

LP1 Assignment AI&R 4

Date: 27th October, 2020

Title: Branch & Bound and Backtracking

Problem Definition:

Implement Crypt-arithmetic problem OR n-queens OR Graph Coloring problem OR Knapsack Problem
(Branch & Bound and Backtracking)

Learning Objectives:

- Learn and Implement Branch & Bound
- Learn and Implement Backtracking

Learning Outcomes:

- I will be able to solve Knapsack problem using Branch and Bound Technique
- I will be able to solve Graph coloring problem using Backtracking

S/W & H/W Packages:

1. Operating System: 64-bit Open source Linux or its derivative
2. Programming Language: Python
3. Google Colaboratory(uses Tesla K80 GPU)

Related Mathematics:

Programmer's Perspective(Knapsack Problem):

Let S be the system set:

$S = \{s; e; X; Y; Fme; DD; NDD; Fc; Sc\}$

s=start state

e=end state

X=set of inputs

$X = \{X1, X2, X3\}$

where

$X1$ = Total Weight

$X2$ = Weights of Items

$X3$ = Value of Items

Y = Optimal Subset of Items

Fme is the set of main functions

$Fme = \{f1, f2\}$

where,

$f1$ = Create Knapsack and select Items

$f2$ = Driver function (to display results)

DD= Deterministic Data: Value and Weight of Items

NDD=Non-deterministic data: No non deterministic data

Fc =failure case: No failure case identified for this application

Programmer's Perspective(Backtracking)

Let S be the system set: $S = \{s; e; X; Y; Fme; DD; NDD; Fc; Sc\}$

s=start state

e=end state

X=set of inputs

$X = \{X1, X2\}$

where

$X1$ = Number of Vertices in Graph

$X2$ = Adjacency Matrix to highlight Connections

Y = Optimal Subset of Colored Nodes

Fme is the set of main functions

$F_{me} = \{f_1, f_2, f_3\}$

where,

f1 = Initialize Graph

f2 = Checking adjacency between vertices

f3 = assigning colors to vertices

DD= Deterministic Data: Number of Vertices

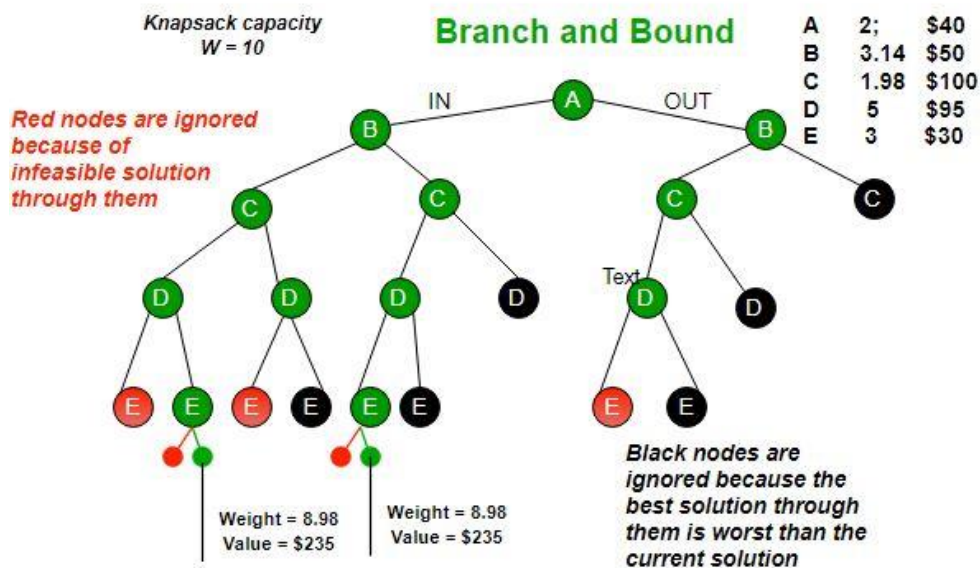
NDD=Non-deterministic data: No non deterministic data

Fc =failure case: No failure case identified for this application

Concepts Related to Theory:

Branch & Bound:

Branch and bound is an algorithm design paradigm which is generally used for solving combinatorial optimization problems. These problems are typically exponential in terms of time complexity and may require exploring all possible permutations in worst case. The Branch and Bound Algorithm technique solves these problems relatively quickly.



Perception

- Open List $V[v_1, \dots, v_n]$ $W[w_1, \dots, w_n]$ denoting value and weights of items.
- Searching through lists V and W using BFS

Cognition

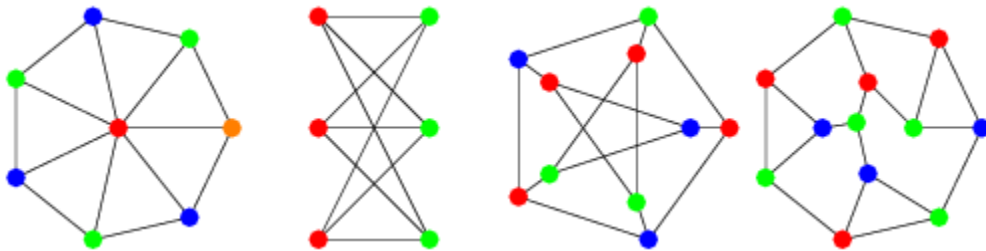
- Hypothesize a Value Profit P , which is a list containing $[v_1/w_1, \dots, v_n/w_n]$
- Algorithm BnB_Knapsack identifies items from V and W where Profit $P(\text{Value/Weight}) = \text{Max}$

Action

- Max Value V_{max} is calculated

Backtracking

Backtracking is an algorithmic-technique for solving problems recursively by trying to build a solution incrementally, one piece at a time, removing those solutions that fail to satisfy the constraints of the problem at any point of time



Algorithm:

1. Create a recursive function that takes the graph, current index, number of vertices and output color array.
2. If the current index is equal to number of vertices. Print the color configuration in output array.
3. Assign color to a vertex (1 to m).
4. For every assigned color, check if the configuration is safe, (i.e. check if the adjacent vertices do not have the same color) recursively call the function with next index and number of vertices
5. If any recursive function returns true break the loop and return true.
6. If no recursive function returns true then return false.

Perception

- Open List $V[v_1, \dots, v_n]$ denoting vertices of graph
- Searching through Adjacency Matrix to perceive connections between vertices V using DFS

Cognition

- Algorithm Graph_Coloring identifies adjacent vertices from V

Action

- Color $C[c_1, \dots, c_n]$ is assigned to vertices
- Minimum items from C are used to color vertices V

Key Differences

Type of Problems

- Backtracking is better for **Decision Problems** while Branch and Bound is useful for **Optimization Problems**

Searching

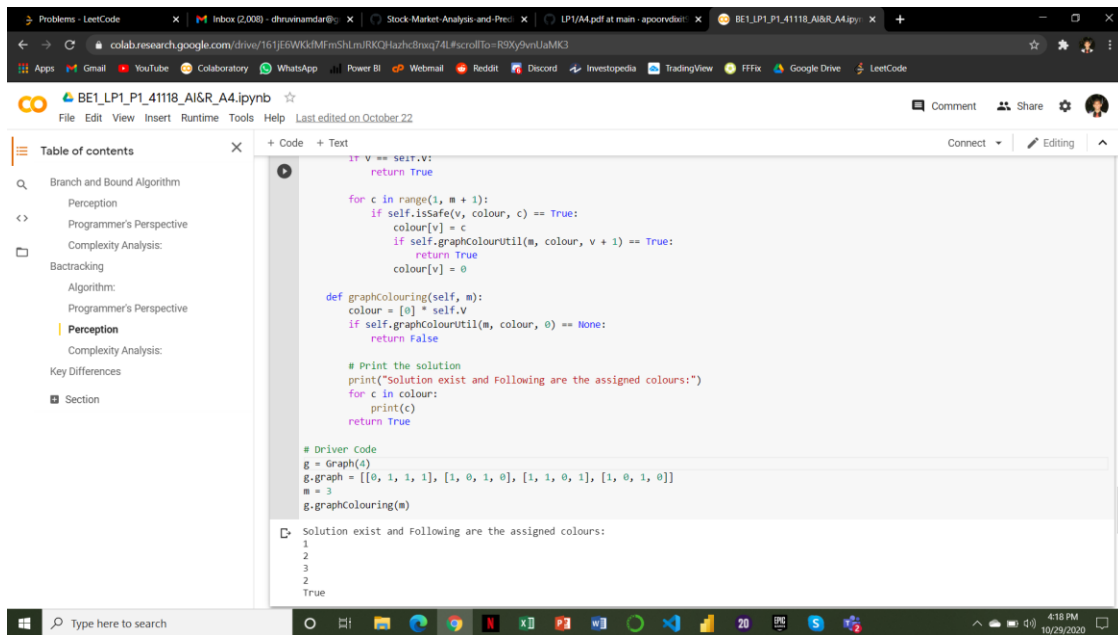
- In **backtracking**, the state space tree is searched until the solution is obtained. whereas, in **Branch and Bound** as the optimum solution may be present any where in the state space tree, so the tree need to be searched completely.

Traversal

- **Backtracking** uses DFS to traverse whereas Branch and Bound can use either of DFS or BFS

Output:

Graph Coloring Algorithm:



The screenshot shows a Google Colab notebook titled "BE1_LP1_P1_41118_AI&R_A4.ipynb". The left sidebar contains a "Table of contents" with sections like "Branch and Bound Algorithm", "Perception", "Programmer's Perspective", "Complexity Analysis", "Backtracking", "Algorithm", "Programmer's Perspective", "Perception", "Complexity Analysis", "Key Differences", and "Section". The main code area contains the following Python code:

```
if v == self.v:
    return True

for c in range(1, m + 1):
    if self.isSafe(v, colour, c) == True:
        colour[v] = c
        if self.graphColourUtil(m, colour, v + 1) == True:
            return True
        colour[v] = 0

def graphColouring(self, m):
    colour = [0] * self.v
    if self.graphColourUtil(m, colour, 0) == None:
        return False

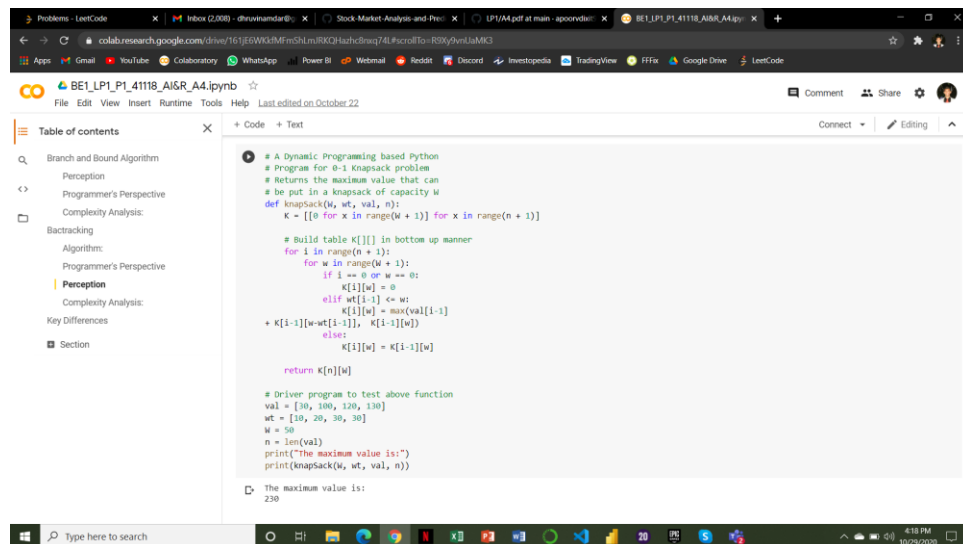
    # Print the solution
    print("Solution exist and Following are the assigned colours:")
    for c in colour:
        print(c)
    return True

# Driver Code
g = Graph(4)
g.graph = [[0, 1, 1, 1], [1, 0, 1, 0], [1, 1, 0, 1], [1, 0, 1, 0]]
m = 3
g.graphColouring(m)
```

The output of the code is:

```
Solution exist and Following are the assigned colours:
1
2
3
2
True
```

Knapsack Problem:



The screenshot shows a Google Colab notebook titled "BE1_LP1_P1_41118_AI&R_A4.ipynb". The left sidebar contains a "Table of contents" with sections like "Branch and Bound Algorithm", "Perception", "Programmer's Perspective", "Complexity Analysis", "Backtracking", "Algorithm", "Programmer's Perspective", "Perception", "Complexity Analysis", "Key Differences", and "Section". The main code area contains the following Python code:

```
# A Dynamic Programming based Python
# Program for 0-1 Knapsack problem
# Returns the maximum value that can
# be put in a knapsack of capacity W
def knapSack(W, wt, val, n):
    K = [[0 for x in range(W + 1)] for x in range(n + 1)]

    # Build table K[][] in bottom up manner
    for i in range(n + 1):
        for w in range(W + 1):
            if i == 0 or w == 0:
                K[i][w] = 0
            elif wt[i-1] <= w:
                K[i][w] = max(val[i-1]
                    + K[i-1][w-wt[i-1]], K[i-1][w])
            else:
                K[i][w] = K[i-1][w]

    return K[n][W]

# Driver program to test above function
val = [10, 100, 120, 130]
wt = [10, 20, 30, 30]
W = 50
n = len(val)
print("The maximum value is:")
print(knapSack(W, wt, val, n))
```

The output of the code is:

```
The maximum value is:
230
```

Notebook Link:

<https://colab.research.google.com/drive/161jE6WKkfMFmShLmJRKQH-azhc8nxq74L?usp=sharing>

Conclusion:

I have successfully designed and implemented Knapsack Problem using Branch & Bound Technique and Graph Coloring using Backtracking