 6 = 14$$

  - Next consider all sets containing two elements:

$$D[v_4][\{v_2, v_3\}] = \underset{j:v_j \in \{v_2, v_3\}}{\text{minimum}} (W[4][j] + D[v_j][\{v_2, v_3\} - \{v_j\}])$$
$$= \text{minimum}(D[4][2] + D[v_2][\{v_3\}],\ D[4][3] + D[v_3][\{v_2\}])$$
$$= \text{minimum}(3 + \infty, 8 + \infty) = \infty$$

Dynamic Programming based Solution (cont...)

- Similarly

$$D[v_3][\{v_2, v_4\}] = \text{minimum}(7 + 10, 8 + 4) = 12$$
$$D[v_2][\{v_3, v_4\}] = \text{minimum}(6 + 14, 4 + \infty) = 20$$

- Finally, compute the length of an optimal tour:

$$D[v_1][\{v_2, v_3, v_4\}] = \underset{j: v_j \in \{v_2, v_3, v_4\}}{\text{minimum}} (D[1][j] + D[v_j][\{v_2, v_3, v_4\} - \{v_j\}])$$
$$= \text{minimum}(D[1][2] + D[v_2][\{v_3, v_4\}], D[1][3] + D[v_3][\{v_2, v_4\}], D[1][4] + D[v_4][\{v_2, v_3\}])$$
$$= \text{minimum}(2 + 20, 9 + 12, \infty + \infty) = 21$$

Dynamic Programming based Solution (cont...)

Algorithm

- **Problem:** Determine an optimal tour in a weighted, directed graph. The weights are nonnegative numbers.
- **Inputs:** a weighted, directed graph, and n, the number of vertices in the graph. The graph is represented by a two-dimensional array W, which has both its rows and columns indexed from 1 to n, where W [i] [j] is the weight on the edge from ith vertex to the jth vertex.
- **Outputs:** a variable minlength, whose value is the length of an optimal tour, and a two-dimensional array P from which an optimal tour can be constructed. P has its rows indexed from 1 to n and its columns indexed by all subsets of V v1. P [i] [A] is the index of the first vertex after vi on a shortest path from vi to v1 that passes through all the vertices in A exactly once.
- Pseudo-code

```
1: procedure travel(i, W[][], P[][])
2:     Integer i, j, k, minlength
3:     Number D[1 ··· n][subset of V − {v1}]
4:     for (i = 2; i <= n; i++) do
5:         D[i][φ] = W[i][1]
6:     end for
7:     for (k = 1; k <= n − 2; k++) do
8:         for (subsets A ⊆ V − {v1} containing k vertices) do
9:             for (i such that i ≠ 1 and vi is not in A) do
10:
    D[i][A] = minimum_{j:vj∈A} (W[i][j] + D[j][A − {vj}])
11:                 P[i][A] = j
12:             end for
13:         end for
14:     end for
15:     D[1][V − {v1}] =
    minimum_{2≤j≤n} (W[1][j] + D[j][V − {v1, vj}])
16:     P[1][V − {v1}] = j
17:     return minlength
18: end procedure
```

- **Time Complexity Analysis**

$$T(n) = \sum_{k=1}^{n-2} (n-1-k)\, k \left(\begin{array}{c} n-1 \\ k \end{array} \right)$$

$$T(n) = (n-1) \sum_{k=1}^{n-2} k \left(\begin{array}{c} n-2 \\ k \end{array} \right)$$

i.e.

$$(n-1-k)\, k \left(\begin{array}{c} n-1 \\ k \end{array} \right) = (n-1) \left(\begin{array}{c} n-2 \\ k \end{array} \right)$$

$$T(n) = (n-1)(n-2)\, 2^{n-3} \in \Theta\left(n^2 2^n\right)$$

Back Tracking Algorithm for Traveling salesman problem

- Using the dynamic programming algorithm, with a time complexity given by

$$T(n) = (n-1)(n-2)\,2^{n-3} \in \Theta\left(n^2 2^n\right)$$

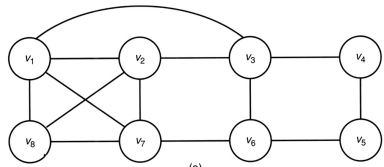
- we assumed that dynamic programming algorithm took 1 microsecond to process its basic operation. A quick calculation shows that it would take years for 40 cities

$$(40-1)(40-2)\,2^{40-3}\mu s = 6.46\,years$$

- The Hamiltonian Circuits problem determines the Hamiltonian Circuits in a connected, undirected graph.

- given a connected, undirected graph, a Hamiltonian Circuit (also called a tour) is a path that starts at a given vertex, visits each vertex in the graph exactly once, and ends at the starting vertex.

- The graph in Figure (a) contains the Hamiltonian Circuit [v1, v2, v8, v7, v6, v5, v4, v3, v1], but the one in Figure (b) does not contain a Hamiltonian Circuit.



Back Tracking Algorithm for Traveling salesman problem (cont...)

- A **state space tree** for this problem is as follows.
  - Put the starting vertex at level 0 in the tree; call it the zeroth vertex on the path.
  - At level 1, consider each vertex other than the starting vertex as the first vertex after the starting one.
  - At level 2, consider each of these same vertices as the second vertex, and so on.
  - Finally, at level $n-1$, consider each of these same vertices as the (n - 1)st vertex.

- The following considerations enable us to backtrack in this state space tree: (Promising function)
  1. The *ith* vertex on the path must be adjacent to the $(i-1)$st vertex on the path.
  2. The $(n-1)$st vertex must be adjacent to the $0th$ vertex (the starting one).
  3. The *ith* vertex cannot be one of the first $(i-1)$ vertices.

Back Tracking Algorithm for Traveling salesman problem (cont...)

Backtracking Algorithm

- **Problem:** Determine all Hamiltonian Circuits in a connected, undirected graph
- **Inputs:** positive integer n and an undirected graph containing n vertices. The graph is represented by a two-dimensional array W, which has both its rows and columns indexed from 1 to n, where $W[i][j]$ is true if there is an edge between the ith vertex and the jth vertex and false otherwise.
- **Outputs:** For all paths that start at a given vertex, visit each vertex in the graph exactly once, and end up at the starting vertex. The output for each path is an array of indices vindex indexed from 0 to $n-1$, where $vindex[i]$ is the index of the ith vertex on the path. The index of the starting vertex is $vindex[0]$.
- Pseudo-code

```
1:  procedure hamiltonian(int i)
2:      int j
3:      if (promising(i)) then
4:          if if(i == n − 1) then
5:              print vindex[0] through vindex[n − 1]
6:          else
7:                                  ▷ Children of the vortex i
8:              for (j = 2; j <= n; j + +) do
9:                  vindex[i + 1] = j
10:                 hamiltonian(i + 1)
11:             end for
12:         end if
13:     end if
14: end procedure
```

Back Tracking Algorithm for Traveling salesman problem (cont...)

- Algorithm for Function Promising (Pseudo-code)

```
1:  procedure Promising(int i)
2:      int j
3:      bool switch
4:      if (i == n − 1  &&  !W[vindex[n − 1]][vindex[0]]) then
5:          switch = false
6:      else
7:          if (i > 0  &&  !W[vindex[i − 1]][vindex[i]]) then
8:              switch = false
9:          else
10:             j = 1  switch = true
11:             while (j && switch) do
12:                 if (vindex[i] == vindex[j]) then
13:                     switch = false
14:                 end if
15:                 j=j+1
16:             end while
17:         end if
18:     end if
19:     return switch
20: end procedure
```

Time Complexity Analysis

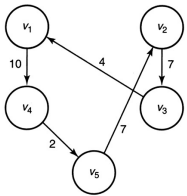
- n, W, and *vindex* are variables and the first call will be *vindex*[0] = 1 //Make v1 the starting vertex *hamiltonian*(0)
- The number of nodes in the state space tree for this algorithm is

$$1 + (n-1) + (n-1)^{\mathbf{2}} + \cdots + (n-1)^{n-\mathbf{1}} = \frac{(n-1)^n - 1}{n-2}$$

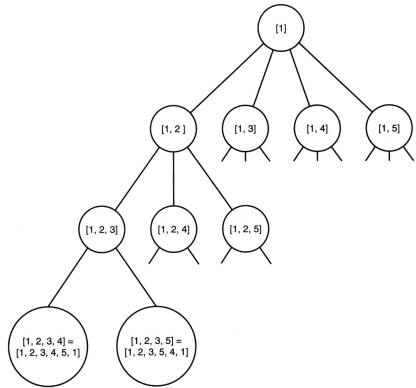
Best-First Search with Branch-and-Bound Pruning Algorithm for the Traveling Salesperson problem

- The goal in this problem is to find the shortest path in a directed graph that starts at a given vertex, visits each vertex in the graph exactly once, and ends up back at the starting vertex. Such a path is called an **optimal tour**.
- Figure shows the adjacency matrix representation of a graph containing five vertices, in which there is an edge from every vertex to every other vertex, and an optimal tour for that graph.

$$\begin{bmatrix} 0 & 14 & 4 & 10 & 20 \\ 14 & 0 & 7 & 8 & 7 \\ 4 & 5 & 0 & 7 & 16 \\ 11 & 7 & 9 & 0 & 2 \\ 18 & 7 & 17 & 4 & 0 \end{bmatrix}$$

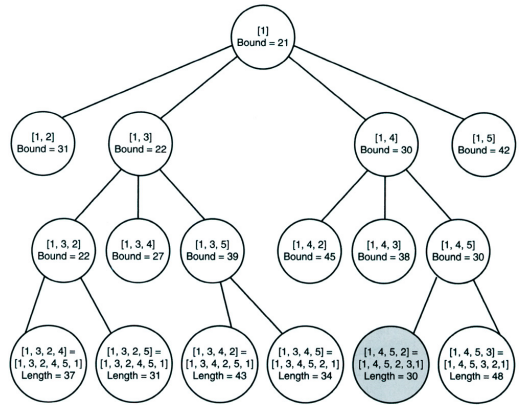


- A portion of this state space tree, in which there are five vertices and in which there is an edge from every vertex to every other vertex, is shown in Figure .



Best-First Search with BnB Algorithm for TSP (cont...)

- The pruned state space tree produced using best-first search with branch-and-bound pruning
- In Figure ,
  - At each node that is not a leaf in the state space tree, the partial tour is at the top and the bound on the length of any tour that could be obtained by expanding beyond the node is at the bottom.
  - At each leaf in the state space tree, the tour is at the top and its length is at the bottom. The node shaded in color is the one at which an optimal tour is found.



Best-First Search with BnB Algorithm for TSP (cont...)

- To use best-first search, we need to be able to determine a bound for each node.
- In this problem, we need to determine a lower bound on the length of any tour that can be obtained by expanding beyond a given node, and we call the node promising only if its bound is less than the current minimum tour length.
- a lower bound on the cost (length of the edge taken) of leaving vertex v1 is given by the minimum of all the nonzero entries in row 1 of the adjacency matrix, a lower bound on the cost of leaving vertex v2 is given by the minimum of all the nonzero entries in row 2, and so on.
- The lower bounds on the costs of leaving the five vertices in the graph are as follows

$$
\begin{aligned}
v_1 &\rightarrow \text{minimum}\,(14, 4, 10, 20) = 4 \\
v_2 &\rightarrow \text{minimum}\,(14, 7, 8, 7) = 7 \\
v_3 &\rightarrow \text{minimum}\,(4, 5, 7, 16) = 4 \\
v_4 &\rightarrow \text{minimum}\,(11, 7, 9, 2) = 2 \\
v_5 &\rightarrow \text{minimum}\,(18, 7, 17, 4) = 4
\end{aligned}
$$

- Because a tour must leave every vertex exactly once, a lower bound on the length of a tour is the sum of these minimums. Therefore, a lower bound on the length of a tour is

$$bound = 4 + 7 + 4 + 2 + 4 = 21$$

Best-First Search with BnB Algorithm for TSP (cont...)

- Suppose we have visited the node containing [1, 2], then bound will be

$$
\begin{aligned}
&v_1 \rightarrow 14 \\
&v_2 \rightarrow \text{minimum}\,(7, 8, 7) = 7 \\
&v_3 \rightarrow \text{minimum}\,(4, 7, 16) = 4 \\
&v_4 \rightarrow \text{minimum}\,(11, 9, 2) = 2 \\
&v_5 \rightarrow \text{minimum}\,(18, 17, 4) = 4 \\
&\textit{bound} = 14 + 7 + 4 + 2 + 4 = 31
\end{aligned}
$$

- suppose we have visited the node containing [1, 2, 3] then bound will be

$$
\begin{aligned}
&v_1 \rightarrow 14 \\
&v_2 \rightarrow 7 \\
&v_3 \rightarrow \text{minimum}\,(7, 16) = 7 \\
&v_4 \rightarrow \text{minimum}\,(11, 2) = 2 \\
&v_5 \rightarrow \text{minimum}\,(18, 4) = 4 \\
&\textit{bound} = 14 + 7 + 7 + 2 + 4 = 34
\end{aligned}
$$

Best-First Search with BnB Algorithm for TSP (cont...)

- Branch and Bound Algorithm

- **Problem:** Determine an optimal tour in a weighted, directed graph. The weights are nonnegative numbers.

- **Inputs:** a weighted, directed graph, and n, the number of vertices in the graph. The graph is represented by a two-dimensional array W, which has both its rows and columns indexed from 1 to n, where $W[i][j]$ is the weight on the edge from the ith vertex to the jth vertex.

- **Outputs:** variable *minlength*, whose value is the length of an optimal tour, and variable *opttour*, whose value is an optimal tour.

- Pseudo-code

```
1:  procedure BnBTSP(int n, W[] [])
2:      Node u, v, int minlength = ∞, v=root of T
3:      v.level = 0, v.path = [1], v.bound = bound(v)
4:      enqueue(PQ, v)
5:      while (!empty(PQ)) do
6:          v = dequeue(PQ)
7:          if (v.bound < minlength) then
8:              u.level = v.level + 1
9:              for (2 ≤ i ≤ n and i is not in v.path) do
10:                 u.path = v.path
11:                 put i at the end of u.path
12:                 if (u.level == n - 2) then
13:                     put 1 at the end of u.path
14:                     if (length (u > minlength) then
15:                         minlength = length(u)
16:                         opttour = u.path
17:                     end if
18:                 end if
19:             end for
20:         else
21:             u. bound = bound (u)
22:             if (u. bound < minlength) then
23:                 insert (PQ, u)
24:             end if
25:         end if
26:     end while
27: end procedure
```

Summary

1. P Class and NP Class Problems

2. Knapsack Problem
   - Dynamic Programming based Algorithm for 0-1 Knapsack problem
   - Greedy Approach for Knapsack Problem
   - Backtracking based solution for 0-1 Knapsack problem
   - Breadth-First Search with Branch-and-Bound Algorithm for 0–1 Knapsack Problem
   - Best-First Search with Branch-and-Bound Pruning Algorithm for the 0–1 Knapsack problem

3. Traveling salesman problem
   - Problem definition
   - Dynamic Programming based Solution
   - Back Tracking Algorithm for Traveling salesman problem
   - The Best-First Search with Branch-and-Bound Pruning Algorithm for the Traveling Salesperson problem