Computationally Hard Problems

Design of Randomized Algorithms – Gametrees

Carsten Witt

Institut for Matematik og Computer Science
Danmarks Tekniske Universitet

Fall 2016

Games

- Games play a central role in computer science (and other disciplines).
- We consider two-person games.
- The players alternatingly make moves.
- The game ends after a number of moves.
- When the game stops, one player has won (no ties).
- For the analysis, we assume that there always are exactly two alternative moves.

Game Trees

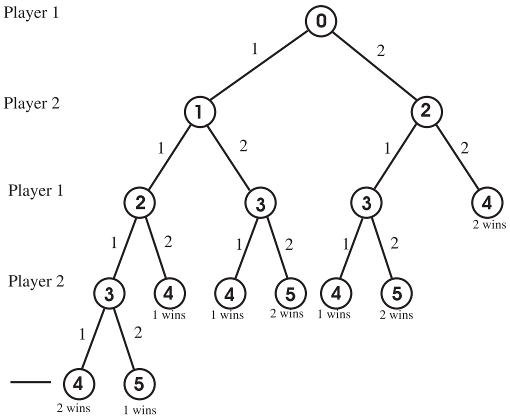
- A game tree is a (in our case binary) tree.
- A node corresponds to a *state* of the game.
- The levels are alternatingly assigned to the players.
- The root is the start state and is a Player 1 level.
- The children of a node contain the states which can be reached by a single move of the respective player.
- The leaves contain final states of the game.

Example "Get 4"

- An empty bowl is put on the table.
- Two players, Player 1 starts. Both players can watch the moves of the other.
- Move: The current player puts 1 or 2 DKK into the bowl.
- The game is over when there are at least 4 DKK in the bowl.
- If there are exactly 4 DKK in the bowl when the game ends, then the player who made the last move wins; otherwise the other player wins.

Example "Get 4" Game Tree

The game tree for "Get 4". The node labels denote the content of the bowl, the edge labels are the DKK added.

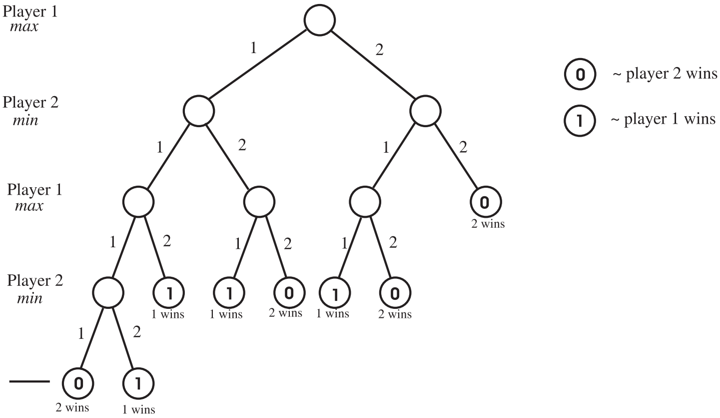


Evaluating a Game Tree

- Player i has a *winning strategy* if he can win the game regardless how the opponent plays.
- The nodes get new labels: 1 if Player 1 has a *winning strategy* from that node.
- A node gets label 0 if Player 2 has a winning strategy from there.
- The label of the root tells us who can win the game.
- In the beginning we can only label the leaves.
- The internal nodes receive their labels bottom-up.

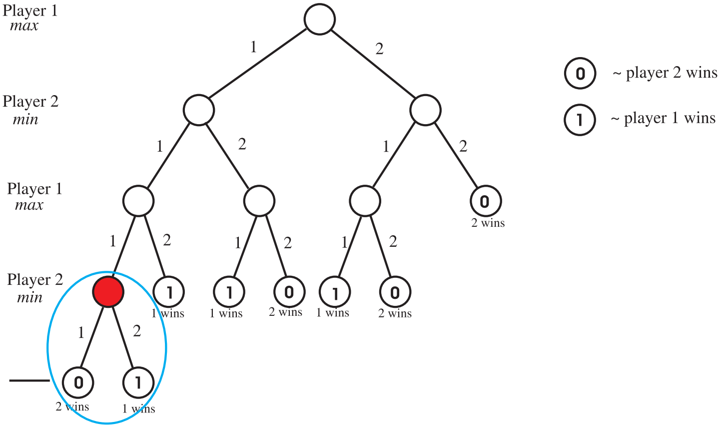
Example "Get 4" Game Tree

The leaves are labeled by the winner.



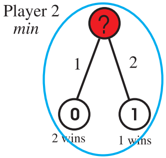
Example "Get 4" Game Tree

Process the internal nodes bottom-up.



Example “Get 4” Game Tree

Process the internal nodes bottom-up.



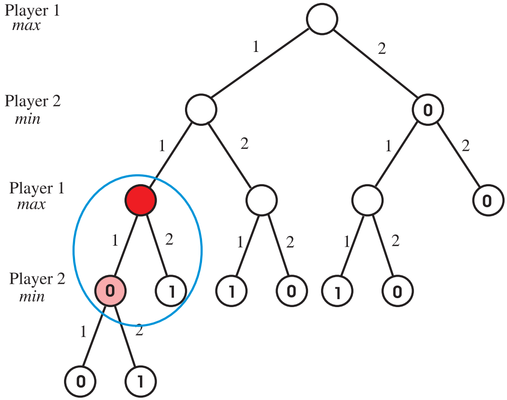
Consider the red Player 2 node.
Player 2 has two choices:
– to go to the left node and win
– to go to the right node and lose

Player 2 will choose the win-node.

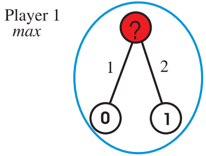
The red nodes gets label 0.

Example "Get 4" Game Tree

Process the internal nodes on the next level.



Example "Get 4" Game Tree



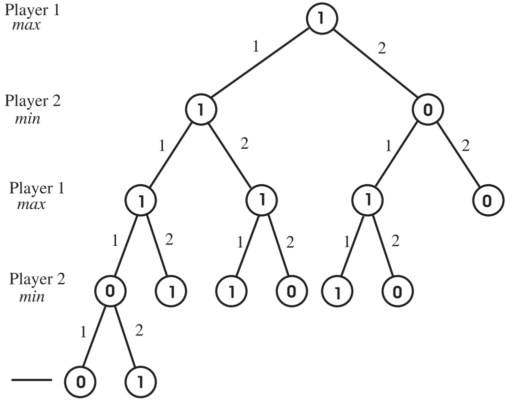
Consider the red Player 1 node.
Player 1 has two choices:
– to go to the left node and lose
– to go to the right node and win

Player 1 will choose the win-node.

The red node gets label 1.

Example “Get 4” Game Tree

Repeating this, we get the tree below. The root is labeled 1, i. e., Player 1 can always win.



General Case

Assume we are at node v which has children a and b.

The labels $\ell(a)$ and $\ell(b)$ of the children are known (either leaves or computed before).

If v is a Player 1-node: $\ell(v) = \max\{\ell(a), \ell(b)\}$

If v is a Player 2-node: $\ell(v) = \min\{\ell(a), \ell(b)\}$

A Recursive Implementation

```
proc EVAL(v)
if (v is a leaf) then
   return(ℓ(v))
else
   if (v is a Player 1 node) then
      return(max{EVAL(a), EVAL(b)})
   else
      return(min{EVAL(a), EVAL(b)})
   end if
end if
end proc
```

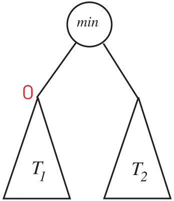
Preparing for Randomization

Observe: If we are at a $\min$-node with children a, b, and one child (say a) has label 0 then the label of v is determined

$$\ell(v) = 0 = \min\{0, \ell(b)\}$$

This is independent of the label of b.

(The case for a $\max$-node v is analogous.)



A Randomized Implementation

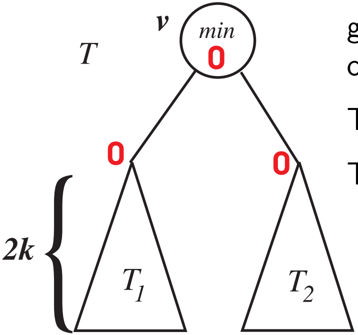
proc $\textsc{Eval-R}(v)$
let w_1 and w_2 be the children of v. (case for leaves is omitted)
pick one child with prob. $1/2$ at random; call this a and the other b.
$t \leftarrow \textsc{Eval-R}(a)$
if ((v is max-node) $\wedge$ $(t = 1)$) **then**
    return(1)
else
    if ((v is min-node) $\wedge$ $(t = 0)$) **then**
        return(0)
    else
        return$(\textsc{Eval-R}(b))$
    end if
end if

Analysis of the Algorithm

- The time for evaluation is proportional to the number of leaves the $\textsc{Eval-R}$ algorithm "looks at".
- We undertake an induction over the depth of the tree (that is, the number of edges from the root to the leaves).
- Consider a tree T of depth $2k+1$ and a $\min$-root.
- Let $M(T)$ be the expected number of leaves that algorithm $\textsc{Eval-R}$ looks at when evaluating T.
- Let $M_{\max}(k) := \max\{M(T) \mid T \text{ has depth } 2k.\}$.
- Note that $M_{\max}(k)$ refers to less deep trees.

Analysis of the Algorithm

Case 1: Both children have label 0.



It does not matter which child of v the algorithm picks first, it determines the label of v.

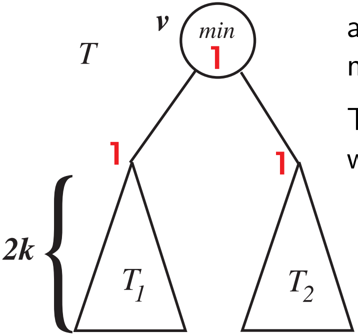
The child is picked with prob. $\frac{1}{2}$.

The other child needs not be evaluated.

$$M(T) = \frac{1}{2}M(T_1) + \frac{1}{2}M(T_2) \leq \frac{1}{2}M_{\max}(k) + \frac{1}{2}M_{\max}(k) = M_{\max}(k).$$

Analysis of the Algorithm

Case 2: Both children have label 1.



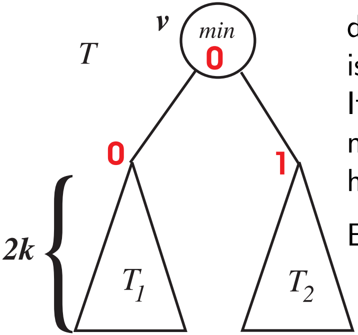
It does not matter which child of v the algorithm picks first, it **does not** determine the label of v.

The other child has to be evaluated as well.

$$M(T) = M(T_2) + M(T_1) \leq 2\, M_{\max}(k)$$

Analysis of the Algorithm

Case 3: One child has label 0, the other 1.



If the algorithm picks the 0-child first, it determines the label of v. The other child is not evaluated.
If the algorithm picks the 1-child first, the min is not determined. The other child has to be evaluated as well.

Each child is picked with prob. $\frac{1}{2}$.

$$M(T) = \frac{1}{2}M(T_1) + \frac{1}{2}\left(M(T_2) + M(T_1)\right) \leq 1.5\, M_{\max}(k)$$

Summary: Min-nodes

Altogether we have

Fact 1 If the label of v is 0, the time is at most $1.5\, M_{\max}(k)$.

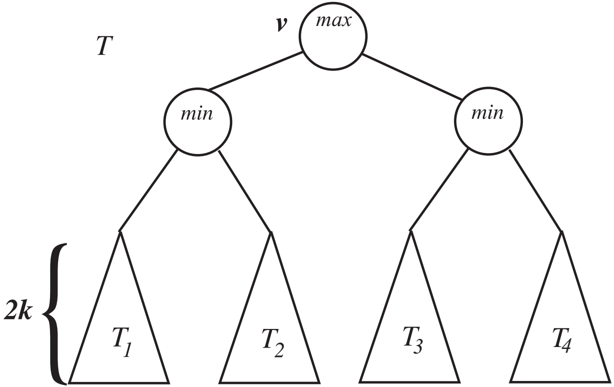
Fact 2 If the label of v is 1 then $2\, M_{\max}(k)$ is sufficient (but may also be required).

The above considerations apply in the case that v is max-node with the obvious changes.

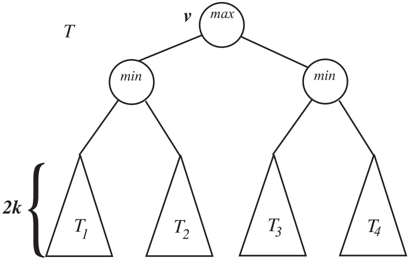
For the induction, however, we need a switch to max-nodes where we can reuse the results for min-nodes and where we (in the end) assume nothing on the label of the node.

Analysis of the Algorithm

Consider max-node v which is the root of a tree of depth $2k+2$:



Analysis of the Algorithm

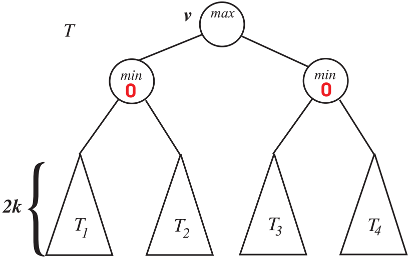


There are three cases

1. Both min-nodes compute 0.
2. One min-node computes 0, the other 1.
3. Both min-nodes compute 1.

Analysis of the Algorithm

Case 1:

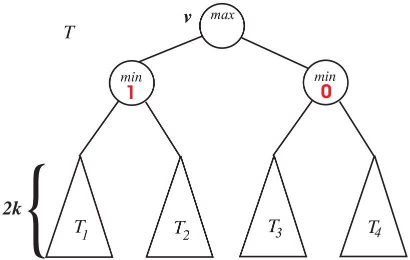


Both min-nodes compute a 0. Then the algorithm evaluates both. By Fact 1 we have

$$M(T) \leq 1.5M_{\max}(k) + 1.5M_{\max}(k) = 3M_{\max}(k)$$

Analysis of the Algorithm

Case 2:

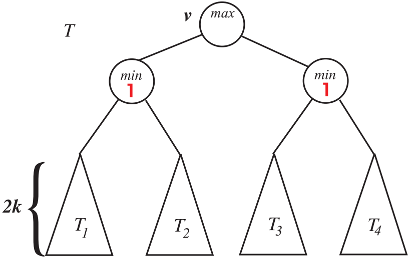


One min-node computes a 0, one 1. With prob. $1/2$ only one is evaluated; with prob. $1/2$ both are. By Facts 1 and 2 we have

$$M(T) \leq \frac{1}{2}\left(2M_{\max}(k) + (2+1.5)M_{\max}(k)\right) < 3M_{\max}(k)$$

Analysis of the Algorithm

Case 3:



Both min-nodes compute a 1. Then the algorithm evaluates only one. By Fact 2 we have

$$M(T) \leq 2M_{\max}(k) < 3M_{\max}(k)$$

Analysis of the Algorithm

Now conclude the proof by induction assuming

$$M_{\max}(k) \le 3^k$$

to get

$$M_{\max}(k+1) \le 3M_{\max}(k) \le 3 \cdot 3^k = 3^{k+1}. \quad \square$$

Note that $3^k \approx 2^{0.793(2k)}$.

Summary

- A complete game tree of depth $2n$ (that is n rounds) has $N = 2^{2n}$ leaves.
- The time for a complete evaluation is N.
- The randomized algorithm has expected running time $2^{0.793(2n)} = N^{0.793}$.
- Note that $N^{0.793} \ll N$.

Comparison

det. N	rand. $N^{0.793}$	gain $N/N^{0.793}$
10	6	1.61
100	39	2.59
1000	239	4.18
10000	1486	6.73
100000	9226	10.84
1000000	57280	17.46
10000000	355631	28.12
100000000	2208005	45.29
1000000000	13708818	72.95
10000000000	85113804	117.49

Computationally Hard Problems

Analysis of Randomized Search Heuristics – Foundations

Carsten Witt

Institut for Matematik og Computer Science
Danmarks Tekniske Universitet

Fall 2016

Using Heuristics = "Off-the-Shelf Algorithms"

Given a poorly understood optimization problem, we would ideally like to

- analyze it,
- design an efficient algorithm,
- prove its correctness and efficiency.

Practice:

- lacking resources (time, money, knowledge) for problem analysis and algorithm design,
- are fine with a "good" (rather than optimal) solution,
- problem is only given as a black box.

In these cases, we often need off-the-shelf heuristics/algorithms.

Black-Box Scenario

Many optimization problems have the following structure:

- set of solutions/search space S
- maximize objective function/fitness function $f\colon S \to \mathbb{R}$.

Black-Box Scenario: can only gain information on f by evaluating it

Often met in engineering, examples

- Edison's team run thousands of experiments to make a working light bulb 
- optimizing the parameters of a production process

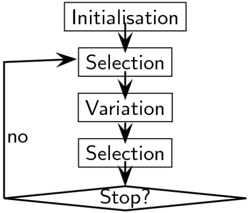
Here algorithms working in the black-box scenario are the only choice!

Randomized Search Heuristics

Our off-the-shelf algorithms are mostly randomized search heuristics

- making random decisions,
- working in the black-box scenario,
- having been used for decades.

Famous example: evolutionary algorithms (EA)



(also called bio-inspired/ nature-inspired search heuristic)

Our focus: analyze simple EAs and randomized local search

Two Very Simple Search Heuristics

Search space $\{0,1\}^n$ $\boxed{x_1}\boxed{x_2}\boxed{x_3}\cdots\boxed{x_n}$, "population" size 1, "offspring population" size 1, selection: "take the better"

Aim: maximize $f\colon \{0,1\}^n \to \mathbb{R}$

(1+1) Evolutionary Algorithm ((1+1) EA)

1. $t := 0$. Choose $x = (x_1, \ldots, x_n) \in \{0,1\}^n$ uniformly at random.
2. $y := x$
3. Independently for each bit in y: flip it with probability $\frac{1}{n}$ (mutation).
4. If $f(y) \geq f(x)$ Then $x := y$ (selection).
5. $t := t + 1$. Continue at line 2.

Two Very Simple Search Heuristics

Search space $\{0,1\}^n$ $\boxed{x_1}\boxed{x_2}\boxed{x_3}\cdots\boxed{x_n}$, "population" size 1, "offspring population" size 1, selection: "take the better"

Aim: maximize $f\colon \{0,1\}^n \to \mathbb{R}$

Randomized Local Search (RLS)

1. $t := 0$. Choose $x = (x_1, \dots, x_n) \in \{0,1\}^n$ uniformly at random.
2. $y := x$
3. Choose one bit in y uniformly and flip it. (mutation).
4. If $f(y) \geq f(x)$ Then $x := y$ (selection).
5. $t := t + 1$. Continue at line 2.

Extremely simple (good for analysis) and surprisingly efficient.

Focus: smallest t ("runtime") to reach optimal solution

Framework for Analysis

Given

- randomized search heuristic A
- fitness function f

study no. T of f-evaluations (black-box) until A finds optimum.

T is random variable

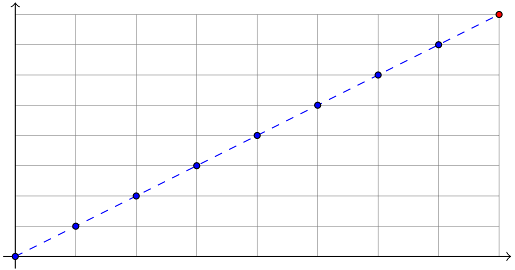
- ideally study whole distribution $\mathrm{Pr}(T \le t)$
- less ambitious: expectation $E(T)$
- even less ambitious (but feasible): bounds on $E(T)$

For (1+1) EA: T equals no. of iterations until optimum found.
Call this runtime/optimization time of (1+1) EA on f.

What problems/fitness functions to study?

Example Problems/Toy Problems

Most famous example problem: $\textsc{OneMax}(x_1, \ldots, x_n) = x_1 + \cdots + x_n$
(the heuristic does not know it is working on it)



Example Problems/Toy Problems

Most famous example problem: $\textsc{OneMax}(x_1, \ldots, x_n) = x_1 + \cdots + x_n$
(the heuristic does not know it is working on it)

Why should we care about such example problems?

- support analysis, help to develop analytical tools
- are easy to understand, are clearly structured
- make important aspects visible
- act as counterexamples
- help to discover general properties
- are important tools for further analysis $\rightarrow \mathcal{NP}$-hard problems
- positive results on easy examples make us trust the algorithm

A First Attempt

Theorem (General Upper Bound)

The expected optimization time of the (1+1) EA on an arbitrary function is $O(n^n)$.

Proof:

- Wait for current bitstring to mutate to optimum.
- At most n bits need to flip: probability at least $1/n^n$.
- Waiting time argument. $\square$

General upper bound for RLS does not exist (∞).

Upper Bound on OneMax

Theorem

The expected optimization time of RLS and the (1+1) EA on OneMax *is* $O(n \log n)$.

Proof for (1+1) EA:

- Consider $\phi \in \{0, \dots, n\}$: current no. one-bits
- Divide run: phase i starts when $\phi = i$ and ends when ϕ increases
- Sufficient for increase: flip a zero-bit, do not flip rest
- $\Pr(\text{increase } \phi \mid \phi = i) \geq \binom{n-i}{1} \cdot \frac{1}{n} \cdot \left(1 - \frac{1}{n}\right)^{n-1} \geq \frac{n-i}{en}$
- $E(\text{length of phase } i) \leq \frac{en}{n-i}$
- Expected duration of all phases $\leq \sum_{i=0}^{n-1} \frac{en}{n-i} = en \sum_{i=1}^{n} \frac{1}{i} = O(n \log n)$ since $\sum_{i=1}^{n} \frac{1}{i} \leq \ln n + 1$.

Upper Bound on OneMax

Theorem

The expected optimization time of RLS and the (1+1) EA on OneMax *is* $O(n \log n)$.

Proof for RLS

- Consider $\phi \in \{0, \dots, n\}$: current no. one-bits
- Divide run: phase i starts when $\phi = i$ and ends when ϕ increases
- Sufficient for increase: flip a zero-bit
- $\Pr(\text{increase } \phi \mid \phi = i) \ge \binom{n-i}{1} \cdot \frac{1}{n} \ge \frac{n-i}{n}$
- $E(\text{length of phase } i) \le \frac{n}{n-i}$
- Expected duration of all phases $\le \displaystyle\sum_{i=0}^{n-1} \frac{n}{n-i} = n \sum_{i=1}^{n} \frac{1}{i} = O(n \log n)$ since $\sum_{i=1}^{n} \frac{1}{i} \le \ln n + 1$.