Master 1 MoSIG

Algorithmic Problem Solving

APP2 Report

Hold'em for n00bs

Team:

Members:
Andrey **SOSNIN**
Majdeddine **ALHAFEZ**
Antoine **COLOMBIER**
Eman **AL-SHAOUR**
Son Tung **DO**

Grenoble, 5 November, 2017

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: $max(value(plyer's\ choice) - value(opponent's\ choice\ given\ player's\ choice))$. However, 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))

Since both sons of any node have to be evaluated separately, the time complexity is exponential: $T(N) = 2^{(N/2)}$
Whereas the space complexity is quadratic: $S(N) = O(N^2)$: at any point we store N elements of size N.

```
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 ≤ j then
        optimal_solution opt;
        if choice = right then
            value ← S[j];
            j − −;
            index ← j;
        else
            value ← S[i];
            i + +;
            index ← i;
        end if
      current_opt ← Dogs_turn(i, j);
        opt.path ← append(current_opt.path, index);
        opt.score ← value + current_opt.score
        return opt;
    else
        return NULL;
    end if
end procedure
```

3 Dynamic approach

Previous solution is too expensive in terms of time complexity: a call to the function with $N = 52$ requires about 2^{31} operations (which amounts to 86 sec in Python implementation). We can however improve it by storing the results for the redundant computations. We introduce an $N \times N$ matrix *cache*. The value of $cache[i][j]$, if initialized, corresponds to the sub-problem produced by removing i cards on the left and j cards on the right. If the coordinates (i, j) correspond to an impossible sub-problem, $cache[i][j]$ is $None$.

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

We fill the matrix diagonal after diagonal, initializing $cache[0][0]$ to represent the initial problem. For every couple (i, j) we check if there's a possible corresponding sub-problem and if so, which way to reach it is the best. A sub-problem can be reached only from it's neighbouring cells on the previous diagonal: either from the top or from the left. The calls to $make\_dogs\_turn(cache, i, k-i)$ simulates the dog's turn by filling the k^{th} diagonal, producing one element for each element at diagonal k-1. Function $make\_move()$ represents player's choice of a card. It returns the net variation in the score.

Algorithm 5

```
procedure compute_cache(size)
    cache ← make_matrix(size, size, None);
    cache[0][0] ← Subproblem(score = 0; path = ϵ);
    dogs_turn = true;
    for k from 1 to size do
        if dogs_turn then
            for i from 0 to k do
                cache[i][k − i] ← make_dogs_turn(cache, i, k − i)
            end for
        else
            for i from 0 to k do
                cache[i][k − i] ← choose_best(cache, i, k − i)
            end for
        end if
        dogs_turn := ¬dogs_turn
    end for
end procedure


procedure choose_best(cache, i, j)
    left_result ← None
    right_result ← None
    if i > 0 ∧ cache[i − 1][j] ≠ None then
        score_change ← make_move(i − 1, j, left)
        left_score ← cache[i − 1][j].score + score_change
        left_path ← cache[i − 1][j].path
        left_result ← Subproblem(score = left_score, path = left_path + "L")
    end if
    if j > 0 ∧ cache[i][j − 1] ≠ None then
        similar, reading from cache[i][j-1], with
        path = right_path + "R"
    end if
    if both results are None then
        return None
    else if one of the results is None then
        return the other one
    else
        return the result with the highest score
    end if
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 traversing the cache's main diagonal, we can find the solution with the maximum score 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
```

3.1 Space complexity

We store at most N^2 elements in the cache, each containing a sting no longer than N characters. Our space complexity is $O(N^2)$

3.2 Time Complexity

We need to compute every the value for every cell of the $N \times N$ matrix at most once.

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

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

3.3 Proof of optimality

We want to show that the dynamic procedure returns the same result as the one we would get from full-space exploration tree.

We start by noticing that for every element with depth k in the tree, there is an initialized cell on the k^{th} diagonal in the cache.

For the outer loop (which iterates over the sequence of diagonals) we consider the following loop invariant: "At the previous diagonal, every initialized element contains the path with the best score for the corresponding sub-problem". Given a filled diagonal of rank k, any element at the diagonal $k+1$ can be reached from at most 2 directions, both of which are checked, so the invariant holds.

In particular, the last diagonal contains a set of best paths for the empty sub-problem, i.e. the best paths to reach the leaves of the tree. Extracting the maximum of this set gives the optimal solution.

3.4 Improved algorithm

Since we access data stored on every diagonal of the cache only once (as the path to reach a cell is stored in the cell), we can improve the space complexity of our solution by using a cache of linear size. We use two arrays of size n: one to store the "diagonal" currently being computed and one to store the previous diagonal. Then the we alternate the roles of the arrays.
Moreover, in order to reduce the space-complexity further, we can store boolean decisions representing a path on an integer value rather than a string. This allows to bring space complexity down to $O(N\ log(N))$, as there's only a finite number of linear-sized arrays and since storing N bits of information takes $log(N)$.

(The following pseudocode contains minimal differences with the previous version)

Algorithm 7

```
procedure compute_cache(size)
    cache ← make_array(2, []);
    cache[0] ← make_array(size, None);
    cache[1] ← make_array(size, None);
    cache[0][0] ← Subproblem(score = 0; path = ϵ);
    dogs_turn = true;
    for k from 1 to size do
        if dogs_turn then
            for i from 0 to k do
                cache[0][k − i] ← make_dogs_turn(cache[1], i, k − i)
            end for
        else
            for i from 0 to k do
                cache[1][k − i] ← choose_best(cache[0], i, k)
            end for
        end if
        dogs_turn := ¬dogs_turn
    end for
end procedure


procedure choose_best(diag, i, k)
    left_result ← None
    right_result ← None
    if i > 0 ∧ diag[i − 1] ≠ None then
        score_change ← make_move(i − 1, left)
        left_score ← diag[i − 1].score + score_change
        left_path ← diag[i − 1].path.path
        left_result ← Subproblem(score = left_score, path = left_path << 1)
    end if
    if i < k ∧ diag[i] ≠ None then
        similar, reading from cache[i], with
        path = right_path << 1 + 1
    end if
    if both results are None then
        return None
    else if one of the results is None then
        return the other one
    else
        return the result with the highest score
    end if
end procedure
```