Master 1 MoSIG

Algorithmic Problem Solving

APP2 Report

Hold'em for n00bs

Team:

SACAD

Members:

Andrey **SOSNIN**

Majdeddine **ALHAFEZ**

Antoine **Colombier**

Eman **AL-SHAOUR**

Son Tung **DO**

Grenoble, 16 October, 2017

Contents

1 Greedy approach

We model the set of cards as an array of integers of size N. We do not consider a more realistic model (N even, at most 52 cards, values between 2 and 14, at most 4 cards of each value) because we focus (except for this section) on exact algorithms which ignore these details.
An implementation of the simulation of a game, where both players employ the same greedy strategy is the following:

Algorithm 1 Simulate greedy

$S \leftarrow create\_random\_array(N)$;
$first \leftarrow 0$;
$last \leftarrow N - 1$;
$wait\_for\_the\_opponent()$;
$N \leftarrow length(S) - 1$;
while $N > 0$ **do**
  if $S[first] > S[last]$ **then**
    $first \leftarrow first + 1$;
  else
    $last \leftarrow last - 1$;
  end if
  $N \leftarrow N - 1$;
  if $N > 0$ **then**
    $wait\_for\_the\_opponent()$;
    $N \leftarrow N - 1$;
  end if
end while

In the code above, $wait\_for\_the\_opponent()$ lets the opponent make their move (if it's the first one, the opponent has an option not to do it). This procedure also updates $fisrt$ or $last$. **This procedure is assumed to be deterministic**: for example, if the first and the last card have the same value, the opponent always chooses the left one.

Using greedy strategy against an opponent playing a greedy strategy is **not optimal**: in the following game the opponent can be defeated, but not with the greedy strategy (whoever is taking the first card):

$$\{3\ 10\ 3\ 9\ 5\ 2\}$$

If the player takes the first card he can go: $right - right - left$ (or $2 - 9 - 10$). Otherwise, if the opponent takes 3, he can go $left - left - left$ (or $10 - 9 - 2$). In both cases this wins the game, while applying greedy strategy doesn't.

Using a greedy algorithm with another metric was taken into consideration. The suggested metric was maximizing the immediate score resulting by making a choice: $max(value(plyer's\ choice) - value(opponent's\ choice\ given\ player's\ choice))$. Howerver, the simulations showed that it resulted in a lower win ratio: about 0.15 versus 0.45 using standard greedy strategy.

2 Optimal solution by exhaustion

We assume that after a dog's choice we have two possible solutions for the player's choice.

- In case the dog goes first, we have:
  optimal_solution opt = Dogs_turn(0, N-1)
- In case the dog chooses to go second, we have:
  optimal_solution opt = best_score(explore_solution(0, N-1, left), explore_solution(0, N-1, right))

Time complexity is exponential: $T(N) = 2^{(N/2)}$
Where the space complexity is linear: $S(N) = O(N)$ The amount of information store is at most K where K is the height of the recursive call function.

```
optimal_solution : {
    string path;        ▷ stores the indexes of the cards
    int score;    ▷ best total score that we can have reaching this sub-problem from the root
}
```

Algorithm 2 Complete space exploration

```
procedure DOGS_TURN(i, j)            ▷ i, j : the indexes of the rightmost and leftmost cards
    if i ≤ j then
        optimal_solution opt;
        if S[i] ≥ S[j] then
            value ← S[j];
            j − −;
            index ← j;
        else
            value ← S[i];
            i + +;
            index ← i;
        end if
        right_opt ← explore_solution(i, j, right);
        left_opt ← explore_solution(i, j, left);
        if left_opt.score ≥ right_opt.score then
            opt.path ← append(left_opt.path, index);
            opt.score ← left_opt.score + value;
        else
            opt.path ← append(right_opt.path, index);
            opt.score ← right_opt.score + value;
        end if
        return opt;
    else
        return NULL;
    end if
end procedure
```

Algorithm 3

procedure $explore\_solution(i, j, choice)$ ▷ i, j : indexes of the rightmost and leftmost cards, choice: which card to choose
    if $i \le j$ **then**
        $optimal\_solution\ opt$;
        if $choice = right$ **then**
            $value \leftarrow S[j]$;
            $j - -$;
            $index \leftarrow j$;
        else
            $value \leftarrow S[i]$;
            $i + +$;
            $index \leftarrow i$;
        end if
        $current\_opt \leftarrow Dogs\_turn(i, j)$;
        $opt.path \leftarrow append(current\_opt.path, index)$;
        $opt.score \leftarrow value + current\_opt.score$
        return opt;
    else
        return $NULL$;
    end if
end procedure

3 Dynamic approach

We use the cache 2 dimensions array of size n by n where n is the total number of cards that we played. We have $cache[n][n]$ is the array of cache. If we move by row $(i + +)$, this means that we choose the leftmost card, and if we move by column $(j + +)$, this means that we choose the rightmost card.

The data structure for an element in cache[i][j]

Algorithm 4 Data structure of cache element

int score ▷ Store the best score to arrive to that elements
string path ▷ String to store the path to reach that elements with best score

Algorithm 5

```
procedure cache_compute(l, r)
    for k from 0 to N do                                        ▷ Diagonal has N elements
        if C[l] ≤ C[r] then                                      ▷ Illustrate Dog move
            r = r − 1
        else
            l = l + 1
        end if
        j = 0
        for i from 0 to k do
            if k − i == 0 then
                cache[0][j].score = C[r − −]
                cache[0][j].path = ‘R‘
            else if j == 0 then
                cache[k − i][0].score = C[l + +]
                cache[k − i][0].path = ‘L‘
            else
                if cache[k − i − 1][j] = NULL then
                    cache[k − i][j].score = cache[k − i][j − 1].score + C[r − −]
                    cache[k − i][j].path = append(cache[k − i][j].path, ‘R‘)
                else if cache[k − i][j − 1] = NULL then
                    cache[k − i][j].score = cache[k − i − 1][j].score + C[l + +]
                    cache[k − i][j].path = append(cache[k − i − 1][j].path, ‘L‘)
                else if cache[k − i − 1][j].score ≤ cache[k − i][j − 1].score then
                    cache[k − i][j].score = cache[k − i][j − 1].score + C[r − −]
                    cache[k − i][j].path = append(cache[k − i][j − 1].path, ‘R‘)
                else
                    cache[k − i][j].score = cache[k − i − 1][j].score + C[l + +]
                    cache[k − i][j].path = append(cache[k − i − 1][j].path, ‘L‘)
                end if
            end if
            j = j + 1
        end for
    end for
end procedure
```

Since we only have N moves, the square matrix (2 dimension array) is only used for half of it (the upper left triangle) By travesing the cache diagonal, we can find the solution with the max value.

Algorithm 6

```
procedure travesing_cache_diagonal
    x = 0
    y = N
    max = cache[x][y]
    for i = 0; i < N; i++ do
        if max.score ≤ cache[x][y].score then
            max = cache[x][y]
        end if
    end for
    return max
end procedure
```

4 Space and Time Complexity of Dynamic Approach

4.1 Space complexity

In this dynamic approach we use a cache of 2 dimensions array. The array has the size of N by N then the space for storing this cache is N^2

In addition, we need to store the card order in another array of 1 dimension with size N.

Our space complexity is O(N^2)

4.2 Time Complexity

We need to compute the upper triangular of cache which is made with $N^2/2$ computations.

We have to traverse the diagonal of the cache which costs O(N)

So overall the computation has a complexity of O(N^2).

5 Conclusion

In this APP, we faced a problem of card playing game which take turn withdraw a card (only left most or right most) in the squence of card face up on the table and winner is the one who has the highest sum of all the card they withdraw till no more card is available.

We played the game by using Greedy Algorithm and Dynamic Programming.

With Greedy Algorithm, we keep taking the card with the highest value possible. By exploring the whole solution space, we may encounter several same computation and duplicate the calculation.

With Dynamic Programming, by using the cache concept. We stored the same problem computation in a 2 dimensions array. so we reduced the amount of computation for the same problem.

With this dynamic approach, we have the complexity in time is O(N^2) and in space is O(N^2)