Implementation of algorithms and data structures

10. seminar

Jirka Fink

https://ktiml.mff.cuni.cz/~fink/

Department of Theoretical Computer Science and Mathematical Logic
Faculty of Mathematics and Physics
Charles University in Prague

Summer semestr 2021/22
Last change 4. května 2022

Building an alternating tree

Initialization of M-alternating tree T on vertices $A \dot\cup B$

$T = A = \emptyset$ and $B = \{r\}$ where r is an M-exposed root.

Use $uv \in E$ to extend T

Input: An edge $uv \in E$ such that $u \in B$ and $v \notin A \cup B$ and v is M-covered.

Action: Let $vz \in M$ and extend T by edges $\{uv, vz\}$ and A by v and B by z.

Use $uv \in E$ to augment M

Input: An edge $uv \in E$ such that $u \in B$ and $v \notin A \cup B$ and v is M-exposed.

Action: Let P be the path obtained by attaching uv to the path from r to u in T. Replace M by $M \triangle E(P)$.

Definition

M-alternating tree T is M-frustrated if every edge of G having one end vertex in B has the other end vertex in A.

Algorithm for finding a maximum matching in bipartite graphs

$M := \emptyset$
for $r \in V$ **do**
  $A := \emptyset$, $B = \{r\}$ # Create M-alternating tree from r
  while *r is M-exposed and exists $uv \in E$ with $u \in B$ and $v \notin A$* **do**
    if *v is M-exposed* **then**
      Use uv to augment the matching M
    else
      Use uv to extend the tree T
return Maximal matching M

While loop is implemented using breath search first

- When extending a tree, the vertex inserted to B is also added to a queue
- While loop is implemented using two loops
  - The outer loop process all vertices in the queue
  - The inner loop tests all edges incident to the vertex u
- Every edges is tested at most once from each end-vertex

Algorithm for finding a maximum matching in bipartite graphs

Data representation

- Graph
  - Array of vertices
- Vertex
  - Array/linked list of pointers/references/indices of neighbor vertices
  - Pointer to the matched vertex (NULL for exposed vertices)
  - Pointer to a parent in the alternating tree
  - Flag determining whether the vertex belong to A or B or neither
- Queue of vertices

Notes

- The parent pointer is needed to find an augmenting path
- No information is stored on edges, so structure for edges is not needed

Odd cycles

Use uv to shrink and update M' and T'

Input: A matching M' of a graph G', an M'-alternating tree T', edge $uv \in E'$ such that $u, v \in B'$

Action: Let C be the circuit formed by uv together with the path in T' from u to v.

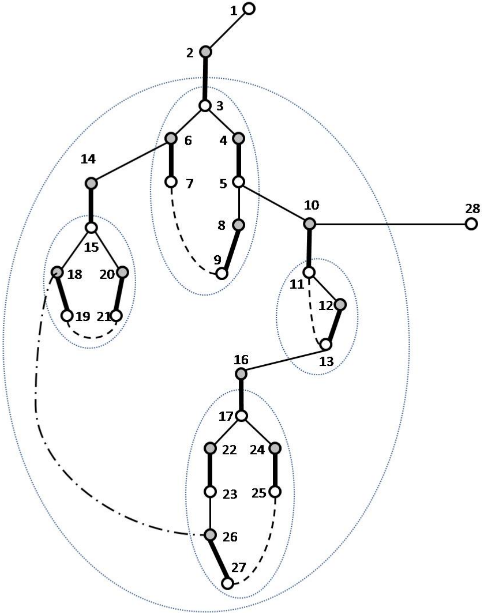
Replace

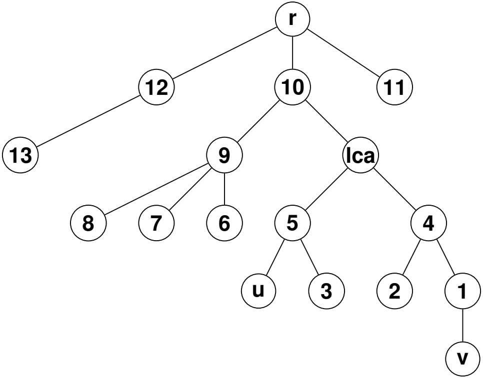
- G' by $G' \times C$
- M' by $M' \setminus E(C)$
- T by the tree having edge-set $E(T) \setminus E(C)$
- $A' := A' \setminus V(C)$
- $B' := B' \setminus V(C) \cup \{c'\}$ where c' is a new pseudo-vertex

Implementation

- Vertices are not contracted, only store a pointer to pseudo-node
- For pseudo-nodes, union-find disjoint data structure is used
- There is no expansion of cycles, only the union-find is initialized
- In a contraction, all vertices of A on the cycle are inserted to the queue
- How to recognize vertices of the cycle?
- How to find an augmenting path?

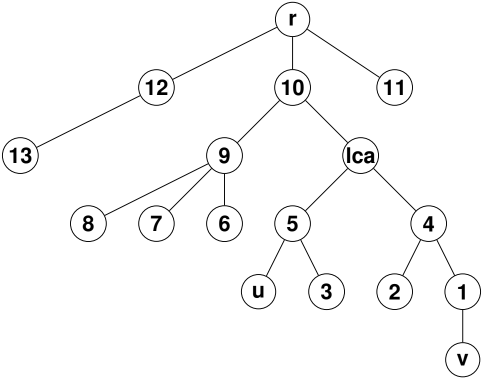
Example (author: Uri Zwick)





- For a given tree and two vertices u and v find the lowest common ancestor on paths from u and v to the root
- For an edges uv joining vertices $u, v \in B$, the odd cycles C is formed by vertices on paths from u and v to lca
- Vertex lca has to be found in time $\mathcal{O}(|C|)$

Lowest common ancestor (LCA): Algorithm



- Add a flag to the structure for vertices to mark predecessors of *u* and *v*
- Initialize the flag by false
- Alternately walk from *u* and *v* to the root and mark visited vertices
- LCA is the first visited vertex that is already marked

Contraction using union-find

- Add a variable uf to the structure for vertices
- Initialize uf to NULL which means that a vertex is not contracted
- Contraction sets uf of all vertices on the cycle to lca(u,v)
- Keep in mind that contacted vertices no longer exists in the graph
- Keep in mind that a pseudo-node can also be contracted
- To find a pseudo-node where a vertex u was contracted, walk on uf to the root of the union-find
  Denote this vertex by Find(u)
- Keep in mind that we use two different trees (forests)
  - Alternating tree
  - Forest of contracted cycles

Maximum matching in general graphs

```
For all vertices u: u.match = NULL
for r ∈ V do
    For all vertices u: u.parent = u.uf = NULL, u.status = NONE
    queue = (r), r.status = B
    while r.match ≠ NULL and queue is not empty do
        u = dequeue()
        for v neighbor of u do
            # Skip vertices contracted to the same pseudo-node
            if Find(u) ≠ Find(v) then
                if v.status == B or v.uf ≠ NULL then
                    Use uv for contraction
                else if v.status == NONE a v.match == NULL then
                    Use uv for augmenting M
                    break # Terminate the inner cycle
                else if v.status == NONE a v.match ≠ NULL then
                    Use uv for extending T
return Maximum matching M
```

Contraction

shrink_cycle(u,v)

To find augmenting path, we need a new variable bridge for every vertex storing the edge that causes the contraction

```
lca = lowest_common_ancestor(u, v)
shrink_path(lca,u,v)
shrink_path(lca,v,u)
```

shrink_path(lca,end,other)

```
x = find(end)
while x ≠ lca do
    union(x,lca)
    x = x.parent

    union(x,lca)
    enqueue(x)
    x.bridge = (end,other)
    x = find(x.parent)
```

Finding augmenting path

Find augmenting path from u to the root of T using recursion

path(x,end) =

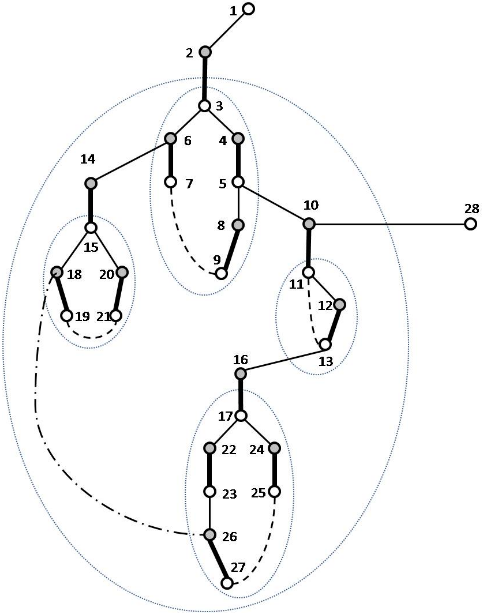
- if x == end:
  (end)
- if x $\neq$ end and x.status == B:
  (x, x.parent) + path(x.parent.parent,end)
- else:
  reverse(path(x.bridge[1],x)) + path(x.bridge[2],end)

Augmenting path is (v) + path(u,root) where uv is the edges used for augmenting our matching

Implementation

We do not need to construct the path, we only traverse it and alternate matching edges.

Example (autor: Uri Zwick)



Disjoint-set data structure

Forest

- A forest has one vertex for every element
- One of the forest corresponds to one set
- Every vertex u stores its parent $p[u]$ in the forest
- The parent of a root of a tree is NULL $\Rightarrow$ initialize $p[u] = NULL$ for every vertices
- Every root stores the size of its tree

Union(u,v)

- Find roots u', v' of trees containing u, v
- If u' contains more element than v', then u' become the parent of v'

Find(u): Find the root of u

- Find the root u' of u
- For all vertices of the path from u do u' change the parent to be u' (except u')
- The amortized complexity is $O(\alpha(n))$

Data representation for finding a maximum matching in general graphs

- Graph
  - Array of vertices
- Vertex
  - Array/linked list of pointers/references/indices of neighbor vertices
  - Pointer to the matched vertex (NULL for exposed vertices)
  - Pointer to a parent in the alternating tree
  - Flag determining whether the vertex belong to A or B or neither
  - Pointer to the parent in union-find data structure
  - Size of Union-find tree
  - Flag for finding lca
  - A pair for pointers to vertices for finding augmenting path
- Queue of vertices

Creating one alternating tree

- Extending tree: $\mathcal{O}(1)$, $\mathcal{O}(n)$-times
- Augmenting matching: $\mathcal{O}(n)$, only once
- Contracting odd cycle C: $\mathcal{O}(|C|\alpha(n))$
  The sum of length of contracted cycles is $\mathcal{O}(n)$
  $\Rightarrow$ Complexity is $\mathcal{O}(n\alpha(n))$
- Breath search first and calling Find on end-vertices of an edge $\mathcal{O}(\alpha(n))$, $\mathcal{O}(m)$-times
- Building one alternating tree takes $\mathcal{O}(m\alpha(n))$

Time compolexity of whole algorithm

- Formally: $\mathcal{O}(n(n+m)\alpha(n))$
- But the algorithm can be run on every component independently
- Time complexity of the algorithm is $\mathcal{O}(nm\alpha(n))$

Hints

Experience

- Algorithm for bipartite graph finds a matching in general graph but it may not be maximal which can be used for creating tests
- For creating large graphs, you can use libraries; e.g. NetworkX in Python, Boost in C++
- Test your algorithms also on huge graphs having thousands of vertices and edges

What should we do when the algorithm fails on a huge graph?

- Test reproducibility (run the program once more)
- Try to create a smaller graph using the same generator
- Try to reduce the buggy graph to find the smallest working example
  - Remove edges one by one
  - Remove isolated vertices
  - Shorted a path by two vertices and edges
  - Combine the above steps with permuting vertices on the input
- Write a script which automatize this process

Implement the algorithm step by step

1. Algorithm for bipartite graphs
2. Test data consistency, unit tests, test feasibility and optimality
3. Thoroughly test the algorithm on bipartite graphs
4. Create tests for general graphs and find graphs on which the algorithm does not find a maximal matching
5. Add variables for general graphs and extend data consistency tests
6. Finding LCA
7. Slower version of union-find
8. Construction odd cycles
9. Finding augmenting path
10. Test everything
11. Faster version of union-find
12. Test everything again
13. Generate as large graph as it fits to your memory
14. Test everything again
15. Submit
16. Enjoy your holidays

Literature and other resources

- Cunningham, Cook, Pulleyblank, Schrijver: Combinatorial optimization, John Wiley & Sons, 1997 (book chapter)
- Jan Vondrák: Polyhedral techniques in combinatorial optimization, 2010 (lecture notes) https://theory.stanford.edu/~jvondrak/CS369P/lec4.pdf
- Michel X. Goemans: Combinatorial Optimization (lecture notes) http://math.mit.edu/~goemans/18433S15/matching-notes.pdf http://math.mit.edu/~goemans/18433S15/matching-nonbip-notes.pdf
- Uri Zwick: Lecture notes on: Maximum matching in bipartite and non-bipartite graphs (lecture notes) https://www.cs.tau.ac.il/~zwick/grad-algo-0910/match.pdf
- Visualization: https://www-m9.ma.tum.de/graph-algorithms/matchings-blossom-algorithm/