

Aayush Shah

19BCE245

17 November 2021

Design and Analysis of Algorithms

Practical 10

0/1 Knapsack Using Branch & Bound

• Code :

```
/*
Aayush Shah
19BCE245
DAA Practical 10(A)      | Knapsack using Branch & Bound
*/

#include <bits/stdc++.h>
using namespace std;

/* Item structure that stores the weight and value of the
item */
struct Item
{
    float weight;
    int value;
};

/* Node structure to store information of decision tree
*/
struct Node
{
    /*
        level : node level in decision tree
        profit : profit including current node
        bound : upper bound of max profit in current
subtree node
    */
    int level, profit, bound;
```

```

    float weight;
};

/* Sort Item by val/weight ratio using the comparison
function. */
bool cmp(Item a, Item b)
{
    double r1 = (double)a.value / a.weight;
    double r2 = (double)b.value / b.weight;
    return r1 > r2;
}

/* Profit bound returns in a subtree rooted with u. The
Greedy solution is mostly used in this function to
determine an upper bound on maximum profit. */
int bound(Node u, int n, int W, Item arr[])
{
    // if weight overcomes the knapsack capacity, return
    // 0 as expected bound
    if (u.weight >= W)
        return 0;

    // initialize bound on profit by current profit
    int profit_bound = u.profit;

    // start including items from index 1 more to current
    // item index
    int j = u.level + 1;
    int totweight = u.weight;

    // checking index condition and knapsack capacity
    // condition
    while ((j < n) && (totweight + arr[j].weight <= W))
    {
        totweight += arr[j].weight;
        profit_bound += arr[j].value;
        j++;
    }

    // If k is not n, include last item partially for
    // upper bound on profit
    if (j < n)
        profit_bound += (W - totweight) * arr[j].value /

```

```

arr[j].weight;

    return profit_bound;
}

// Returns maximum profit we can get with capacity W
int knapsack(int W, Item arr[], int n)
{
    // sorting Item on basis of value per unit
    // weight.
    sort(arr, arr + n, cmp);

    // make a queue for traversing the node
    queue<Node> Q;
    Node u, v;

    // dummy node at starting
    u.level = -1;
    u.profit = u.weight = 0;
    Q.push(u);

    // One by one extract an item from decision tree
    // compute profit of all children of extracted item
    // and keep saving maxProfit
    int maxProfit = 0;
    while (!Q.empty())
    {
        // Dequeue a node
        u = Q.front();
        Q.pop();

        // If it is starting node, assign level 0
        if (u.level == -1)
            v.level = 0;

        // If there is nothing on next level
        if (u.level == n-1)
            continue;

        // Else if not last node, then increment level,
        // and compute profit of children nodes.
        v.level = u.level + 1;
    }
}

```

```

        // Taking current level's item add current
        // level's weight and value to node u's
        // weight and value
        v.weight = u.weight + arr[v.level].weight;
        v.profit = u.profit + arr[v.level].value;

        // If cumulated weight is less than W and
        // profit is greater than previous profit,
        // update maxprofit
        if (v.weight <= W && v.profit > maxProfit)
            maxProfit = v.profit;

        // Get the upper bound on profit to decide
        // whether to add v to Q or not.
        v.bound = bound(v, n, W, arr);

        // If bound value is greater than profit,
        // then only push into queue for further
        // consideration
        if (v.bound > maxProfit)
            Q.push(v);

        // Do the same thing, but Without taking
        // the item in knapsack
        v.weight = u.weight;
        v.profit = u.profit;
        v.bound = bound(v, n, W, arr);
        if (v.bound > maxProfit)
            Q.push(v);
    }

    return maxProfit;
}

// driver program to test above function
int main()
{
    int W = 8; // Weight of knapsack
    // Item arr[] = {{2, 40}, {3.14, 50}, {1.98, 100}, {5,
    95}, {3, 30}};

```

```

    Item arr[] = {{2,1}, {3,2}, {4,5}, {5,6}};
    int n = sizeof(arr) / sizeof(arr[0]);
    /*
    int n,w;
    printf("Enter number of objects : ");
    scanf("%d",&n);

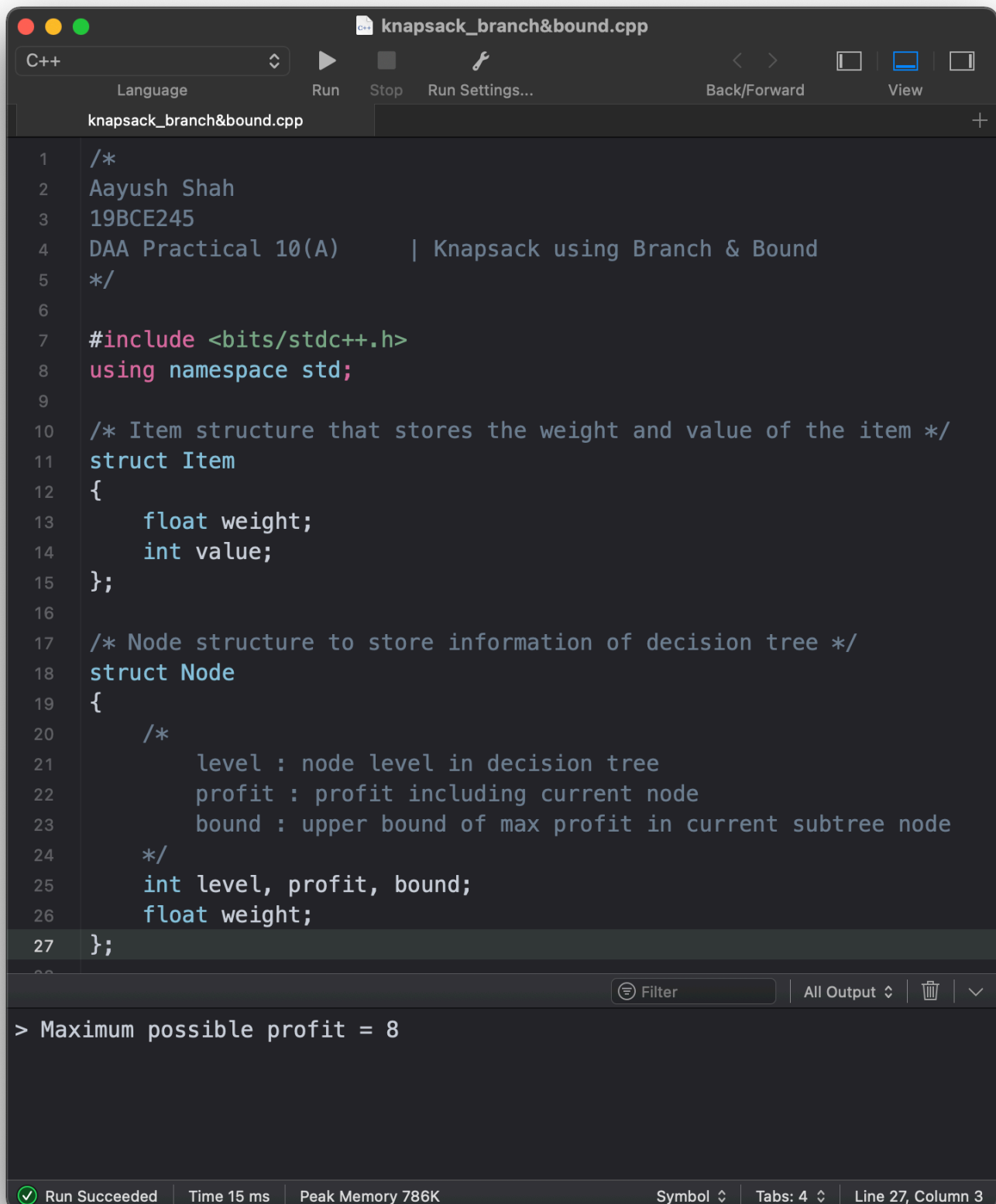
    Item arr[n][n];

    printf("Enter maximum weight capacity : ");
    scanf("%d",&W);
    for(int i=0;i<n;i++){
        printf("\n\tEnter weight of item %d : ", i+1);
        scanf("%d",&arr[i][0]);
        printf("\tEnter value  of item %d : ", i+1);
        scanf("%d",&arr[i][1]);
    }
    */

    cout << "> Maximum possible profit = "
        << knapsack(W, arr, n);

    return 0;
}

```

• Output :

```
knapsack_branch&bound.cpp
C++
Run Stop Run Settings... Back/Forward View
knapsack_branch&bound.cpp +
1  /*
2  Aayush Shah
3  19BCE245
4  DAA Practical 10(A) | Knapsack using Branch & Bound
5  */
6
7  #include <bits/stdc++.h>
8  using namespace std;
9
10 /* Item structure that stores the weight and value of the item */
11 struct Item
12 {
13     float weight;
14     int value;
15 };
16
17 /* Node structure to store information of decision tree */
18 struct Node
19 {
20     /*
21      level : node level in decision tree
22      profit : profit including current node
23      bound : upper bound of max profit in current subtree node
24     */
25     int level, profit, bound;
26     float weight;
27 };
28
29
> Maximum possible profit = 8
Run Succeeded Time 15 ms Peak Memory 786K Symbol Tabs: 4 Line 27, Column 3
```

N-Queen using Backtracking

• Code :

```
/*
Aayush Shah
19BCE245
DAA Practical 10(B)          | N-Queen using Backtracking
*/

/* Program to solve N Queen Problem using backtracking */

#define N 5
#include <stdbool.h>
#include <stdio.h>

/* A utility function to print solution */
void printSolution(int board[N][N])
{
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++)
            printf(" %d ", board[i][j]);
        printf("\n");
    }
}

/* A utility function to check if a queen can
be placed on board[row][col]. Note that this
function is called when "col" queens are
already placed in columns from 0 to col -1.
So we need to check only left side for
attacking queens */
bool isSafe(int board[N][N], int row, int col)
{
    int i, j;

    /* Check this row on left side */
    for (i = 0; i < col; i++)
        if (board[row][i])
            return false;

    /* Check upper diagonal on left side */
```

```

    for (i = row, j = col; i >= 0 && j >= 0; i--, j--)
        if (board[i][j])
            return false;

    /* Check lower diagonal on left side */
    for (i = row, j = col; j >= 0 && i < N; i++, j--)
        if (board[i][j])
            return false;

    return true;
}

/* A recursive utility function to solve N
Queen problem */
bool solveNQUtil(int board[N][N], int col)
{
    /* base case: If all queens are placed
    then return true */
    if (col >= N)
        return true;

    /* Consider this column and try placing
    this queen in all rows one by one */
    for (int i = 0; i < N; i++) {
        /* Check if the queen can be placed on
        board[i][col] */
        if (isSafe(board, i, col)) {
            /* Place this queen in board[i][col] */
            board[i][col] = 1;

            /* recur to place rest of the queens */
            if (solveNQUtil(board, col + 1))
                return true;

            /* If placing queen in board[i][col]
            doesn't lead to a solution, then
            remove queen from board[i][col] */
            board[i][col] = 0; // BACKTRACK
        }
    }

    /* If the queen cannot be placed in any row in
    this column col then return false */

```



```

    return false;
}

/* This function solves the N Queen problem using
Backtracking. It mainly uses solveNQUtil() to
solve the problem. It returns false if queens
cannot be placed, otherwise, return true and
prints placement of queens in the form of 1s.
Please note that there may be more than one
solutions, this function prints one of the
feasible solutions.*/
bool solveNQ()
{
    int board[N][N] = { { 0, 0, 0, 0, 0},
                        { 0, 0, 0, 0, 0},
                        { 0, 0, 0, 0, 0},
                        { 0, 0, 0, 0, 0},
                        { 0, 0, 0, 0, 0} };

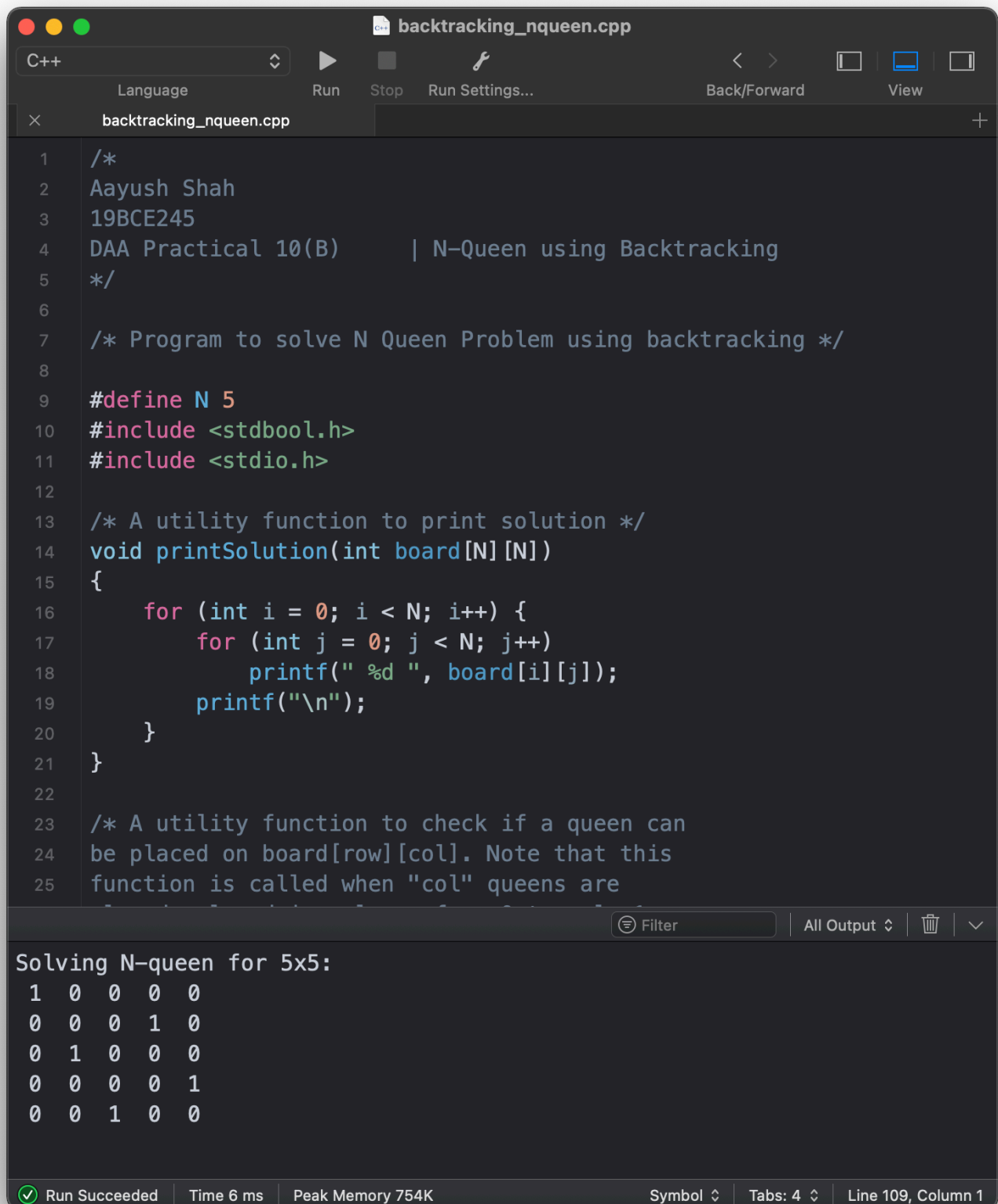
    if (solveNQUtil(board, 0) == false) {
        printf("Solution does not exist");
        return false;
    }

    printSolution(board);
    return true;
}

/* Main function */
int main()
{
    printf("Solving N-queen for %dx%d: \n",N,N);
    solveNQ();
    return 0;
}

```

• Output



```
1  /*
2  Aayush Shah
3  19BCE245
4  DAA Practical 10(B)      | N-Queen using Backtracking
5  */
6
7  /* Program to solve N Queen Problem using backtracking */
8
9  #define N 5
10 #include <stdbool.h>
11 #include <stdio.h>
12
13 /* A utility function to print solution */
14 void printSolution(int board[N][N])
15 {
16     for (int i = 0; i < N; i++) {
17         for (int j = 0; j < N; j++)
18             printf(" %d ", board[i][j]);
19         printf("\n");
20     }
21 }
22
23 /* A utility function to check if a queen can
24 be placed on board[row][col]. Note that this
25 function is called when "col" queens are
```

Solving N-queen for 5x5:

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

Run Succeeded | Time 6 ms | Peak Memory 754K | Symbol | Tabs: 4 | Line 109, Column 1