A lot of ideas. Some good, some bad to know. It is crucial to KISS

C ?

(keep it simple, stupid)

Master 1 MoSIG

Algorithmic Problem Solving

APP2 Report

Hold'em for n00bs

Team:

Procedural Uniform Translation Across International Negotiations

hum...

pas de très bon goût...

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 ← create_random_array(N);
first ← 0;
last ← N − 1;
wait_for_the_opponent();
N ← length(S) − 1;
while N > 0 do
    if S[first] > S[last] then
        first ← first + 1;
    else
        last ← last − 1;
    end if
    N ← N − 1;
    if N > 0 then
        wait_for_the_opponent();
        N ← 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: $T(N) = 2^{(N/2)}$ explain?

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.

if you only store the score, but what if you need the way to play?

```
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
}
```

Seems $O(n^2)$ in space

Algorithm 2 Complete space exploration

procedure DOGS_TURN(i, j) ▷ i, j : the indexes of the rightmost and leftmost cards
  if $i \le j$ **then**
    optimal_solution opt;
    if $S[i] \ge S[j]$ **then**
      $value \leftarrow S[j]$;
      $j--$;
      $index \leftarrow j$;
    else
      $value \leftarrow S[i]$;
      $i++$;
      $index \leftarrow i$;
    end if
    $right\_opt \leftarrow explore\_solution(i, j, right)$;
    $left\_opt \leftarrow explore\_solution(i, j, left)$;
    if $left\_opt.score \ge right\_opt.score$ **then**
      $opt.path \leftarrow append(left\_opt.path, index)$;
      $opt.score \leftarrow left\_opt.score + value$;
    else
      $opt.path \leftarrow append(right\_opt.path, index)$;
      $opt.score \leftarrow right\_opt.score + value$;
    end if
    return *opt*;
  else
    return $NULL$;
  end if
end procedure

3 Dynamic approach

) why use dyn. prog. justify

We use an array of arrays of increasing sizes *cache*[][]. The big array corresponds to the "levels" of the problem: each level is an array corresponding to all sub-problems of the same size. Little arrays contain elements of type Subproblem:

Subproblem : {
  int *index*; ▷ first index of the subproblem in the initial problem
  int *best_score*; ▷ best total score that we can have reaching this subproblem from the root
  int *best_parent*; ▷ parent which maximizes the score
  bool *card*; ▷ last card taken: 0 - left, 1 - right
}

Sub problem is not really independent from initial pb then.

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
```

This is defined more in-detail in the section 4.1.

The entry P of the following procedure is the initial problem *after* the opponent has made their move, if any. N is the number of cards.

Algorithm 4 Generate good solutions

```
procedure COMPUTE_CACHE(subproblem P, int N)
    height ← (N + 1)/2 + 1;   magic ?
    cache ← create_array(height, []);
    cache[level] ← create_array(1, P);
    for level = 1 . . . height do
        cache[level] ← create_array(2 × level + 1, None);
        last ← add_sons(cache[level − 1][0], cache, level, −1);
        j ← 1;
        while (j < 2 × level − 1) ∧ (cache[level − 1][j] ≠ None) do
            last ← add_sons(cache[level − 1][j], cache, level, last);
            incrj;
        end while
    end for
    return cache
end procedure
```

(In the pseudocode "/" denotes integer division).

```
procedure ADD_SONS(subproblem P, array cache, int level, int last)
    S ← generate_left_son(P, level − 1);
    if last = −1 then                              ▷ Is it the first one on this level?
        cache[level][last] ← S;
        last ← last + 1;
    else
        S′ ← cache[level][last];
        if S′.index = S.index then          ▷ Was the problem already calculated?
            if S′.score < S.score then
                cache[level][last] ← S;                  ▷ We found a better solution
            end if
        else
            cache[level][last + 1] ← S;
            last ← last + 1;
        end if
    end if
    S ← generate_right_son(P, level − 1);
    S′ ← cache[level][last];
    if S′.index = S.index then
        if S′.score < S.score then
            cache[level][last] = S;
        else
            cache[level][last + 1] = S;
            last ← last + 1;
        end if
    end if
    return last;
end procedure
```

the "optimization" of memory makes the algorithm super difficult to read.

Procedures *generate_left_son* and *generate_rigth_son* work the same way, as in the 2nd section: they each generate a problem obtained by player choosing one of the two cards, then letting the opponent take its move. The *best_score* fields are obtained by taking the *best_score* problem and adding new cards. The *card* field is set to 0 for *generate_left_son* and to 1 for *generate_right_son*.

After generating the whole structure, we go through the last level of $cash[][]$ looking for the element with the highest score (linear complexity): when it is found we access its best parent, then its best parent and so on, until reaching the root. During this procedure we generate a sequence of 0s and 1s corresponding to the moves we take.

First: present a clean and simple algorithm, then, if necessary (justify!) an improved version.

procedure PLAY_THE_DAMN_GAME(subproblem P, array $cache$, int N)
  $height \leftarrow (N+1)/2+1$;
  $moves \leftarrow create\_array(height, 0)$;
  $best = \text{argmax}_{x.best\_score}(cache[height-1])$;
  for $level = height \dots 1$ **do**
    $moves[level-1] \leftarrow best.card$
    $best \leftarrow cache[level-1][best.best\_parent]$;
  end for
  $wait\_for\_the\_opponent()$;
  for $i = 1 \dots height$ **do**
    $play(moves[i])$
    $wait\_for\_the\_opponent()$;
  end for
end procedure

If the last level of *cache* contains elements of length 0, *argmax* looks for the biggest *best_score*; if it contains elements of length 1, (i.e. the player can take the last card) it looks for the highest *best_score plus value_of_the_remaining_card*.

need to come soon

4 Complexity and correctness of the dynamic approach

4.1 Definitions and manipulated objects

- The initial sequence S of N cards is called the **problem**. For the sake of clarity, we will write $S = \{x_0, x_1, ..., x_{N-1}\}$ to represent cards inside S (even though we only care about cards' indices, not the values). A **sub-problem** $P = \{x_i, x_{i+1}, ..., x_{i+r-1}\}$ of length r is a sub-string (*i.e.* a contiguous sub-sequence) of S that can be obtained after some number of moves by both the player and the opponent. Note that since we cannot choose opponent's moves, not every sub-string of S is a sub-problem. For example, $\{3,\ 11,\ 4\}$ is a sub-problem of $S_0 = \{8,\ 5,\ 3,\ 11,\ 4\}$, but $\{5,\ 3,\ 11\}$ is not: if the player takes 11, the opponent will take 5 and if the player takes 4, the opponent will take 8.

- The **index** $ind(P)$ of P in S is the position of the first card of P in S. For example, the index of $\{3,\ 11,\ 4\}$ in S_0 is 2.

- For two sub-problems P_1 and P_2 we say that $P_1 \leq P_2$ when $ind(P_1) \leq ind(P_2)$. For example $\{8,\ 5,\ 3\} \leq S_0 \leq \{3,\ 11,\ 4\}$.

- A **level** is a sequence $(P_1, ..., P_l)$ of all (different) sub-problems of the same length. If level 0 is $\{S\}$ then level Λ contains the sub-problems of length $k = N - 2 \times \Lambda$. We say that a level is **ordered** when all its elements are in

the same order they appear in S (*i.e.* $\forall(i,j),\ 1 \le i < j \le l, \quad P_i < P_j$). We will show later that our procedure produces ordered levels. An example of (ordered) levels:
$0 : (\{8,\ 5,\ 3,\ 11,\ 4\})$
$1 : (\{8,\ 5,\ 3\},\ \{3,\ 11,\ 4\})$
$2 : (\{5\},\ \{3\},\ \{4\})$

- Finally, even though the structure we are going to manipulate is not a tree, we will say that a sub-problem P' is the **left son** of a sub-problem P if P' is obtained by taking out the rightmost card in P and letting the opponent take their move; similarly, the **right son** is obtained by removing the the leftmost card. The two sons of P are called **siblings**.

We want to show that our procedure finds the optimal sequence of moves and does so in a quadratic time.

sub problem is $i \to j$ So at most $n \times n = n^2$

4.2 An upper bound for the number of sub-problems per level

The problem that we are going to address is *what is the maximal number of sub-problems that can we have on one level* or, in other words, *how do we know that there's enough merging?*

Let S be a string of length $N > 0$. How many different sub-strings of length r $(1 \le r \le N)$ can we extract from S?

Since a sub-string has to be contiguous, and its length r is fixed, the only choice we have is for its index. Which indices are possible? We cannot have it too far near the end of S, otherwise it won't fit: last possible index we can choose for a sub-string is $N - r$. Since all indices between 0 and $N - r$ are valid choices, the answer to the question above is: $N - r + 1$ sub-strings. ■

For our game this means that there can be at most $N - N + 1 = 1$ sub-problem of length N (on level 0), $N - (N-2) + 1 = 3$ sub-problems on level 1, 5 sub-problems on level 2, 7 sub-problems on level 3, etc. Note that this upper bound is not optimal: for example, we know that levels for the sub-problems of size $N - 2$ and $N - 4$ cannot contain more than $2^1 = 2$ and $2^2 = 4$ elements respectively. However, it ensures that the number of sub-problems of length $N - r$ is only a linear function of N (and not an exponential one, as we had with the solution by exhaustive enumeration).

4.3 Overlap detection

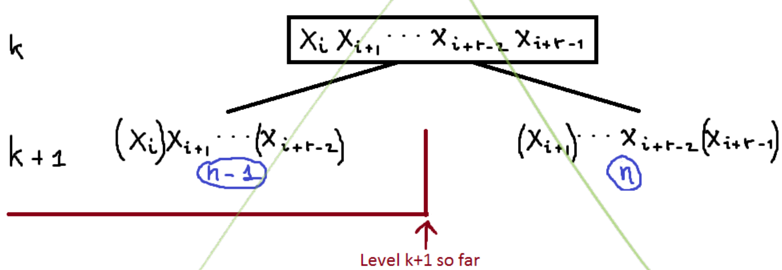
Our procedure fills each level from left to right, merging overlapping sub-problems. We want to show that it is possible to detect in $O(1)$ if a sub-problem has already been calculated. Indeed, when adding a new sub-problem P_i to a partially filled ordered level $\{P_0, ..., P_{i-1}\}$, we can simply check if $ind(P_{i-1}) = ind(P_i)$ (in which case P_i and P_{i-1} represent the same sub-problem), due to the fact that $\forall j < i-1, \quad P_j < P_{i-1}$, as the level is ordered. Hence we only need to show that:

The algorithm produces only ordered levels.

We proceed by induction on k : index of the level.

Level $k = 0$ has only one element, so the statement holds. Now suppose that we have (entirely) generated some level k. In order to generate level $k+1$, we take every element on level k from left to right, calculate its left and right sons and add them on level $k+1$, occasionally performing merging. Consider the loop invariant "after adding n^{th} element to the level $k+1$, the sequence of its n first elements is ordered". First element to be added to level $k+1$ is added unconditionally. Now suppose that $n-1$ elements have been added to the level $k+1$ without violating the invariant. If the n^{th} element P_n is a sibling of the P_{n-1}, then the loop invariant holds:

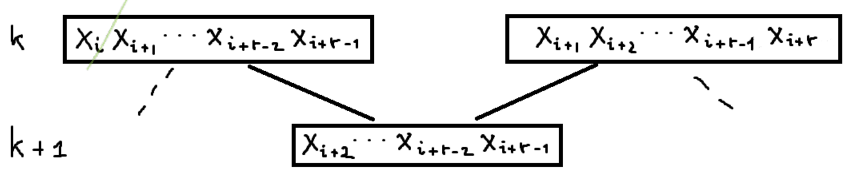
unnecessarily complicated!



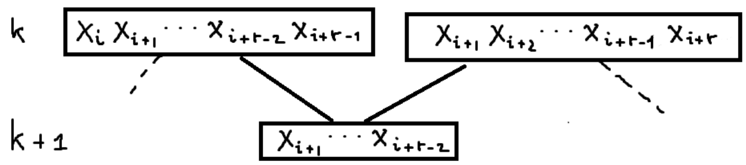
use a 2D matrix

the simplest is the easiest & the best!

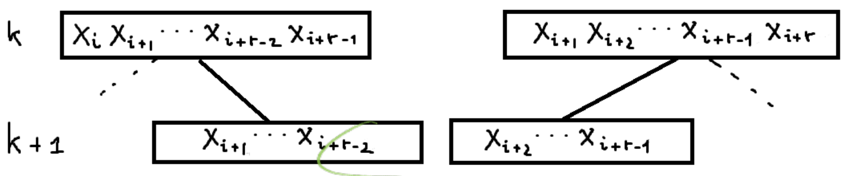
In the drawing above, we do not specify whichever card do P_n and P_{n-1} lack: in any case $P_{n-1} < P_n$. Now let's explore the situation where P_n is not being added after its sibling. All possible cases are listed below:



In this case, the opponent took the leftmost card in both P_n and P_{n-1}. Since the two problems are the same, they are merged into one.

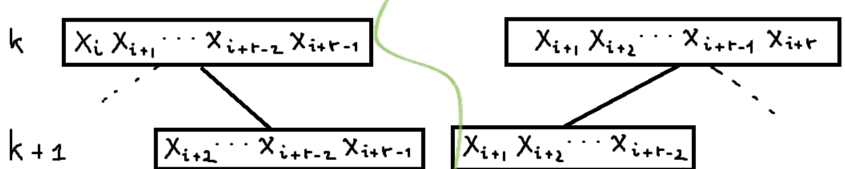


Same, but with the opponent taking the rightmost card.



The opponent chooses the the rightmost card in P_{n-1} and the leftmost in P_n. The problems are different, so P_n gets added to the level $k+1$ without being merged with P_{n-1}.

The following case would seemingly pose a problem:



However this situation is impossible. Indeed in P_{n-1} the opponent chooses x_{i+1} over x_{i+r-1}, but in P_n he does the opposite. Since the opponent's behaviour is deterministic, we cannot have this case.
By induction principle, this proves that level $k+1$ is ordered, which in its turn, concludes the proof. ■

4.4 Conclusion

We showed, that in order to apply the dynamic programming approach, one has to explore $\frac{N}{2} = O(N)$ levels; every level having at most $O(N)$ elements means that the amount of operations necessary in order to generate the next level is also $O(N)$, amount of computations to generate each element being constant. Therefore, the total complexity is $O(N^2)$.

This also proves, that the algorithm terminates.

In order to show that the algorithm finds the optimal solution, we need to

If you really want to prove optimality, you must prove that an optimal solution contains optimal sub-solutions to sub-problem (Counter ex: graph coloring)

show that the procedure produces a solution that is both feasible and optimal. Again we proceed by induction on levels showing $P(k)$:= *"on level k, all sub-problems can be reached, and their field best_score corresponds to the best score we can get doing so"*
The initial problem is feasible and its score is optimal. Suppose some level k satisfies $P(k)$. In add_sons we neither add nor modify elements of the level $k+1$, unless they derive from a problem on the level above, hence the feasibility. In addition, the level consist of all different problems, uniquely represented. A problem is overwritten when its score is below optimal. Hence $P(k)$. ■

4.5 Another point of view

In the previous section, we assume that overlaps were in fact same slices from our original array of card using the couple (index, lenght). We figured out tho that overlaps might also occurred and different segments of the array, if we could find some pattern repetition. For instance, let's take a board such as: $S = \{2,\ 3,\ 2,\ 3,\ 2,\ 3,\ 2,\ 3\}$

We can see that overlaps actually occurred on the couple (0, 2) and (6, 2) for instance. In order to deal with this kind of problem, we came up with a second solution which is based on a solution where we cached every different problem. The space complexity become then $\frac{N^2-N}{2}$ for the cache, and $\frac{N}{2}$ for the concrete computation. No algorithm details will be given in pseudo code, however you can find the Python algorithm in appendix.

is it useful? how often does this situation arise? is it worth the added compute time?

4.6 An improvement to the dynamic approach

The space occupied by the cache in the previous procedure is a $O(n^2)$ function. In the following algorithm we improve on that by making it linear.

what is this & why is it here?

```
procedure COMPUTE_BEST_PATH(subproblem P, int N)
    height ← (N + 1)/2 + 1;
    cache ← make_array(height, None);
    cache[0] ← P;
    global_best_score ← −∞;
    best_path ← make_array(height − 1, 0);
    stack ← Empty_stack();
    for level = 1...height − 1 do                              ▷ Initializing the cache
        cache[level] = generate_left_son(P, level − 1);
        best_path[level − 1] ← cache[level].card;
        push(stack, (generate_right_son(P, level − 1), level));      ▷ Put to
stack what needs to be computed in the future
    end for
    global_best_score ← −cache[height − 1].best_score;
    while stack is not empty do
        (S, level) ← pop(stack);
        if level = height − 1 then                                        ▷ leaf
            update(S, &cache, &best_path, &best_score_global);
        else
            if cache[level].index = S.index then
                pass                                           ▷ Already computed
            else
                ▷ Push two sons to the stack and replace the last problem on
the level with this one
                L ← left_rigth_son(S, level − 1);
                R ← generate_rigth_son(S, level − 1);
                push(stack, R, level + 1));
                push(stack, L, level + 1));
                cache[level] = S;
            end if
        end if
    end while
    return best_path;
end procedure
```

In the pseudocode above, we consider not the whole cache like in the previous algorithm, but only the set of last problems on the level. The *stack* variable contains information about problems, which need to be explored in the future. Procedure *update* updates *cache*, *best_path* and *best_score_global* with the leaf S as usual, performing merging if necessary.
Adding and removing elements to and from the stack happens in a constant time. The amount of problems evaluated is the same as in the previous version of the algorithm (although they are obtained in different order). The length of the stack is bound by a linear function in N. Therefore, this algorithm has a linear space complexity and a quadratic time complexity.