Lecture 6.1

- Dr. Anurag Sharma

Dynamic Programming

About the topic

- Dynamic programming is similar to the divide-and-conquer approach in that an instance of a problem is divided into smaller instances.
- In dynamic programming, we solve the small instances first, store the results, and look them up when we need them instead of recomputing them.

Cont.

- Dynamic programming is a bottom-up approach since the solution is constructed from the bottom up in the array.
- There are two steps in the development of this approach.
  - Establish the recursive property that gives the solution to an instance of the problem.
  - Solve an instance of the problem in a bottom-up fashion by solving smaller instances first.

Example

- The Binomial coefficient

$$\binom{n}{k} = \frac{n!}{k!\,(n-k)!}$$

- We cannot compute the binomial coefficient directly from this definition because $n!$ is very large, even for moderate values of n.
- Solution?
  - Eliminate the need to compute $n!$ or $k!$ by using the recursive property.

Recursive binomial coefficient

- $\binom{n}{k} = \begin{cases} \binom{n-1}{k-1} + \binom{n-1}{k}, & 0 < k < n \\ 1\,, & k = 0 \text{ or } k = n \end{cases}$
- Problem solved?
- Same instances are being solved in each recursion.
- Solve $\binom{4}{2}$
- Remember Fibonacci (and worst case complexities)?
  - It is always inefficient when an instance is divided into almost as large as original instance using D&C approach.

DP version of Binomial Coefficient

- These kinds of problems can be developed in a more efficient way using dynamic programming.
  - Establish the recursive property that gives the solution to an instance of the problem.

$$B[i][j] = \begin{cases} B[i-1][j-1] + B[i-1][j], & 0 < j < i \\ 1\,, & j = 0 \text{ or } j = i \end{cases}$$

  - Solve an instance of the problem in a bottom-up fashion by solving smaller instances first.
    - How? See next slide.

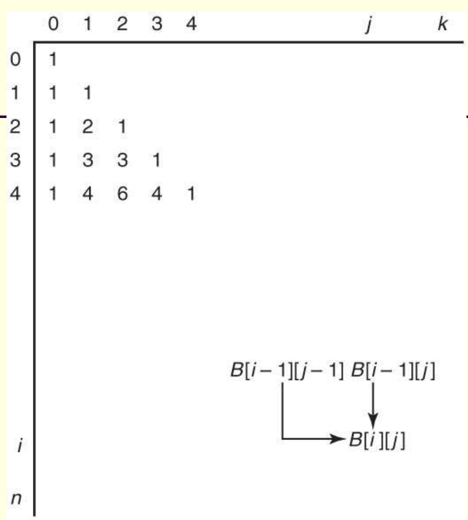


Figure 3.1: The array B used used to compute the binomial coefficient.

Algorithm for Binomial Coefficient

▶ **Algorithm 3.2** **Binomial Coefficient Using Dynamic Programming**

Problem: Compute the binomial coefficient.

Inputs: nonnegative integers n and k, where $k \leq n$.

Outputs: $bin2$, the binomial coefficient $\binom{n}{k}$.

```
int bin2 (int n, int k)
{
  index i, j;
  int B[0..n][0..k];

  for (i = 0; i <= n; i++)
     for (j = 0; j <= minimum(i, k); j++)
        if (j == 0 || j == i)
           B[i][j] = 1;
        else
           B[i][j] = B[i-1][j-1] + B[i-1][j];
  return B[n][k];
}
```

What is T(n) for binomial coeff.?

- Look at the algorithm and analyze number of computations needed.
- The inner loop would require minimum of following values
- $T(n) = min(1,k) + min(2,k) + \cdots + min(k,k) + min(k+1,k) + \cdots + min(n,k)$
- $T(n) = 1 + 2 + \cdots + k + \sum_{i=1}^{n-k+1} k$
- $T(n) = k\frac{(k+1)}{2} + k(n-k+1)$
- $T(n) = \frac{1}{2}k^2 + \frac{1}{2}k + nk - k^2 + k$
- $T(n) = nk - \frac{1}{2}k^2 + \frac{3}{2}k$
- Since, $nk - \frac{1}{2}k^2 + \frac{3}{2}k \leq nk + 2k^2 \leq nk$
- ∴ Big O order would be $O(nk)$

Example – 2

- Floyd's Algorithm for Shortest Path
  - To understand this let us first review graph theory.

Graph Theory

- In a pictorial representation of a graph, circles represent **vertices**, and a line from one circle to another represents an **edge** (sometimes also called an arc).
- If each edge has a direction associated with it, the graph is called a **directed graph**, or **digraph**.
- If the edges have values associated with them, the values are called **weights**, and the graph is called a **weighted graph**.

Cont.

- A **path** is a sequence of adjacent vertices in a graph, while a simple-path is a path but with distinct vertices (that is you can not pass through the same vertex twice).
- A **cycle** is a simple path with three or more vertices such that the last is adjacent to the first. A graph is said to be **acyclic** if it has no cycles and cyclic if it has one or more cycles.
- A **path** is called **simple** if it never passes through the same vertex twice. The **length** of a path in a weighted graph is the sum of the weights on the path. In an unweighted graph, it is the number of edges in the path.

A weighted, directed graph

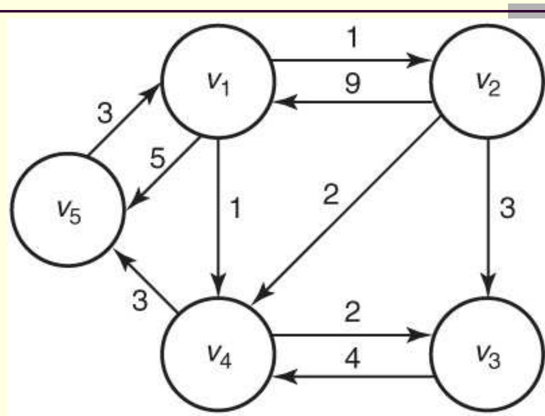


Figure 3.2: A weighted, directed graph.

Shortest Path Problem

- A problem that has many applications is finding the **shortest path** from each vertex to all other vertices
- Examples: Google map, telecommunication, & networking, Airline flight times etc.
- A shortest path must be a simple path
- The Shortest Paths problem is an **optimization problem**

Optimization Problem

- There can be more than one candidate solution to an instance of an optimization problem.
- Each candidate solution has a value associated with it, and a solution to the instance is any candidate solution that has an optimal value.
- Depending on the problem, the optimal value is either the maximum or minimum of these lengths.

Find shortest path

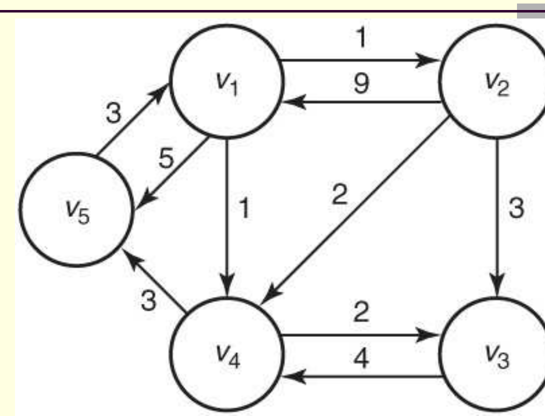


Figure 3.2: A weighted, directed graph.

	1	2	3	4	5
1	0	1	∞	1	5
2	9	0	3	2	∞
3	∞	∞	0	4	∞
4	∞	∞	2	0	3
5	3	∞	∞	∞	0

W

	1	2	3	4	5
1	0	1	3	1	4
2	8	0	3	2	5
3	10	11	0	4	7
4	6	7	2	0	3
5	3	4	6	4	0

D

Figure 3.3: W represents the graph in figure 3.2 and D contains the lengths of the shortest paths. Our algorithm for the Shortest Paths problem computes the values in D from those in W.

Some examples

We will calculate some exemplary values of $D^{(k)}[i][j]$ for the graph in Figure 3.2.

$$D^{(0)}[2][5] = length[v_2, v_5] = \infty.$$

$$D^{(1)}[2][5] = minimum(length[v_2, v_5], length[v_2, v_1, v_5]) = minimum(\infty, 14) = 14.$$

$D^{(2)}[2][5] = D^{(1)}[2][5] = 14.$ {For any graph these are equal because a} {shortest path starting at v_2 cannot pass } {through v_2.}

$D^{(3)}[2][5] = D^{(2)}[2][5] = 14.$ {For this graph these are equal because} {including v_3 yields no new paths} {from v_2 to v_5.}

Shortest path formulation

- **Case 1:** At least one shortest path from v_i to v_j using only vertices in $\{v_1, v_2, ..., v_{k-1}\}$ as intermediate vertices;
  - Thus, $D^k[i][j] = D^{k-1}[i][j]$
- **Case 2:** All shortest paths from v_i to v_j using only vertices in $\{v_1, v_2, ..., v_k\}$ as intermediate vertices;
  - Thus, $D^k[i][j] = D^{k-1}[i][k] + D^{k-1}[k][j]$

"The principle of optimality states that an optimal solution to an instance of a problem always contains optimal solutions to all sub-instances."

Therefore:

$$D^k[i][j] = minimum\,(D^{k-1}[i][j], D^{k-1}[i][k] + D^{k-1}[k][j])$$

It means, either the shortest path goes through k or without k.

Cont.

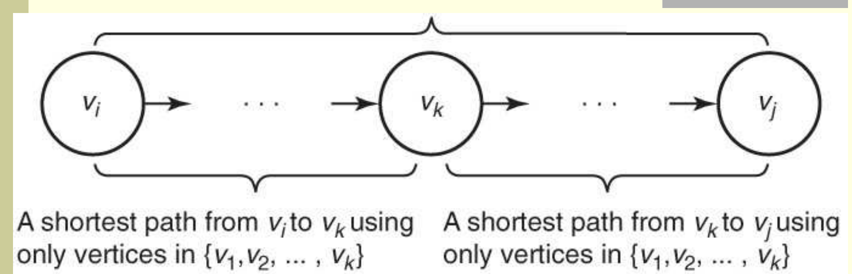


Figure 3.4: The shortest path uses vk

Floyd's Algorithm for Shortest Path

Input: n — number of vertices
a — adjacency matrix
Output: Transformed a that contains the shortest path lengths

```
for k ← 0 to n − 1
    for i ← 0 to n − 1
        for j ← 0 to n − 1
            a[i, j] ← min(a[i, j], a[i, k] + a[k, j])
        endfor
    endfor
endfor
```


Time complexity of FA?

- $T(n) = ?$
- $T(n) = n * n * n = n^3$
- What would be $T(n)$ of D&C version of Floyd's algorithm? Better or worse?

Wait! Where is the shortest path?

```
void floyd2 (int n,
             const number W[][],
                   number D[][],
                   index  P[][])
{
  index, i, j, k;

  for (i = 1; i <= n; i++)
    for (j = 1; j <= n; j++)
      P[i][j] = 0;
  D = W;
  for (k = 1; k <= n; k++)
    for (i = 1; i <= n; i++)
      for (j = 1; j <= n; j++)
        if (D[i][k] + D[k][j] < D[i][j]){
          P[i][j] = k;
          D[i][j] = D[i][k] + D[k][j];
        }
}
```

	1	2	3	4	5
1	0	0	4	0	4
2	5	0	0	0	4
3	5	5	0	0	4
4	5	5	0	0	0
5	0	1	4	1	0

What is the shortest path from 5 to 3?

- Look at the path table:
- 5 – 3
- 5 – (4) – 3 ($v_k = 4$)
- 5 – (1) – 4 – 3
- => 3+1+2 = 6

Lecture 6.2

- Dr. Anurag Sharma

Dynamic Programming (TSP)

Travelling Salesman 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 (TSP)**.

TSP

- An instance of this problem can be represented by a weighted graph, in which each vertex represents a city.
- A tour in a directed graph is a path from a vertex to itself that passes through each of the other vertices only once.
- An optimal tour in a weighted, directed graph is such a path of minimum length.
- No one has ever found an algorithm for the Traveling Salesperson problem whose worst-case time complexity is better than exponential. Yet, no one has ever proved that the algorithm is not possible.

Principle of optimality

- "The principle of optimality states that an optimal solution to an instance of a problem always contains optimal solutions to all sub-instances."
- In case of shortest path problem if v_k is a vertex on an optimal path from v_i to v_j, then the subpaths* from v_i to v_k and from v_k to v_j must also be optimal. [*with all same nodes.]

Solve TSP?

- The TSP is to find an optimal tour when at least one tour exists
- One method is to apply the **Brute-force** approach – i.e. start with one city and consider each remaining city in turn, but this will yield a factorial time!
- However, **dynamic programming** can also be applied to this problem.
  - Use DP paradigm and principle of optimality to divide the problem using bottom up approach.

DP for TSP

- 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
- Let

  W = adjacency matrix for a graph

  V = set of all the vertices

  A = a subset of V
- $D[vi]\ [A]$ = length of shortest path from v_i to v_1 passing through each vertex in A exactly once

Cont.

- $V - \{v_1, v_j\}$ contains all vertices except $v1$ and vj and since principle of optimality applies, length of an optimal tour =

$$\min_{2 \le j \le n} (W[1][\,j\,] + D[v_j][V - \{v_1, v_j\}])$$

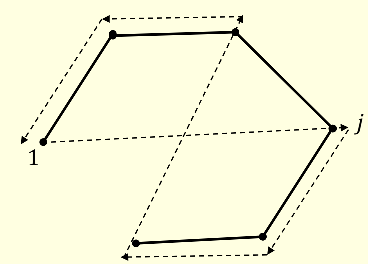
- In general for $i \neq 1$ and v_i not in A, $D[v_i][A]\ =$

$$\min_{j:\, v_j \in A} (W[i][j]\ +\ D[v_j][A - \{v_j\}], \text{ if } A \neq \emptyset$$

- $D[v_i][\emptyset] = W[i][1]$

Cont.

$$\min_{2 \le j \le n} (W[1][\,j\,] + D[v_j][V - \{v_1, v_j\}])$$



Example

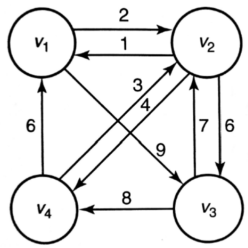


Figure 3.16 ● The optimal tour is $[v_1, v_3, v_4, v_2, v_1]$.

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

Figure 3.17 ● The adjacency matrix representation

Cont.

- $D[v_2][\phi] = 1$
- $D[v_3][\phi] = \infty$
- $D[v_4][\phi] = 6$

- $D[v_3][\{v_2\}] = 7 + 1 = 8$
- $D[v_4][\{v_2\}] = 4$

- $D[v_2][\{v_3\}] = 6 + \infty = \infty\ [v_2 - v_1 \Rightarrow v_2 - v_3 - v_1]$
- $D[v_4][\{v_3\}] = \infty$

- $D[v_2][\{v_4\}] = 4 + 6 = 10$
- $D[v_3][\{v_4\}] = 8 + 6 = 14\ \ [v_3 - v_1 \Rightarrow v_3 - v_4 - v_1]$

Cont.

- $D[v_4][\{v_2, v_3\}] = \min_{j:\, v_j \in \{v_2, v_3\}} \left(W[4][j] + D[v_j][\{v_2, v_3\} - \{v_j\}]\right)$

$= \min_{j:\, v_j \in \{v_2, v_3\}} \left((W[4][2] + D[v_2][\{v_2, v_3\} - \{v_2\}]), (W[4][3] + D[v_3][\{v_2, v_3\} - \{v_3\}])\right)$

$= \min_{j:\, v_j \in \{v_2, v_3\}} \left((W[4][2] + D[v_2][\{v_3\}]), (W[4][3] + D[v_3][\{v_2\}])\right)$

$= \min(3 + \infty, \infty + 8) = \infty$

- And, $D[v_1][\{v_2, v_3, v_4\}] = \min_{j:\, v_j \in \{v_2, v_3, v_4\}} \left(W[1][j] + D[v_j][\{v_2, v_3, v_4\} - \{v_j\}]\right)$

$$= \min_{j:\, v_j \in \{v_2, v_3, v_4\}} \begin{pmatrix} W[1][2] + D[v_2][\{v_3, v_4\}, \\ W[1][3] + D[v_3][\{v_2, v_4\}, \\ W[1][4] + D[v_4][\{v_2, v_3\} \end{pmatrix}$$

$= \min(2 + 20, 9 + 12, \infty + \infty) = 21$

- ~~What is $D[v_3][\{v_2, v_4\}]$ pictorially?~~

Algorithm

```
void travel (int n,
             const number W[][],
             index P[][],
             number& minlength)
{
  index i, j, k;
  number D[1..n][subset of V - {v1}];

  for (i = 2; i <= n; i++)
    D[i][∅] = W[i][1];
  for (k = 1; k <= n - 2; k++)
    for (all subsets A ⊆ V - {v1} containing k vertices)
      for (i such that i ≠ 1 and vi is not in A){
        D[i][A] = minimum (W[i][j] + D[j][A - {vj}]);
                  j:vj ∈ A
        P[i][A] = value of j that gave the minimum;
      }
  D[1][V - {v1}] = minimum (W[1][j] + D[j][V - {v1, vj}]);
                   2 ≤ j ≤ n
  P[1][V - {v1}] = value of j that gave the minimum;
  minlength = D[1][V - {v1}];
}
```

Time complexity?

- $T(n) = ?$
- According to the for loops in the algorithm:
- First loop: $k = 1 : \sim n$ i.e., n times
- Second loop: $\binom{n}{k}$ for every k
- Third loop: $n - k \approx n$
- Roughly, $T(n) = (n) \sum_{i=1}^{n} k \binom{n}{k}$
- $T(n) = (n) n 2^n = n^2 2^n$
- Big O order is $O(2^n)$ (better than $n!$)

What is $\sum_{k=0}^{n} k \binom{n}{k}$? (from https://math.stackexchange.com)

Since the binomial coefficients have the $n-k$ symmetry, we can put

$$\sum_{k=0}^{n} (n-k) \binom{n}{n-k}$$

thus

$$S_n = \sum_{k=0}^{n} k \binom{n}{k} = \sum_{k=0}^{n} (n-k) \binom{n}{n-k}$$

But the RHS is

$$n \sum_{k=0}^{n} \binom{n}{n-k} - \sum_{k=0}^{n} k \binom{n}{n-k}$$

Now

$$S_n = n \sum_{k=0}^{n} \binom{n}{k} - \sum_{k=0}^{n} k \binom{n}{k}$$

or

$$S_n = n \sum_{k=0}^{n} \binom{n}{k} - S_n$$

$$S_n = n2^n - S_n$$

$2S_n = n2^n$

$S_n = n2^{n-1}$

Lecture 7.1

- Dr. Anurag Sharma

Meta-heuristic Algorithms

Course Learning Outcomes

- 2. Assess the suitability of different algorithms for solving a given problem
- 3. Solve computationally difficult real world problems using appropriate algorithmic techniques.

Algorithm Design Paradigms

So far we have seen various kinds of algorithm design approaches. No design is perfect and they come with their pros and cons.

- Brute force
- Divide and Conquer
- Dynamic Programming
- Greedy Approach (not yet studied)

Solve complex problems like TSP

- No one has ever found an algorithm for the Traveling Salesperson problem whose worst-case time complexity is better than exponential. Yet, no one has ever proved that the algorithm is not possible.
- Worst case Time complexity with various approaches:
  - Brute Force – factorial
  - Dynamic Programming – exponential
  - Divide & Conquer?

Do we have a satisfactory solution?

- There are no ‘deterministic method’ that can solve TSP better than exponential.
- That means we cannot have solutions for large problems in a reasonable time frame.
- But we have seen solutions can be obtained very quickly. Such as GPS tracking, shortest route etc.

Artificial Intelligence Methods

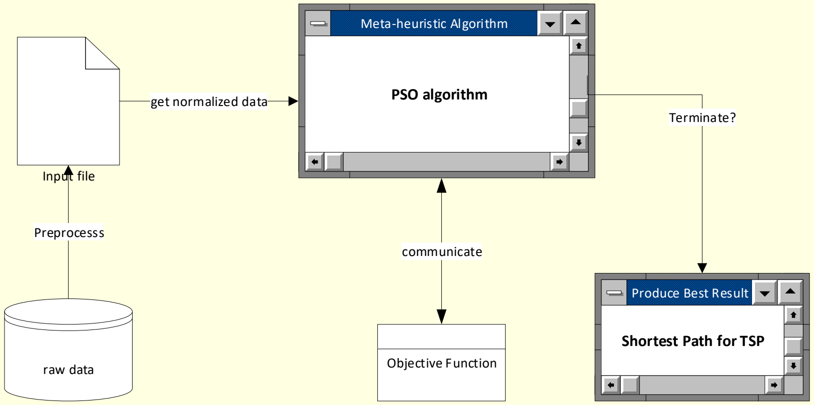
- These complex optimization problems can be solved by meta-heuristic algorithms.
- Like:
  - Genetic Algorithm
  - Particle Swarm Optimization
  - Ant Colony Optimization
  - Etc.

Particle Swarm Optimization (PSO)

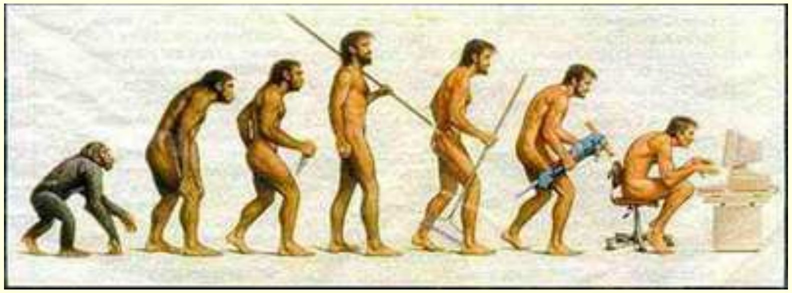
PSO Algorithm

```
//PSO Algorithm:
//Input: problem in matrix form;
//output: optimum solution
read(); // read the matrix from the file;
parameter_setting();
for(int i=0; i<max_iter; i++)
      for(int j=0; j<swarm_size; j++)
            best_neighbor = get_best_neighbor(particle[j]);
      if(best_neighbor<global_best)
            global_best = best_neighbor;
      End if;
      extra_best = move_towards(particle[j], best_neighbor);
      If(global_best<extra_best)
            solution = global_best;
      End if;
      else
            solution = extra_best;
      End for_loop;
End for_loop;
```

Framework



Evolutionary Algorithm



Lecture 8.1 & 8.2

- Anurag Sharma & Shymal Chandra

Greedy Algorithms

Greedy Uncle Scrooge

Greedy Algorithms

- Greedy algorithms, like dynamic programming are often used to solve optimization problems.
- A greedy approach arrives at the solution by making a sequence of decisions, each of which simply looks like the best decision at that moment.
- Each choice is the locally optimal choice, hoping to arrive at a globally optimal solution.
- The final solution however may not always be the optimal with the greedy approach.

Cont.

- Each iteration in the greedy algorithm consists of the following steps:

1. A selection procedure – chooses the next item according to some greedy criterion
2. A feasibility check – determines whether the decision is locally optimal, i.e. whether the chosen item can lead to a solution.
3. A solution check – determines whether the solution is reached or not.

Examples

- Problem: To minimize the number of coins while giving some change
- Example 1: From available coins 50c, 20c, 10c, 5c, 2c, 1c, give a change of 75c

1. Selection – pick coin with highest value
2. Feasibility – check if change value ≤ 75c
3. Solution – check if change value reached
   Greedy Solution: 50c + 20c + 5c (optimal)

Cont.

- Example 2: Assuming you only have coins 12c, 10c, 5c, 1c, how would you give a change of 16c?
- Greedy Solution: 12c + 1c + 1c + 1c + 1c (not optimal)
- Optimal solution would have been 10c + 5c + 1c

Huffman Code

- Huffman coding is an efficient method of compressing data without losing information.
- It uses a particular type of optimal prefix code for data compression.

Data Compression

- Even though capacity of secondary storage devices keeps getting larger and their cost keeps getting smaller, the devices continue to fill up due to increased storage demands.
- Thus, given a data file, it would be desirable to find a way to store the file as efficiently as possible.
- The problem of data compression is to find an efficient method for encoding a data file.

Encoding Data

- A common way to represent a file is to use a binary code – each character represented by a unique binary string, called the **codeword**
- A fixed-length binary code represents each character using the same number of bits
- Example: Using code A: 00, B: 01, C: 11 and given a file ABABCBBBC, the encoding is 000100011101010111
- Can you obtain more efficient encoding?

Cont.

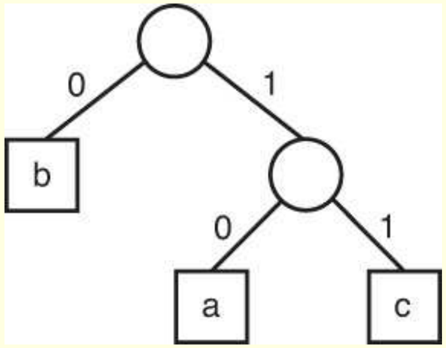
- A variable-length binary code can represent different characters using different numbers of bits
- Example: Using code A: 10, B: 0, C: 11 and given a file ABABCBBBC, the encoding is 1001001100011
- This encoding uses lesser bits than the previous

Prefix Codes

- A particular type of variable-length code is a prefix code.
- In prefix code, no codeword for one character constitutes the beginning of the codeword for another character, e.g. If codeword for A is 01, then another character B cannot have codeword as 011
- Every prefix code can be represented by a binary tree whose leaves are the characters to be encoded

Cont.

- Binary Tree for code A: 10, B: 0, C: 11



Parsing prefix codes

- Start at the first bit on ~~the left in the file and~~ the root of the tree, sequence through the bits, and go left or right down the tree depending on whether a 0 or 1 is encountered
- When a leaf is reached, obtain the character at that leaf.
- Return to the root and repeat the procedure starting with the next bit (or branch) ~~in sequence in the file~~

Generating prefix code

- Huffman developed a greedy algorithm that produces an optimal binary character code by constructing a binary tree corresponding to an optimal code
- A code produced by this algorithm is called a Huffman code

Huffman's Algorithm

- The basic approach is to get the frequency of each character used in a given a text file, and store these in a priority queue.
- In a priority queue, the element with the highest priority is the character with the lowest frequency in the file.
- The priority queue can be implemented as a linked list, but more efficiently as a heap.
-

Character	Frequency
A	15
B	5
C	12
D	17
E	10
F	25

Cont.

n = number of characters in the file;

Arrange n pointers to nodetype records in a priority queue PQ as follows: For each pointer p in PQ

p-> $symbol$ = a distinct character in the file;
p-> $frequency$ = the frequency of that character in the file;
p-> $left$ = p-> $right$ = NULL;

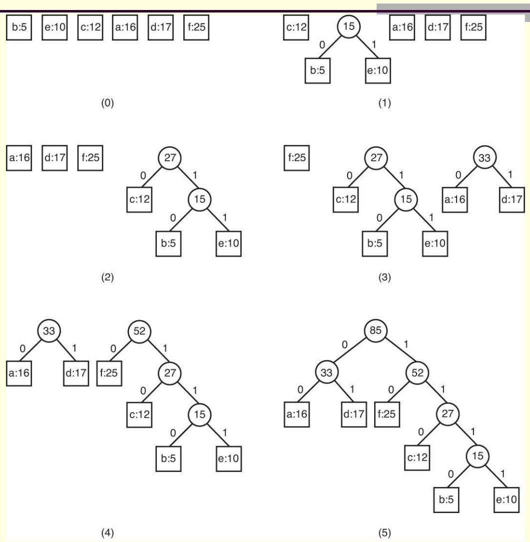
```
for (i=1; i <= n-1; i++) {   // There is no solution check; rather,
    remove(PQ, p);            // solution is obtained when i = n - 1.
    remove(PQ, q);            // Selection procedure.
    r = new nodetype;         // There is no feasibility check.
    r->left = p;
    r->right = q;
    r->frequency = p->frequency + q->frequency;
    insert(PQ, r);
}
remove(PQ, r);
return r;
```

Cont.

- Every iteration takes the two most priority items and join them together in left and right branch.
- The aim is to have lesser bits for most common characters and more bits for less common characters so as to minimize total number of bits.

Huffman tree construction

Character	Frequency
A	15
B	5
C	12
D	17
E	10
F	25



Final verdict

- Finally, parsing through the tree would result in the following optimal codes for each character:

Character	Frequency	Code
b	5	1110
e	10	1111
c	12	110
a	16	00
d	17	01
f	25	10

- Why not just 3 digits for each character? Try it!

Lecture 9.1

- Anurag Sharma & Shymal Chandra

Minimum Spanning Trees with Greedy Algorithms

Spanning Trees

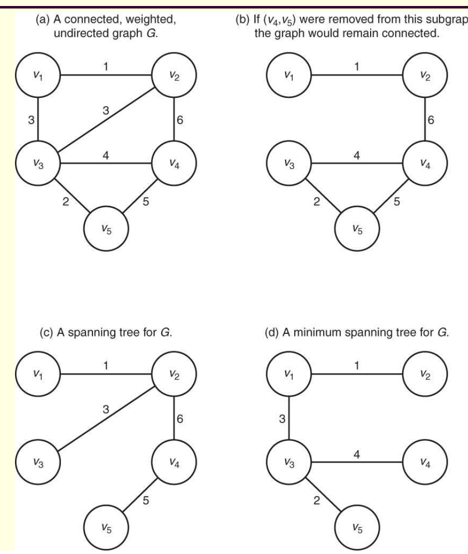
A review of graph theory:

- A graph is **undirected** when its edges do not have direction. An undirected graph is called **connected** if there is a path between every pair of vertices.
- An undirected graph with no simple cycles is called **acyclic**. A tree is an acyclic, connected, undirected graph.
- A **spanning tree** for a given graph is a connected subgraph that contains all the vertices of the given graph and is a tree.
- A spanning tree can be defined as an undirected graph G = , where V is the set of vertices and E is the set of edges

Minimum Spanning Trees

- If G is a weighted graph, the spanning tree will have a total weight.
- A graph may have different spanning trees, but not every spanning tree has the minimum weight
- A spanning tree with minimum weight is called minimum spanning tree
- The problem of finding the minimum spanning tree in an undirected, weighted, connected graph has many applications such as Google Maps, networking in telecommunications, operations research etc.
- We can represent such graphs using an adjacency matrix
- We will look at two algorithms (Prim's and Kruskal's) which produce minimum spanning trees

Minimum Spanning Tree Formation



Prim's Algorithm

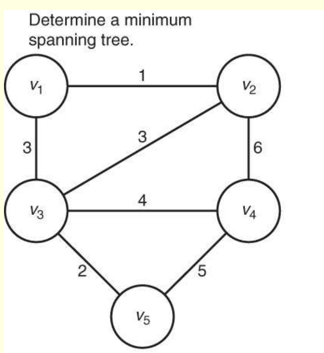
- Prim's algorithm starts with an empty subset of edges F and a subset of vertices Y initialized to contain an arbitrary vertex - we can initialize Y to {v1}
- The set of all vertices is the set V
- A vertex nearest to Y is a vertex in V – Y that is connected to a vertex in Y by an edge of minimum weight
- The vertex that is nearest to Y is added to Y and the edge is added to F
- Repeat these steps until all vertices have been included in the set Y

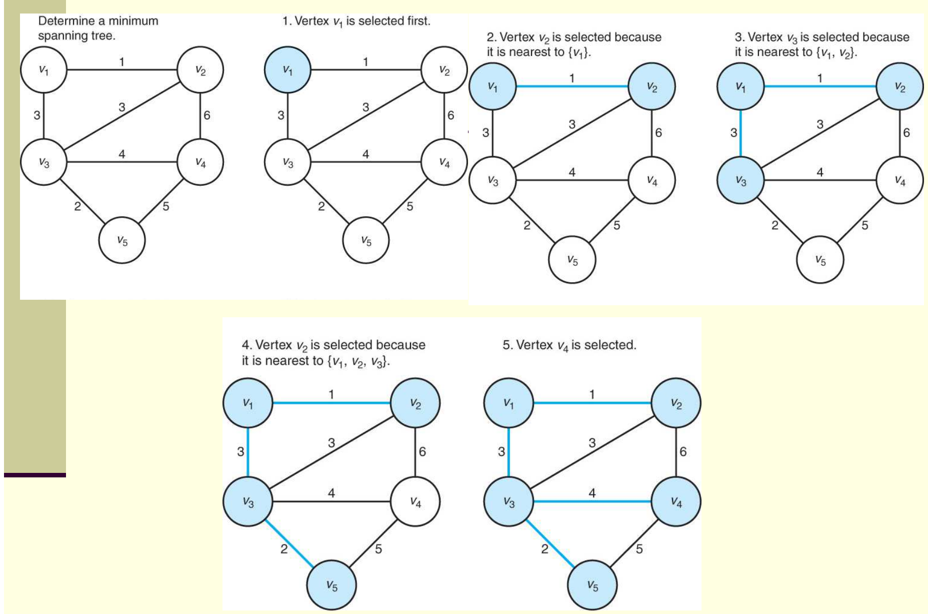
Pseudocode for Prim's Algorithm

```
F = ∅;                                       // Initialize set of edges
                                             // to empty.
Y = {v1};                                    // Initialize set of vertices to
                                             // contain only the first one.
while (the instance is not solved){

  select a vertex in V − Y that is           // selection procedure and
  nearest to Y;                              // feasibility check

  add the vertex to Y;
  add the edge to F;

  if (Y == V)                                // solution check
     the instance is solved;
}
```


Example

- Find the minimum spanning tree for the following graph:



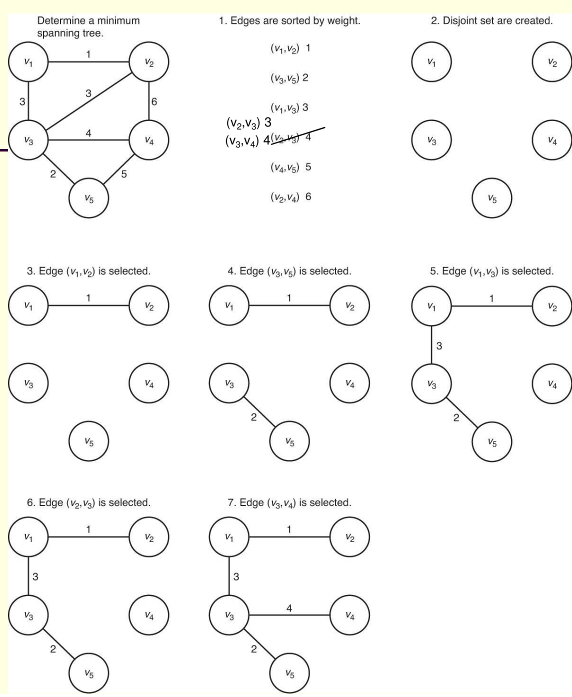


Kruskal's Algorithm

- Kruskal's algorithm starts by creating disjoint subsets of V – one for each vertex and containing only that vertex, and an empty set of edges F
- It then inspects the edges according to increasing weight
- If an edge connects two vertices in disjoint subsets, the edge is added to F and the subsets are merged into one set
- This process is repeated until all the subsets are merged into one set and the final set F gives the minimum spanning tree

Pseudocode – Kruskal's algorithm

```
F = Ø;                                             // Initialize set of
                                                   // edges to empty.
create disjoint subsets of V, one for each
vertex and containing only that vertex;

sort the edges in E in nondecreasing order;

while (the instance is not solved){

  select next edge;                                // selection procedure

  if (the edge connects two vertices in            // feasibility check
                       disjoint subsets){
      merge the subsets;
      add the edge to F;
  }
  if (all the subsets are merged)                  // solution check
     the instance is solved;
}
```

Lecture 9.2

- Anurag Sharma & Shymal Chandra

Greedy Algorithms: Dijkistra's Algorithm

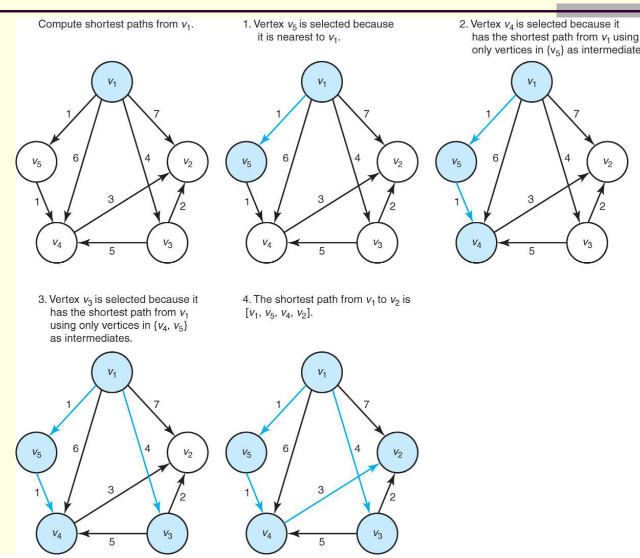
Single-source shortest path

- Problem: Given a graph $G = \langle E, V \rangle$, find the shortest path from a given source vertex $s \in V$ to every vertex $v \in V$
- A greedy algorithm to solve the above problem is the Dijkstra's Algorithm for Single-Source Shortest Path
- Dijkstra's algorithm is similar to Prim's algorithm for the Minimum Spanning Tree.

Dijkstra's Algorithm

```
Y = {v1};
F = Ø;

while (the instance is not solved){

  select a vertex v from V − Y, that has a    // selection
  shortest path from v1, using only vertices  // procedure and
  in Y as intermediates;                      // feasibility check

  add the new vertex v to Y;
  add the edge (on the shortest path) that touches v to F;

  if (Y == V)
    the instance is solved;                   // solution check
}
```


Example



How to solve?

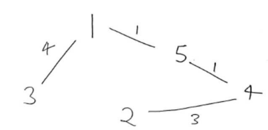
- Make Adjacency matrix

	1	2	3	4	5
1	0	7	4	6	1
2	∞	0	∞	∞	∞
3	∞	2	0	5	∞
4	∞	3	∞	0	∞
5	∞	∞	∞	1	0

Cont.

- Pick min in each row

	1	2	3	4	5
1	—	7^1	4^1	6^1	1^1
5	—	$\infty, 7$ 7^1	4^1	$1+1$ 2^5	—
4	—	$3+2 = 5^4$	4^1	—	—
3	—	$2+4 \times$ 5^4	—	—	—
2	—	—	—	—	—



Lecture 10.1

- Anurag Sharma & Shymal Chandra

Greedy Algorithms vs Dynamic Programming

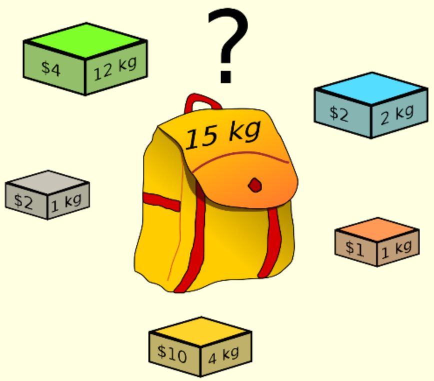
Greedy vs Dynamic Programming

- The greedy approach and dynamic programming are two ways to solve optimization problems
- When a greedy approach solves a problem, the result may be a simpler
- However, it can be difficult to determine whether a greedy algorithm always produces an optimal solution
- We will now look at The Knapsack Problem and try to solve it using both the algorithms!

Smart thief?

The Knapsack Problem

- The Knapsack Problem can be described as follows:
  - A thief breaks into a jewellery store carrying a knapsack wanting to steal items and pack it into his knapsack
  - Given n items S = {item1, item2, …, item n}, each item having a weight w_i and providing a profit p_i , which items should the thief put into his knapsack that has a maximum capacity W in order to obtain the maximum profit?



- The Knapsack problem has two variations: 0-1 Knapsack and Fractional Knapsack

Greedy Approach: The 0-1 Knapsack Problem

- This problem requires a subset A of S to be determined such that
- $maximize \sum_{i \in A} p_i \mid \sum_{i \in A} w_i \leq W$
- Greedy Strategies: Steal (select) items with the largest profit or steal items with the lightest weight etc.
- These however may not be optimal
- Another greedy strategy would be to steal items with the largest profit per unit weight

Example

- 3 items with w = [5, 10, 20], p = [50, 60,140] and W = 30
- Profits per unit = [10, 6, 7]
- Greedy Approach: Select items 1 and 3: Profit is $190, although optimal profit would be $200 (items 2 and 3)
- The problem is that even if items 1 and 3 are selected, there is wastage of space (5 units) in the knapsack (since knapsack is not filled to capacity)
- However, in the 0-1 Knapsack problem, you can either select the whole of an item or none of it – no fractions allowed, so the above problem is expected

Greedy Approach: The Fractional Knapsack Problem

- In a slight variation, the Fractional Knapsack Problem is where the thief does not have to steal all of an item, but rather can take any fraction of the item
- Greedy approach to the fractional knapsack problem yields the optimal solution
- Example: With the same strategy in the previous example (i.e. select items with the highest profit per weight value): Profit = 50 + 140 + (5/10) * 60 = $220 (where you select items 1 and 3 first and take 5/10 of item 2 to avoid any wastage of space in the knapsack) This gives you the optimal profit!

Dynamic Programming Approach: The 0-1 Knapsack Problem

- For a dynamic programming algorithm, the principle of optimality should apply
- Let A be an optimal subset of n items. There are two cases:

1. If A contains item i, the total profit of items in A is equal to $p_i +$ the optimal profit obtained from the first $i - 1$ items, where the total weight cannot exceed $W - w_i$
2. If A does not contain item i, the total profit of items in A is equal to the optimal subset of the first $i - 1$ items

Cont.

- You can create a 2-D array P (whose rows are indexed from 0 to n and columns indexed from 0 to W)
- In general, the two cases discussed on the previous slide can be represented by the following formula:
- $P[i][w] = \begin{cases} \max(P[i-1][w], p_i + P[i-1][w-w_i]), if\ w_i \le w \\ P[i-1][w] \qquad\qquad\qquad\qquad\qquad\quad , if\ w_i > w \end{cases}$
- The maximum profit is given by the value at $P[n][w]$

Lecture 11.1

- Anurag Sharma & Shymal Chandra

Backtracking Algorithms

Backtracking Technique

- Backtracking is used to solve problems in which a sequence of objects is selected from a specified set so that the sequence satisfies some criterion
- Often the goal is to find any feasible solution rather than an optimal solution – example, when solving a maze that could have many possible solutions
- Backtracking is a modified **depth-first search** of a state-space tree
- What is depth-first search? A preorder traversal of a tree!

Cont.

- A **state space tree** of a problem is a tree that contains nodes indicating the object chosen or the direction chosen. A path from the root of the tree to a leaf (node with no children) is a candidate solution
- Backtracking is a procedure whereby, after determining a node can lead to nothing but dead ends, we go back (backtrack) to parent node and search on the next child
- A node is **nonpromising,** if it is determined that it cannot possibly lead to a solution and promising otherwise

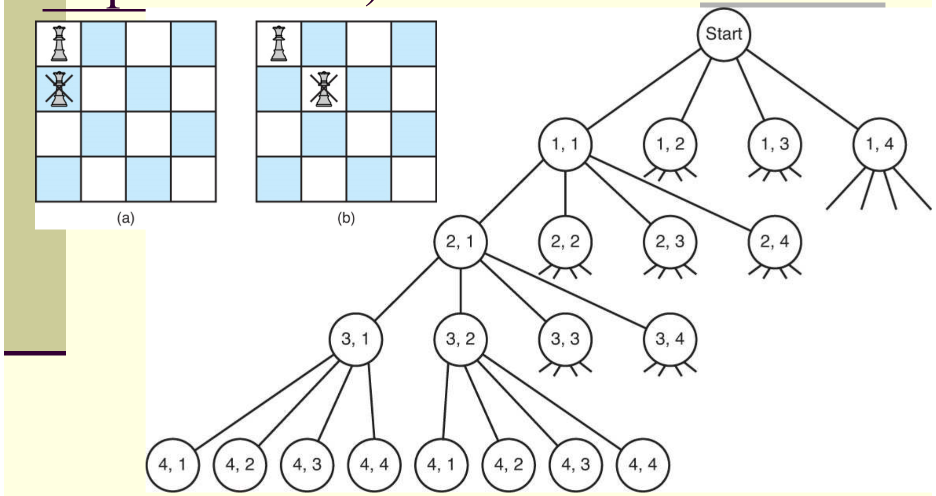
Cont.

- **Pruning** a state space tree is doing a depth-first search and checking whether each node is promising or not; if not promising then backtrack to parent node
- Pruning helps shorten the entire state space tree
- The subtree consisting of the visited nodes is called **pruned state space tree**

Example (The n-queen problem)

- The idea in the n-Queens problem is to place n queens on an n x n chess board, such that none of the queens can attack another queen
- Remember that queens can move horizontally, vertically, or diagonally any distance
- We will illustrate backtracking using n = 4 i.e. placing 4 Queens on a 4 x 4 chess board such that no queen can attack any other

4-queen problem (4x4x4x4=256 possibilities)

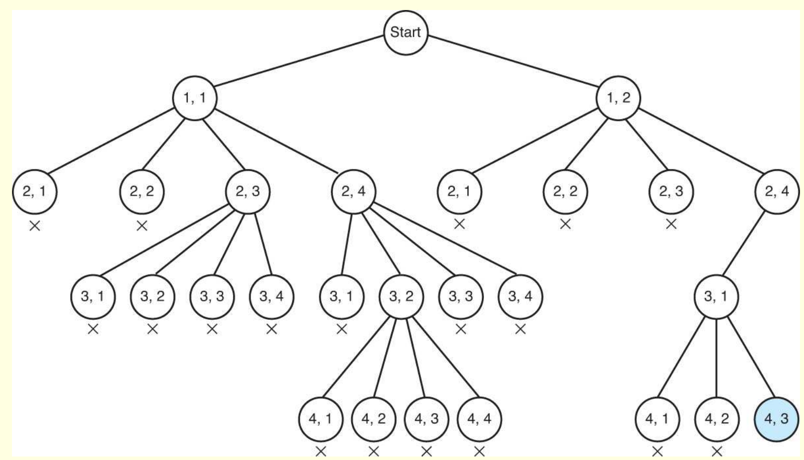


Use backtracking approach

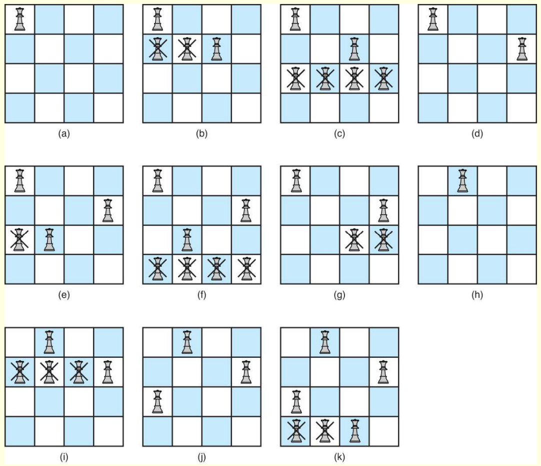
Backtracking is the procedure whereby, after determining that a node can lead to nothing but dead ends, we go back ("backtrack") to the node's parent and proceed with the search on the next child. We call a node ***nonpromising*** if when visiting the node we determine that it cannot possibly lead to a solution. Otherwise, we call it ***promising***. To summarize, backtracking consists of doing a depth-first search of a state space tree, checking whether each node is promising, and, if it is nonpromising, backtracking to the node's parent. This is called ***pruning*** the state space tree, and the subtree consisting of the visited nodes is called the ***pruned state space tree***. A general algorithm for the backtracking approach is as follows:

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

  if (promising(v))
     if (there is a solution at v)
        write the solution;
     else
        for (each child u of v)
           checknode(u);
}
```


4-queen problem (promising solutions)



Solution for 4-queen problem



Efficiency of backtracking

- N-queen problem with backtracking has efficiency of O(n!).

• **Table 5.1** An illustration of how much checking is saved by backtracking in the n-Queens problem *

n	Number of Nodes Checked by Algorithm 1†	Number of Candidate Solutions Checked by Algorithm 2‡	Number of Nodes Checked by Backtracking	Number of Nodes Found Promising by Backtracking
4	341	24	61	17
8	19,173,961	40,320	15,721	2057
12	9.73×10^{12}	4.79×10^{8}	1.01×10^{7}	8.56×10^{5}
14	1.20×10^{16}	8.72×10^{10}	3.78×10^{8}	2.74×10^{7}

Any better solution? [optional]

- https://link.springer.com/chapter/10.1007/978-3-642-35101-3_21

Table 1. Comparative test results on no problem specific information extraction

N	CMA-ES [25]	DE [25]	GA	NSGA II	ICHEA
4	456 NFC (SR = 1.00)	134 NFC (SR = 1.00)	367 NFC (SR = 1.00)	93 NFC (SR= 1.00)	39 NFC (SR = 1.00)
5	656 NFC (SR = 1.00)	254 NFC (SR = 1.00)	750 NFC (SR = 1.00)	217 NFC (SR = 1.00)	37 NFC (SR = 1.00)
6	22,013 NFC (SR = 1.00)	1,11,136 NFC (SR = 0.65)	30,086 NFC (SR = 0.75)	694 NFC (SR = 1.00)	51 NFC (SR = 1.00)
7	9,964 NFC (SR = 1.00)	24,338 NFC (SR = 0.95)	1,400 NFC (SR = 1.00)	2631 NFC (SR = 1.00)	34 NFC (SR = 1.00)
8	84,962 NFC (SR = 1.00)	7,576 NFC (SR = 0.75)	3,786 NFC (SR = 0.80)	1273 NFC (SR = 1.00)	41 NFC (SR = 1.00)
9	133,628 NFC (SR = 1.00)	19,296 NFC (SR = 0.50)	18,333 NFC (SR = 0.80)	27,852 NFC (SR = 1.00)	72 NFC (SR = 1.00)
10	263,572 NFC (SR = 0.95)	286,208 NFC (SR = 0.30)	3,300 NFC (SR = 0.30)	1,737 NFC (SR = 1.00)	83 NFC (SR = 1.00)
11	284,382 NFC (SR = 0.95)	68,255 NFC (SR = 0.10)	15,550 NFC (SR = 0.40)	SR = 0.00	132 NFC (SR = 1.00)
12	295,740 NFC (SR = 0.75)	99,120 NFC (SR = 0.25)	23,000 NFC (SR = 0.70)	SR = 0.00	122 NFC (SR = 1.00)
13	376,631 NFC (SR = 0.85)	95,485 NFC (SR = 0.15)	3,400 NFC (SR = 0.10)	SR = 0.00	293 NFC (SR = 1.00)
14	450,654 NFC (SR = 0.85)	160,475 NFC (SR = 0.10)	47,350 NFC (SR = 0.40)	SR = 0.00	308 NFC (SR = 1.00)
15	627,391 NFC (SR = 0.50)	223,425 NFC (SR = 0.10)	95,625 NFC (SR = 0.40)	SR = 0.00	381 NFC (SR = 1.00)

Hamiltonian Circuits Problem (optional)

- A Hamiltonian circuit (tour) of a graph is a path that starts at a given vertex, visits each vertex in the graph exactly once, and ends at the starting vertex. The problem is to find all the Hamiltonian circuits in a graph
- A state space tree for this problem is as follows: Put the starting vertex at level 0 in the tree; the zeroth vertex on the path. At level 1, create a child node for the root node for each remaining vertex that is adjacent to the first vertex. At each node in level 2, create a child node for each of the adjacent vertices that are not in the path from the root to this vertex, and so on…

Cont.

- In order to backtrack in this state space tree: The i^{th} vertex on the path must be adjacent to the $(i - 1)^{st}$ vertex on the path
- The $(n - 1)^{st}$ vertex must be adjacent to the 0th vertex
- The i^{th} vertex cannot be one of the first i - 1 vertices