practice-3

November 25, 2024

1 Practice-3

[14]:
```
#include <iostream>
#include <vector>
#include <algorithm>
#include <limits.h>
#include <queue>
#include <climits>
#include <chrono>
using namespace std;
using namespace std::chrono;
```

Implement the activity selection problem to get a clear understanding of greedy approach.

[7]:
```
struct Activity {
    int start;
    int finish;
};
```

[8]:
```
bool activityCompare(Activity s1, Activity s2) {
    return (s1.finish < s2.finish);
}
```

[9]:
```
void printMaxActivities(vector<Activity> activities) {
    sort(activities.begin(), activities.end(), activityCompare);

    cout << "Selected activities are: " << endl;

    int i = 0;
    cout << "(" << activities[i].start << ", " << activities[i].finish << ")"␣
 ↪<< endl;

    for (int j = 1; j < activities.size(); j++) {
        if (activities[j].start >= activities[i].finish) {
            cout << "(" << activities[j].start << ", " << activities[j].finish␣
 ↪<< ")" << endl;
            i = j;
        }
```

```
    }
}
```

[10]:
```
vector<Activity> activities = {{5, 9}, {1, 2}, {3, 4}, {0, 6}, {5, 7}, {8, 9}};
printMaxActivities(activities);
```

```
Selected activities are:
(1, 2)
(3, 4)
(5, 7)
(8, 9)
(1, 2)
(3, 4)
(5, 7)
(8, 9)
```

Get a detailed insight of dynamic programming approach by the implementation of Matrix Chain Multiplication problem and see the impact of parenthesis positioning on time requirements for matrix multiplication.

[12]:
```
int MatrixChainOrder(vector<int> &p, int n) {
    vector<vector<int>> m(n, vector<int>(n, 0));

    for (int L = 2; L < n; L++) {
        for (int i = 1; i < n - L + 1; i++) {
            int j = i + L - 1;
            m[i][j] = INT_MAX;
            for (int k = i; k <= j - 1; k++) {
                int q = m[i][k] + m[k + 1][j] + p[i - 1] * p[k] * p[j];
                if (q < m[i][j]) {
                    m[i][j] = q;
                }
            }
        }
    }

    return m[1][n - 1];
}
```

[13]:
```
vector<int> arr = {1, 2, 3, 4};
int size = arr.size();

cout << "Minimum number of multiplications is " << MatrixChainOrder(arr, size)␣
 ↪<< endl;
```

```
Minimum number of multiplications is 18
```

Compare the performance of Dijkstra and Bellman ford algorithm for the single source shortest path problem.

[15]:
```
struct Edge {
    int src, dest, weight;
};
```

[16]:
```
void dijkstra(vector<vector<pair<int, int>>> &adj, int src, int V) {
    vector<int> dist(V, INT_MAX);
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int,␣
 ↪int>>> pq;

    dist[src] = 0;
    pq.push({0, src});

    while (!pq.empty()) {
        int u = pq.top().second;
        pq.pop();

        for (auto &neighbor : adj[u]) {
            int v = neighbor.first;
            int weight = neighbor.second;

            if (dist[u] + weight < dist[v]) {
                dist[v] = dist[u] + weight;
                pq.push({dist[v], v});
            }
        }
    }
}
```

[17]:
```
void bellmanFord(vector<Edge> &edges, int src, int V) {
    vector<int> dist(V, INT_MAX);
    dist[src] = 0;

    for (int i = 1; i <= V - 1; i++) {
        for (auto &edge : edges) {
            int u = edge.src;
            int v = edge.dest;
            int weight = edge.weight;

            if (dist[u] != INT_MAX && dist[u] + weight < dist[v]) {
                dist[v] = dist[u] + weight;
            }
        }
    }
}
```

[18]:
```
int V = 5;
vector<vector<pair<int, int>>> adj(V);
```

```
vector<Edge> edges;
```

[19]:
```
adj[0].push_back({1, 10});
adj[0].push_back({4, 5});
adj[1].push_back({2, 1});
adj[1].push_back({4, 2});
adj[2].push_back({3, 4});
adj[3].push_back({2, 6});
adj[3].push_back({0, 7});
adj[4].push_back({1, 3});
adj[4].push_back({2, 9});
adj[4].push_back({3, 2});
```

[20]:
```
edges.push_back({0, 1, 10});
edges.push_back({0, 4, 5});
edges.push_back({1, 2, 1});
edges.push_back({1, 4, 2});
edges.push_back({2, 3, 4});
edges.push_back({3, 2, 6});
edges.push_back({3, 0, 7});
edges.push_back({4, 1, 3});
edges.push_back({4, 2, 9});
edges.push_back({4, 3, 2});
```

[ ]:
```
auto start = high_resolution_clock::now();
dijkstra(adj, 0, V);
auto end = high_resolution_clock::now();
auto duration = duration_cast<microseconds>(end - start).count();
cout << "Dijkstra's algorithm execution time: " << duration << " microseconds"␣
↪<< endl;

start = high_resolution_clock::now();
bellmanFord(edges, 0, V);
end = high_resolution_clock::now();
duration = duration_cast<microseconds>(end - start).count();
cout << "Bellman-Ford algorithm execution time: " << duration << "␣
↪microseconds" << endl;
```

Through 0/1 Knapsack problem, analyze the greedy and dynamic programming approach for the same dataset.

[ ]:
```
struct Item {
    int value, weight;
};

bool cmp(Item a, Item b) {
    double r1 = (double)a.value / a.weight;
```

```
    double r2 = (double)b.value / b.weight;
    return r1 > r2;
}

int knapSackGreedy(int W, vector<Item> &items) {
    sort(items.begin(), items.end(), cmp);
    int curWeight = 0;
    int finalValue = 0;

    for (auto &item : items) {
        if (curWeight + item.weight <= W) {
            curWeight += item.weight;
            finalValue += item.value;
        } else {
            int remain = W - curWeight;
            finalValue += item.value * ((double)remain / item.weight);
            break;
        }
    }

    return finalValue;
}

int knapSackDP(int W, vector<Item> &items, int n) {
    vector<vector<int>> K(n + 1, vector<int>(W + 1));

    for (int i = 0; i <= n; i++) {
        for (int w = 0; w <= W; w++) {
            if (i == 0 || w == 0)
                K[i][w] = 0;
            else if (items[i - 1].weight <= w)
                K[i][w] = max(items[i - 1].value + K[i - 1][w - items[i - 1].
  ↪weight], K[i - 1][w]);
            else
                K[i][w] = K[i - 1][w];
        }
    }

    return K[n][W];
}

vector<Item> items = {{60, 10}, {100, 20}, {120, 30}};
int W = 50;
int n = items.size();

cout << "Maximum value in Knapsack (Greedy): " << knapSackGreedy(W, items) <<␣
  ↪endl;
```

```
cout << "Maximum value in Knapsack (DP): " << knapSackDP(W, items, n) << endl;
```

Implement the sum of subset and N Queen problem.

```
[ ]: void printSolution(vector<int> &board) {
    for (int i = 0; i < board.size(); i++) {
        for (int j = 0; j < board.size(); j++) {
            if (board[i] == j)
                cout << "Q ";
            else
                cout << ". ";
        }
        cout << endl;
    }
    cout << endl;
}

bool isSafe(vector<int> &board, int row, int col) {
    for (int i = 0; i < row; i++) {
        if (board[i] == col || abs(board[i] - col) == abs(i - row))
            return false;
    }
    return true;
}

void solveNQueen(vector<int> &board, int row) {
    if (row == board.size()) {
        printSolution(board);
        return;
    }

    for (int col = 0; col < board.size(); col++) {
        if (isSafe(board, row, col)) {
            board[row] = col;
            solveNQueen(board, row + 1);
            board[row] = -1;
        }
    }
}

void subsetSum(vector<int> &arr, vector<int> &subset, int index, int sum, int␣
 ↪target) {
    if (sum == target) {
        for (int i = 0; i < subset.size(); i++) {
            cout << subset[i] << " ";
        }
        cout << endl;
```

```
        return;
    }

    for (int i = index; i < arr.size(); i++) {
        if (sum + arr[i] <= target) {
            subset.push_back(arr[i]);
            subsetSum(arr, subset, i + 1, sum + arr[i], target);
            subset.pop_back();
        }
    }
}

int main() {
    int N = 4;
    vector<int> board(N, -1);
    cout << "Solutions for " << N << " Queen problem:" << endl;
    solveNQueen(board, 0);

    vector<int> arr = {10, 7, 5, 18, 12, 20, 15};
    int target = 35;
    vector<int> subset;
    cout << "Subsets with sum " << target << ":" << endl;
    subsetSum(arr, subset, 0, 0, target);

    return 0;
}
```

Compare the Backtracking and Branch & Bound Approach by the implementation of 0/1 Knapsack problem. Also compare the performance with dynamic programming approach.

[ ]:
```
struct Item {
    int value, weight;
};

int knapSackDP(int W, vector<Item> &items, int n) {
    vector<vector<int>> K(n + 1, vector<int>(W + 1));

    for (int i = 0; i <= n; i++) {
        for (int w = 0; w <= W; w++) {
            if (i == 0 || w == 0)
                K[i][w] = 0;
            else if (items[i - 1].weight <= w)
                K[i][w] = max(items[i - 1].value + K[i - 1][w - items[i - 1].
 ↪weight], K[i - 1][w]);
            else
                K[i][w] = K[i - 1][w];
        }
```

```
    }

    return K[n][W];
}

int knapSackBacktracking(int W, vector<Item> &items, int n, int idx = 0) {
    if (idx == n || W == 0)
        return 0;

    if (items[idx].weight > W)
        return knapSackBacktracking(W, items, n, idx + 1);

    int include = items[idx].value + knapSackBacktracking(W - items[idx].
    ↪weight, items, n, idx + 1);
    int exclude = knapSackBacktracking(W, items, n, idx + 1);

    return max(include, exclude);
}

struct Node {
    int level, profit, bound, weight;
};

int bound(Node u, int n, int W, vector<Item> &items) {
    if (u.weight >= W)
        return 0;

    int profit_bound = u.profit;
    int j = u.level + 1;
    int totweight = u.weight;

    while ((j < n) && (totweight + items[j].weight <= W)) {
        totweight += items[j].weight;
        profit_bound += items[j].value;
        j++;
    }

    if (j < n)
        profit_bound += (W - totweight) * items[j].value / items[j].weight;

    return profit_bound;
}

int knapSackBranchAndBound(int W, vector<Item> &items, int n) {
    sort(items.begin(), items.end(), [](Item a, Item b) {
        return (double)a.value / a.weight > (double)b.value / b.weight;
    });
```

```
    queue<Node> Q;
    Node u, v;
    u.level = -1;
    u.profit = u.weight = 0;
    Q.push(u);

    int maxProfit = 0;
    while (!Q.empty()) {
        u = Q.front();
        Q.pop();

        if (u.level == -1)
            v.level = 0;

        if (u.level == n - 1)
            continue;

        v.level = u.level + 1;

        v.weight = u.weight + items[v.level].weight;
        v.profit = u.profit + items[v.level].value;

        if (v.weight <= W && v.profit > maxProfit)
            maxProfit = v.profit;

        v.bound = bound(v, n, W, items);

        if (v.bound > maxProfit)
            Q.push(v);

        v.weight = u.weight;
        v.profit = u.profit;
        v.bound = bound(v, n, W, items);
        if (v.bound > maxProfit)
            Q.push(v);
    }

    return maxProfit;
}

vector<Item> items = {{60, 10}, {100, 20}, {120, 30}};
int W = 50;
int n = items.size();

auto start = high_resolution_clock::now();
cout << "Maximum value in Knapsack (DP): " << knapSackDP(W, items, n) << endl;
```

```
auto end = high_resolution_clock::now();
auto duration = duration_cast<microseconds>(end - start).count();
cout << "DP execution time: " << duration << " microseconds" << endl;

start = high_resolution_clock::now();
cout << "Maximum value in Knapsack (Backtracking): " << knapSackBacktracking(W, items, n) << endl;
end = high_resolution_clock::now();
duration = duration_cast<microseconds>(end - start).count();
cout << "Backtracking execution time: " << duration << " microseconds" << endl;

start = high_resolution_clock::now();
cout << "Maximum value in Knapsack (Branch and Bound): " << knapSackBranchAndBound(W, items, n) << endl;
end = high_resolution_clock::now();
duration = duration_cast<microseconds>(end - start).count();
cout << "Branch and Bound execution time: " << duration << " microseconds" << endl;
```