Graph Decomposition

— DFS&BFS, Cycle, DAG, SCC, and Biconnectivity

hengxin0912@gmail.com

May 19, 2016

Graph Decomposition

Contents of Tutorials

1. Graph ~~Traversal~~ Decomposition
2. MST (Greedy Algorithm) & Path
3. DP: Dynamic Programming

Graph Decomposition

Graph decomposition vs. Graph traversal

Graph Decomposition

Classifying edges

Definition (Classifying edges)

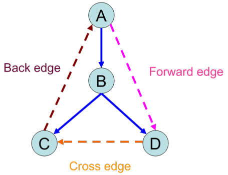
Given a dfs/bfs traversal:

- Tree edge
- Back edge
- Forward edge
- Cross edge

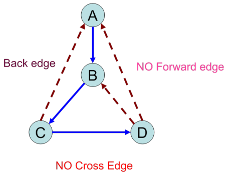
Remarks:

- Applicable to both DFS and BFS
- With respect to DFS/BFS trees

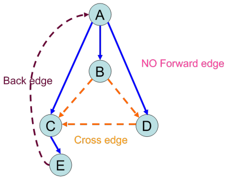
Classifying edges [Problem: 3.4.1]



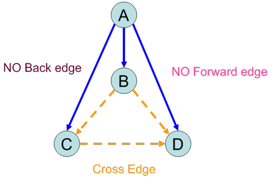
(a) DFS on directed graph.



(b) DFS on undirected graph.



(c) BFS on directed graph.



(d) BFS on undirected graph.

Classifying edges

DFS tree and BFS tree coincide [Problem: 3.4.30]

$G = (V, E), v \in V$. DFS tree T = BFS tree T'.

- G is an undirected graph $\Rightarrow G = T$.
- G is a digraph $\Rightarrow^{?} G = T$.

Solution.

- T: tree + back; T': tree + cross
- T: tree + back + forward + cross; T': tree + back + cross

Distance constraints for BFS

Distance constraints for BFS [Problem: 3.4.4]

BFS on digraph:

$$\text{TE: } d[v] = d[u] + 1$$

$$\text{BE: } 0 \leq d[v] \leq d[u]$$

$$\text{CE: } d[v] \leq d[u] + 1$$

BFS on undirected graph:

$$\text{TE: } d[v] = d[u] + 1$$

$$\text{CE: } d[v] = d[u] \vee d[v] = d[u] + 1$$

Solution to "*CE* in BFS on *undirected* graph".

- $d[v] = d[u], d[v] = d[u] + 1$
- $d[v] < d[u], d[v] > d[u] + 1$

Remark.

- BFS tree defines a *shortest-path* from its root to every other node.
- Layers in BFS on *undirected* graph $\Rightarrow$ bipartite testing [Problem: 3.4.26]

Graph Decomposition

Cycle detection

Table: Cycle detection [Problem: 3.4.21]

	Digraph	Undirected graph
DFS	back edge $\iff$ cycle	back edge $\iff$ cycle
BFS	back edge $\Rightarrow$ cycle cycle $\not\Rightarrow$ back edge	cross edge $\iff$ cycle

Remark.

- cycle in undirected graphs
- cycle in digraphs
  - DAG
  - SCC

Orientation of undirected graph

Orientation of undirected graph [Problem: 3.4.13]

Undirected (connected) graph G, edge oriented s.t. $\forall v, \text{in}[v] \geq 1$.

Solution.

orientation $\iff$ $\exists$ cycle; DFS

Bipartite graph

Bipartite graph [Problem: 3.4.26; 3.4.32]

To test bipartiteness of an undirected graph.

Solution.

BFS + Coloring:

- pick any s, c[s] $= 0$
- $u \leftarrow$ Dequeue(Q)
- $\forall (u, v)$:
  - tree edge
  - cross edge: check

Proof.

Check cross edge (u, v):

- $(\exists)$ $d[v] = d[u] \Rightarrow$ the same layer $\Rightarrow$ odd cycle ([Problem: 3.4.17]; EX) $\Rightarrow$ non-bipartite
- $(\forall)$ $d[v] = d[u] + 1 \Rightarrow$ different layers $\Rightarrow$ different colors

$\square$

Graph Decomposition

DAG

$$\text{no back edge} \iff \text{DAG} \iff \exists \text{ topo. ordering}$$

Topo. sorting

DFS on digraph, $u \to v$:

- back edge: $f(u) < f(v)$
- others: $f(u) > f(v)$

$$u \to v \Rightarrow u \prec v$$

$$u \to v \Rightarrow f(u) > f(v)$$

Topo. sorting: sort vertices in *decreasing* order of their *finish* times.

Digraph as DAG

Digraph as DAG [Problem: 3.4.6]

Theorem

Every digraph is a dag of its SCCs.

Remark.

- SCC algorithm
- SCC: reachability/connectivity equivalence class
- two tiered structure of digraphs

Kahn's toposort algorithm

Kahn's toposort algorithm (1962) [Problem: 3.4.19]

- queue for source vertices ($\text{in}[v] = 0$)
- repeat: dequeue v, delete it, output it

Solution.

Lemma

Every DAG has at least one source and at least one sink vertex.

Remark

DFS on DAG:

- $\arg\max_v f(v) \Rightarrow$ source (used in SCC algorithm)
- $\arg\min_v f(v) \Rightarrow$ sink

Hamiltonian path in DAG

Hamilton path in DAG [Problem: 3.4.16]

- DAG G
- path visiting each vertex once

Solution.

- general digraph: NP-hard
- dag: $\exists$ HP $\iff$ $\exists!$ topo. ordering
  - $\Leftarrow$: By contridiction. $\exists u \sim v : u \nrightarrow v$; swap

Algorithm:

- toposort, check edges
- the Kahn toposort algorithm

Semi-connected DAG

Definition

Semi-connected digraph $\forall u, v : u \leadsto v \vee v \leadsto u$

Semi-connected DAG [Problem: 3.4.21 (c) + (d)]

To test whether a DAG is semi-connected.

Solution.

$$\text{dag: } \exists \text{ HP} \iff \exists! \text{ topo. ordering} \iff \text{semi-connected}$$

Proof.

- $\Leftarrow$: by contradiction; total order $(\forall u, v : u \prec v \vee v \prec u)$
- $\Rightarrow$: $\exists HP$

□

Minimum cost reachable

Minimun cost reachable [Problem: 3.4.22]

Compute cost$[u] = \min\{\text{cost}[v] \mid u \rightsquigarrow v\}$.

Solution.

- dag: reverse topo. ordering
  - backtracking: cost$[u] = \min_{u \to v}\{\text{cost}[v]\}$
- digraph: dag of scc

Line up

Line up [Problem: 3.4.29]

- i hates j: $i \prec j$
- i hates j: $\#i < \#j$

Solution.

i hates j: $i \to j$;

- DAG?
- longest path; critical path

One-to-all reachability

One-to-all reachability [Problem: 3.4.28]

- given $v : v \leadsto^{?} \forall u$
- $\exists ? v : v \leadsto \forall u$

Solution.

- DFS/BFS
- SCC; $\exists!$ source vertex $v \iff v \leadsto \forall u$

Proof.

- $\Rightarrow$: By contradiction. $\exists u : v \not\leadsto u \wedge \text{in}[u] > 0 \Rightarrow \exists u' \to u \wedge v \not\leadsto u'$. Cycle.
- $\Leftarrow$: (1) source (2) $\exists!$

$\square$

Graph Decomposition

SCC

Kosaraju SCC algorithm, 1978 [Problem: 3.4.7]

- 1st DFS $\Rightarrow^?$ BFS
- 2nd DFS $\Rightarrow^?$ BFS

Solution.

digraph = dag of SCCs

- (2nd round) Repeat: traversal from $s \in$ sink scc; delete
- (1nd round) dag: toposort $\Rightarrow$ decreasing finish time $\Rightarrow$ $s \in$ source scc

Remark.

- DFS on G^T; DFS/BFS on G
- DFS on G; DFS/BFS on G^T

One-way streets

One-way streets [Problem: 3.4.23]

- $\forall u, v : u \leftrightsquigarrow v$
- $s : s \rightsquigarrow v \rightsquigarrow s$

Solution.

- G is an SCC
- $\{v \mid s \rightsquigarrow v\}$ is an SCC
  - compute $\{v \mid s \rightsquigarrow v\}$
  - compute SCCs
  - compare

Infinite path

Infinite path [Problem: 3.4.25]

- prove: $\text{Inf}(p) \subseteq \exists$SCC
- an infinite path?
- $\ldots \land$ visiting $\exists g \in V_G$ infinitely often
- $\ldots \land$ not visiting $\exists b \in V_B$ infinitely often

Solution.

- cycle $\subseteq \exists$scc
- $\exists$scc : $s \rightsquigarrow$ scc
- $\exists$scc : $s \rightsquigarrow$ scc $\land$ scc $\cap V_G \neq \emptyset$
- delete V_B; $\exists$scc : scc $\cap V_G \neq \emptyset$; in G: $s \rightsquigarrow$ scc
  - wrong: $\exists$scc : $s \rightsquigarrow$ scc $\land$ scc $\cap V_G \neq \emptyset \land$ scc $\cap V_B = \emptyset$
  - wrong: delete V_B; $\exists$scc : $s \rightsquigarrow$ scc $\land$ scc $\cap V_G \neq \emptyset$

Odd cycle in digraph

Odd cycle in digraph [Problem: 3.4.15]

Find an odd cycle in a digraph G.

Lemma

A digraph G has an odd directed cycle $\iff$ $\exists$scc : scc is non-bipartite (when treated undirected).

Proof.

- $\Leftarrow$: undirected C; oriented
  - odd directed cycle
  - choose a direction $\forall u \to v$: Len$(v \rightsquigarrow u)$

$\square$

Remark.

DFS $+$ Coloring on G

Graph Decomposition

Biconnectivity algorithm

G is biconnected $\iff$ G has no articulation vertex.

v is an articulation vertex $\iff \exists x, y \in V : \forall p \equiv x \sim y, x \sim v \sim y$.

Biconnectivity algorithm by Tarjan&Hopcroft, 1971

Root of DFS tree as an articulation point [Problem: 3.4.8]

DFS on undirected graph $G \Rightarrow$ DFS tree T:

- leaf node
- root node r: out$[r] \geq 2$

Proof.

- $\Leftarrow$: ($G =$ tree $+$ back); delete r; disconnect subtrees
- $\Rightarrow$: By contradiction. out$[r] = 1 \Rightarrow r$ is not an articulation vertex ($\forall x, y$).

$\square$

- internal node

Biconnectivity algorithm

Articulation checking, 1971 [Problem: 3.4.10]

DFS on undirected graph $G \Rightarrow$ back edges:

$$v \text{ is not an articulation vertex}$$
$$\iff \forall \text{subtree } T_v \text{ of } v, \text{back}[T_v] = v.\text{ancestor}$$
$$v \text{ is an articulation vertex} \iff \exists T_v \text{ of } v, \text{back}[T_v] = v \vee v.\text{descendant}$$

$\exists \Rightarrow$ checking articulation (wBack $\geq$ d$[v]$) when backtracking from w to v

Biconnectivity algorithm

Initialization of back$[v]$, 1971 [Problem: 3.4.9]

$$\text{back}[v] = \infty, 2n+1 \text{ vs. } \text{back}[v] = d[v]$$

Solution.

back$[v]$: the earliest reachable ancestor of v by tree + back edges

- tree edge ($\rightarrow v$; discovered): back$[v] = d[v]$
- back edge ($v \rightarrow w$): back$[v] = \min\{\text{back}[v], d[w]\}$
- backtracking from w: back$[v] = \min\{\text{back}[v], \text{back}[w] = \text{wBack}\}$

back$[v] = \infty$:

- if updated
- if never updated: wBack $= \infty > d[v]$ vs. wBack $= d[w] > d[v]$

Bridge

Bridge [Problem: 3.4.24]