Part I

Simple Methods of design of Randomized Algorithms

Chapter 4. SIMPLE METHODS for DESIGN of RANDOMIZED ALGORITHMS

Chapter 4. SIMPLE METHODS for DESIGN of RANDOMIZED ALGORITHMS

In this chapter we present a new way how to see randomized algorithms and an application of some simple basic techniques how to design randomized algorithms.

Chapter 4. SIMPLE METHODS for DESIGN of RANDOMIZED ALGORITHMS

In this chapter we present a new way how to see randomized algorithms and an application of some simple basic techniques how to design randomized algorithms.

Especially we deal with:

Chapter 4. SIMPLE METHODS for DESIGN of RANDOMIZED ALGORITHMS

In this chapter we present a new way how to see randomized algorithms and an application of some simple basic techniques how to design randomized algorithms.

Especially we deal with:

- A unified approach to deterministic, randomized and quantum algorithms

Chapter 4. SIMPLE METHODS for DESIGN of RANDOMIZED ALGORITHMS

In this chapter we present a new way how to see randomized algorithms and an application of some simple basic techniques how to design randomized algorithms.

Especially we deal with:

- A unified approach to deterministic, randomized and quantum algorithms
- Application of the linearity of expectations method

Chapter 4. SIMPLE METHODS for DESIGN of RANDOMIZED ALGORITHMS

In this chapter we present a new way how to see randomized algorithms and an application of some simple basic techniques how to design randomized algorithms.

Especially we deal with:

- A unified approach to deterministic, randomized and quantum algorithms
- Application of the linearity of expectations method
- Design of randomized algorithms for games trees.

PROLOGUE

A way to see basics of deterministic, randomized and quantum computations and their differences.

MATHEMATICAL VIEWS of COMPUTATION 1/3

MATHEMATICAL VIEWS of COMPUTATION 1/3

Let us consider an n bits strings set $S \subset \{0,1\}^n$.

MATHEMATICAL VIEWS of COMPUTATION 1/3

Let us consider an n bits strings set $S \subset \{0,1\}^n$.

To describe a **deterministic computation** on S we need to specify:

MATHEMATICAL VIEWS of COMPUTATION 1/3

Let us consider an n bits strings set $S \subset \{0,1\}^n$.

To describe a **deterministic computation** on S we need to specify:

an initial state - by an n-bit string - say s_0

MATHEMATICAL VIEWS of COMPUTATION 1/3

Let us consider an n bits strings set $S \subset \{0, 1\}^n$.

To describe a **deterministic computation** on S we need to specify:

an initial state - by an n-bit string - say s_0

and an **evolution** (computation) mapping $E : S \to S$ which can be described by a vector of the length 2^n, the elements and indices of which are n-bit strings.

MATHEMATICAL VIEWS of COMPUTATION 1/3

Let us consider an n bits strings set $S \subset \{0,1\}^n$.

To describe a **deterministic computation** on S we need to specify:

an initial state - by an n-bit string - say s_0

and an **evolution** (computation) mapping $E : S \to S$ which can be described by a vector of the length 2^n, the elements and indices of which are n-bit strings.

A **computation step** is then an application of the evolution mapping E to the current state represented by an n-bit string s.

MATHEMATICAL VIEWS of COMPUTATION 1/3

Let us consider an n bits strings set $S \subset \{0,1\}^n$.

To describe a **deterministic computation** on S we need to specify:

an initial state - by an n-bit string - say s_0

and an **evolution** (computation) mapping $E : S \rightarrow S$ which can be described by a vector of the length 2^n, the elements and indices of which are n-bit strings.

A **computation step** is then an application of the evolution mapping E to the current state represented by an n-bit string s.

However, for any at least a bit significant task, the number of bits needed to describe such an evolution mapping, $n2^n$, is much too big.

MATHEMATICAL VIEWS of COMPUTATION 1/3

Let us consider an n bits strings set $S \subset \{0,1\}^n$.

To describe a **deterministic computation** on S we need to specify:

an initial state - by an n-bit string - say s_0

and an **evolution** (computation) mapping $E : S \rightarrow S$ which can be described by a vector of the length 2^n, the elements and indices of which are n-bit strings.

A **computation step** is then an application of the evolution mapping E to the current state represented by an n-bit string s.

However, for any at least a bit significant task, the number of bits needed to describe such an evolution mapping, $n2^n$, is much too big. **The task of programming is then/therefore** to replace an application of such an enormously huge mapping by an application of a much shorter circuit/program.

MATHEMATICAL VIEWS of COMPUTATION 2/3

MATHEMATICAL VIEWS of COMPUTATION 2/3

To describe a **randomized computation** we need;

MATHEMATICAL VIEWS of COMPUTATION 2/3

To describe a **randomized computation** we need;

1:) to specify an **initial probability distribution** on all n-bit strings.

MATHEMATICAL VIEWS of COMPUTATION 2/3

To describe a **randomized computation** we need;

1:) to specify an **initial probability distribution** on all n-bit strings. That can be done by a vector of length 2^n, indexed by n-bit strings, the elements of which are non-negative numbers that sum up to 1.

MATHEMATICAL VIEWS of COMPUTATION 2/3

To describe a **randomized computation** we need;

1:) to specify an **initial probability distribution** on all n-bit strings. That can be done by a vector of length 2^n, indexed by n-bit strings, the elements of which are non-negative numbers that sum up to 1.

2:) to specify a **randomized evolution**, which has to be done, in case of a homogeneous evolution, by a $2^n \times 2^n$ matrix A of conditional probabilities for obtaining a new state/string from an old state/string.

MATHEMATICAL VIEWS of COMPUTATION 2/3

To describe a **randomized computation** we need;

1:) to specify an **initial probability distribution** on all n-bit strings. That can be done by a vector of length 2^n, indexed by n-bit strings, the elements of which are non-negative numbers that sum up to 1.

2:) to specify a **randomized evolution**, which has to be done, in case of a homogeneous evolution, by a $2^n \times 2^n$ matrix A of conditional probabilities for obtaining a new state/string from an old state/string.

The matrix A has to be **stochastic** - all columns have to sum up to one and $A[i, j]$ is a probability of going from a string representing j to a string representing i.

MATHEMATICAL VIEWS of COMPUTATION 2/3

To describe a **randomized computation** we need;

1:) to specify an **initial probability distribution** on all n-bit strings. That can be done by a vector of length 2^n, indexed by n-bit strings, the elements of which are non-negative numbers that sum up to 1.

2:) to specify a **randomized evolution**, which has to be done, in case of a homogeneous evolution, by a $2^n \times 2^n$ matrix A of conditional probabilities for obtaining a new state/string from an old state/string.

The matrix A has to be **stochastic** - all columns have to sum up to one and $A[i,j]$ is a probability of going from a string representing j to a string representing i.

To perform a computation step, one then needs to multiply by A the 2^n-elements vector specifying the current probability distribution on 2^n states.

MATHEMATICAL VIEWS of COMPUTATION 2/3

To describe a **randomized computation** we need;

1:) to specify an **initial probability distribution** on all n-bit strings. That can be done by a vector of length 2^n, indexed by n-bit strings, the elements of which are non-negative numbers that sum up to 1.

2:) to specify a **randomized evolution**, which has to be done, in case of a homogeneous evolution, by a $2^n \times 2^n$ matrix A of conditional probabilities for obtaining a new state/string from an old state/string.

The matrix A has to be **stochastic** - all columns have to sum up to one and $A[i, j]$ is a probability of going from a string representing j to a string representing i.

To perform a computation step, one then needs to multiply by A the 2^n-elements vector specifying the current probability distribution on 2^n states.

However, for any nontrivial problem the number 2^n is larger than the number of particles in the universe.

MATHEMATICAL VIEWS of COMPUTATION 2/3

To describe a **randomized computation** we need;

1:) to specify an **initial probability distribution** on all n-bit strings. That can be done by a vector of length 2^n, indexed by n-bit strings, the elements of which are non-negative numbers that sum up to 1.

2:) to specify a **randomized evolution**, which has to be done, in case of a homogeneous evolution, by a $2^n \times 2^n$ matrix A of conditional probabilities for obtaining a new state/string from an old state/string.

The matrix A has to be **stochastic** - all columns have to sum up to one and $A[i,j]$ is a probability of going from a string representing j to a string representing i.

To perform a computation step, one then needs to multiply by A the 2^n-elements vector specifying the current probability distribution on 2^n states.

However, for any nontrivial problem the number 2^n is larger than the number of particles in the universe. Therefore, **the task of programming is to design a small circuit/program** that can implement such a multiplication by a matrix of an enormous size.

In case of **quantum computation** on n quantum bits:

---

[1] A matrix A is usually called unitary if its inverse matrix can be obtained from A by transposition around the main diagonal and replacement of each element by its complex conjugate.

MATHEMATICAL VIEWS of COMPUTATION 3/3

In case of **quantum computation** on n quantum bits:

1:) **Initial state** has to be given by an 2^n vector of complex numbers (probability amplitudes) the sum of the squares of which is one.

[1] A matrix A is usually called unitary if its inverse matrix can be obtained from A by transposition around the main diagonal and replacement of each element by its complex conjugate.

MATHEMATICAL VIEWS of COMPUTATION 3/3

In case of **quantum computation** on n quantum bits:

1:) **Initial state** has to be given by an 2^n vector of complex numbers (probability amplitudes) the sum of the squares of which is one.

2:) Homogeneous **quantum evolution** has to be described by an $2^n \times 2^n$ unitary matrix of complex numbers - at which inner products of any two different columns and any two different rows are 0.[1]

[1] A matrix A is usually called unitary if its inverse matrix can be obtained from A by transposition around the main diagonal and replacement of each element by its complex conjugate.

MATHEMATICAL VIEWS of COMPUTATION 3/3

In case of **quantum computation** on n quantum bits:

1:) **Initial state** has to be given by an 2^n vector of complex numbers (probability amplitudes) the sum of the squares of which is one.

2:) Homogeneous **quantum evolution** has to be described by an $2^n \times 2^n$ unitary matrix of complex numbers - at which inner products of any two different columns and any two different rows are 0.[1]

Concerning a **computation step**, this has to be again a multiplication of a vector of the probability amplitudes, representing the current state, by a very huge $2^n \times 2^n$ unitary matrix which has to be realized by a "small" quantum circuit (program).

[1] A matrix A is usually called unitary if its inverse matrix can be obtained from A by transposition around the main diagonal and replacement of each element by its complex conjugate.

LINEARITY OF EXPECTATIONS

LINEARITY OF EXPECTATIONS

A very simple, but very often very useful, fact is that for any random variables $X_1, X_2, \ldots$ it holds

$$\mathbf{E}[\sum_i X_i] = \sum_i \mathbf{E}[X_i].$$

LINEARITY OF EXPECTATIONS

A very simple, but very often very useful, fact is that for any random variables $X_1, X_2, \ldots$ it holds

$$\mathbf{E}[\sum_i X_i] = \sum_i \mathbf{E}[X_i].$$

even if X_i are dependent and dependencies among X_i's are very complex.

LINEARITY OF EXPECTATIONS

A very simple, but very often very useful, fact is that for any random variables $X_1, X_2, \ldots$ it holds

$$\mathbf{E}[\sum_i X_i] = \sum_i \mathbf{E}[X_i].$$

even if X_i are dependent and dependencies among X_i's are very complex.

Example: A ship arrives at a port, and all 40 sailors on board go ashore to have fun. At night, all sailors return to the ship and, being drunk, each chooses randomly a cabin to sleep in.

LINEARITY OF EXPECTATIONS

A very simple, but very often very useful, fact is that for any random variables $X_1, X_2, \ldots$ it holds

$$\mathbf{E}[\sum_i X_i] = \sum_i \mathbf{E}[X_i].$$

even if X_i are dependent and dependencies among X_i's are very complex.

Example: A ship arrives at a port, and all 40 sailors on board go ashore to have fun. At night, all sailors return to the ship and, being drunk, each chooses randomly a cabin to sleep in. Now comes the **question:** *What is the expected number of sailors sleeping in their own cabins?*

LINEARITY OF EXPECTATIONS

A very simple, but very often very useful, fact is that for any random variables $X_1, X_2, \ldots$ it holds

$$\mathbf{E}[\sum_i X_i] = \sum_i \mathbf{E}[X_i].$$

even if X_i are dependent and dependencies among X_i's are very complex.

Example: A ship arrives at a port, and all 40 sailors on board go ashore to have fun. At night, all sailors return to the ship and, being drunk, each chooses randomly a cabin to sleep in. Now comes the **question:** *What is the expected number of sailors sleeping in their own cabins?*

Solution: Let X_i be a random variable, so called *(indicator variable)*, which has value 1 if the i-th sailor chooses his own cabin, and 0 otherwise.

LINEARITY OF EXPECTATIONS

A very simple, but very often very useful, fact is that for any random variables $X_1, X_2, \ldots$ it holds

$$\mathbf{E}[\sum_i X_i] = \sum_i \mathbf{E}[X_i].$$

even if X_i are dependent and dependencies among X_i's are very complex.

Example: A ship arrives at a port, and all 40 sailors on board go ashore to have fun. At night, all sailors return to the ship and, being drunk, each chooses randomly a cabin to sleep in. Now comes the **question:** *What is the expected number of sailors sleeping in their own cabins?*

Solution: Let X_i be a random variable, so called *(indicator variable)*, which has value 1 if the i-th sailor chooses his own cabin, and 0 otherwise.

Expected number of sailors who get to their own cabin is

$$\mathbf{E}[\sum_{i=1}^{40} X_i] = \sum_{i=1}^{40} \mathbf{E}[X_i]$$

LINEARITY OF EXPECTATIONS

A very simple, but very often very useful, fact is that for any random variables $X_1, X_2, \ldots$ it holds

$$\mathbf{E}[\sum_i X_i] = \sum_i \mathbf{E}[X_i].$$

even if X_i are dependent and dependencies among X_i's are very complex.

Example: A ship arrives at a port, and all 40 sailors on board go ashore to have fun. At night, all sailors return to the ship and, being drunk, each chooses randomly a cabin to sleep in. Now comes the **question:** *What is the expected number of sailors sleeping in their own cabins?*

Solution: Let X_i be a random variable, so called *(indicator variable)*, which has value 1 if the i-th sailor chooses his own cabin, and 0 otherwise.

Expected number of sailors who get to their own cabin is

$$\mathbf{E}[\sum_{i=1}^{40} X_i] = \sum_{i=1}^{40} \mathbf{E}[X_i]$$

Since cabins are chosen randomly $\mathbf{E}[X_i] =$

LINEARITY OF EXPECTATIONS

A very simple, but very often very useful, fact is that for any random variables $X_1, X_2, \ldots$ it holds

$$\mathbf{E}[\sum_i X_i] = \sum_i \mathbf{E}[X_i].$$

even if X_i are dependent and dependencies among X_i's are very complex.

Example: A ship arrives at a port, and all 40 sailors on board go ashore to have fun. At night, all sailors return to the ship and, being drunk, each chooses randomly a cabin to sleep in. Now comes the **question:** *What is the expected number of sailors sleeping in their own cabins?*

Solution: Let X_i be a random variable, so called *(indicator variable)*, which has value 1 if the i-th sailor chooses his own cabin, and 0 otherwise.

Expected number of sailors who get to their own cabin is

$$\mathbf{E}[\sum_{i=1}^{40} X_i] = \sum_{i=1}^{40} \mathbf{E}[X_i]$$

Since cabins are chosen randomly $\mathbf{E}[X_i] = \frac{1}{40}$

LINEARITY OF EXPECTATIONS

A very simple, but very often very useful, fact is that for any random variables $X_1, X_2, \ldots$ it holds

$$\mathbf{E}[\sum_i X_i] = \sum_i \mathbf{E}[X_i].$$

even if X_i are dependent and dependencies among X_i's are very complex.

Example: A ship arrives at a port, and all 40 sailors on board go ashore to have fun. At night, all sailors return to the ship and, being drunk, each chooses randomly a cabin to sleep in. Now comes the **question:** *What is the expected number of sailors sleeping in their own cabins?*

Solution: Let X_i be a random variable, so called *(indicator variable)*, which has value 1 if the i-th sailor chooses his own cabin, and 0 otherwise.

Expected number of sailors who get to their own cabin is

$$\mathbf{E}[\sum_{i=1}^{40} X_i] = \sum_{i=1}^{40} \mathbf{E}[X_i]$$

Since cabins are chosen randomly $\mathbf{E}[X_i] = \frac{1}{40}$ and

LINEARITY OF EXPECTATIONS

A very simple, but very often very useful, fact is that for any random variables $X_1, X_2, \ldots$ it holds

$$\mathbf{E}[\sum_i X_i] = \sum_i \mathbf{E}[X_i].$$

even if X_i are dependent and dependencies among X_i's are very complex.

Example: A ship arrives at a port, and all 40 sailors on board go ashore to have fun. At night, all sailors return to the ship and, being drunk, each chooses randomly a cabin to sleep in. Now comes the **question:** *What is the expected number of sailors sleeping in their own cabins?*

Solution: Let X_i be a random variable, so called *(indicator variable)*, which has value 1 if the i-th sailor chooses his own cabin, and 0 otherwise.

Expected number of sailors who get to their own cabin is

$$\mathbf{E}[\sum_{i=1}^{40} X_i] = \sum_{i=1}^{40} \mathbf{E}[X_i]$$

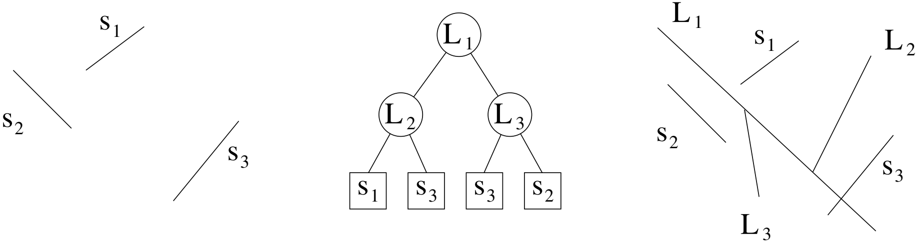
Since cabins are chosen randomly $\mathbf{E}[X_i] = \frac{1}{40}$ and $\mathbf{E}[\sum_{i=1}^{40} X_i] = 40.\frac{1}{40} = 1$.

EXAMPLE - BINARY PARTITION of a SET of LINE SEGMENTS 1/3

Problem Given a set $S = \{s_1, \ldots, s_n\}$ of non-intersecting line segments, find a partition of the plane such that every region will contain at most one line segment (or at most a part of a line segment).

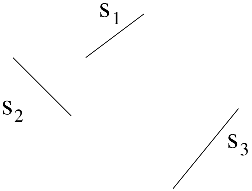
EXAMPLE - BINARY PARTITION of a SET of LINE SEGMENTS 1/3

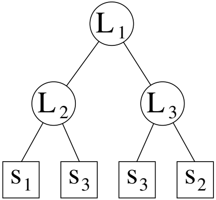
Problem Given a set $S = \{s_1, \ldots, s_n\}$ of non-intersecting line segments, find a partition of the plane such that every region will contain at most one line segment (or at most a part of a line segment).

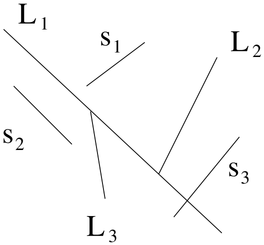


EXAMPLE - BINARY PARTITION of a SET of LINE SEGMENTS 1/3

Problem Given a set $S = \{s_1, \ldots, s_n\}$ of non-intersecting line segments, find a partition of the plane such that every region will contain at most one line segment (or at most a part of a line segment).



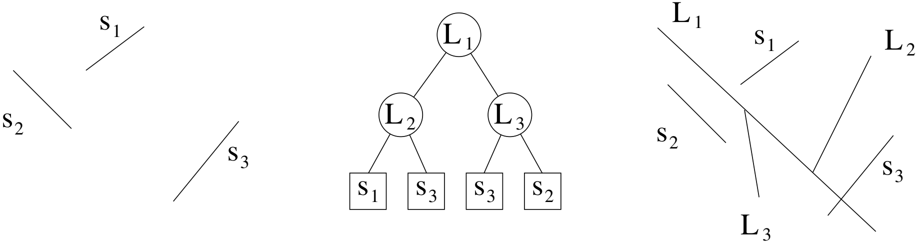




A (binary) partition will be described by a binary tree + additional information (about nodes).

EXAMPLE - BINARY PARTITION of a SET of LINE SEGMENTS 1/3

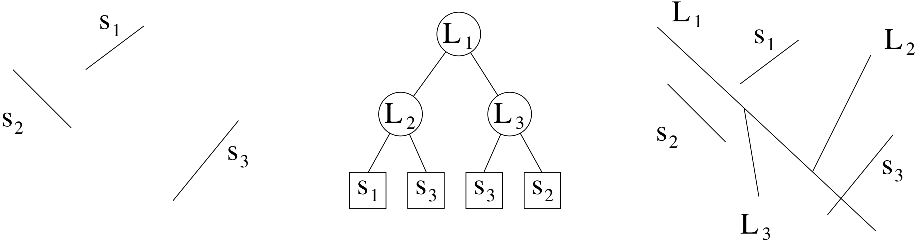
Problem Given a set $S = \{s_1, \dots, s_n\}$ of non-intersecting line segments, find a partition of the plane such that every region will contain at most one line segment (or at most a part of a line segment).



A (binary) partition will be described by a binary tree + additional information (about nodes). With each node v a region r_v of the plane will be associated (the whole plane will be represented by the root) and also a line L_v intersecting r_v.

EXAMPLE - BINARY PARTITION of a SET of LINE SEGMENTS 1/3

Problem Given a set $S = \{s_1, \ldots, s_n\}$ of non-intersecting line segments, find a partition of the plane such that every region will contain at most one line segment (or at most a part of a line segment).



A (binary) partition will be described by a binary tree + additional information (about nodes). With each node v a region r_v of the plane will be associated (the whole plane will be represented by the root) and also a line L_v intersecting r_v.

Each line L_v will partition the region r_v into two regions $r_{l,v}$ and $r_{r,v}$ which correspond to two children of v - to the left and right one.

EXAMPLE - BINARY PARTITION of a SET of LINE SEGMENTS 2/3

EXAMPLE - BINARY PARTITION of a SET of LINE SEGMENTS 2/3

Notation: $l(s_i)$ will denote a **line-extension of the segment** s_i.

EXAMPLE - BINARY PARTITION of a SET of LINE SEGMENTS 2/3

Notation: $l(s_i)$ will denote a **line-extension of the segment** s_i.
autopartitions will use only line-extensions of given segments.

EXAMPLE - BINARY PARTITION of a SET of LINE SEGMENTS 2/3

Notation: $l(s_i)$ will denote a **line-extension of the segment** s_i.
autopartitions will use only line-extensions of given segments.
Algorithm RandAuto:

EXAMPLE - BINARY PARTITION of a SET of LINE SEGMENTS 2/3

Notation: $l(s_i)$ will denote a **line-extension of the segment** s_i.
autopartitions will use only line-extensions of given segments.
Algorithm RandAuto:

Input: A set $S = \{s_1, \ldots, s_n\}$ of non-intersecting line segments.

EXAMPLE - BINARY PARTITION of a SET of LINE SEGMENTS 2/3

Notation: $l(s_i)$ will denote a **line-extension of the segment** s_i.
autopartitions will use only line-extensions of given segments.
Algorithm RandAuto:

> **Input:** A set $S = \{s_1, \ldots, s_n\}$ of non-intersecting line segments.
> *Output:* A binary autopartition P_Π of S.

EXAMPLE - BINARY PARTITION of a SET of LINE SEGMENTS 2/3

Notation: $l(s_i)$ will denote a **line-extension of the segment** s_i.
autopartitions will use only line-extensions of given segments.
Algorithm RandAuto:

Input: A set $S = \{s_1, \ldots, s_n\}$ of non-intersecting line segments.
Output: A binary autopartition P_Π of S.
1: Pick a permutation Π of $\{1, \ldots, n\}$ uniformly and randomly.

EXAMPLE - BINARY PARTITION of a SET of LINE SEGMENTS 2/3

Notation: $l(s_i)$ will denote a **line-extension of the segment s_i**.
autopartitions will use only line-extensions of given segments.
Algorithm RandAuto:

Input: A set $S = \{s_1, \ldots, s_n\}$ of non-intersecting line segments.

Output: A binary autopartition P_Π of S.

1: Pick a permutation Π of $\{1, \ldots, n\}$ uniformly and randomly.

2: **While** there is a region R that contains more than one segment, choose one of them randomly and cut it with $l(s_i)$ where i is the first element in the ordering induced by Π such that $l(s_i)$ cuts the region R.

EXAMPLE - BINARY PARTITION of a SET of LINE SEGMENTS 2/3

Notation: $l(s_i)$ will denote a **line-extension of the segment s_i**.
autopartitions will use only line-extensions of given segments.
Algorithm RandAuto:

Input: A set $S = \{s_1, \ldots, s_n\}$ of non-intersecting line segments.
Output: A binary autopartition P_Π of S.
1: Pick a permutation Π of $\{1, \ldots, n\}$ uniformly and randomly.
2: **While** there is a region R that contains more than one segment, choose one of them randomly and cut it with $l(s_i)$ where i is the first element in the ordering induced by Π such that $l(s_i)$ cuts the region R.

Theorem: The expected size of the autopartition P_Π of S, produced by the above RandAuto algorithm is $\theta(n \ln n)$.

EXAMPLE - BINARY PARTITION of a SET of LINE SEGMENTS 2/3

Notation: $l(s_i)$ will denote a **line-extension of the segment** s_i.
autopartitions will use only line-extensions of given segments.

Algorithm RandAuto:

Input: A set $S = \{s_1, \ldots, s_n\}$ of non-intersecting line segments.

Output: A binary autopartition P_Π of S.

1: Pick a permutation Π of $\{1, \ldots, n\}$ uniformly and randomly.

2: **While** there is a region R that contains more than one segment, choose one of them randomly and cut it with $l(s_i)$ where i is the first element in the ordering induced by Π such that $l(s_i)$ cuts the region R.

Theorem: The expected size of the autopartition P_Π of S, produced by the above RandAuto algorithm is $\theta(n \ln n)$.

Proof: Notation (for line segments u, v).

EXAMPLE - BINARY PARTITION of a SET of LINE SEGMENTS 2/3

Notation: $l(s_i)$ will denote a **line-extension of the segment** s_i.
autopartitions will use only line-extensions of given segments.
Algorithm RandAuto:

Input: A set $S = \{s_1, \ldots, s_n\}$ of non-intersecting line segments.
Output: A binary autopartition P_Π of S.
1: Pick a permutation Π of $\{1, \ldots, n\}$ uniformly and randomly.
2: **While** there is a region R that contains more than one segment, choose one of them randomly and cut it with $l(s_i)$ where i is the first element in the ordering induced by Π such that $l(s_i)$ cuts the region R.

Theorem: The expected size of the autopartition P_Π of S, produced by the above RandAuto algorithm is $\theta(n \ln n)$.

Proof: **Notation** (for line segments u, v).

$$index(u,v) = \begin{matrix} i & \text{if} & l(u) \text{ intersects } i-1 \text{ segments before hitting } v; \\ \infty & & \text{if } l(u) \text{ does not hit } v. \end{matrix}$$

EXAMPLE - BINARY PARTITION of a SET of LINE SEGMENTS 2/3

Notation: $l(s_i)$ will denote a **line-extension of the segment** s_i.
autopartitions will use only line-extensions of given segments.
Algorithm RandAuto:

Input: A set $S = \{s_1, \ldots, s_n\}$ of non-intersecting line segments.
Output: A binary autopartition P_Π of S.
1: Pick a permutation Π of $\{1, \ldots, n\}$ uniformly and randomly.
2: **While** there is a region R that contains more than one segment, choose one of them randomly and cut it with $l(s_i)$ where i is the first element in the ordering induced by Π such that $l(s_i)$ cuts the region R.

Theorem: The expected size of the autopartition P_Π of S, produced by the above RandAuto algorithm is $\theta(n \ln n)$.

Proof: **Notation** (for line segments u, v).

$$index(u,v) = \begin{array}{lll} i & \text{if} & l(u) \text{ intersects } i-1 \text{ segments before hitting } v; \\ \infty & & \text{if } l(u) \text{ does not hit } v. \end{array}$$

$u \dashv v$ will be an event that $l(u)$ cuts v in the constructed (autopartition) tree.

EXAMPLE - BINARY PARTITION of a SET of LINE SEGMENTS 3/3

EXAMPLE - BINARY PARTITION of a SET of LINE SEGMENTS 3/3

Probability: Let u and v be segments, $index(u, v) = i$ and let $u_1, \ldots, u_{i-1}$ be segments the line $l(u)$ intersects before hitting v.

EXAMPLE - BINARY PARTITION of a SET of LINE SEGMENTS 3/3

Probability: Let u and v be segments, $index(u, v) = i$ and let $u_1, \ldots, u_{i-1}$ be segments the line $l(u)$ intersects before hitting v.
The event $u \dashv v$ happens, during an execution of RandPart, only if u occurs before any of $\{u_1, \ldots, u_{i-1}, v\}$ in the permutation Π.

EXAMPLE - BINARY PARTITION of a SET of LINE SEGMENTS 3/3

Probability: Let u and v be segments, $index(u, v) = i$ and let $u_1, \ldots, u_{i-1}$ be segments the line $l(u)$ intersects before hitting v.

The event $u \dashv v$ happens, during an execution of RandPart, only if u occurs before any of $\{u_1, \ldots, u_{i-1}, v\}$ in the permutation Π. Therefore the probability that event $u \dashv v$ happens is $\frac{1}{i+1} = \frac{1}{\text{index}(u,v)+1}$.

Notation: Let $C_{u,v}$ be the indicator variable that has value 1 if $u \dashv v$ and 0 otherwise.

EXAMPLE - BINARY PARTITION of a SET of LINE SEGMENTS 3/3

Probability: Let u and v be segments, $index(u, v) = i$ and let $u_1, \ldots, u_{i-1}$ be segments the line $l(u)$ intersects before hitting v.

The event $u \dashv v$ happens, during an execution of RandPart, only if u occurs before any of $\{u_1, \ldots, u_{i-1}, v\}$ in the permutation Π. Therefore the probability that event $u \dashv v$ happens is $\frac{1}{i+1} = \frac{1}{\text{index}(u,v)+1}$.

Notation: Let $C_{u,v}$ be the indicator variable that has value 1 if $u \dashv v$ and 0 otherwise.

$$\mathbf{E}[C_{u,v}] = Pr[u \dashv v] = \frac{1}{index(u, v) + 1}.$$

EXAMPLE - BINARY PARTITION of a SET of LINE SEGMENTS 3/3

Probability: Let u and v be segments, $index(u, v) = i$ and let $u_1, \ldots, u_{i-1}$ be segments the line $l(u)$ intersects before hitting v.
The event $u \dashv v$ happens, during an execution of RandPart, only if u occurs before any of $\{u_1, \ldots, u_{i-1}, v\}$ in the permutation Π.Therefore the probability that event $u \dashv v$ happens is $\frac{1}{i+1} = \frac{1}{\text{index}(u,v)+1}$.

Notation: Let $C_{u,v}$ be the indicator variable that has value 1 if $u \dashv v$ and 0 otherwise.

$$\mathbf{E}[C_{u,v}] = Pr[u \dashv v] = \frac{1}{index(u, v) + 1}.$$

Clearly, the size of the created partition P_Π equals n plus the number of intersections due to cuts.

EXAMPLE - BINARY PARTITION of a SET of LINE SEGMENTS 3/3

Probability: Let u and v be segments, $index(u, v) = i$ and let $u_1, \ldots, u_{i-1}$ be segments the line $l(u)$ intersects before hitting v.
The event $u \dashv v$ happens, during an execution of RandPart, only if u occurs before any of $\{u_1, \ldots, u_{i-1}, v\}$ in the permutation Π. Therefore the probability that event $u \dashv v$ happens is $\frac{1}{i+1} = \frac{1}{\text{index}(u,v)+1}$.

Notation: Let $C_{u,v}$ be the indicator variable that has value 1 if $u \dashv v$ and 0 otherwise.

$$\mathbf{E}[C_{u,v}] = Pr[u \dashv v] = \frac{1}{index(u, v) + 1}.$$

Clearly, the size of the created partition P_Π equals n plus the number of intersections due to cuts. Its expectation value is therefore

$$n + E[\sum_u \sum_{v \neq u} C_{u,v}] = n + \sum_u \sum_{v \neq u} Pr[u \dashv v] = n + \sum_u \sum_{v \neq u} \frac{1}{index(u, v) + 1}.$$

EXAMPLE - BINARY PARTITION of a SET of LINE SEGMENTS 3/3

Probability: Let u and v be segments, $index(u, v) = i$ and let $u_1, \ldots, u_{i-1}$ be segments the line $l(u)$ intersects before hitting v.

The event $u \dashv v$ happens, during an execution of RandPart, only if u occurs before any of $\{u_1, \ldots, u_{i-1}, v\}$ in the permutation Π.Therefore the probability that event $u \dashv v$ happens is $\frac{1}{i+1} = \frac{1}{\text{index}(u,v)+1}$.

Notation: Let $C_{u,v}$ be the indicator variable that has value 1 if $u \dashv v$ and 0 otherwise.

$$\mathbf{E}[C_{u,v}] = Pr[u \dashv v] = \frac{1}{index(u, v) + 1}.$$

Clearly, the size of the created partition P_Π equals n plus the number of intersections due to cuts. Its expectation value is therefore

$$n + E[\sum_u \sum_{v \neq u} C_{u,v}] = n + \sum_u \sum_{v \neq u} Pr[u \dashv v] = n + \sum_u \sum_{v \neq u} \frac{1}{index(u, v) + 1}.$$

For any line segment u and integer i there are at most two v, w such that $index(u, v) = index(u, w) = i$.

EXAMPLE - BINARY PARTITION of a SET of LINE SEGMENTS 3/3

Probability: Let u and v be segments, $index(u, v) = i$ and let $u_1, \ldots, u_{i-1}$ be segments the line $l(u)$ intersects before hitting v.

The event $u \dashv v$ happens, during an execution of RandPart, only if u occurs before any of $\{u_1, \ldots, u_{i-1}, v\}$ in the permutation Π.Therefore the probability that event $u \dashv v$ happens is $\frac{1}{i+1} = \frac{1}{\mathrm{index}(u,v)+1}$.

Notation: Let $C_{u,v}$ be the indicator variable that has value 1 if $u \dashv v$ and 0 otherwise.

$$\mathbf{E}[C_{u,v}] = Pr[u \dashv v] = \frac{1}{index(u, v) + 1}.$$

Clearly, the size of the created partition P_Π equals n plus the number of intersections due to cuts. Its expectation value is therefore

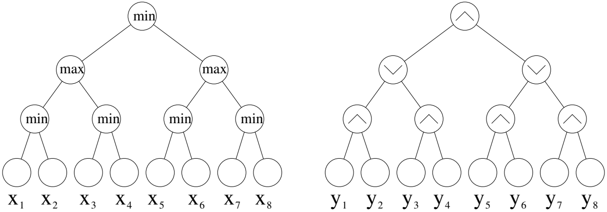
$$n + E[\sum_u \sum_{v \neq u} C_{u,v}] = n + \sum_u \sum_{v \neq u} Pr[u \dashv v] = n + \sum_u \sum_{v \neq u} \frac{1}{index(u, v) + 1}.$$

For any line segment u and integer i there are at most two v, w such that $index(u, v) = index(u, w) = i$.Hence $\sum_{v \neq u} \frac{1}{index(u,v)+1} \leq \sum_{i=1}^{n-1} \frac{2}{i+1}$ and therefore $n + \mathbf{E}[\sum_u \sum_{v \neq u} C_{u,v}] \leq n + \sum_u \sum_{i=1}^{n-1} \frac{2}{i+1} \leq n + 2nH_n$.

Game trees

GAME TREE EVALUATION - I.

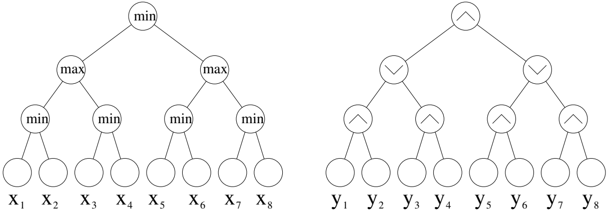
Game trees



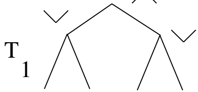
Game trees are trees with operations **max** and **min** alternating in internal nodes and with values assigned to their leaves.

GAME TREE EVALUATION - I.

Game trees



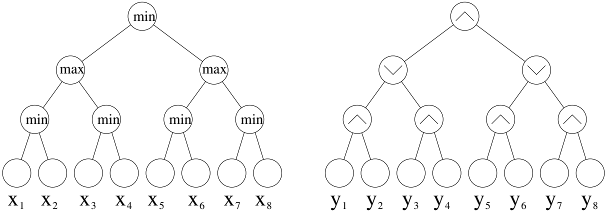
Game trees are trees with operations **max** and **min** alternating in internal nodes and with values assigned to their leaves. In case all such values are Boolean - **0** or **1** Boolean operation **OR** and **AND** are considered instead of **max** and **min**.



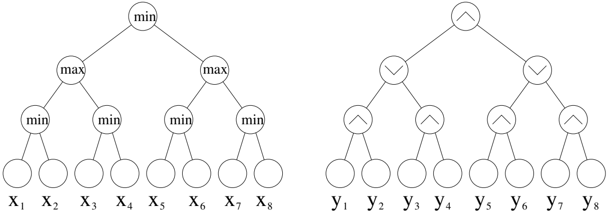
T_k – binary game tree of depth $2k$.
Goal is to evaluate the tree - the root.

GAME TREE EVALUATION - II.

GAME TREE EVALUATION - II.

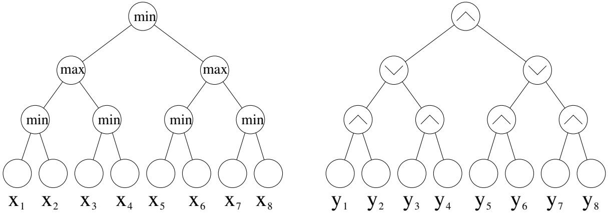


GAME TREE EVALUATION - II.



Evaluation of game trees plays a crucial role in AI, in various game playing programs.

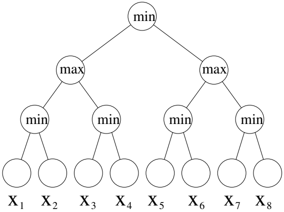
GAME TREE EVALUATION - II.

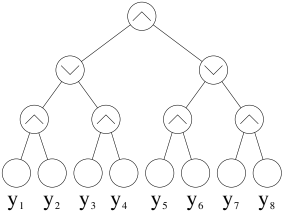


Evaluation of game trees plays a crucial role in AI, in various game playing programs.

Assumption: An evaluation algorithm chooses at each step (somehow) a leaf, reads its value and performs all evaluations of internal nodes it can perform.

GAME TREE EVALUATION - II.

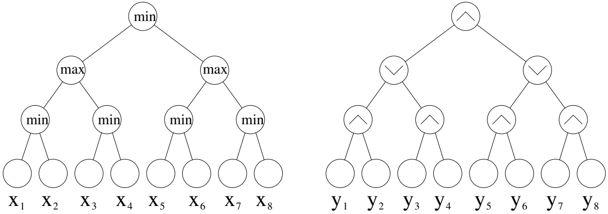




Evaluation of game trees plays a crucial role in AI, in various game playing programs.

Assumption: An evaluation algorithm chooses at each step (somehow) a leaf, reads its value and performs all evaluations of internal nodes it can perform. Cost of an evaluation algorithm is the number of leaves inspected.

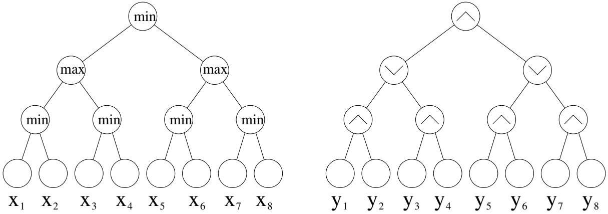
GAME TREE EVALUATION - II.



Evaluation of game trees plays a crucial role in AI, in various game playing programs.

Assumption: An evaluation algorithm chooses at each step (somehow) a leaf, reads its value and performs all evaluations of internal nodes it can perform. Cost of an evaluation algorithm is the number of leaves inspected. Determine the total number of such steps needed.

GAME TREE EVALUATION - II.



Evaluation of game trees plays a crucial role in AI, in various game playing programs.

Assumption: An evaluation algorithm chooses at each step (somehow) a leaf, reads its value and performs all evaluations of internal nodes it can perform. Cost of an evaluation algorithm is the number of leaves inspected. Determine the total number of such steps needed.

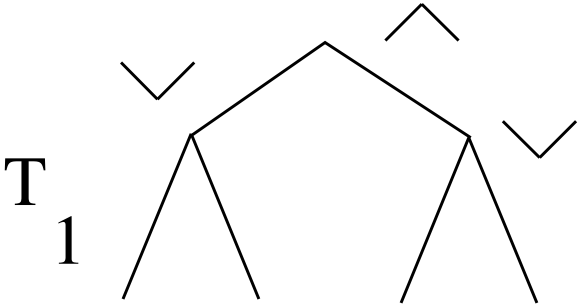
WORST CASE COMPLEXITY

WORST CASE COMPLEXITY

T_k – will denote the binary game tree of depth $2k$.

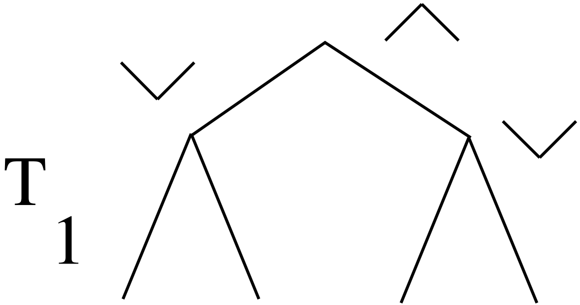
WORST CASE COMPLEXITY

T_k – will denote the binary game tree of depth $2k$.



WORST CASE COMPLEXITY

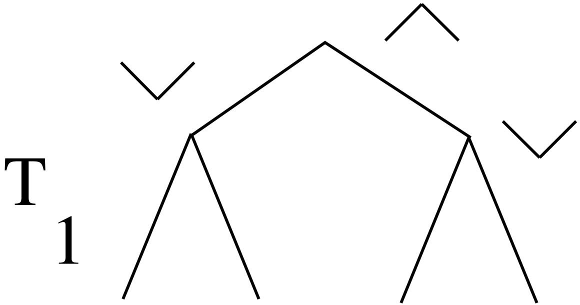
T_k – will denote the binary game tree of depth $2k$.



Every deterministic algorithm can be forced to inspect all leaves.

WORST CASE COMPLEXITY

T_k – will denote the binary game tree of depth $2k$.



Every deterministic algorithm can be forced to inspect all leaves. The worst-case complexity of a deterministic algorithm to evaluate T_k is therefore:

$$n = 4^k = 2^{2k}.$$

A RANDOMIZED ALGORITHM - BASIC IDEA:

A RANDOMIZED ALGORITHM - BASIC IDEA:

To evaluate an AND-node v, the algorithm chooses randomly one of its children and evaluates it.

A RANDOMIZED ALGORITHM - BASIC IDEA:

To evaluate an AND-node v, the algorithm chooses randomly one of its children and evaluates it.

If 1 is returned,

A RANDOMIZED ALGORITHM - BASIC IDEA:

To evaluate an AND-node v, the algorithm chooses randomly one of its children and evaluates it.

If 1 is returned, algorithm proceeds to evaluate other children subtree and returns as the value of v the value of that subtree.

A RANDOMIZED ALGORITHM - BASIC IDEA:

To evaluate an AND-node v, the algorithm chooses randomly one of its children and evaluates it.

If 1 is returned, algorithm proceeds to evaluate other children subtree and returns as the value of v the value of that subtree. If 0 is returned,

A RANDOMIZED ALGORITHM - BASIC IDEA:

To evaluate an AND-node v, the algorithm chooses randomly one of its children and evaluates it.

If 1 is returned, algorithm proceeds to evaluate other children subtree and returns as the value of v the value of that subtree. If 0 is returned, algorithm returns immediately 0 for v (without evaluating other subtree).

A RANDOMIZED ALGORITHM - BASIC IDEA:

To evaluate an AND-node v, the algorithm chooses randomly one of its children and evaluates it.

If 1 is returned, algorithm proceeds to evaluate other children subtree and returns as the value of v the value of that subtree. If 0 is returned, algorithm returns immediately 0 for v (without evaluating other subtree).

To evaluate an OR-node v, algorithm chooses randomly one of its children and evaluates it.

A RANDOMIZED ALGORITHM - BASIC IDEA:

To evaluate an AND-node v, the algorithm chooses randomly one of its children and evaluates it.

If 1 is returned, algorithm proceeds to evaluate other children subtree and returns as the value of v the value of that subtree. If 0 is returned, algorithm returns immediately 0 for v (without evaluating other subtree).

To evaluate an OR-node v, algorithm chooses randomly one of its children and evaluates it.

If 0 is returned, algorithm proceeds to evaluate other subtree and returns as the value of v the value of the subtree.

A RANDOMIZED ALGORITHM - BASIC IDEA:

To evaluate an AND-node v, the algorithm chooses randomly one of its children and evaluates it.

If 1 is returned, algorithm proceeds to evaluate other children subtree and returns as the value of v the value of that subtree. If 0 is returned, algorithm returns immediately 0 for v (without evaluating other subtree).

To evaluate an OR-node v, algorithm chooses randomly one of its children and evaluates it.

If 0 is returned, algorithm proceeds to evaluate other subtree and returns as the value of v the value of the subtree. If 1 is returned, the algorithm returns 1 for v.

RANDOMIZED ALGORITHMS - SUMMARY of THE BASIC IDEA

Start at the root and in order to evaluate a node evaluate (recursively) a random child of the current node.

RANDOMIZED ALGORITHMS - SUMMARY of THE BASIC IDEA

Start at the root and in order to evaluate a node evaluate (recursively) a random child of the current node.

If this does not determine the value of the current node, evaluate the node of other child.

Theorem: Given any instance of T_k, the expected number of steps for the above randomized algorithm is at most 3^k.

Theorem: Given any instance of T_k, the expected number of steps for the above randomized algorithm is at most 3^k.

Proof by induction:
Base step: Case $k = 1$ easy - verify by computations for all possible inputs.

Theorem: Given any instance of T_k, the expected number of steps for the above randomized algorithm is at most 3^k.

Proof by induction:
Base step: Case $k = 1$ easy - verify by computations for all possible inputs.
Inductive step: Assume that the expected cost of the evaluation of any instance of T_{k-1} is at most 3^{k-1}.

Theorem: Given any instance of T_k, the expected number of steps for the above randomized algorithm is at most 3^k.

Proof by induction:
Base step: Case $k = 1$ easy - verify by computations for all possible inputs.
Inductive step: Assume that the expected cost of the evaluation of any instance of T_{k-1} is at most 3^{k-1}.

Consider an OR-node tree T with both children being T_{k-1}-trees.

Theorem: Given any instance of T_k, the expected number of steps for the above randomized algorithm is at most 3^k.

Proof by induction:
Base step: Case $k = 1$ easy - verify by computations for all possible inputs.
Inductive step: Assume that the expected cost of the evaluation of any instance of T_{k-1} is at most 3^{k-1}.

Consider an OR-node tree T with both children being T_{k-1}-trees.
If the root of T were to return 1,

Theorem: Given any instance of T_k, the expected number of steps for the above randomized algorithm is at most 3^k.

Proof by induction:
Base step: Case $k = 1$ easy - verify by computations for all possible inputs.
Inductive step: Assume that the expected cost of the evaluation of any instance of T_{k-1} is at most 3^{k-1}.

Consider an OR-node tree T with both children being T_{k-1}-trees.
If the root of T were to return 1, at least one of its T_{k-1}-subtrees has to return 1.

Theorem: Given any instance of T_k, the expected number of steps for the above randomized algorithm is at most 3^k.

Proof by induction:
Base step: Case $k = 1$ easy - verify by computations for all possible inputs.
Inductive step: Assume that the expected cost of the evaluation of any instance of T_{k-1} is at most 3^{k-1}.

Consider an OR-node tree T with both children being T_{k-1}-trees.
If the root of T were to return 1, at least one of its T_{k-1}-subtrees has to return 1. With probability $\frac{1}{2}$ this child is chosen first, given in average at most 3^{k-1} leaf-evaluations.

Theorem: Given any instance of T_k, the expected number of steps for the above randomized algorithm is at most 3^k.

Proof by induction:
Base step: Case $k = 1$ easy - verify by computations for all possible inputs.
Inductive step: Assume that the expected cost of the evaluation of any instance of T_{k-1} is at most 3^{k-1}.

Consider an OR-node tree T with both children being T_{k-1}-trees.
If the root of T were to return 1, at least one of its T_{k-1}-subtrees has to return 1. With probability $\frac{1}{2}$ this child is chosen first, given in average at most 3^{k-1} leaf-evaluations. With probability $\frac{1}{2}$ both subtrees are to be evaluated.

Theorem: Given any instance of T_k, the expected number of steps for the above randomized algorithm is at most 3^k.

Proof by induction:
Base step: Case $k = 1$ easy - verify by computations for all possible inputs.
Inductive step: Assume that the expected cost of the evaluation of any instance of T_{k-1} is at most 3^{k-1}.

Consider an OR-node tree T with both children being T_{k-1}-trees.
If the root of T were to return 1, at least one of its T_{k-1}-subtrees has to return 1. With probability $\frac{1}{2}$ this child is chosen first, given in average at most 3^{k-1} leaf-evaluations. With probability $\frac{1}{2}$ both subtrees are to be evaluated. The expected cost of determining the value of T is therefore:

Theorem: Given any instance of T_k, the expected number of steps for the above randomized algorithm is at most 3^k.

Proof by induction:
Base step: Case $k = 1$ easy - verify by computations for all possible inputs.
Inductive step: Assume that the expected cost of the evaluation of any instance of T_{k-1} is at most 3^{k-1}.

Consider an OR-node tree T with both children being T_{k-1}-trees.
If the root of T were to return 1, at least one of its T_{k-1}-subtrees has to return 1. With probability $\frac{1}{2}$ this child is chosen first, given in average at most 3^{k-1} leaf-evaluations. With probability $\frac{1}{2}$ both subtrees are to be evaluated. The expected cost of determining the value of T is therefore:

$$\frac{1}{2} \times 3^{k-1} + \frac{1}{2} \times 2 \times 3^{k-1} = \frac{1}{2} \times 3^k = \frac{3}{2} \times 3^{k-1}.$$

If the root of T were to return 0 both subtrees have to be evaluated, giving the cost $2 \times 3^{k-1}$.

If the root of T were to return 0 both subtrees have to be evaluated, giving the cost $2 \times 3^{k-1}$.

Consider now the root of T_k.

If the root of T were to return 0 both subtrees have to be evaluated, giving the cost $2 \times 3^{k-1}$.

Consider now the root of T_k.

If the root evaluates to 1, both of its OR-subtrees have to evaluate to 1.

If the root of T were to return 0 both subtrees have to be evaluated, giving the cost $2 \times 3^{k-1}$.

Consider now the root of T_k.

If the root evaluates to 1, both of its OR-subtrees have to evaluate to 1. The expected cost is therefore

$$2 \times \frac{3}{2} \times 3^{k-1} = 3^k.$$

If the root of T were to return 0 both subtrees have to be evaluated, giving the cost $2 \times 3^{k-1}$.

Consider now the root of T_k.

If the root evaluates to 1, both of its OR-subtrees have to evaluate to 1. The expected cost is therefore

$$2 \times \frac{3}{2} \times 3^{k-1} = 3^k.$$

If the root evaluates to 0, at least one of the subtrees evaluates to 0. The expected cost is therefore

$$\frac{1}{2} \times 2 \times 2 \times 3^{k-1} + \frac{1}{2} \times \frac{3}{2} \times 3^{k-1} \leq 3^k = n^{\lg_4 3} = n^{0.793}.$$

If the root of T were to return 0 both subtrees have to be evaluated, giving the cost $2 \times 3^{k-1}$.

Consider now the root of T_k.

If the root evaluates to 1, both of its OR-subtrees have to evaluate to 1. The expected cost is therefore

$$2 \times \frac{3}{2} \times 3^{k-1} = 3^k.$$

If the root evaluates to 0, at least one of the subtrees evaluates to 0. The expected cost is therefore

$$\frac{1}{2} \times 2 \times 2 \times 3^{k-1} + \frac{1}{2} \times \frac{3}{2} \times 3^{k-1} \le 3^k = n^{\lg_4 3} = n^{0.793}.$$

Our algorithm is therefore a *Las Vegas algorithm*.

If the root of T were to return 0 both subtrees have to be evaluated, giving the cost $2 \times 3^{k-1}$.

Consider now the root of T_k.

If the root evaluates to 1, both of its OR-subtrees have to evaluate to 1. The expected cost is therefore

$$2 \times \frac{3}{2} \times 3^{k-1} = 3^k.$$

If the root evaluates to 0, at least one of the subtrees evaluates to 0. The expected cost is therefore

$$\frac{1}{2} \times 2 \times 2 \times 3^{k-1} + \frac{1}{2} \times \frac{3}{2} \times 3^{k-1} \leq 3^k = n^{\lg_4 3} = n^{0.793}.$$

Our algorithm is therefore a *Las Vegas algorithm*. Its running time (number of leaves evaluations) is: $n^{0.793}$.

APPENDIX

APPENDIX

The concept of the number of wisdom introduced in the following and related results helped to show that randomness is deeply rooted even in arithmetic.

APPENDIX

The concept of the number of wisdom introduced in the following and related results helped to show that randomness is deeply rooted even in arithmetic.

In order to define numbers of wisdom the concept of self-delimiting programs is needed.

APPENDIX

The concept of the number of wisdom introduced in the following and related results helped to show that randomness is deeply rooted even in arithmetic.

In order to define numbers of wisdom the concept of self-delimiting programs is needed.

A program represented by a binary word p, is self-delimiting for a computer C, if for any input pw the computer C can recognize where p ends after reading p only..

Another way to see self-delimiting programs is to consider only such programming languages L that no program in L is a prefix of another program in L.

Ω - numbers of wisdom

For a universal computer C with only self-delimiting programs, the number of wisdom Ω_C is the probability that randomly constructed program for C halts.

Ω - numbers of wisdom

For a universal computer C with only self-delimiting programs, the number of wisdom Ω_C is the probability that randomly constructed program for C halts. More formally

Ω - numbers of wisdom

For a universal computer C with only self-delimiting programs, the number of wisdom Ω_C is the probability that randomly constructed program for C halts. More formally

$$\Omega_C = \sum_{p \text{ halts}} 2^{-|p|}$$

where p are (self-delimiting) halting programs for C.

Ω - numbers of wisdom

For a universal computer C with only self-delimiting programs, the number of wisdom Ω_C is the probability that randomly constructed program for C halts. More formally

$$\Omega_C = \sum_{p \text{ halts}} 2^{-|p|}$$

where p are (self-delimiting) halting programs for C.

Ω_C is therefore the probability that a self-delimiting computer program for C generated at random, by choosing each of its bits using an independent toss of a fair coin, will eventually halt.

Properties of numbers of wisdom

Properties of numbers of wisdom

- $0 \leq \Omega_C \leq 1$

Properties of numbers of wisdom

- $0 \leq \Omega_C \leq 1$
- Ω_C is an uncomputable and random real number.

Properties of numbers of wisdom

- $0 \leq \Omega_C \leq 1$
- Ω_C is an uncomputable and random real number.
- At least n-bits long theory is needed to determine n bits of Ω_C.

Properties of numbers of wisdom

- $0 \leq \Omega_C \leq 1$
- Ω_C is an uncomputable and random real number.
- At least n-bits long theory is needed to determine n bits of Ω_C.
- At least n bits long program is needed to determine n bits of Ω_C
- Bits of Ω can be seen as mathematical facts that are true for no reason.

- Greg Chaitin, who introduced numbers of wisdom, designed a specific universal computer C and a two hundred pages long Diophantine equation E, with 17,000 variables and with one parameter k, such that for a given k the equation E has a finite (infinite) number of solutions if and only if the k-th bit of Ω_C is 0 (is 1).

- Greg Chaitin, who introduced numbers of wisdom, designed a specific universal computer C and a two hundred pages long Diophantine equation E, with 17,000 variables and with one parameter k, such that for a given k the equation E has a finite (infinite) number of solutions if and only if the k-th bit of Ω_C is 0 (is 1).{ As a consequence, we have that randomness, unpredictability and uncertainty occur even in the theory of Diophantine equations of elementary arithmetic.}

- Greg Chaitin, who introduced numbers of wisdom, designed a specific universal computer C and a two hundred pages long Diophantine equation E, with 17,000 variables and with one parameter k, such that for a given k the equation E has a finite (infinite) number of solutions if and only if the k-th bit of Ω_C is 0 (is 1).$\{$ As a consequence, we have that randomness, unpredictability and uncertainty occur even in the theory of Diophantine equations of elementary arithmetic.$\}$
- Knowing the value of Ω_C with n bits of precision allows to decide which programs for C with at most n bits halt.