Overview

- A-sort,
- Sparse polynomials and matrices,
- Low-level Access to Memory,
- Hashing,
- Heaps,
- Arithmetic expressions, notations and conversion between them,
- Graphs and their representation,
- Graph-algorithms.

A-sort

- A-sort: Sorting using an A-B-tree with your finger.

A-sort

- A-sort: Sorting using an A-B-tree with your finger.
- Your finger points at the leaf (of the B-tree) where we inserted last.

A-sort

- A-sort: Sorting using an A-B-tree with your finger.
- Your finger points at the leaf (of the B-tree) where we inserted last.
- We are not start the insertion process at the root but where our finger is pointing.

A-sort

- A-sort: Sorting using an A-B-tree with your finger.
- Your finger points at the leaf (of the B-tree) where we inserted last.
- We are not start the insertion process at the root but where our finger is pointing.
- Yields good results when the input is pre-sorted (we don't bubble too often to the root).

Representation using linked lists

- We want to represent sparse polynomials (i.e., polynomials with just a few non-zero coefficients). How do we do that?

Representation using linked lists

- We want to represent sparse polynomials (i.e., polynomials with just a few non-zero coefficients). How do we do that?
- We use linked lists.

Representation using linked lists

- We want to represent sparse polynomials (i.e., polynomials with just a few non-zero coefficients). How do we do that?
- We use linked lists.
- It is an advantage to use a bi-directional version...

Representation using linked lists

- We want to represent sparse polynomials (i.e., polynomials with just a few non-zero coefficients). How do we do that?
- We use linked lists.
- It is an advantage to use a bi-directional version...
- ...maybe even cyclic – and with a head.

Representation using linked lists

- We want to represent sparse polynomials (i.e., polynomials with just a few non-zero coefficients). How do we do that?
- We use linked lists.
- It is an advantage to use a bi-directional version...
- ...maybe even cyclic – and with a head.
- Summing the polynomials up: Pass through the lists (and merge them).

Representation using linked lists

- We want to represent sparse polynomials (i.e., polynomials with just a few non-zero coefficients). How do we do that?
- We use linked lists.
- It is an advantage to use a bi-directional version...
- ...maybe even cyclic – and with a head.
- Summing the polynomials up: Pass through the lists (and merge them).
- Polynomial multiplication: We have to pass in both directions.

Representation using linked lists

- We want to represent sparse polynomials (i.e., polynomials with just a few non-zero coefficients). How do we do that?
- We use linked lists.
- It is an advantage to use a bi-directional version...
- ...maybe even cyclic – and with a head.
- Summing the polynomials up: Pass through the lists (and merge them).
- Polynomial multiplication: We have to pass in both directions.
- Head can be used to find that we reached the end of the polynomial.

Sparse matrix representation

- Again, we have just a few non-zero elements, thus we compress the matrix.

Sparse matrix representation

- Again, we have just a few non-zero elements, thus we compress the matrix.
- Theoretically there are many possibilities and we have to be careful which one to use, because a bad design can be much worse than brute force. For example:

Sparse matrix representation

- Again, we have just a few non-zero elements, thus we compress the matrix.
- Theoretically there are many possibilities and we have to be careful which one to use, because a bad design can be much worse than brute force. For example:
- Linked list of the elements (ordered in both dimensions),

Sparse matrix representation

- Again, we have just a few non-zero elements, thus we compress the matrix.
- Theoretically there are many possibilities and we have to be careful which one to use, because a bad design can be much worse than brute force. For example:
- Linked list of the elements (ordered in both dimensions),
- linked list of linked lists (list of rows consisting of list of columns),

Sparse matrix representation

- Again, we have just a few non-zero elements, thus we compress the matrix.
- Theoretically there are many possibilities and we have to be careful which one to use, because a bad design can be much worse than brute force. For example:
- Linked list of the elements (ordered in both dimensions),
- linked list of linked lists (list of rows consisting of list of columns),
- Dividing into quarters (divide the matrix into four parts – left top, right top, left bottom, right bottom). If the submatrix is "too large" and non-zero, we divide again.

Low-level functions

- `procedure mark(var p:pointer);` – returns the heap-top do not use it (proposes colleague Kryl)

Low-level functions

- `procedure mark(var p:pointer);` – returns the heap-top
  do not use it (proposes colleague Kryl)
- `procedure release(var p:pointer);` – set the heap-top
  (deallocates everything above the top)
  do not use it (DTTO)

Low-level functions

- `procedure mark(var p:pointer);` – returns the heap-top
  do not use it (proposes colleague Kryl)
- `procedure release(var p:pointer);` – set the heap-top (deallocates everything above the top)
  do not use it (DTTO)
- It seems these functions are not in Free Pascal (they may be dangerous). It is a simple form of garbage-collector.

Low-level functions

- `procedure mark(var p:pointer);` – returns the heap-top
  do not use it (proposes colleague Kryl)
- `procedure release(var p:pointer);` – set the heap-top
  (deallocates everything above the top)
  do not use it (DTTO)
- It seems these functions are not in Free Pascal (they may be dangerous). It is a simple form of garbage-collector.
- `function MemAvail: longint;` – returns number of available bytes on heap
  (unavailable in Free Pascal since 2.0)

Low-level functions

- `procedure mark(var p:pointer);` – returns the heap-top
  do not use it (proposes colleague Kryl)
- `procedure release(var p:pointer);` – set the heap-top (deallocates everything above the top)
  do not use it (DTTO)
- It seems these functions are not in Free Pascal (they may be dangerous). It is a simple form of garbage-collector.
- `function MemAvail: longint;` – returns number of available bytes on heap
  (unavailable in Free Pascal since 2.0)
- `function MaxAvail: longint;` – returns size of the largest free block (largest allocable size)
  (DTTO)

Low-level functions

- `GetMem` – allocates memory, compared to `new` it does not examine how much – use only in emergency (advice of F. Klaempfl)

Low-level functions

- `GetMem` – allocates memory, compared to `new` it does not examine how much – use only in emergency (advice of F. Klaempfl)
- `FreeMem` – deallocates memory allocated by `GetMem` – (DTTO)

Example of GetMem/FreeMem

We create an array of uncertain length

```
type parr=^tarr;
     tarr=array[1..10000] of longint;
var arr:parr;
begin
      GetMem(arr,500);{get 500 bytes}
      arr^[10]:=1000;{This is OK}
      arr^[500]:=1024;{Problem -- array too small!}
      FreeMem(arr,500);{FreeMem(arr); should suffice}
end.
```

Hashing

- Consider data that can be indexed, but where the range is too large for explicit enumeration (e.g., strings). It makes sense to compute some function (i.e., sum up ordinal values of individual characters mod 256). This function gets denoted h.

Hashing

- Consider data that can be indexed, but where the range is too large for explicit enumeration (e.g., strings). It makes sense to compute some function (i.e., sum up ordinal values of individual characters mod 256). This function gets denoted h.
- Then we allocate a table much smaller than the universum (range).

Hashing

- Consider data that can be indexed, but where the range is too large for explicit enumeration (e.g., strings). It makes sense to compute some function (i.e., sum up ordinal values of individual characters mod 256). This function gets denoted h.
- Then we allocate a table much smaller than the universum (range).
- This is called the *hashing*.

Hashing

- Consider data that can be indexed, but where the range is too large for explicit enumeration (e.g., strings). It makes sense to compute some function (i.e., sum up ordinal values of individual characters mod 256). This function gets denoted h.
- Then we allocate a table much smaller than the universum (range).
- This is called the *hashing*.
- It may happen that more candidates want to access the same cell in the table. This is called a *collision*.

Hashing

how to solve collisions

- There are several methods how to solve collisions:

Hashing

how to solve collisions

- There are several methods how to solve collisions:
- Each member contains a list of elements belonging to the same cell,

Hashing

how to solve collisions

- There are several methods how to solve collisions:
- Each member contains a list of elements belonging to the same cell,
- coalescent, cuckoo-hashing,... – generally we are placing the values into incorrect cells,

Hashing

how to solve collisions

- There are several methods how to solve collisions:
- Each member contains a list of elements belonging to the same cell,
- coalescent, cuckoo-hashing,... – generally we are placing the values into incorrect cells,
- implies problem with `delete` (almost unimplementable as we could break the chain).

Hashing

how to solve collisions

- There are several methods how to solve collisions:
- Each member contains a list of elements belonging to the same cell,
- coalescent, cuckoo-hashing,... – generally we are placing the values into incorrect cells,
- implies problem with `delete` (almost unimplementable as we could break the chain).
- Where to place the element in collision? There are many possibilities. Either we pick next free cell or we design a function that proposes next cell.

Hashing

how to solve collisions

- There are several methods how to solve collisions:
- Each member contains a list of elements belonging to the same cell,
- coalescent, cuckoo-hashing,... – generally we are placing the values into incorrect cells,
- implies problem with `delete` (almost unimplementable as we could break the chain).
- Where to place the element in collision? There are many possibilities. Either we pick next free cell or we design a function that proposes next cell.
- If we know the size of the data, we may try to implement *perfect hashing*, i.e., hashing without collisions. Hashing should be in more detail explained in the lecture of Algorithms and Data Structures (proofs).

Notations

- How can we notate (write) the arithmetic expression?

Notations

- How can we notate (write) the arithmetic expression?
- $(10 + 5) * (15 - 4)/2$ – infix notation (operator is between operands),

Notations

- How can we notate (write) the arithmetic expression?
- $(10+5)*(15-4)/2$ – infix notation (operator is between operands),
- $/ \ * \ + 10 \ 5 \ - \ 15 \ 4 \ 2$ – prefix notation (operator preceeds the operands),

Notations

- How can we notate (write) the arithmetic expression?
- $(10+5)*(15-4)/2$ – infix notation (operator is between operands),
- $/ \; * \; + \; 10 \; 5 \; - \; 15 \; 4 \; 2$ – prefix notation (operator preceeds the operands),
- $/ \; (* \; (+ \; 10 \; 5) \; (- \; 15 \; 4)) \; 2$ (with brackets to make it clear),

Notations

- How can we notate (write) the arithmetic expression?
- $(10 + 5) * (15 - 4)/2$ – infix notation (operator is between operands),
- $/ \ * \ + 10 \ 5 \ - \ 15 \ 4 \ 2$ – prefix notation (operator preceeds the operands),
- $/ \ (* \ (+ \ 10 \ 5) \ (- \ 15 \ 4)) \ 2$ (with brackets to make it clear),
- $10 \ 5 \ + \ 15 \ 4 \ - \ * \ 2 \ /$ – postfix notation (operator is behind the operands),

Notations

- How can we notate (write) the arithmetic expression?
- $(10 + 5) * (15 - 4)/2$ – infix notation (operator is between operands),
- $/ \; * \; + \; 10 \; 5 \; - \; 15 \; 4 \; 2$ – prefix notation (operator preceeds the operands),
- $/ \; (* \; (+ \; 10 \; 5) \; (- \; 15 \; 4)) \; 2$ (with brackets to make it clear),
- $10 \; 5 \; + \; 15 \; 4 \; - \; * \; 2 \; /$ – postfix notation (operator is behind the operands),
- $((10 \; 5 \; +) \; (15 \; 4 \; -) \; *) \; 2 \; /$ (with superfluous parentheses),

Notations

- How can we notate (write) the arithmetic expression?
- $(10 + 5) * (15 - 4)/2$ – infix notation (operator is between operands),
- $/ \ * \ + 10 \ 5 \ - \ 15 \ 4 \ 2$ – prefix notation (operator preceeds the operands),
- $/ \ (* \ (+ \ 10 \ 5) \ (- \ 15 \ 4)) \ 2$ (with brackets to make it clear),
- $10 \ 5 \ + \ 15 \ 4 \ - \ * \ 2 \ /$ – postfix notation (operator is behind the operands),
- $((10 \ 5 \ +) \ (15 \ 4 \ -) \ *) \ 2 \ /$ (with superfluous parentheses),
- by a tree: Each node contains an operator (and has two sons - operands) or a value (leaf).

Notations

- How can we notate (write) the arithmetic expression?
- $(10+5)*(15-4)/2$ – infix notation (operator is between operands),
- $/\ *\ +\ 10\ 5\ -\ 15\ 4\ 2$ – prefix notation (operator preceeds the operands),
- $/\ (*\ (+\ 10\ 5)\ (-\ 15\ 4))\ 2$ (with brackets to make it clear),
- $10\ 5\ +\ 15\ 4\ -\ *\ 2\ /$ – postfix notation (operator is behind the operands),
- $((10\ 5\ +)\ (15\ 4\ -)\ *)\ 2\ /$ (with superfluous parentheses),
- by a tree: Each node contains an operator (and has two sons - operands) or a value (leaf).
- Evaluation will be only sketched, pseudocode has to be creatively interpreted!

Arithmetic expressions and notations

- Advantages and disadvantages (of individual notations)?

Arithmetic expressions and notations

- Advantages and disadvantages (of individual notations)?
- Is it possible to evaluate expressions in all these notations?

Arithmetic expressions and notations

- Advantages and disadvantages (of individual notations)?
- Is it possible to evaluate expressions in all these notations?
- Can we convert one notation into another one?

Arithmetic expressions and notations

- Advantages and disadvantages (of individual notations)?
- Is it possible to evaluate expressions in all these notations?
- Can we convert one notation into another one?
- Yes, e.g., using a tree.

Evaluating the prefix notation

We use recursion: `function evaluate:integer;`

```
begin
    if (we read a number) then
        evaluate:=value_of_the_input_number
    else
    begin operator:=read_operator();
        arg1:=evaluate;
        arg2:=evaluate;
        evaluate:=perform(operator,arg1,arg2);
    end;
end;
```

Tree from the prefix notation

```
function pref_tree:tree;
begin
      if(we read a number) then
            pref_tree:=leaf(value_of_input);
      else
      begin tmp:=inner_node(operator);
            tmp.arg1:=evaluate;
            tmp.arg2:=evaluate;
            evaluate:=tmp;
      end;
end;
```

function `leaf` creates a leaf,

function `inner_node` creates a node of out_deg 2,

vertex of out_deg 2 has sons `arg1` a `arg2`.

How to convert tree to all notations?

- Recursively:

How to convert tree to all notations?

- Recursively:
- We search the tree in such a way that in one phase we search the left child,

How to convert tree to all notations?

- Recursively:
- We search the tree in such a way that in one phase we search the left child,
- in one phase we search the right child and in one phase we write the operator.

How to convert tree to all notations?

- Recursively:
- We search the tree in such a way that in one phase we search the left child,
- in one phase we search the right child and in one phase we write the operator.
- All three notations arise by correct ordering of these phases.

How to convert tree to all notations?

- Recursively:
- We search the tree in such a way that in one phase we search the left child,
- in one phase we search the right child and in one phase we write the operator.
- All three notations arise by correct ordering of these phases.
- Even though we always visit the left child before the right one, we change the time when we output the operator!

Generating prefix notation

```
procedure gen_pref(v:tree);
begin
      if(leaf(v)) then
            output(v);
      else
      begin output(v);
            gen_pref(v.arg1);
            gen_pref(v.arg2);
      end;
end;
```

Function `output` outputs the operator or number (resp.), function `leaf` decides whether a given node is a leaf.

Generating postfix notation

```
procedure gen_pref(v:tree);
begin
      if(leaf(v)) then
            output(v);
      else
      begin gen_pref(v.arg1);
            gen_pref(v.arg2);
            output(v);
      end;
end;
```

Function `output` outputs the operator or number (resp.), function `leaf` decides whether a given node is a leaf.

Generating infix notation

almost correctly!

```
procedure gen_pref(v:tree);
begin
      if(leaf(v)) then
            output(v);
      else
      begin gen_pref(v.arg1);
            output(v);
            gen_pref(v.arg2);
      end;
end;
```

Function `output` outputs the operator or number (resp.), function `leaf` decides whether a given node is a leaf.

Generating infix notation

ugly but correct!

```
procedure gen_pref(v:tree);
begin
      if(leaf(v)) then
            output(v);
      else
      begin write('(');
            gen_pref(v.arg1);
            output(v);
            gen_pref(v.arg2);
            write(')');
      end;
end;
```

Function `output` outputs the operator or number (resp.), function `leaf` decides whether a given node is a leaf.

Evaluating postfix notation

...towards the solution

Revision of our knowledge:

Buffer is a data structure with the following operations:

- `push` – insert at the top of the buffer,
- `pop` – remove from the top of the buffer,
- i.e., last in, first out.

Evaluating postfix notation

```
function eval_post:integer;
begin
      while not eof do
      begin if (we read a number) then
                  push(number);
            if (we read an operator) then
            begin arg2:=pop;
                  arg1:=pop;
                  push(operator(arg1,arg2));
            end;
      end;
      writeln(pop);{Result is on the buffer-top}
end;
```

Tree from the prefix notation

```
function tree_post:tree;
begin
      while not eof do
      begin if (we read a number) then
                  push(leaf(number));
            if (we read an operator) then
            begin pom:=node(operator);
                  pom.arg2:=pop;
                  pom.arg1:=pop;
                  push(pom);
            end;
      end;
      tree_post:=pop;{Result is on the buffer-top}
end;
```

Evaluating the tree

should be clear, but let's go:

```
function eval_tree(v:tree);
begin
      if(leaf(v)) then
            eval_tree:=value(v)
      else
      begin arg1:=eval_tree(v.arg1);
            arg2:=eval_tree(v.arg2);
            op:=operator(v);
            eval_tree:=op(arg1,arg2);
      end;
end;
```

Evaluating the infix notation

alias Massacre at the hangman's tree

- One possibility is to find the operator that gets performed as the last one,

Evaluating the infix notation

alias Massacre at the hangman's tree

- One possibility is to find the operator that gets performed as the last one,
- divide the expression into two parts, evaluate (recursively) and perform the operation.

Evaluating the infix notation

alias Massacre at the hangman's tree

- One possibility is to find the operator that gets performed as the last one,
- divide the expression into two parts, evaluate (recursively) and perform the operation.
- Advantage: After thinking over how to find the last operation it is simple.

Evaluating the infix notation

alias Massacre at the hangman's tree

- One possibility is to find the operator that gets performed as the last one,
- divide the expression into two parts, evaluate (recursively) and perform the operation.
- Advantage: After thinking over how to find the last operation it is simple.
- Disadvantage: We are still traversing through the expression (looking for the operators).

Evaluating the infix notation

alias Massacre at the hangman's tree

- One possibility is to find the operator that gets performed as the last one,
- divide the expression into two parts, evaluate (recursively) and perform the operation.
- Advantage: After thinking over how to find the last operation it is simple.
- Disadvantage: We are still traversing through the expression (looking for the operators).
- How to find the operator that is being performed last?

How to find operator that is being performed last?

1. Find the last operator of addition or subtraction outside brackets,

Next year (or term) you'll learn how to do it using grammars.

How to find operator that is being performed last?

1. Find the last operator of addition or subtraction outside brackets,
2. else find the last operator of multiplication of division outside brackets,

Next year (or term) you'll learn how to do it using grammars.

How to find operator that is being performed last?

1. Find the last operator of addition or subtraction outside brackets,
2. else find the last operator of multiplication of division outside brackets,
3. if the expression is a number, evaluate it,

Next year (or term) you'll learn how to do it using grammars.

How to find operator that is being performed last?

1. Find the last operator of addition or subtraction outside brackets,
2. else find the last operator of multiplication of division outside brackets,
3. if the expression is a number, evaluate it,
4. else "peel the brackets"(remove the parenthesis at the beginning and at the end).

Next year (or term) you'll learn how to do it using grammars.

How to find operator that is being performed last?

1. Find the last operator of addition or subtraction outside brackets,
2. else find the last operator of multiplication of division outside brackets,
3. if the expression is a number, evaluate it,
4. else "peel the brackets"(remove the parenthesis at the beginning and at the end).
5. in cases 2^i (i.e., 1, 2, 4) employ recursion.

Next year (or term) you'll learn how to do it using grammars.

Graph – definition

Definition

Graph is an ordered pair $G = (V, E)$ where V is a set of vertices and $E \subseteq \binom{V}{2}$ is a set of edges.

Graph – definition

Definition

Graph is an ordered pair $G = (V, E)$ where V is a set of vertices and $E \subseteq \binom{V}{2}$ is a set of edges.

Definition

An ordered pair $G = (V, E)$ is called an oriented graph with a vertex set V and an edge set E, if $E \subseteq V \times V$.

Graph – definition

Definition

Graph is an ordered pair $G = (V, E)$ where V is a set of vertices and $E \subseteq \binom{V}{2}$ is a set of edges.

Definition

An ordered pair $G = (V, E)$ is called an oriented graph with a vertex set V and an edge set E, if $E \subseteq V \times V$.

- Graphs are generally discussed in Discrete mathematics, so you have probably heard about particular algorithms. And the algorithms can actually be implemented.

Graph – definition

Definition

Graph is an ordered pair $G = (V, E)$ where V is a set of vertices and $E \subseteq \binom{V}{2}$ is a set of edges.

Definition

An ordered pair $G = (V, E)$ is called an oriented graph with a vertex set V and an edge set E, if $E \subseteq V \times V$.

- Graphs are generally discussed in Discrete mathematics, so you have probably heard about particular algorithms. And the algorithms can actually be implemented.
- However, a formal definition of a graph is fine, but it does not help much with programming. We have to ask ourselves:

Graph – definition

Definition

Graph is an ordered pair $G = (V, E)$ where V is a set of vertices and $E \subseteq \binom{V}{2}$ is a set of edges.

Definition

An ordered pair $G = (V, E)$ is called an oriented graph with a vertex set V and an edge set E, if $E \subseteq V \times V$.

- Graphs are generally discussed in Discrete mathematics, so you have probably heard about particular algorithms. And the algorithms can actually be implemented.
- However, a formal definition of a graph is fine, but it does not help much with programming. We have to ask ourselves:
- How should one represent a graph when programming?

Graphs – remarks to definition

- Goal: Advantages and disadvantages of individual representations.

Graphs – remarks to definition

- Goal: Advantages and disadvantages of individual representations.
- Define basic notions (walk, trail, path, connectivity, trees).

Graph Representation

- From the Discrete Mathematics lectures you know the following:

Graph Representation

- From the Discrete Mathematics lectures you know the following:
- Adjacency matrix A_G
  – is a square 0/1-matrix whose rows and columns are indexed by individual vertices. One corresponds to an edge between appropriate vertices (zero means no edge there).

Graph Representation

- From the Discrete Mathematics lectures you know the following:
- Adjacency matrix A_G
  – is a square 0/1-matrix whose rows and columns are indexed by individual vertices. One corresponds to an edge between appropriate vertices (zero means no edge there).
- Incidence matrix B_G – rows index by vertices, columns by edges, one in $B[i,j]$ means that the edge j is incident with the vertex i.

Graph Representation

- From the Discrete Mathematics lectures you know the following:
- Adjacency matrix A_G
  – is a square 0/1-matrix whose rows and columns are indexed by individual vertices. One corresponds to an edge between appropriate vertices (zero means no edge there).
- Incidence matrix B_G – rows index by vertices, columns by edges, one in $B[i,j]$ means that the edge j is incident with the vertex i.
- Advantages and disadvantages?

Graph Representation

- From the Discrete Mathematics lectures you know the following:
- Adjacency matrix A_G
  – is a square 0/1-matrix whose rows and columns are indexed by individual vertices. One corresponds to an edge between appropriate vertices (zero means no edge there).
- Incidence matrix B_G – rows index by vertices, columns by edges, one in $B[i,j]$ means that the edge j is incident with the vertex i.
- Advantages and disadvantages?
- Can we convert these representations?

Converting A_G to B_G and back

```
init_with_0s(B_G);
edge_index:=1;
for i:=1 to n do begin
      for j:=i+1 to n do begin
      if(A_G[i,j]=1) then
      begin
            B_G[i,edge_index]:=1;
            B_G[j,edge_index]:=1;
            inc(edge_index);
      end;
end;
```

B_G to A_G

Either we analyze the incidence matrix (in a similar way) or:
$A_G := B_G \times {B_G}^T$;
`for i:=1 to n do`
    $A_G[i, i] := 0$;

Důkaz.

Exercise in Combinatorics and Graph Theory I □

Further graph representations

- List of vertices and edges incident to individual vertices.

Further graph representations

- List of vertices and edges incident to individual vertices.
- I.e., we are keeping a list of edges incident to each vertex in the graph.

Further graph representations

- List of vertices and edges incident to individual vertices.
- I.e., we are keeping a list of edges incident to each vertex in the graph.
- We employ linked-lists. If we employ an array, what do we get?

Further graph representations

- List of vertices and edges incident to individual vertices.
- I.e., we are keeping a list of edges incident to each vertex in the graph.
- We employ linked-lists. If we employ an array, what do we get?
- The adjacency matrix!

Further graph representations

- List of vertices and edges incident to individual vertices.
- I.e., we are keeping a list of edges incident to each vertex in the graph.
- We employ linked-lists. If we employ an array, what do we get?
- The adjacency matrix!

Functions necessary/sufficient to work with a graph:

Further graph representations

- List of vertices and edges incident to individual vertices.
- I.e., we are keeping a list of edges incident to each vertex in the graph.
- We employ linked-lists. If we employ an array, what do we get?
- The adjacency matrix!

Functions necessary/sufficient to work with a graph:

- `find_neighbors(v)`,

Further graph representations

- List of vertices and edges incident to individual vertices.
- I.e., we are keeping a list of edges incident to each vertex in the graph.
- We employ linked-lists. If we employ an array, what do we get?
- The adjacency matrix!

Functions necessary/sufficient to work with a graph:

- `find_neighbors(v)`,
- `vertices`,

Further graph representations

- List of vertices and edges incident to individual vertices.
- I.e., we are keeping a list of edges incident to each vertex in the graph.
- We employ linked-lists. If we employ an array, what do we get?
- The adjacency matrix!

Functions necessary/sufficient to work with a graph:

- `find_neighbors(v)`,
- `vertices`,
- `edges` or `edge(u,v)` – we can find it through `vertices` and `find_neighbors`,

Further graph representations

- List of vertices and edges incident to individual vertices.
- I.e., we are keeping a list of edges incident to each vertex in the graph.
- We employ linked-lists. If we employ an array, what do we get?
- The adjacency matrix!

Functions necessary/sufficient to work with a graph:

- `find_neighbors(v)`,
- `vertices`,
- `edges` or `edge(u,v)` – we can find it through `vertices` and `find_neighbors`,
- further, e.g., (`vertex_weight(v), edge_weight(e)...`).

Further graph representations

- List of vertices and edges incident to individual vertices.
- I.e., we are keeping a list of edges incident to each vertex in the graph.
- We employ linked-lists. If we employ an array, what do we get?
- The adjacency matrix!

Functions necessary/sufficient to work with a graph:

- `find_neighbors(v)`,
- `vertices`,
- `edges` or `edge(u,v)` – we can find it through `vertices` and `find_neighbors`,
- further, e.g., (`vertex_weight(v), edge_weight(e)...`).
- Advantages/disadvantages?

Further graph representations

- List of vertices and edges incident to individual vertices.
- I.e., we are keeping a list of edges incident to each vertex in the graph.
- We employ linked-lists. If we employ an array, what do we get?
- The adjacency matrix!

Functions necessary/sufficient to work with a graph:

- `find_neighbors(v)`,
- `vertices`,
- `edges` or `edge(u,v)` – we can find it through `vertices` and `find_neighbors`,
- further, e.g., (`vertex_weight(v), edge_weight(e)...`).
- Advantages/disadvantages?
- In the oriented case we have to modify the representation.

Walk, trail, path, circle

Definition

Walk, trail, path, circle

Definition

- A walk of length k is a sequence of edges $\{v_0, v_1\}, \{v_1, v_2\}, \{v_2, v_3\}, \ldots, \{v_{k-1}, v_k\}$.

Walk, trail, path, circle

Definition

- A walk of length k is a sequence of edges $\{v_0, v_1\}, \{v_1, v_2\}, \{v_2, v_3\}, \ldots, \{v_{k-1}, v_k\}$.
- A trail is a walk where each edge occurs at most once.

Walk, trail, path, circle

Definition

- A walk of length k is a sequence of edges $\{v_0, v_1\}, \{v_1, v_2\}, \{v_2, v_3\}, \ldots, \{v_{k-1}, v_k\}$.
- A trail is a walk where each edge occurs at most once.
- A path is a trail (or a walk) where each vertex occurs only once (i.e., each vertex is incident to two consecutive edges).

Walk, trail, path, circle

Definition

- A walk of length k is a sequence of edges $\{v_0, v_1\}, \{v_1, v_2\}, \{v_2, v_3\}, \ldots, \{v_{k-1}, v_k\}$.
- A trail is a walk where each edge occurs at most once.
- A path is a trail (or a walk) where each vertex occurs only once (i.e., each vertex is incident to two consecutive edges).
- A trail is a circle if it starts and ends in the same vertex and each vertex occurs there exactly once.

Connectivity, tree

Definition

Connectivity, tree

Definition

- A graph is connected if from any its vertices we can reach any other vertex.

Connectivity, tree

Definition

- A graph is connected if from any its vertices we can reach any other vertex.
- A graph is a tree if it is connected and it contains no circles.

Connectivity, tree

Definition

- A graph is connected if from any its vertices we can reach any other vertex.
- A graph is a tree if it is connected and it contains no circles.

- Definitions are fine, but how we use them while programming?

Connectivity, tree

Definition

- A graph is connected if from any its vertices we can reach any other vertex.
- A graph is a tree if it is connected and it contains no circles.

- Definitions are fine, but how we use them while programming?
- How do we verify that a graph is connected?

Connectivity, tree

Definition

- A graph is connected if from any its vertices we can reach any other vertex.
- A graph is a tree if it is connected and it contains no circles.

- Definitions are fine, but how we use them while programming?
- How do we verify that a graph is connected?
- We use a suitable claim.

Connectivity, tree

Definition

- A graph is connected if from any its vertices we can reach any other vertex.
- A graph is a tree if it is connected and it contains no circles.

- Definitions are fine, but how we use them while programming?
- How do we verify that a graph is connected?
- We use a suitable claim.
- How do we decide whether a graph is a tree?

Connectivity, tree

Definition

- A graph is connected if from any its vertices we can reach any other vertex.
- A graph is a tree if it is connected and it contains no circles.

- Definitions are fine, but how we use them while programming?
- How do we verify that a graph is connected?
- We use a suitable claim.
- How do we decide whether a graph is a tree?
- Similarly!

Graph connectivity

A graph is connected iff from one (fixed) vertex we can reach all the other vertices.

```
for i in vertices do
      unvisit(i); {so far we visited nothing}
i:=start_vertex;
queue:={i};{for reachable vertices}
while nonempty(queue) do begin
      visit(i);
      queue:=queue+unvisited_neighbors(i);
end;
connected:=true;
for i in vertices do begin
      if unvisited(i) then
            connected:=false;
```

Algorithm analysis

- the for-cycle takes place at most n-times.

Algorithm analysis

- the for-cycle takes place at most n-times.
- while-cycle passes for each vertex at most once and inspects the neighbors of this vertex.

Algorithm analysis

- the for-cycle takes place at most n-times.
- while-cycle passes for each vertex at most once and inspects the neighbors of this vertex.
- Complexity depends on the representation (how quickly we find the neighbors of a given vertex).

Algorithm analysis

- the for-cycle takes place at most n-times.
- while-cycle passes for each vertex at most once and inspects the neighbors of this vertex.
- Complexity depends on the representation (how quickly we find the neighbors of a given vertex).
- Complexity is $\Omega(m)$ (Each edge must be inspected).

Algorithm analysis

- the for-cycle takes place at most n-times.
- while-cycle passes for each vertex at most once and inspects the neighbors of this vertex.
- Complexity depends on the representation (how quickly we find the neighbors of a given vertex).
- Complexity is $\Omega(m)$ (Each edge must be inspected).
- Considering a list of edges incident to each vertex, complexity is $O(m)$,

Algorithm analysis

- the for-cycle takes place at most n-times.
- while-cycle passes for each vertex at most once and inspects the neighbors of this vertex.
- Complexity depends on the representation (how quickly we find the neighbors of a given vertex).
- Complexity is $\Omega(m)$ (Each edge must be inspected).
- Considering a list of edges incident to each vertex, complexity is $O(m)$,
- in the adjacency matrix, complexity is $O(n^2)$,

Algorithm analysis

- the for-cycle takes place at most n-times.
- while-cycle passes for each vertex at most once and inspects the neighbors of this vertex.
- Complexity depends on the representation (how quickly we find the neighbors of a given vertex).
- Complexity is $\Omega(m)$ (Each edge must be inspected).
- Considering a list of edges incident to each vertex, complexity is $O(m)$,
- in the adjacency matrix, complexity is $O(n^2)$,
- considering the incidence matrix, complexity may be $\Theta(mn^2)$.

Algorithm analysis

- the for-cycle takes place at most n-times.
- while-cycle passes for each vertex at most once and inspects the neighbors of this vertex.
- Complexity depends on the representation (how quickly we find the neighbors of a given vertex).
- Complexity is $\Omega(m)$ (Each edge must be inspected).
- Considering a list of edges incident to each vertex, complexity is $O(m)$,
- in the adjacency matrix, complexity is $O(n^2)$,
- considering the incidence matrix, complexity may be $\Theta(mn^2)$.
- Thus a good representation yields the complexity $\Theta(m+n)$.

Remarks

- If we use a queue, we are implementing the wave-algorithm (BFS).

Remarks

- If we use a queue, we are implementing the wave-algorithm (BFS).
- We may use a buffer and get a DFS.

Remarks

- If we use a queue, we are implementing the wave-algorithm (BFS).
- We may use a buffer and get a DFS.
- Advantages/disadvantages:

Remarks

- If we use a queue, we are implementing the wave-algorithm (BFS).
- We may use a buffer and get a DFS.
- Advantages/disadvantages:
- DFS may get implemented using recursion (and thus without an auxiliary data-structure).

Remarks

- If we use a queue, we are implementing the wave-algorithm (BFS).
- We may use a buffer and get a DFS.
- Advantages/disadvantages:
- DFS may get implemented using recursion (and thus without an auxiliary data-structure).
- BFS visits the vertex using the shortest path.

Looking for a cycle

A graph has a cycle if we return to a particular vertex while searching the graph.

```
cycle:=false; {so far no cycle}
for i in vertices do unvisit(i);
for i in vertices do
    if unvisited(i) then{new component}
    begin queue:={i};
        while(nonempty(queue)) do
        begin dequeue_from_queue_and_assing_into(i);
if(visited(i)) then
                cycle:=true;
            else  for j in neighbors(i) do
                begin queue:=queue+{j};
                    erase_edge({i,j});
                end;
    end; end;
```

Tree

- We may test whether the graph is connected without cycles (use previous algorithms).

Tree

- We may test whether the graph is connected without cycles (use previous algorithms).
- Or we test cycle-freeness and connectivity (one component).

Tree

- We may test whether the graph is connected without cycles (use previous algorithms).
- Or we test cycle-freeness and connectivity (one component).
- Or we test connectivity (or cycle-freeness) and an appropriate number of edges (Euler's formular).

Shortest path

When looking for the shortest path, it depends on the representation:

- Perform BFS (considering the list of vertices and edges),

Theorem

In ${A_G}^k$ position i, j gives number of walks with length k from (vertex) i to j.

Corollary

In $(A_G + I)^k$ position i, j says the number of walks of length at most k from i to j.

Shortest path

When looking for the shortest path, it depends on the representation:

- Perform BFS (considering the list of vertices and edges),
- make the power of adjacency-matrix using matrix-representation.

Theorem

In ${A_G}^k$ position i, j gives number of walks with length k from (vertex) i to j.

Corollary

In $(A_G + I)^k$ position i, j says the number of walks of length at most k from i to j.

Dijkstra's algorithm

Looks for the shortest path from a given vertex into all other vertices

Input: Graph with `nonnegatively` evaluated edges.

- We keep the "queue" for vertices ordered by the shortest so far found path.
- At the beginning we initialize the distances to all vertices [except start] by infinity [large-enough value], distance to start is 0.
- We add `start` into the queue for reachable vertices.
- Remove the first vertex of the "queue" and inspect its neighbors.
- Repeat this while the "queue" is non-empty.

Extending the path

When extending the path, for a vertex v in distance $d(v)$ we try for each edge $\{v, w\}$ whether

$$d(w) > d(v) + length(\{v, w\}).$$

If so, let $d(w) := d(v) + length(\{v, w\})$ and correct the position of w in the "queue".

Analysis

- The algorithm is proven in Invitation to Discrete Mathematics (and many other books).

Analysis

- The algorithm is proven in Invitation to Discrete Mathematics (and many other books).
- Finiteness: Each iteration removes one vertex from the "queue". This vertex never appears there anew (as all the edge-lengths are non-negative).

Analysis

- The algorithm is proven in Invitation to Discrete Mathematics (and many other books).
- Finiteness: Each iteration removes one vertex from the "queue". This vertex never appears there anew (as all the edge-lengths are non-negative).
- Partial correctness uses the invariant:
  In each iteration we have the shortes paths using only the vertices that were already removed from the "queue".

Analysis

- The algorithm is proven in Invitation to Discrete Mathematics (and many other books).
- Finiteness: Each iteration removes one vertex from the "queue". This vertex never appears there anew (as all the edge-lengths are non-negative).
- Partial correctness uses the invariant:
  In each iteration we have the shortes paths using only the vertices that were already removed from the "queue".
- Invariant shows the correctness.

Analysis

- The algorithm is proven in Invitation to Discrete Mathematics (and many other books).
- Finiteness: Each iteration removes one vertex from the "queue". This vertex never appears there anew (as all the edge-lengths are non-negative).
- Partial correctness uses the invariant:
  In each iteration we have the shortes paths using only the vertices that were already removed from the "queue".
- Invariant shows the correctness.
- The algorithm is kind of modification of BFS!

Analysis

- The algorithm is proven in Invitation to Discrete Mathematics (and many other books).
- Finiteness: Each iteration removes one vertex from the "queue". This vertex never appears there anew (as all the edge-lengths are non-negative).
- Partial correctness uses the invariant:
  In each iteration we have the shortes paths using only the vertices that were already removed from the "queue".
- Invariant shows the correctness.
- The algorithm is kind of modification of BFS!
- Complexity depends on the representation of the graph and of the 'queue'!

Remarks

- If the graph is unweighted, Dijkstra's algorithm collapses into BFS!

Remarks

- If the graph is unweighted, Dijkstra's algorithm collapses into BFS!
- Applications of graph algorithms – the whole theoretical computer science.

Remarks

- If the graph is unweighted, Dijkstra's algorithm collapses into BFS!
- Applications of graph algorithms – the whole theoretical computer science.
- Examples of graph-algorithms:

Remarks

- If the graph is unweighted, Dijkstra's algorithm collapses into BFS!
- Applications of graph algorithms – the whole theoretical computer science.
- Examples of graph-algorithms:
- Théseus and Minotaurus,

Remarks

- If the graph is unweighted, Dijkstra's algorithm collapses into BFS!
- Applications of graph algorithms – the whole theoretical computer science.
- Examples of graph-algorithms:
- Théseus and Minotaurus,
- King (and other garbage) on chessboard with some squares prohibited,

Remarks

- If the graph is unweighted, Dijkstra's algorithm collapses into BFS!
- Applications of graph algorithms – the whole theoretical computer science.
- Examples of graph-algorithms:
- Théseus and Minotaurus,
- King (and other garbage) on chessboard with some squares prohibited,
- ...

Further graph-optimization problems (algorithms)

- Minimum spanning tree – a.k.a. how to optimally place electric power lines.

Further graph-optimization problems (algorithms)

- Minimum spanning tree – a.k.a. how to optimally place electric power lines.
- Eulerian graph, i.e., can we draw all the edges of the graph by one trail (without passing any edge twice)?

Further graph-optimization problems (algorithms)

- Minimum spanning tree – a.k.a. how to optimally place electric power lines.
- Eulerian graph, i.e., can we draw all the edges of the graph by one trail (without passing any edge twice)?
- Hamiltonicity – a circle that visits all the vertices (each exactly once),

Further graph-optimization problems (algorithms)

- Minimum spanning tree – a.k.a. how to optimally place electric power lines.
- Eulerian graph, i.e., can we draw all the edges of the graph by one trail (without passing any edge twice)?
- Hamiltonicity – a circle that visits all the vertices (each exactly once),
- Clique – maximum complete subgraph,

Further graph-optimization problems (algorithms)

- Minimum spanning tree – a.k.a. how to optimally place electric power lines.
- Eulerian graph, i.e., can we draw all the edges of the graph by one trail (without passing any edge twice)?
- Hamiltonicity – a circle that visits all the vertices (each exactly once),
- Clique – maximum complete subgraph,
- Chromaticity – minimum number of colors s. t. we can color all vertices in such a way that neighboring vertices have distinct colors.

Further graph-optimization problems (algorithms)

- Minimum spanning tree – a.k.a. how to optimally place electric power lines.
- Eulerian graph, i.e., can we draw all the edges of the graph by one trail (without passing any edge twice)?
- Hamiltonicity – a circle that visits all the vertices (each exactly once),
- Clique – maximum complete subgraph,
- Chromaticity – minimum number of colors s. t. we can color all vertices in such a way that neighboring vertices have distinct colors.
- Edge chromaticity – we color edges and no pair of edges incident with a common vertex has the same color.

Further graph-optimization problems (algorithms)

- Minimum spanning tree – a.k.a. how to optimally place electric power lines.
- Eulerian graph, i.e., can we draw all the edges of the graph by one trail (without passing any edge twice)?
- Hamiltonicity – a circle that visits all the vertices (each exactly once),
- Clique – maximum complete subgraph,
- Chromaticity – minimum number of colors s. t. we can color all vertices in such a way that neighboring vertices have distinct colors.
- Edge chromaticity – we color edges and no pair of edges incident with a common vertex has the same color.
- ...

End

Thank you for your attention...