Embedding a Planar Graph using a PQ-Tree

Niklas Kotowski

RWTH Aachen

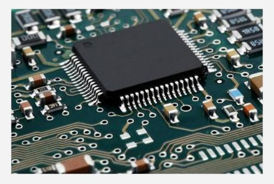
May 4, 2018

Table of Contents

motivation

Practical Use of Planarity:

- design of VLSI Circuits



- determining isomorphism of chemical structures

last week: planarity? how to test if a graph is planar ?
this week: how to give an embedding of a planar graph ?

planarity

intuitive definition

the graph can be drawn without any crossing edges

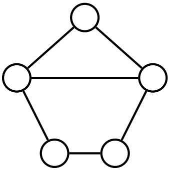
alternative definition

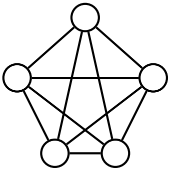
graph is planar, if the nonseperable components are planar

nonseperable component

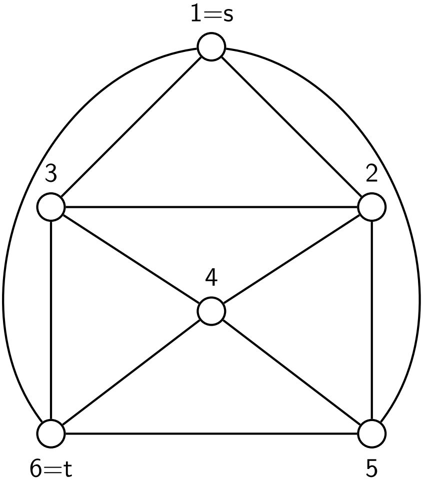
we cannot split the graph into two components by deleting any vertex

example:



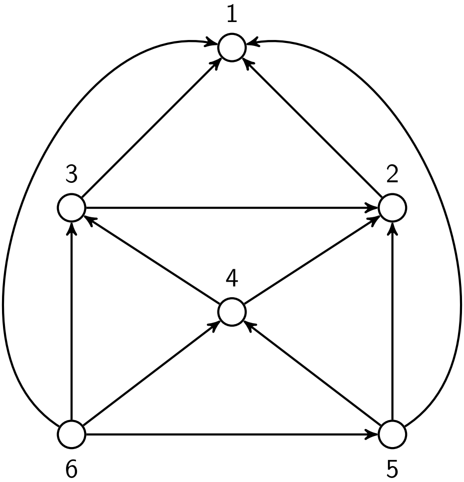


st-numbering

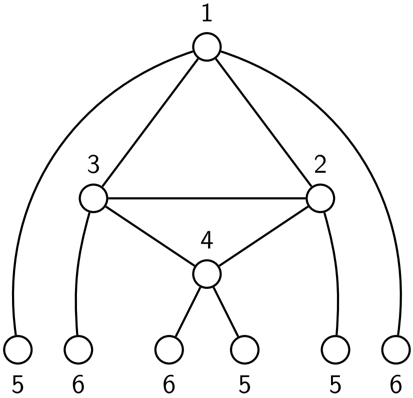


- each vertex gets a number
- 1 is called source denoted by s
- n is called sink denoted by t
- s and t have to be adjacent
- every vertex j except s and t have to fullfill:
  $i < j < k$
  for adjacent vertices i,k

upward graph



bush form B_k



- Graph G is reduced to k Nodes
- contains all **virtual edges**
- places **virtual vertices** on a horizontal line

pq-tree

P-Node:

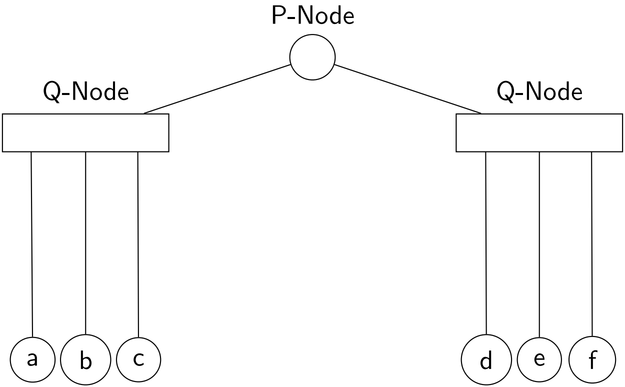
- a cut vertex
- can be permuted arbitrarily
- represented by a circle

Q-Node:

- a non seperable component
- can only be reversed
- represented by a rectangle

a pq-tree represents all possible permutations of the elements of a given set

example



- Elements are shown in the leafs a,b,c,d,e,f
- all possible combinations:
- abcdef, abcfed, cbadef, cbafed, fedabc, fedcba, defabc, defbca

PLANAR

some preliminaries regarding PLANAR:

pertinent

vertices labelled $v+1$ are pertinent

pertinent subtree

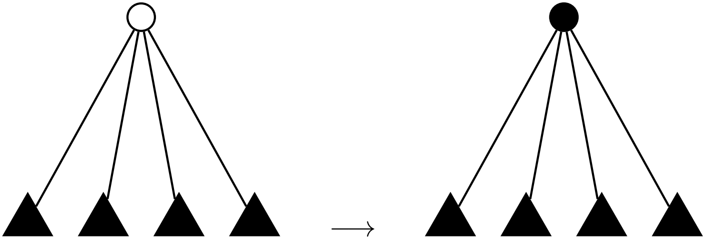
the subtree with all pertinent vertices

full

a node is full if all descendants are pertinent

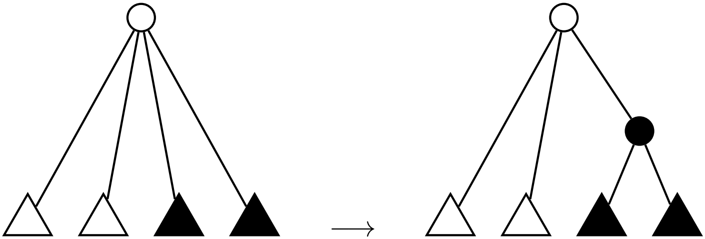
template matchings

- if a node has only full childs, the node is marked full too



template matchings

- if a node has a pair of full childnodes, a new P-node is created with the full nodes as children



PLANAR

```
initialization;
assign st-numbers to all vertices of G;
construct a PQ-tree corresponding to G_1';
begin
    for v ← 2 to n do
        reduction step → align vertices v + 1;
        if reduction step fails then
            "G is nonplanar"
        end
        vertex addition step → replace all full nodes of the PQ-tree by a
        new P-node;
    end
    "G is planar";
end
```

Algorithm 1: Planar testing algorithm

PLANAR

PLANAR

the algorithm has two steps:

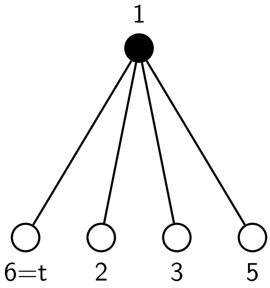
- **reduction step:**
  align the vertices v+1
- **vertex addition step:**
  replace full node by a new P-node
  add all neighbours larger than v to the P-node

⟶ let's look closer at it with an example

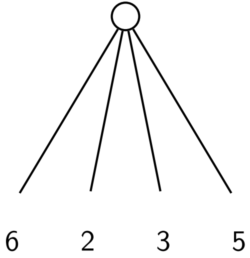
example

<u>initialize:</u>

- assign st-numbers to all vertices of G



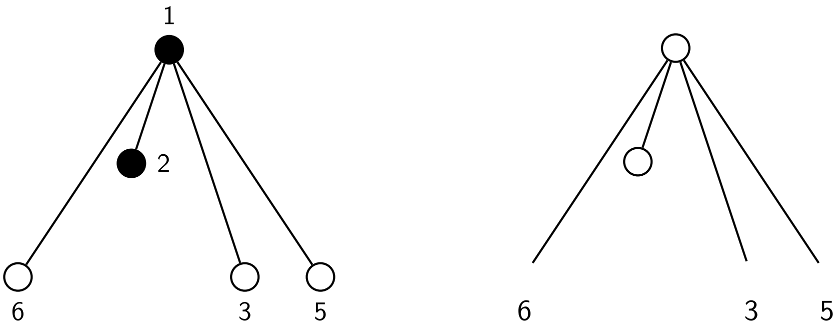
PQ-tree corresponding G_1



example

vertex addition step:

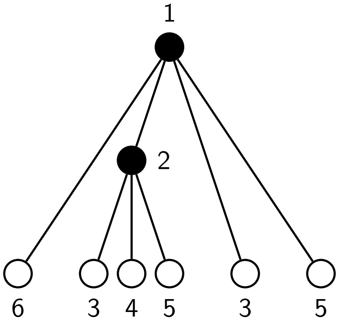
- replace full node by a new P-node
- add all neighbours larger than v to the P-node

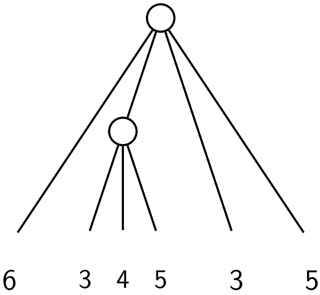


example

vertex addition step:

- replace full node by a new P-node
- add all neighbours larger than v to the P-node

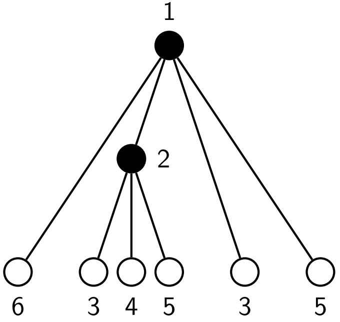


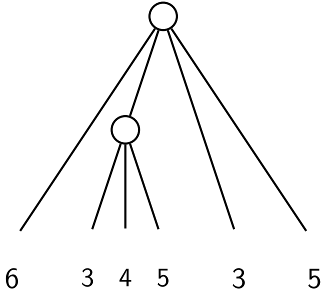


example

reduction step:

- align vertices v+1

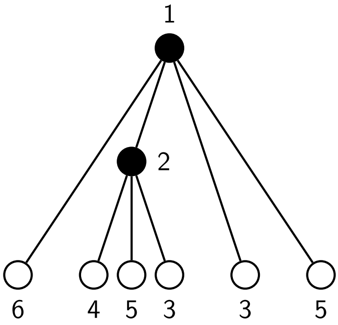


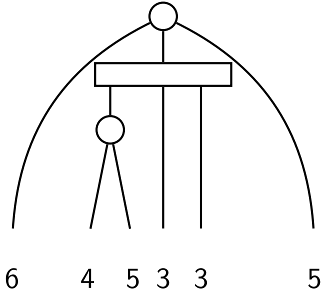


example

reduction step:

- align vertices v+1

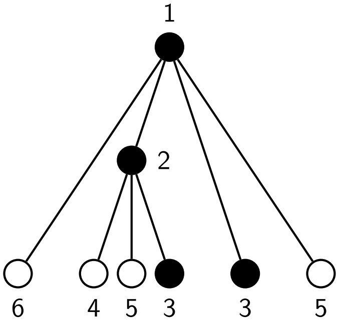


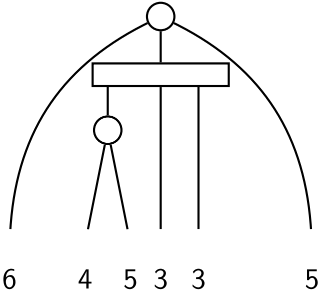


example

vertex addition step:

- replace full node by a new P-node
- add all neighbours larger than v to the P-node

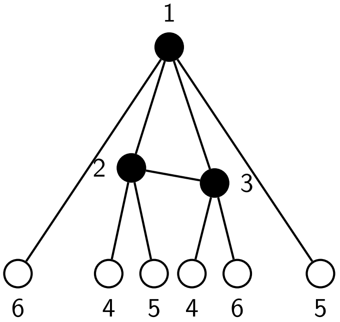


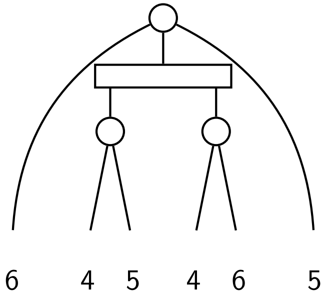


example

vertex addition step:

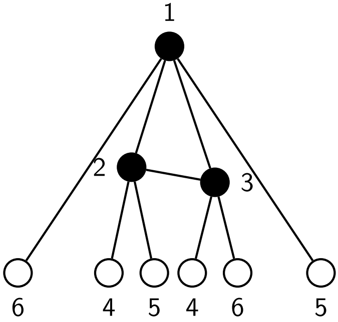
- replace full node by a new P-node
- add all neighbours larger than v to the P-node

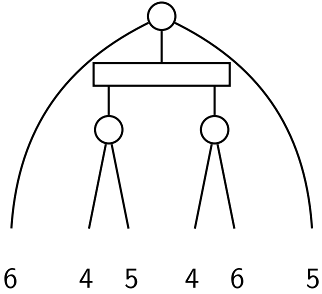




example

reduction step:

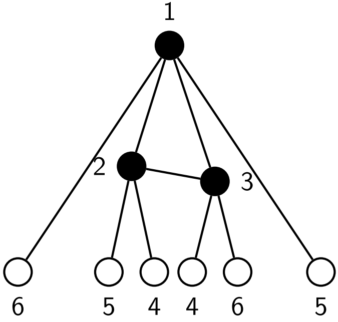


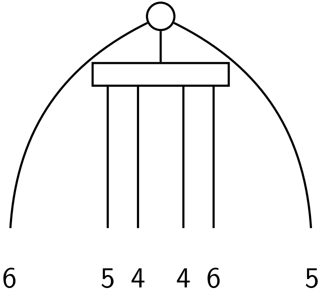


example

reduction step:

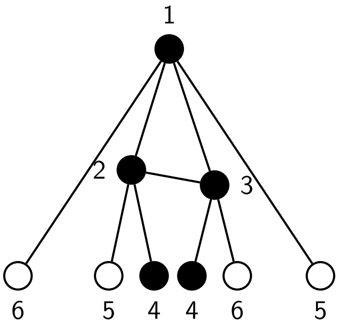
- align vertices v+1

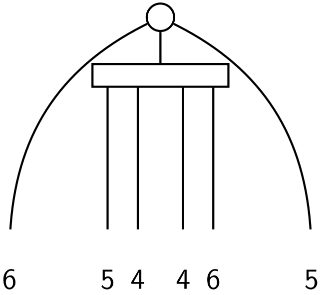




example

vertex addition step:

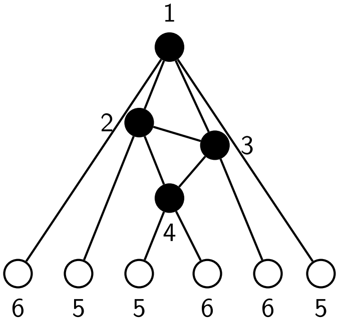


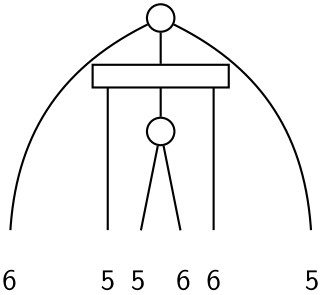


example

vertex addition step:

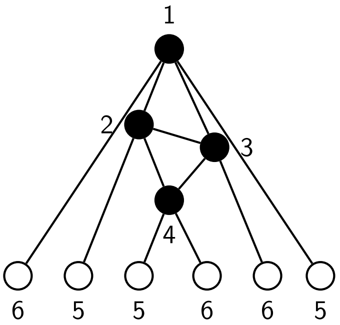
- replace full node by a new P-node
- add all neighbours larger than v to the P-node

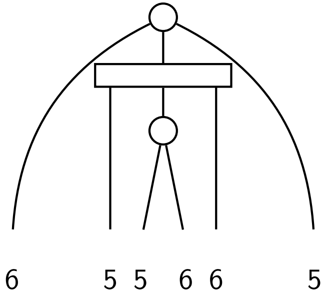




example

reduction step:

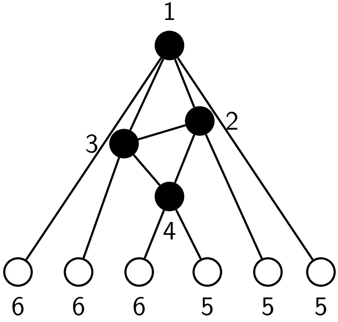


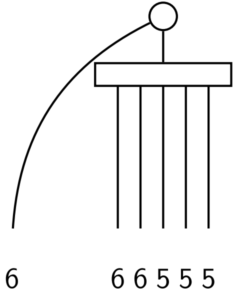


example

reduction step:

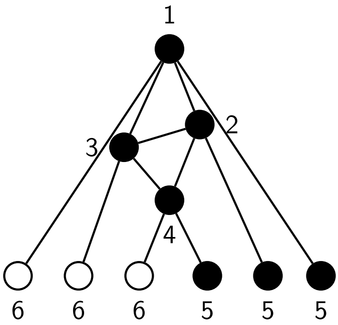
- align vertices v+1

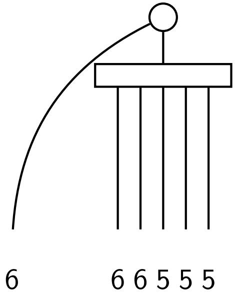




example

vertex addition step:

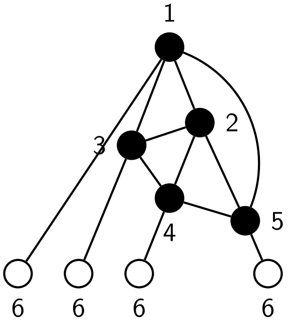


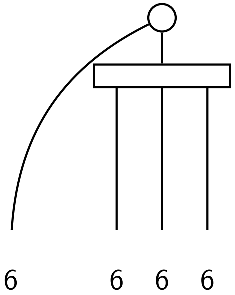


example

vertex addition step:

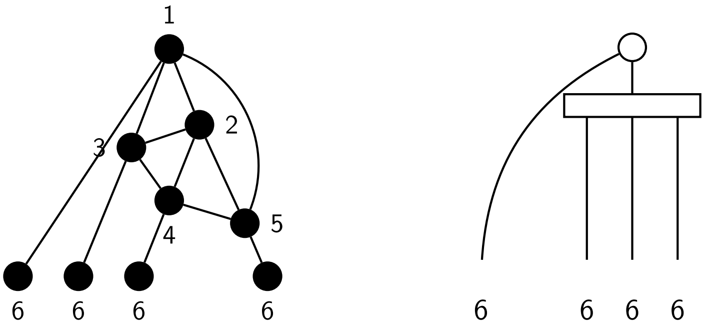
- replace full node by a new P-node
- add all neighbours larger than v to the P-node





example

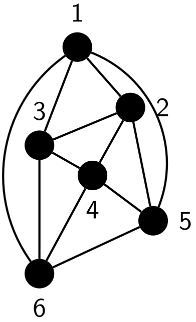
vertex addition step:



example

vertex addition step:

- replace full node by a new P-node
- add all neighbours larger than v to the P-node



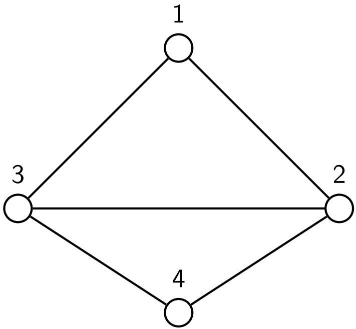
runtime

- we have **two steps** in this algorithm
- the vertex addition steps execution time depends on the vertex degree, so at most $\mathbf{O(n)}$
- the reduction step applies template matchings and aligns vertices in each step, it is not straightforward to see that all together need $\mathbf{O(n)}$
- the whole algorithm takes linear time $\mathbf{O(n)}$

embedding

until here, we can just **test** whether a given graph is planar or not ?

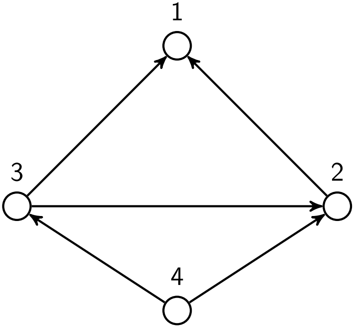
embedding of B_4:



- $Adj(1) = 2, 3$
- $Adj(2) = 1, 3, 4$
- $Adj(3) = 1, 2, 4$
- $Adj(4) = 3, 2$
- vertices ordered clockwise

upward embedding

upward embedding of B_4:



- $Adj(1) = \emptyset$
- $Adj(2) = 1$
- $Adj(3) = 1, 2$
- $Adj(4) = 3, 2$
- vertices ordered clockwise

naive embedding

algorithm:

- modify PLANAR
- in every step we write down the adjacency list of the bush form
- after n steps we have an embedding of the graph
- every step takes $O(n)$
- runtime $O(n^2)$

$O(n^2)$ is too much, therefore I present a linear time algorithm

EMBED

EMBED

algorithm runs in two phases:

- 1.phase:
  obtains an upward embedding A_u of G
  UPWARD-EMBED
- 2.phase:
  with A_u, we construct a complete embedding A of G
  ENTIRE-EMBED

ENTIRE-EMBED

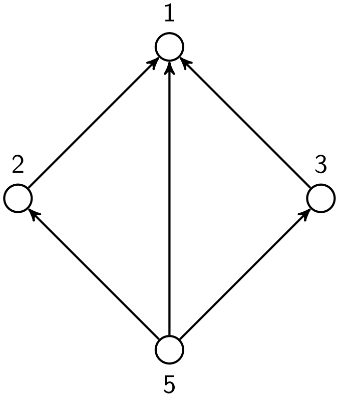
initialize:

- copy the upward embedding A_u and mark every vertex as new
- start a depth first search(DFS) on the copy

in detail DFS(x):

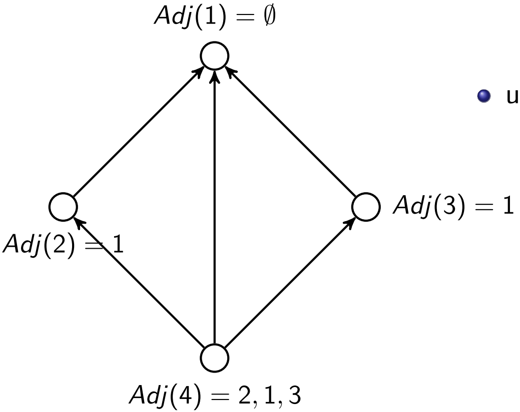
- x is marked as old
- for every adjacent vertex v insert x to the top of $A_u(v)$
- if v is marked a new, execute DFS(v)

example



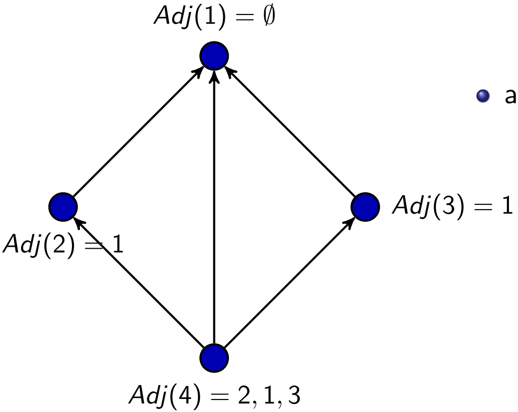
- basic graph with st-numbering

example



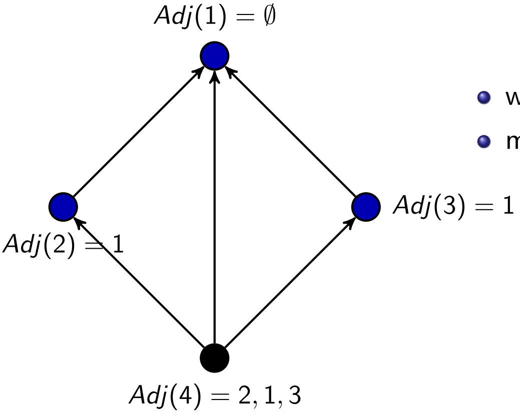
- upward embedding of the graph

example



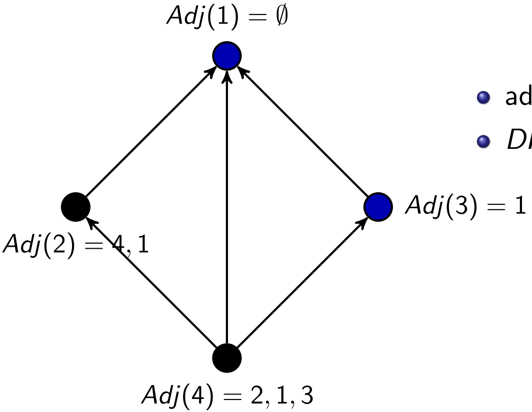
- all nodes marked as new

example



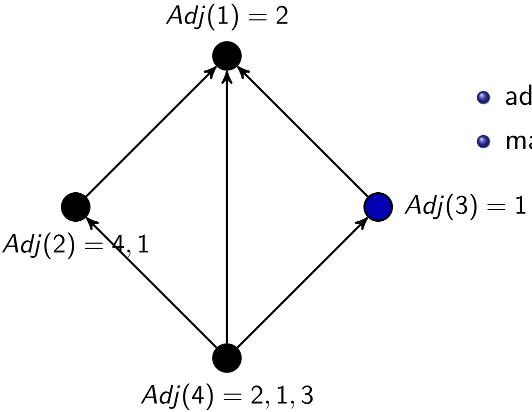
- we start with $DFS(4)$
- mark vertex 4 as old

example



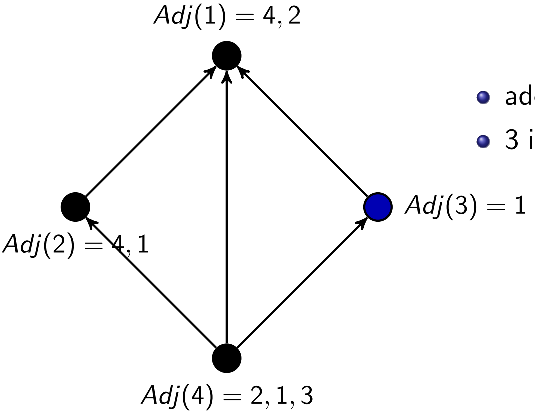
- add 4 to the top of $Adj(2)$
- $DFS(2)$ mark 2 as old

example



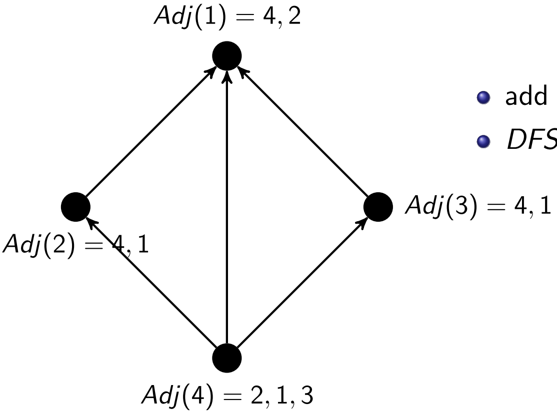
- add 2 to the top of $Adj(1)$
- mark 1 as old, DFS(1)

example



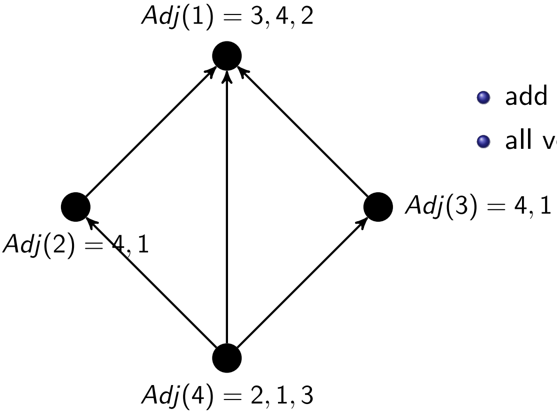
- add 4 to the top of $Adj(1)$
- 3 is the next adjacent vertex of 4

example



- add 4 to the top of $Adj(3)$
- $DFS(3)$ mark 3 as old

example



- add 3 to the top of $Adj(1)$
- all vertices marked "old"

Constructing A_u

naive algorithm:

- a naive algorithm works by scanning the leaves in each addition step
- fix the direction by counting the number of reversions made and if its odd, reverse A_u
- this easy implementation takes $O(n^2)$, thats again too much

UPWARD-EMBED:

- modification of vertex addition step, mentioned earlier
- uses direction indicator vertices
- takes linear time $O(n)$

direction indicator

- a **node** represented by a **triangle**

  pointing left: ◁

  pointing right: ▷

- traces the **reversions** of A_u
- will be reversed with each parent reversion
- **indicates** the **order** of the adjacency list

UPWARD-EMBED

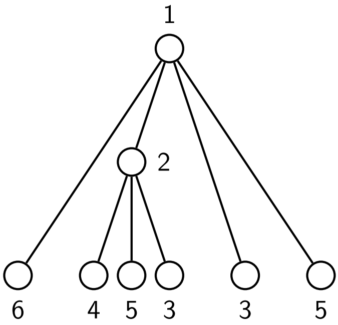
Two things are different in the vertex addition step:

- **when** and **where** to add the indicators ?
  - we add a query which tests if the root of the pertinent subtree is not full
  - if not, we add a direction indicator pointing from the higher labelled node to the lower as a childnode
  - else we do the regular vertex addition step
- **when** to reverse the adjacency list ?
  - for each element x in A_u check if it is a direction indicator
  - delete x and if the direction is opposite, reverse the list

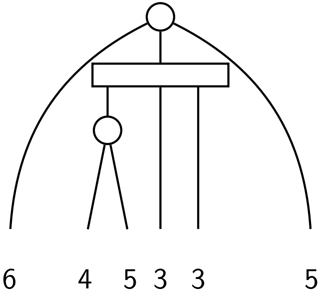
example

new vertex addition step:

- if the root of the pertinent subtree is not full



PQ-tree corresponding B_2'



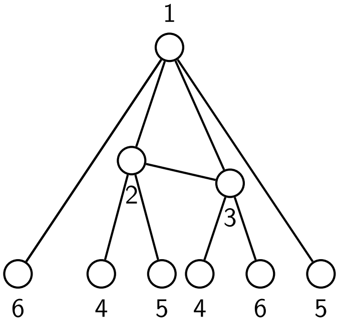
example

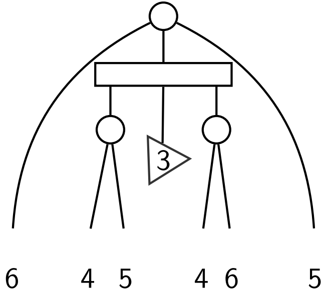
new vertex addition step:

- if the root of the pertinent subtree is not full
- add an indicator directed from l_k to l_1

PQ-tree corresponding B_3'

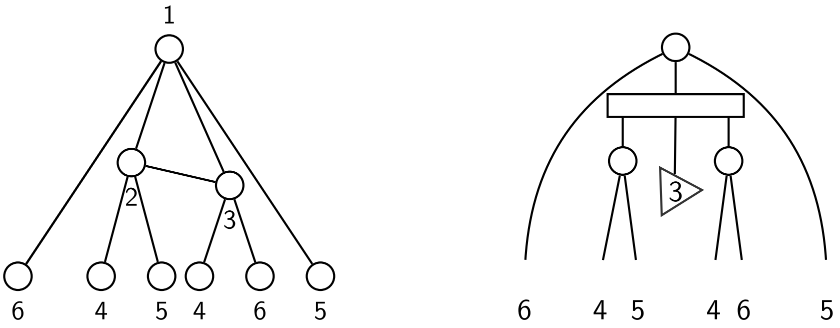
$A_u(3) = 2, 1$





example

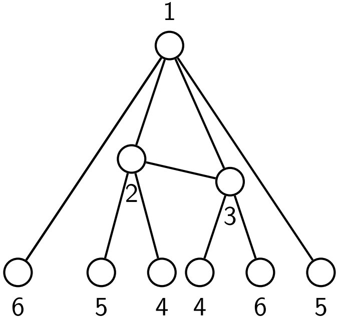
reduction step:

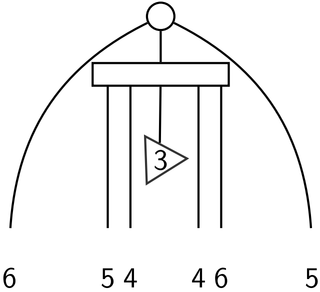


example

reduction step:

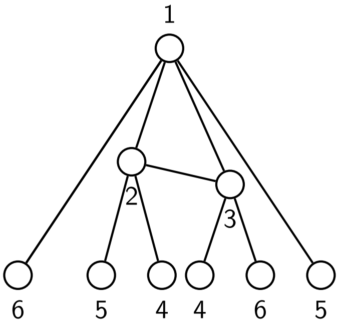
- align vertices v+1

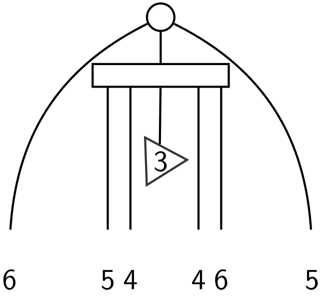




example

new vertex addition step:





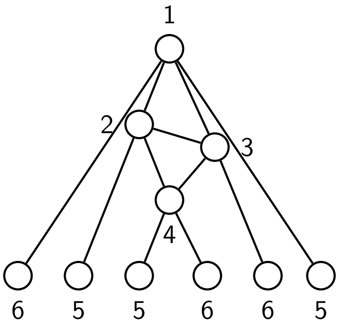
example

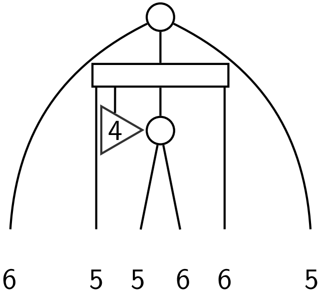
new vertex addition step:

- if the root of the pertinent subtree is not full
- add an indicator directed from l_k to l_1

PQ-tree corresponding B'_3

$$A_u(4) = 2, di(3), 3$$





example

correction step:

- for **each direction indicator x**, delete x and check if it has the opposite direction to that of $A_u(v)$, if yes **reverse** the list $A_u(x)$
- $A_u(2) = 1$
- $A_u(3) = 1, 2$
- $A_u(4) = 2, \overrightarrow{di(3)}, 3$
- $A_u(5) = 4, \overleftarrow{di(4)}, 2, 1$
- $A_u(6) = 1, 3, 4, 5, \overrightarrow{di(5)}$ $\rightarrow$ **no reversion!**

example

correction step:

- for **each direction indicator x**, delete x and check if it has the opposite direction to that of $A_u(v)$, if yes **reverse** the list $A_u(x)$
- $A_u(2) = 1$
- $A_u(3) = 1, 2$
- $A_u(4) = 2, \overrightarrow{di(3)}, 3$
- $A_u(5) = 4, \overleftarrow{di(4)}, 2, 1 \rightarrow$ **reverse** $A_u(4)$
- $A_u(6) = 1, 3, 4, 5$

example

correction step:

- for **each direction indicator x**, delete x and check if it has the opposite direction to that of $A_u(v)$, if yes **reverse** the list $A_u(x)$
- $A_u(2) = 1$
- $A_u(3) = 1, 2$
- $A_u(4) = 2, \overrightarrow{di(3)}, 3 \rightarrow A_u(4) = 3, \overleftarrow{di(3)}, 3 \rightarrow$ **reverse** $A_u(3)$
- $A_u(5) = 4, 2, 1$
- $A_u(6) = 1, 3, 4, 5$

example

correction step:

- for **each direction indicator x**, delete x and check if it has the opposite direction to that of $A_u(v)$, if yes **reverse** the list $A_u(x)$
- $A_u(2) = 1$
- $A_u(3) = 1, 2 \rightarrow A_u(3) = 2, 1$
- $A_u(4) = 3, 2$
- $A_u(5) = 4, 2, 1$
- $A_u(6) = 1, 3, 4, 5$
- **no** $di(x)$ **left !**

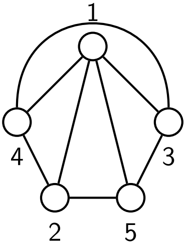
runtime

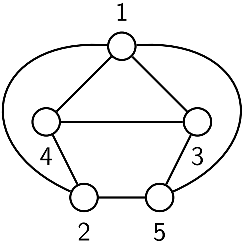
runtime

- we add our direction indicators directly into our adjacency lists
- in the last step we check the lists and reverse them if the indicator is reversed
- at maximum n indicators, therefore $O(n)$ runtime

find all possible embeddings

- what if we want to gather **all possible embeddings** ?
- we have to permute and reverse the adjacency lists regarding different rules





- first of all we have to write down some **definitions** and
- there are different graph structures we have to handle correctly

find all possible embeddings

idea of the algorithm

- we can categorize different parts of the adjacency lists to be reversed or permuted
- we are adding parantheses and brackets to display the possible operations
- if we execute all possible operations we get all planar embeddings

find all possible embeddings

$\{x, y\}$ pair of vertices in G

equivalence classes of edges E_i

two edges are in the same **class** if the edges lie on the same path and contain any vertex $\{x, y\}$ only as an end vertex

seperation pair

if there exist at least two equivalence classes E_i, E_j with minimal 2 elements each, the selected pair $\{x, y\}$ is labelled **seperation pair**

split-component

a subgraph $G_i = (V_i, E_i)$ induced by an equivalence class is called an $\{x, y\}$ **split-component**

find all possible embeddings

$\{x, y\}$ pair of vertices in G
$\{s, t\}$ as defined in the beginning is used as a reference of all embeddings

$\{s, t\}$-component

if $\{s, t\}$ is not a seperation pair, the graph without the vertices $\{s, t\}$ is labelled $\{s, t\}$-**component**

find all possible embeddings

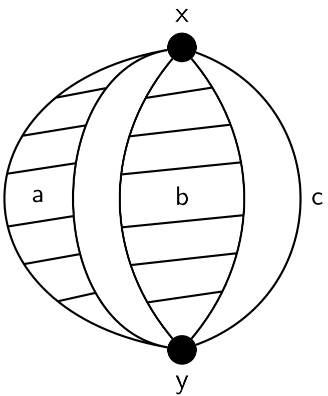
we get different embeddings for these operations:

- if $\{x, y\}$ **is a seperation pair:**
  (i) swap the $\{x, y\}$-split-components with the $\{x, y\}$-edge
  (ii) flip over the $\{x, y\}$-split-components
- if $\{x, y\}$ **is not a seperation pair:**
  (iii) reverse the $\{s, t\}$-component

- if we execute all possible operations, we get **all possible embeddings**

example

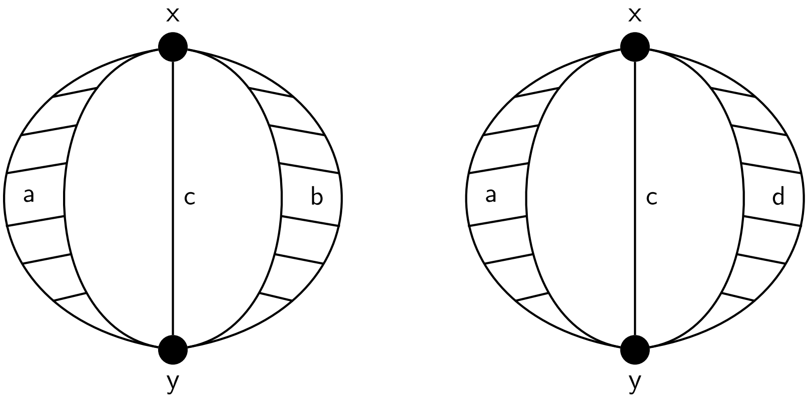
- Fig.1:



- permutation of $\{x, y\}$- split components and the edge (x,y)

example

- Fig.2:



- reverse the $\{x, y\}$- split components

Find all possible Embeddings

operations on the **adjacency lists** A_u:

(a) case (i): permute the sublists $L(u_1), L(u_2), ..L(u_l)$ of $A_u(v)$,where $u_1, .., u_l$ are the sons of v

(b) case (ii): permute the sublists $L(u_2), ..L(u_l)$ of $A_u(t)$,where $u_2, .., u_l$ are the sons of t

(c) case (iii): reverse the sublists $L(u_1), L(u_2), ..L(u_l)$ of $A_u(v)$,where $u_1, .., u_l$ are the sons of v

Find all possible Embeddings

formal structures applied to the adjacency lists:

- we define $L(v)$ to be the list which contains all descendants of the PQ-tree node v
- we use parenthese to signalize that we can permute a sublist $L(u_i)$
- we use brackets to display a possible reversion of the sublist $L(u_i)$

example:

- $A_u(v) = (L(u_1), L(u_2), L(u_3))$
- indicates, a possible permutation of u_1, u_2 and u_3

GENERATE

GENERATE

- apply the operations a), b) and c) to the sublists to specify all possible permutations and reversions
- UPWARD-EMBED
- ENTIRE-EMBED

Questions

thank you for your attention

any questions ?