Fundamental Algorithms

Lecture #10

Cluj-Napoca

January 8, 2020

Agenda

- **Maximum Flow**
- **Connected components**
- **Conclusions on graphs**
- **B trees (definition & utilization; Operations next time)**

Maximum Flow – the method

- The simplest problem concerning flow networks
- Many real world applications
- The problem: Given G=(V, E), directed, weighted, s=source vertex; t=target vertex, find the greatest rate at which some material can be transported from s to t without violating the capacity constraints

Maximum Flow - formalism

- $\forall (u, v) \in E$, $c(u, v) >= 0$ (capacity from u to v)
- Define the **flow** $f: V \times V \rightarrow R$ with properties:

  P1 (cap constr.): $\forall u, v \in V$, $f(u, v) <= c(u, v)$

  P2 (antisymmetry): $\forall u, v \in V$, $f(u, v) = -f(v, u)$

  P3 (conservation): $\forall u \in V - \{s, t\}$, $\Sigma_{v \in V} f(u, v) = 0$
- Def: **Network flow** $|f| = \Sigma_{v \in V} f(s, v)$ (Total flow leaving the source)
- Def: **Max flow problem** in a flow network G, find the max flow value from s to t.

Maximum Flow - remarks

- **$f(u, u) = 0$**

P2: $f(u, v) = -f(v, u)$

Apply to u, u => $f(u, u) = -f(u, u) \Rightarrow f(u, u) = 0$

- If **$f(u, v) <> 0$** then $(u, v) \in E$ or $(v, u) \in E$

Consequence of the previous obs.

Assume $(u, v) \neg\in E \Rightarrow c(u, v) = 0$

$(v, u) \neg\in E \Rightarrow c(v, u) = 0$

P1 => $f(u, v) <= c(u, v)$

$f(v, u) <= c(v, u)$

$\Rightarrow f(u, v) <= 0$ and $f(v, u) <= 0 \Rightarrow f(u, v) = f(v, u) = 0$!

Maximum Flow – the method – contd.

- A method/strategy, not an alg.
- Different implementations => different running times
- Iterative approach
- Starts with no flow (value 0) on each edge
- At each iterative step increases the flow value on an augmented path (with the max allowed val)
- Repeats the above step until no augmented path exists
- The **min-cut theorem** proves the process computes the max flow.

Maximum Flow – additional constructs

- **Residual network**

  = defines the remaining capacity after some flow passes a given edge $(u, v) \in E$,
  with $c(u, v)$, and $f(u, v)$

$$\mathbf{c_f(u, v) = c(u, v) - f(u, v)}$$

flow from v to u (even in the absence of v, u capacity)

$$\begin{aligned} c_f(v, u) &= c(v, u) - f(v, u) = \\ &= 0 - f(v, u) = \\ &= f(u, v) \text{ (P2)} \end{aligned}$$

(ex. Blackboard)

Maximum Flow – additional constructs

- **Augmented path**

A *simple path* from s to t on which some *residual capacity* exists

By adding the residual capacity to the flow network in G, the flow will be closer to max

Repeat until no augmented path exists completes the method

Maximum Flow – additional constructs

- **Cut in a flow network**

  def: is a partition of the V set;

  $(S, T), S \cup T = V, S \cap T = \varnothing, s \in S, t \in T$

  Not: f(S, T) = network flow across the cut

  c (S, T) = capacity of the cut (ex: blackboard)

$f(S, T) = f(1, 3)+ f(2, 3) + f(2, 4) = 12 + (-4) + 11 = 19$

$c(S, T) = c(1, 3) + c(2, 4) = 12 + 14 = 26$

Obs: flow negative components (based on P2), yet capacity only positive components

Properties of the flow network

Lema: Let f the flow in a flow network G,

and (S, T) a cut in G,

then $f(S, T) = |f|$

i.e. cut flow = net flow (without proof; check textbook)

Corollary: the *value of any flow f* in G is upper bounded by the *capacity of any cut* in G

Max flow min cut Theorem

If f is a flow in a flow network G,

and s = source, t = target,

the following conditions are equivalent:

1. F is the ***max flow*** in G
2. The residual network Gf contains ***no augmented path***
3. ***|f| = c(S, T)*** **for some cut** (S, T)

Proof:

1=> 2 contradiction

2=> 3 contradiction + Lema

3 => 1 Lema

Ford – Fulkerson alg.

```
Ford - Fulkerson (G, s, t)
for each edge (u, v) ∈E
  do f[u, v] <- 0
     f[v, u] <- 0
while ∃ a path from s to t in the residual network Gf
  do cf(p)= min {cf(u,v)| (u,v) ∈p}
     for each edge (u,v) ∈p
          do  f[u, v] <- f[u, v] + cf(p)
              f[v, u] <- f[u, v]
```

Ford – Fulkerson alg. analysis

- The running time depends on how the augmented path is found
- If a bad technique is used, the method does not converge
- bfs ensures polynomial running time

  =>Edmond-Karp method, $O(V, E^2)$
- If all capacities are integers, for a random selection of the path, $O(E\ |f^*|)$, where $|f^*|$= max flow

  $c \in Q$, transformation could be applied to meet the above mentioned conditions

Connected Components

- G=(V,E) a directed graph
- *G* is strongly connected
  - if $\forall u, v \in V$ (u and v are reachable from one another)
  - $\exists$ path(u,v) and
  - $\exists$ path(v,u).
- A **strongly connected component** (***SCC***) of *G* is a **maximal** set of vertices $C \subseteq V$ such that C is strongly connected (for all $u, v \in C$, both path(u, v) and path(v, u) exist).

Strongly Connected Components

SCC(G)

1. call DFS(G) to compute finishing times f[u] for all u
2. compute G^T
3. call DFS(G^T), considering vertices main loop in order of decreasing f[u] (as computed in first DFS)
4. output the vertices in each tree of the depth-first forest formed in second DFS as a separate SCC

Strongly Connected Components

- **Evaluation: O(V+E)**
  - **O(V+E)**
  - **O(E)**
  - **O(V+E)**
  - **O(V)**
- **Approach**
  - second DFS in decreasing order of finishing times from first DFS, vertices are visited in topologically sorted order (see seminar #7 for topo sort)
  - running DFS on G^T, no *v* from a *u*, where *v* and *u* are in different components is visited
  - G and G^T have the same SCC

Strongly Connected Components

- **Notation:**
  - d/f refer to discovery /finishing times of the dfs on G
  - Let $C \subseteq V$ a SCC of G
  - $d(C) = \min \{d[u] \mid \forall\, u \in C\}$ //first discovered vertex in C
  - $f(C) = \max \{f[u] \mid \forall\, u \in C\}$ //last finished vertex in C

Strongly Connected Components

- **Lema (22.14)**

Let C and C' be distinct SCC's in G. Suppose there is an edge $(u, v) \in E$ such that $u \in C$ and $v \in C'$. Then $f(C) > f(C')$.

Justification:

- **if d(C)<d(C')**
  - Let x $\in C$ s.t. d[x]=d(C) (x first discovered in C)
  - At moment d[x], $\forall$ u'$\in C$, $\forall$ v' $\in C'$, u' and v' are white (d[x] min among all) => $\exists$ white path(x,u') (as x and u are in the same C, and d[u']>d[x] and d[x]=d(C)) and white path(x,v') (as d(C)<d(C') and (u, v) $\in$ E)
  - Existence of white path => $\forall$ u'$\in C$, $\forall$ v' $\in C'$ are descendants of x in depth first tree
  - Descendants + parentheses theorem => f[x]=f(C) hence max among all, therefore, f(C) > f(C')

Strongly Connected Components

- **Lema (22.14)**

Let C and C' be distinct SCC's in G. Suppose there is an edge $(u, v) \in E$ such that $u \in C$ and $v \in C'$. Then $f(C) > f(C')$.

Justification:

- **if d(C)>d(C')**
  - Let y $\in C'$ s.t. d[y]=d(C') (y first discovered in C')
  - At moment d[y], $\forall$ v' $\in C'$, v' are white (d[y] min among all) => $\exists$ white path(y,v') (as y and v' are in the same C)
  - =>f[y]=f(C')
  - At moment d[y], $\forall$ u' $\in C$, u' are white, and NO vertex in C is reachable from y
  - =>at moment f[y] all u' $\in C$ are still white
  - => $\forall$ u' $\in C$ f[u']>f(y) => f(C)> f(C')

Strongly Connected Components

- **Corollary (22.15):**

Let C and C' be distinct SCC's in $G = (V, E)$. Suppose there is an edge $(u, v) \in E^T$, where $u \in C$ and $v \in C'$. Then $f(C) < f(C')$.

- **Justification:**
  - $(u, v) \in E^T \Rightarrow (v, u) \in E$
  - G and G^T have the same SCC + Lema $\Rightarrow f(C) > f(C')$

Strongly Connected Components

- **Correctness – just a template**
  - second DFS (on G^T), starts with SCC *C* such that f(*C*) is maximum according to step 3 of the strategy (let x be in C so that d[x]=d(*C*))
    - It visits all vertices in *C*
    - By Corollary, f(*C*)>f(*C′*)

      => there is **no** edge (u,v) $\in E^T$, with u $\in C$, v$\in C'$
    - Thus, dfs visits *only* vertices in C
    - depth-first tree rooted at x contains only *C* vertices
  - second DFS (on G^T), continues with SCC *C ′* such that f(*C ′*) is maximum
    - dfs visits *only* vertices in *C ′* but the only edges out of *C ′* go to C, which were already visited.
  - Process continues the same

Graphs – instead of conclusions

- **most complex DS**
- **Algorithm complexity in terms of both size of V and E**
- **Various strategies, rather than algorithms**
- **Strategies to derive trees (with static representation) out of graphs**
- **On some graph a linear ordering can be imposed, an thus graph-> tree-> list!!!**

B-trees

- Previous DS reside in the primary memory
- Trees on secondary storage devices (disk)
- A node may have many children
- Goal: decrease the number of pages accessed when search for a node
- Store a very large number of keys
- Maintain the height of the tree under control (h very small)

B-trees – contd.

- **Typical pattern while working with B-trees:**

```
x<-pointer to some object
Disk-Read(x)
Operations that access/modify some fields of x
Disk-Write(x)
```

- **Once in memory, operations are performed fast**
- **Objective: as few pages read/write operations**

B-trees – contd.

- P1: n[x] keys in node x
- P2: keys are ordered

$key_1[x] <= key_2[x] <= \ldots <= key_{n[x]}[x]$

- P3: An internal node contains n[x]+1 pointers to the children $c_i[x]$
- P4: is a search tree:

$key[c_1[x]] <= key_1[x] <= keyc[_2[x]] <=$

- P5: All leaves are at the same level = height of the tree = h
- P6: t = degree of the tree; min(t)=2. Every node (except for the root) has at **least *t-1 keys***, and ***t children***
- P7: Every node (except for the root) has at **most *2t-1 keys***, and ***2t children***

B-trees – contd.

- **One pass procedures (top-down ONLY, NO back up; as opposed to PBT, AVL, RB trees)**
- **For ALL operations, the process is JUST top->down!!!**
- **Search**
  - **Straightforward generalization of BST search**
  - **Combined with ordered list search**
  - **#pages accessed (worst) O(h)**
  - **Access time in a page (worst) O(t)**
  - **Overall O(th)**

B-trees – contd.

- **One pass procedures (top-down ONLY, NO back up rebalance; as opposed to PBT, AVL, RB trees)**
- **Insert**
  - **Search for a LEAF position to insert**
  - **Insertion is performed in an EXISTING leaf**
  - **Along the path while searching (top-down), ensure there is space for a safe insert (*split full nodes* on the path down to avoid overflowing, so that the insertion is successful in an existing node!!!)**
  - **A key added to a full leaf will induce the *migration of the median key to the parent node*, and the split of the given leaf into 2 leaves**
  - **O(h) disk accesses, O(t) CPU time in one page => O(th)**

B-trees – contd.

- **One pass procedures (top-down, NO back up; as opposed to AVL, RB trees)**
- **Delete**
  - **Search for the node to contain the key to be removed and identify its type (leaf/no leaf – all nodes are either internal, or leaves; the tree is complete) (z from BST)**
  - **Physical *removal* of one object which *belongs to a leaf* (y in BST)**
  - **Along the path while searching (top-down), ensure the constraints for a safe delete are met (*merge nodes with degree t* on the path down to avoid underflowing, so that the deletion is possible)**
  - **O(h) disk accesses O(t) CPU time in one page => O(th)**

B Trees - insert

- **Like in any BST tree insert in a leaf (in a leaf, NOT as leaf)**
- **Stages:**
  - **Search the path for the position (leaf) to insert**
  - **Ensure the search path is safe**
  - **Insert the key in the corresponding leaf**
- **Types of nodes to store a key:**
  - **Leaf (key to be inserted)**
  - **Non-leaf (the "safe path" step = in the attempt to make room = in the split stage with median migration up)**
- **Possible issues**
  - **Attempt to store in a full node, with (2t-1) keys – issue – node overflows! Not allowed.**
- **Cases to analyze**
  - **Not overflowing node – no issue**
  - **Overflowing node – issue – needs a strategy to handle it**

B Trees -insert

- **Strategy**
  - **Along the searched path, ANY full node along the path (with already (2t-1) keys) is "fixed" (allowing for a potential full node to accept a new key to be added):**
    - **Divide the full node in 2 nodes with (t-1) keys**
    - **The median key in the full node is promoted to the parent node (there *is room*, as we proceed top-down, and an upper node was "fixed", is not overflowing)**
    - **if *root is full*, increase the height (by adding 1 more node = new root); the ONLY case of *height increase*.**
- **Insert procedure**
  - **Top-down approach (descendent)**
  - **There is NO operation performed on return (bottom up)**

B Trees – delete

- **Like in ANY other BST tree**
  - **Search for the key to remove** (pointed by **z**)
  - **If in leaf, delete it**
  - **If not in the leaf, remove (physically) a node** (pointed by **y**; its content is moved in the node pointed by z) **with one-single child** (pointed by **x**) **- the predecessor/successor – (in B-trees y is in a leaf, hence x points to nil)**
- **Types of nodes containing the key to be deleted**
  - **Leaf**
  - **Non-leaf (i.e. internal)**

Node type	Capacity
Leaf	Does not underflow
Non-leaf	Underflows

- **Possible issues**
  - **Attempt to delete from a node with only (t-1) keys – issue – node underflows! Not allowed**
- **Cases to analyze**

B Trees – delete

- **Cases to analyze**

Issue: attempt to delete from a node with only (t-1) keys –
node underflows! Not allowed

Node type	Capacity
Leaf	Does not underflow
Non-leaf	Underflows

- **Solution**
  - **Similar strategy as in case of insert: prevent, rather than repair**
  - **In the search stage (for the key to delete), ensures that each node (along the searched path) has the ability to allow for delete (does not underflow)**
  - **Along the searched path, any node with only (t-1) keys is "fixed"**
    - **If any of the sibling nodes has at least t keys, "borrow" a key from it (promote the last/first key from the left/right brother node to the parent node, and move down the appropriate key from parent to the almost underflowing node). (see examples for case 3.1 – next slide)**
    - **If both sibling neighbors have only (t-1) keys, merge the underflowing node with one of the 2 siblings, and put the key from the parent node in between them in the new generated (by merge) node. (see examples for case 3.2.1 next slide). Maybe a height shrink occurs (see examples for case 3.2.2 next slide).**
- **Delete procedure**
  - **Top-down approach (descendent)**
  - **There is no operation performed on return (bottom up)**

B Trees –delete contd.

Node type	Capacity
Leaf	Does not underflow
Non-leaf	Underflows

- **Cases – only 3 to be considered (not 4): non-leaf/underflow is not considered (since along the path, the underflow situation is solved, anyway).**
- **Cases: (follow the examples – blackboard)**
  - **Case1 – key in leaf, node does not underflow**
    - **Simply delete the key**
  - **Case2 – key not in leaf, node does not underflow**
    - **Remove pred/succ (from a leaf for a BT) like in BST. The case reduces to case 1 or 3.**
  - **Case3 – key in leaf, node underflows**
    - **3.1 sibling consistent (at least one sibling neighbor has at least t keys) sol: "borrow" from sibling**
      - **Promote the last/first key in consistent neighbor to parent**
      - **Move down the key in parent node in between the leaf and consistent sibling to the underflowing leaf**
    - **3.2 neighbor siblings both with just (t-1) keys**
      - **Merge the underflowing leaf with one sibling neighbor by adding in the middle the key in the parent in between the leaf and sibling**
        - **3.2.1 keep the height of the tree**
        - **3.2.2 *decrease the height* of the tree (if the *parent* is the *root* and has just one single key)**