Computationally Hard Problems

Design of Randomized Algorithms – Game Trees

Carsten Witt

Institut for Matematik og Computer Science
Danmarks Tekniske Universitet

Fall 2020

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

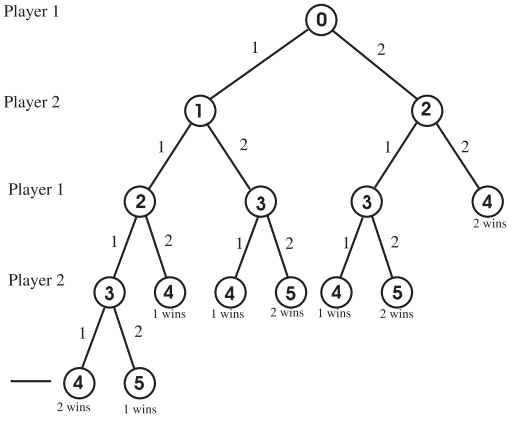
- A game tree is a (in our case binary) tree.
- A node corresponds to a *state* of the game.
- The levels are alternatingly assigned to the players.
- The root is the start state and is a Player 1 level.
- The children of a node contain the states which can be reached by a single move of the respective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

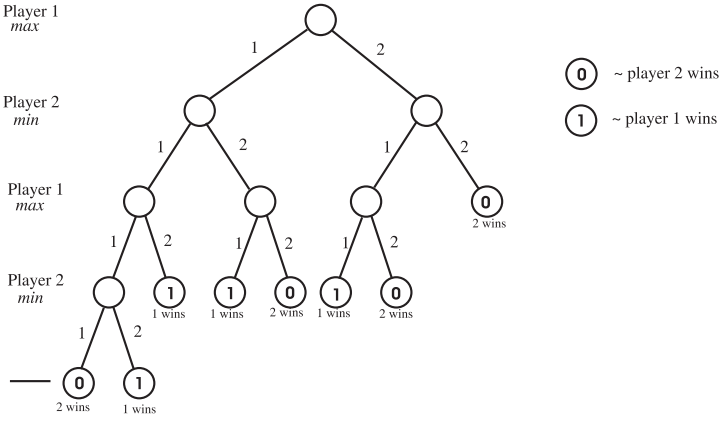


Evaluating a Game Tree

- 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.

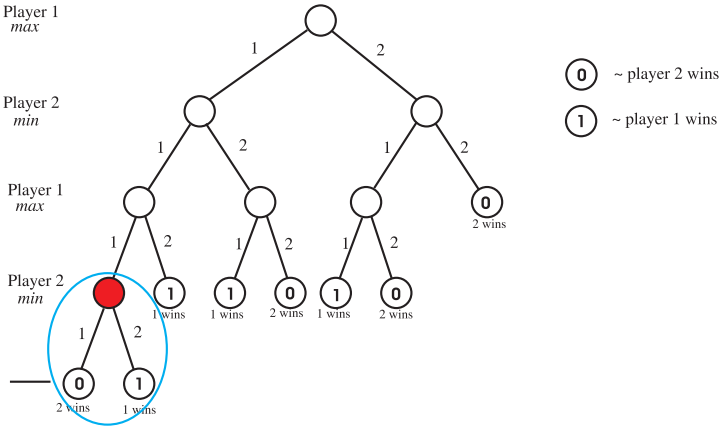
Example "Get 4" Game Tree

The leaves are labeled by the winner.



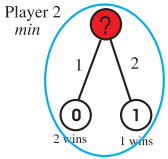
Example "Get 4" Game Tree

Process the internal nodes bottom-up.



Example "Get 4" Game Tree

Process the internal nodes bottom-up.



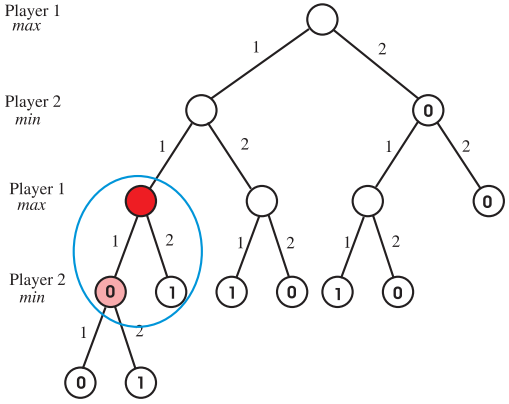
Consider the red Player 2 node.
Player 2 has two choices:
– to go to the left node and win
– to go to the right node and lose

Player 2 will choose the win-node.

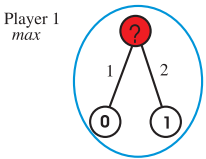
The red nodes gets label 0.

Example "Get 4" Game Tree

Process the internal nodes on the next level.



Example "Get 4" Game Tree



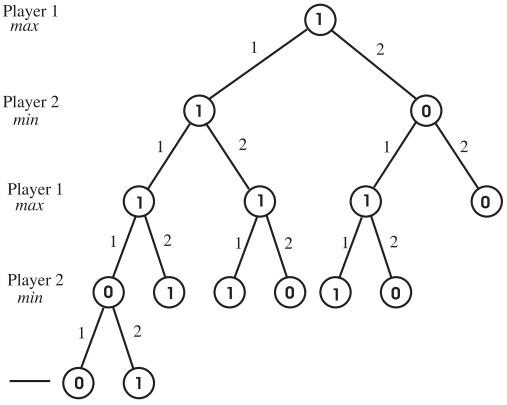
Consider the red Player 1 node.
Player 1 has two choices:
– to go to the left node and lose
– to go to the right node and win

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 $\textsc{Eval}(v)$
if (v is a leaf) **then**
  return$(\ell(v))$
else
  if (v is a Player 1 node) **then**
    return$(\max\{\textsc{Eval}(a), \textsc{Eval}(b)\})$
  else
    return$(\min\{\textsc{Eval}(a), \textsc{Eval}(b)\})$
  end if
end if
end proc

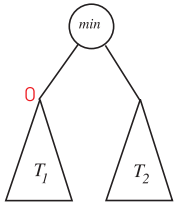
Preparing for Randomization

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

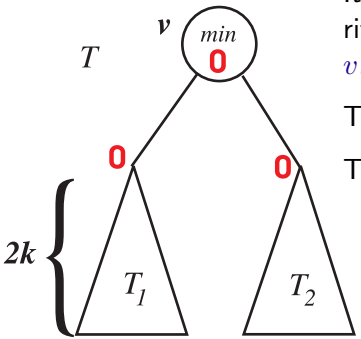
proc EVAL-R(v)
let w_1 and w_2 be the children of v. (case for leaves is omitted)
pick one child with prob. $1/2$ at random; call this a and the other b.
$t \leftarrow$ EVAL-R(a)
if $((v$ is max-node$) \wedge (t = 1))$ **then**
  return(1)
else
  if $((v$ is min-node$) \wedge (t = 0))$ **then**
    return(0)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

Analysis of the Algorithm

- The time for evaluation is proportional to the number of leaves the 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).
- Let $M(T)$ be the expected number of leaves that algorithm EVAL-R looks at when evaluating a tree T.
- Let $M_{\max}(k) := \max\{M(T) \mid T \text{ has depth } 2k.\}$.
- The aim is to show $M_{\max}(k) \leq 3^k$ using induction on k.
- We next consider a tree T of depth $2k+1$ and a min-root.
- Note that $M_{\max}(k)$ refers to trees less deep than T.

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.

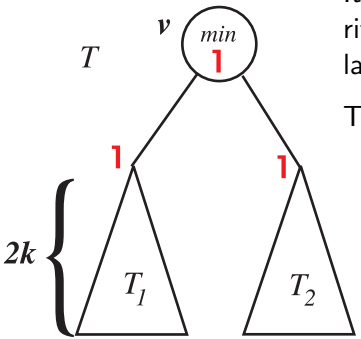
The child is picked with prob. $\frac{1}{2}$.

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.



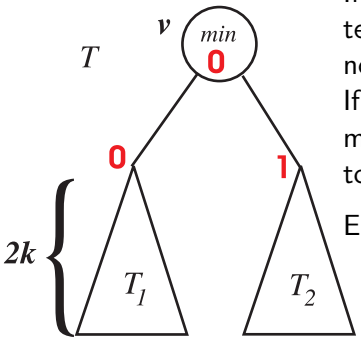
It does not matter which child of v the algorithm picks first, it **does not** determine the label of v.

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 $1.5\,M_{\max}(k)$.

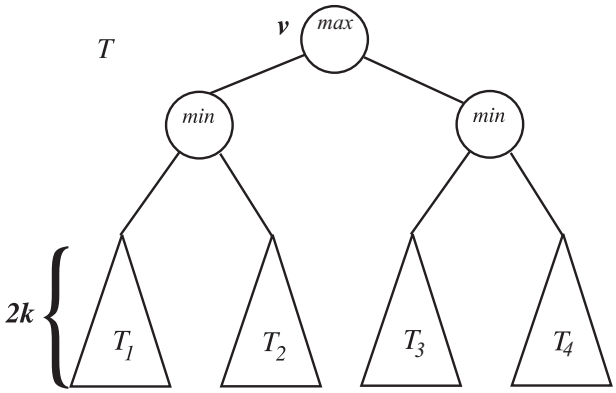
Fact 2 If the label of v is 1 then $2\,M_{\max}(k)$ is sufficient (but may also be required).

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

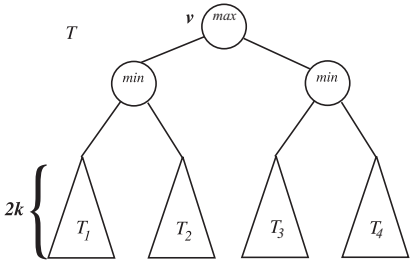
For the induction, however, we need a switch to max-nodes where we can reuse the results for min-nodes and where we (in the end) assume nothing on the label of the node.

Analysis of the Algorithm

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



Analysis of the Algorithm

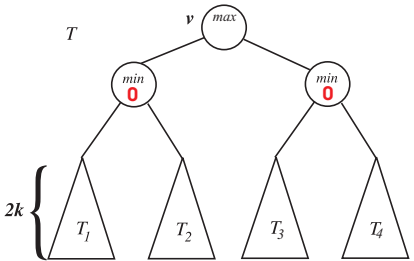


There are three cases

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:

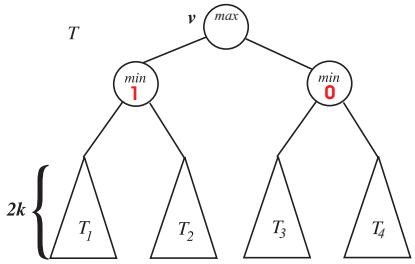


Both min-nodes compute a 0. Then the algorithm evaluates both. By Fact 1 we have

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

Analysis of the Algorithm

Case 2:

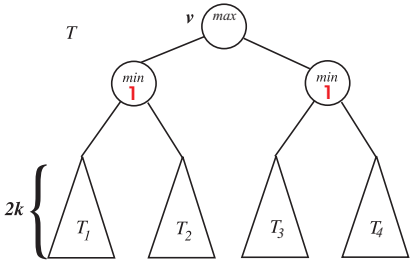


One min-node computes a 0, one 1. With prob. $1/2$ only one is evaluated; with prob. $1/2$ both are. By Facts 1 and 2 we have

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

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

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

to get

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

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

sectionRandomized Search Heuristics

Computationally Hard Problems

Randomized Search Heuristics – Introduction

Carsten Witt

Institut for Matematik og Computer Science
Danmarks Tekniske Universitet

Fall 2020

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

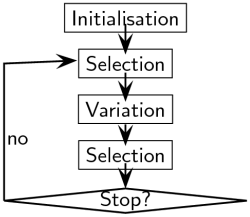
Here algorithms working in the black-box scenario are the only choice!

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)

Components of Evolutionary Algorithms (1/5)

Search space S. A typically finite set, in which certain solution (e. g., of optimal quality) are sought. Examples:

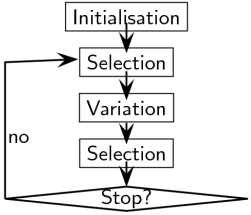
- $\{0,1\}^n$, all bit strings of length n. In this way, an assignment of truth values to n boolean variables can be coded.
- $\Pi_n :=$ set of all permutations on $\{1,\ldots,n\}$. In this way, a solution to the TSP can be coded.

Fitness function f. Maps $x \in S \mapsto f(x) \in \mathbb{R}$, i. e., returns a value for every search point. Examples:

- $f\colon \{0,1\}^n \to \mathbb{R}$ returns the number of satisfied clauses w. r. t. the MAX-3-SAT problem, using the search space $\{0,1\}^n$ as described above.
- $f\colon \Pi_n \to \mathbb{R}$ returns the length of a TSP-tour.

Components of Evolutionary Algorithms (2/5)

Population and general procedure. Search points from S are called *individuals* and collected in a *population*. Population size is denoted by μ. A population P is transformed to a population P' by applying selection and variation (including mutation and crossover). All explained below.



One cycle, transforming a population to a new one, is called *generation*.

Components of Evolutionary Algorithms (3/5)

Selection. Individuals from current population are chosen to be varied. Typically, selection is stochastic and individuals of better fitness are often preferred, in any case not disfavored. In a generation, typically selection is applied μ times.

Examples:

- Uniform selection. Every individual has the same probability of being chosen.
- Tournament selection. Temporarily choose k individuals uniformly at random without replacement and choose one of best fitness from them.
- Fitness-proportional selection Given $P = \{x_1, \ldots, x_\mu\}$, individual x_i is chosen with probability $\frac{f(x_i)}{\sum_{j=1}^{\mu} f(x_j)}$. Requires non-negative f and makes only sense if f has to be maximized.

Usually, only the selection operators of an EA use the fitness values of the individuals.

Components of Evolutionary Algorithms (4/5)

Mutation. Transforms an individual $x \in S$ into a new individual $x' \in S$, typically in a randomized way.

Examples:

- Bit-flip mutation. Given $x = (x_1, \ldots, x_n) \in \{0,1\}^n$, an index $k \in \{1, \ldots, n\}$ is picked uniformly at random. x' is obtained by flipping bit k in x, formally $x' = (x_1, \ldots, x_{k-1}, 1 - x_k, x_{k+1}, \ldots, x_n)$.
- Swap mutation Given $x = (i_1, \ldots, i_n) \in \Pi_n$, two indices $k < \ell \in \{1, \ldots, n\}$ are picked u. a. r. and x' is obtained by swapping the elements at these two places. Formally, $x' = (i_1, \ldots, i_{k-1}, i_\ell, i_{k+1}, \ldots, i_{\ell-1}, i_k, i_{\ell+1}, \ldots, i_n)$.

There are many more mutation operators.

Components of Evolutionary Algorithms (5/5)

Crossover. Transforms two individuals $x, y \in S$ into a new individual $z \in S$ (or two new individuals z and z') by recombining them.

Example:

- Uniform crossover. Given $x, y \in \{0,1\}^n$, create z by uniformly taking each element from either x or y. Formally, $z_i = x_i$ with probability $1/2$ and $z_i = y_i$ with probability $1/2$, independently for $i = 1, \ldots, n$. If two offspring are required, let z' be the string consisting of the elements not chosen.

It is not obvious how to define "natural" crossover operators for Π_n (permutations).

A Typical Evolutionary Algorithm

Initialize population P_0 of size μ. Set $t \leftarrow 0$.
while stopping criterion not fulfilled **do**
  for $i \leftarrow 1, \ldots, \mu$ **do**
    Choose two individuals x and y from P_t by applying some selection operator.
    Create z by applying some crossover operator to x and y.
    Create z' by applying some mutation operator to z.
    Add z' to P_{t+1}. (assumption: P_{t+1} initially empty)
  end for
  $t \leftarrow t + 1$.
end while

The scheme is typical but does not cover all variants of EAs. E. g., varying population size is not allowed. Also often mutation and crossover are not necessarily applied in every step but only depending a the so-called mutation and crossover probability.

Operators for the TSP: Mutation

Mutation of a tour $(i_1, \ldots, i_n) \in \Pi_n$.

- Swap mutation usually does not give good results on practical TSP instances. For example, it does not exploit symmetry in the distances ($d(i,j) \approx d(j,i)$).
- Better: Jump: Pick two indices $k < \ell$ and let element at pos. k jump to pos. ℓ, shift intermediate values to the left: $(1,3,4,6,2,5) \mapsto (1,4,6,2,3,5)$ for $k = 2$, $\ell = 5$
- More popular: 2-OPT: Pick two *edges* and connect their endpoints "crossing over"