RAJIV GANDHI PROUDYOGIKI VISHWAVIDYALAYA, BHOPAL New Scheme
Based On AICTE Flexible Curricula
COMPUTER SCIENCE AND ENGINEERING IV-Semester

CS-402 Analysis Design of Algorithm

Topic Covered
Dynamic Programming, 0/1 Knapsack Problem, Branch and Bound, N Queen problem

Introduction:

Dynamic programming is a name, coined by Richard Bellman in 1955. Dynamic programming, as greedy method, is a powerful algorithm design technique that can be used when the solution to the problem may be viewed as the result of a sequence of decisions. In the greedy method we make irrevocable decisions one at a time, using a greedy criterion. However, in dynamic programming we examine the decision sequence to see whether an optimal decision sequence contains optimal decision subsequence.

When optimal decision sequences contain optimal decision subsequences, we can establish recurrence equations, called dynamic-programming recurrence equations, that enable us to solve the problem in an efficient way.

Dynamic programming is based on the principle of optimality (also coined by Bellman). The principle of optimality states that no matter whatever the initial state and initial decision are, the remaining decision sequence must constitute an optimal decision sequence with regard to the state resulting from the first decision. The principle implies that an optimal decision sequence is comprised of optimal decision subsequences. Since the principle of optimality may not hold for some formulations of some problems, it is necessary to verify that it does hold for the problem being solved. Dynamic programming cannot be applied when this principle does not hold.

The steps in a dynamic programming solution are:

- Verify that the principle of optimality holds
- Set up the dynamic-programming recurrence equations
- Solve the dynamic-programming recurrence equations for the value of the optimal solution.
- Perform a trace back step in which the solution itself is constructed.

Dynamic programming differs from the greedy method since the greedy method produces only one feasible solution, which may or may not be optimal, while dynamic programming produces all possible sub-problems at most once, one of which

guaranteed to be optimal. Optimal solutions to sub-problems are retained in a table, thereby avoiding the work of recomputing the answer every time a sub-problem is encountered

The divide and conquer principle solve a large problem, by breaking it up into smaller problems which can be solved independently. In dynamic programming this principle is carried to an extreme: when we don't know exactly which smaller problems to solve, we simply solve them all, then store the answers away in a table to be used later in solving larger problems. Care is to be taken to avoid recomputing previously computed values, otherwise the recursive program will have prohibitive complexity. In some cases, the solution can be improved and in other cases, the dynamic programming technique is the best approach.

Dynamic programming and Greedy strategy both are used for solving optimization problem. Optimization problem are those which require either minimum or maximum result. In greedy, we have predefined method like Prims, Kruskals but in dynamic programming to find all possible solution and pickup the best solution, but its time consuming process.

In greedy method decision takes in one time and follow the procedure. In dynamic programming in every stage to take the decision. Dynamic programming based on principal of optimality, problem can be solved by sequence of decision.

0/1 Knapsack problem:

The Knapsack problem is an optimization problem, where constrained is the number of object that can be placed inside a fixed size knapsack. Given a set of object with specific weight and profit(value), the aim is to get as much profit into the knapsack as possible given the weight.

The knapsack problem is an example of a combinational optimization problem, a topic in mathematics and computer science about finding the optimal object among a set of objects. In 0/1 knapsack problem objects are not divisible i.e. you can not break an object, you either take an object or not, like objects-fan, light, computer, washing machine etc.

Example: Solve the following instance of 0/1 knapsack problem using dynamic programming . (RGPV DEC 2016)

Object:	1	2	3	4
Weight:	4	7	5	3
Profit:	$40	$42	$25	$12

The capacity of Knapsack is 10

Solution:

In this problem we have maximum capacity is 10.
problem solved by tabulation method for this we have to use following formula

V[i,w]= Max[V(i-1,w), v[(i-1,w-w(i))+p(i)]]

Where
i- is the index of object(row number)
w- weight of object
p- profit of object

In the table arrange the weight in increasing order and their profit and object.

		W---->	1	2	3	4	5	6	7	8	9	10
Pi	Wi	Object										
12	3	1	0	0	12	12	12	12	12	12	12	12
40	4	2	0	0	12	40	40	40	52	52	52	52
25	5	3	0	0	12	40	40	40	52	52	65	65
42	7	4	0	0	12	40	40	40	52	52	65	65

*In first row weight of first object is 3 and profit is 12. In this row we have only one object so maximum profit is 12 filled up to last column.

*In second row weight is 4 and profit is 40. but in this row we have two object first object weight is 3 and second object weight is 4. So total weight is 7 and total profit is 52, filled from 7^{th} column to last column.

*Remaining entries filled by above formula:

Suppose we have to find value of V[3,7]
V[i,w]= Max[V(i-1,w), v[(i-1,w-w(i))+p(i)]]
V[3,7]=Max[V(2,7),v[(2,7-5)+25]]
V[3,7]= Max[52,V[(2,2+25)]]
V[3,7]= Max[52,[0+25]]
V[3,7]= Max[52,25]
So Maximum value of V[3,7] is 52. Similarly find the remaining values.

Final solution :
Highest profit is 65
How to know which object should be included (from Bottom to top in the table)
*Highest profit is 65. 65 present in last and second last row. So second last row object (3^{rd} object) is included. Profit of this object is 25 so remaining profit is 65-25=40
*40 is available in 3^{rd} and 2^{nd} row, 3^{rd} row object is already included so 2^{nd} row object(1^{st} object) is included in knapsack, profit of this object is 40, so remaining profit is 40-40=0.

Object that is included 1,3 and profit is 65.

N Queen Problem

This problem is to find an arrangement of N queens on a chess board, such that no queen can attack any other queens on the board.

The chess queens can attack in any direction as horizontal, vertical, horizontal and diagonal way.

A binary matrix is used to display the positions of N Queens, where no queens can attack other queens.

Input and Output

Input:
The size of a chess board. Generally, it is 8. as (8 x 8 is the size of a normal chess board.)
Output:
The matrix that represents in which row and column the N Queens can be placed.
If the solution does not exist, it will return false.

```
1 0 0 0 0 0 0 0
0 0 0 0 0 0 1 0
0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 1
0 1 0 0 0 0 0 0
0 0 0 1 0 0 0 0
0 0 0 0 0 1 0 0
0 0 1 0 0 0 0 0
```

```
In this output, the value 1 indicates the correct place for
the queens.
The 0 denotes the blank spaces on the chess board.
```

Algorithm

isValid(board, row, col)

Input: The chess board, row and the column of the board.

Output: True when placing a queen in row and place position is a valid or not.

```
Begin
  if there is a queen at the left of current col, then
    return false
  if there is a queen at the left upper diagonal, then
    return false
  if there is a queen at the left lower diagonal, then
    return false;
  return true //otherwise it is valid place
End
```

solveNQueen(board, col)

Input: The chess board, the col where the queen is trying to be placed.

Output: The position matrix where queens are placed.

```
Begin
```

```
  if all columns are filled, then
    return true
  for each row of the board, do
    if isValid(board, i, col), then
      set queen at place (i, col) in the board
      if solveNQueen(board, col+1) = true, then
        return true
      otherwise remove queen from place (i, col) from board.
  done
  return false
End
```

Source Code (C++)

```
#include<iostream>
using namespace std;
#define N 8

void printBoard(int board[N][N]) {
  for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++)
      cout << board[i][j] << " ";
    cout << endl;
  }
}

bool isValid(int board[N][N], int row, int col) {
  for (int i = 0; i < col; i++)    //check whether there is queen in the left or not
    if (board[row][i])
      return false;
  for (int i=row, j=col; i>=0 && j>=0; i--, j--)
    if (board[i][j])       //check whether there is queen in the left upper diagonal or not
      return false;
  for (int i=row, j=col; j>=0 && i<N; i++, j--)
    if (board[i][j])      //check whether there is queen in the left lower diagonal or not
      return false;
  return true;
}

bool solveNQueen(int board[N][N], int col) {
  if (col >= N)          //when N queens are placed successfully
    return true;
  for (int i = 0; i < N; i++) {     //for each row, check placing of queen is possible or not
    if (isValid(board, i, col) ) {
      board[i][col] = 1;      //if validate, place the queen at place (i, col)
      if ( solveNQueen(board, col + 1))    //Go for the other columns recursively
        return true;

      board[i][col] = 0;       //When no place is vacant remove that queen
    }
  }
  return false;       //when no possible order is found
```

```
}

bool checkSolution() {
  int board[N][N];
  for(int i = 0; i<N; i++)
    for(int j = 0; j<N; j++)
      board[i][j] = 0;      //set all elements to 0

  if ( solveNQueen(board, 0) == false )
  {     //starting from 0th column
    cout << "Solution does not exist";
    return false;
  }
  printBoard(board);
  return true;
}

int main() {
  checkSolution();
}
```

Output

```
1 0 0 0 0 0 0 0
0 0 0 0 0 0 1 0
0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 1
0 1 0 0 0 0 0 0
0 0 0 1 0 0 0 0
0 0 0 0 0 1 0 0
0 0 1 0 0 0 0 0
```

Branch and bound Technique

A branch and bound algorithm is an optimization technique to get an optimal solution to the problem. It looks for the best solution for a given problem in the entire space of the solution. The bounds in the function to be optimized are merged with the value of the latest best solution. It allows the algorithm to find parts of the solution space completely. A branch and bound algorithm is an optimization technique to get an optimal solution to the problem. It looks for the best solution for a given problem in the entire space of the solution. The bounds in the function to be optimized are merged with the value of the latest best solution. It allows the algorithm to find parts of the solution space completely.

Problem Statement

A traveler needs to visit all the cities from a list, where distances between all the cities are known and each city should be visited just once. What is the shortest possible route that he visits each city exactly once and returns to the origin city?

Solution

Travelling salesman problem is the most notorious computational problem. We can use brute-force approach to evaluate every possible tour and select the best one. For n number of vertices in a graph, there are $(n - 1)!$ number of possibilities.
Instead of brute-force using dynamic programming approach, the solution can be obtained in lesser time, though there is no polynomial time algorithm.
Let us consider a graph $G = (V, E)$, where V is a set of cities and E is a set of weighted edges. An edge $e(u, v)$ represents that vertices u and v are connected. Distance between vertex u and v is $d(u, v)$, which should be non-negative.
Suppose we have started at city 1 and after visiting some cities now we are in city j. Hence, this is a partial tour. We certainly need to know j, since this will determine which cities are most convenient to visit next. We also need to know all the cities visited so far, so that we don't repeat any of them. Hence, this is an appropriate sub-problem.
For a subset of cities $S \in \{1, 2, 3, \ldots, n\}$ that includes 1, and $j \in S$, let $C(S, j)$ be the length of the shortest path visiting each node in S exactly once, starting at 1 and ending at j.
When $|S| > 1$, we define $C(S, 1) = \infty$ since the path cannot start and end at 1.
Now, let express C(S, j) in terms of smaller sub-problems. We need to start at 1 and end at j. We should select the next city in such a way that $C(S,j)=\min C(S-\{j\},i)+d(i,j)$ where $i \in S$ and $i \neq j$ $C(S,j)=\min C(s-\{j\},i)+d(i,j)$ where $i \in S$ and $i \neq j$

Algorithm: Traveling-Salesman-Problem

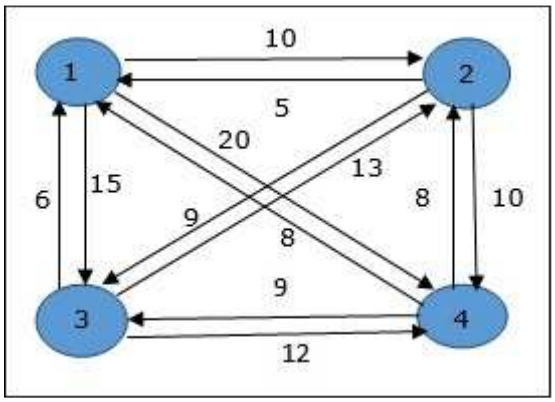
```
C ({1}, 1) = 0
for s = 2 to n do
   for all subsets S ∈ {1, 2, 3, … , n} of size s and containing 1
      C (S, 1) = ∞
   for all j ∈ S and j ≠ 1
      C (S, j) = min {C (S – {j}, i) + d(i, j) for i ∈ S and i ≠ j}
Return minj C ({1, 2, 3, …, n}, j) + d(j, i)
```

Analysis

There are at the most 2^n.n sub-problems and each one takes linear time to solve. Therefore, the total running time is O(2^n.n^2).

Example

In the following example, we will illustrate the steps to solve the travelling salesman problem.



From the above graph, the following table is prepared.

	1	2	3	4
1	0	10	15	20
2	5	0	9	10
3	6	13	0	12
4	8	8	9	0

S = Φ

Cost(2,Φ,1)=d(2,1)=5
Cost(3,Φ,1)=d(3,1)=6
Cost(4,Φ,1)=d(4,1)=8

S = 1

Cost(i,s)=min{Cost(j,s–(j))+d[i,j]}

Cost(2,{3},1)=d[2,3]+Cost(3,Φ,1)=9+6=15
Cost(2,{4},1)=d[2,4]+Cost(4,Φ,1)=10+8=18
Cost(3,{2},1)=d[3,2]+Cost(2,Φ,1)=13+5=18
Cost(3,{4},1)=d[3,4]+Cost(4,Φ,1)=12+8=20
Cost(4,{3},1)=d[4,3]+Cost(3,Φ,1)=9+6=15
Cost(4,{2},1)=d[4,2]+Cost(2,Φ,1)=8+5=13

S=2

Cost(2,{3,4},1) =d[2,3]+Cost(3,{4},1)=9+20=29
Cost(2,{3,4},1)=d[2,4]+Cost(4,{3},1)=10+15=25=25Cost(2,{3,4},1)=25
Cost(2,{3,4},1)=d[2,3]+Cost(2,{4},1)=9+20=29d[2,4]+Cost(4,{3},1)=10+15=25

Cost(3,{2,4},1)=d[3,2]+Cost(2,{4},1)=13+18=31
Cost(3,{2,4},1)=d[3,4]+Cost(4,{2},1)=12+13=25=25Cost(3,{2,4},1)

Cost(3,{2,4},1)=d[3,2]+Cost(2,{4},1)=13+18=31

Cost(4,{2,3},1)=

d[4,2]+Cost(2,{3},1)=8+15=23

d[4,3]+Cost(3,{2},1)=9+18=27=23Cost(4,{2,3},1)

d[3,2]+Cost(2,{3},1)=8+15=23

S=3

Cost(1,{2,3,4},1)=

d[1,2]+Cost(2,{3,4},1)=10+25=35

d[1,3]+Cost(3,{2,4},1)=15+25=40

d[1,4]+Cost(4,{2,3},1)=20+23=43=35Cost(1,{2,3,4},1)

d[1,2]+Cost(2,{3,4},1)=10+25=35

d[1,3]+Cost(3,{2,4},1)=15+25=40

d[1,4]+Cost(4,{2,3},1)=20+23=43

The minimum cost path is 35.

Start from cost {1, {2, 3, 4}, 1}, we get the minimum value for d [1, 2]. When s = 3, select the path from 1 to 2 (cost is 10) then go backwards. When s = 2, we get the minimum value for d [4, 2]. Select the path from 2 to 4 (cost is 10) then go backwards.

When s = 1, we get the minimum value for d [4, 3]. Selecting path 4 to 3 (cost is 9), then we shall go to then go to s = Φ step. We get the minimum value for d [3, 1] (cost is 6).