Harvard University
SIMR (Summer Introduction to Mathematical Research)
Dinic's Algorithm

Yongwhan Lim
Research Mentor in SIMR at Harvard
Lecturer in EECS at MIT
Associate in Computer Science at Columbia
7:45pm ET, Thursday, July 7, 2022

Yongwhan Lim

- Currently:
  - (July 11~) Senior Quantitative Software Engineer at Two Sigma
  - Research Mentor in SIMR at Harvard University
  - Lecturer in EECS at MIT
  - Associate in Computer Science at Columbia University
  - ICPC Head Coach at Columbia University
  - ICPC Judge for Mid-central and Greater New York regions in N.A.
- Previously:
  - Research Software Engineer at Google Research
- Education:
  - Stanford: Math & CS (BS '11) and CS (MS '13)
  - MIT: Operations Research (PhD, started in 2013 but on an extended leave-of-absence since 2016)

Lecture Overview on Dinic's Algorithm

- Definitions
- **Dinic's Algorithm**
- Number of Phases
- Proof of Correctness
- Finding Blocking Flow
- Complexity
- Unit Networks
- Unit Capacity Networks
- Implementations in C++
- Looking ahead!
- References
- Contact Information

Lecture Overview on Dinic's Algorithm

- Definitions
- **Dinic's Algorithm**
- Number of Phases
- Proof of Correctness
- Finding Blocking Flow
- Complexity
- Unit Networks
- Unit Capacity Networks
- **Implementations in C++**
- Looking ahead!
- References
- Contact Information

Definitions

- A **residual network** G^R of network G is a network which contains two edges for each edge $(v, u) \in G$:
  - (v, u) with capacity $c^R_{vu} = c_{vu} - f_{vu}$
  - (u, v) with capacity $c^R_{uv} = f_{vu}$
- A **blocking flow** of some network is such a flow that every path from s to t contains at least one edge which is saturated by this flow.
  - Note that a blocking flow is not necessarily maximal.
- A **layered network** of a network G is a network built in the following way:
  - For each vertex v we calculate level[v]: the shortest path (unweighted) from s to this vertex using only edges with positive capacity.
  - We keep only those edges (v, u) for which level[v]+1 = level[u].
  - Obviously, this network is acyclic.

Dinic's Algorithm

- The algorithm consists of several phases.
- On each phase:
  - Construct the layered network of the residual network of G.
  - Find an arbitrary blocking flow in the layered network and add it to the current flow.

Number of Phases

- The algorithm terminates in less than V phases.
- Let's prove it using following two lemmas:
    - Lemma 1. The distances from s to each vertex do not decrease after each iteration: that is, $level_{i+1}[v] \geq level_i[v]$.
    - Lemma 2. $level_{i+1}[t] > level_i[t]$.
- There are less than V phases because level[t] increases, but it can't be greater than V - 1.

Lemma 1: Proof

- $level_{i+1}[v] \geq level_i[v]$.
- Fix a phase i and a vertex v. Consider any shortest path P from s to t in G^R_{i+1}. The length of P equals $level_{i+1}[v]$. Note that G^R_{i+1} can only contain edges from G^R_i and back edges for edges from G^R_i.
- If P has no back edges for G^R_i then $level_{i+1}[v] \geq level_i[v]$ because P is also a path in G^R_i. Suppose P has at least one back edge, say, (u, w). Then, $level_{i+1}[u] \geq level_i[u]$.
    - $level_i[u] = level_i[w] + 1$: Since (u, w) does not belong to G^R_i, (w, u) was affected by the blocking flow on the previous iteration. Also, $level_{i+1}[w] = level_{i+1}[u] + 1$ and $level_{i+1}[u] \geq level_i[u]$.
- So, $level_{i+1}[w] = level_{i+1}[u] + 1 \geq level_i[u] + 1 = (level_i[w] + 1) + 1 = level_i[w] + 2$.
- Similarly for the rest of the path.

Lemma 2: Proof (by contradiction)

- $level_{i+1}[t] > level_i[t]$
- From the previous lemma, $level_{i+1}[t] \geq level_i[t]$.
- Suppose that $level_{i+1}[t] = level_i[v]$.
- Note that G^R_{i+1} can only contain edges from G^R_i and back edges for edges from G^R_i.
- It means that there is a shortest path in G^R_i which was not blocked by the blocking flow.
- This is a contradiction.

Proof of Correctness

- The algorithm terminates in less than V phases.
- Now, when the algorithm terminated, it could not find a blocking flow in the layered network.
- So, the layered network does not have any path from s to t.
- So, the residual network does not have any path from s to t.
- Therefore, the flow has to be maximum.

Finding Blocking Flow

- In order to find the blocking flow on each iteration, we may simply try pushing flow with DFS (Depth-First Search) from s to t in the layered network while it can be pushed.
- In order to do it more efficiently, we must remove the edges which cannot be used to push anymore.
- We can keep a **pointer** in each vertex which points to the next edge which can be used.

Finding Blocking Flow (con't)

- A single DFS run takes O(k + V) time, where k is the number of pointer advances on this run.
- Over all runs, a number of pointer advances cannot exceed E.
- A total number of runs would not exceed E, as every run saturates at least one edge.
- So, a total running time of finding a blocking flow is **O(VE)**.

Complexity

- Since there are less than V phases, the total time complexity is **$O(V^2E)$**.

Unit Networks

- A **unit network** is a network in which for any vertex except vertex s and vertex t, **either incoming or outgoing edge is unique and has unit capacity**.
  - This is the setting with the network built to solve a maximum matching problem with flows.
- On unit networks, Dinic's algorithm runs in **$O(EV^{1/2})$**.

Unit Networks: Dinic's algorithm in $O(EV^{1/2})$

- Each phase now works in $O(E)$ because each edge will be considered at most once. So, it suffices to see a number of phases is $O(V^{1/2})$.
- Suppose there have already been $V^{1/2}$ phases.
  - All the augmenting paths with the length ≤ $V^{1/2}$ have been found.
  - Let f be the current flow, f' be the maximum flow and consider their difference f' - f.
  - It is a flow in G^R of value val(f') - val(f) where on each edge it is either 0 or 1.
  - It can be decomposed into val(f') - val(f) paths from s to t and possibly cycles.
  - As the network is unit, they cannot share common vertices.
  - Hence, (val(f') - val(f)) $V^{1/2}$ ≤ (the total number of vertices) ≤ V.
- Therefore, in another $V^{1/2}$ iterations, we will definitely find the maximum flow.

Unit Capacity Networks: : Dinic in O($E^{3/2}$)

- In a more general settings where all edges still have unit capacities (but a constraint on *the number of incoming and outgoing edges* is relaxed), the paths cannot have common edges as opposed to common vertices.
- So, a number of phases is O($E^{1/2}$), leading to the algorithm running in **O($E^{3/2}$)**.
- Alternatively, a number of phases can be shown to be O($V^{2/3}$), which gives even better bound on dense graphs.

Implementations in C++: *FlowEdge* struct

```cpp
struct FlowEdge {
	int v, u;
	long long cap, flow = 0;
	FlowEdge(int v, int u, long long cap) : v(v), u(u), cap(cap) {}
};
```

Implementations in C++: *Dinic* struct

```
struct Dinic {
	const long long flow_inf = 1e18;
	vector<FlowEdge> edges;
	vector<vector<int>> adj;
	int n, m = 0;
	int s, t;
	vector<int> level, ptr;
	queue<int> q;
	Dinic(int n, int s, int t) : n(n), s(s), t(t) {
		adj.resize(n);
		level.resize(n);
		ptr.resize(n);
	}
	void add_edge(int v, int u, long long cap);
	bool bfs();
	long long dfs(int v, long long pushed);
	long long flow();
};
```

Implementations in C++: *flow* function

```
long long flow() {
	long long f = 0;
	while (true) {
		fill(level.begin(), level.end(), -1);
		level[s] = 0;
		q.push(s);
		if (!bfs())
			break;
		fill(ptr.begin(), ptr.end(), 0);
		while (long long pushed = dfs(s, flow_inf)) {
			f += pushed;
		}
	}
	return f;
}
```

Implementations in C++: *bfs* function

```cpp
bool bfs() {
	while (!q.empty()) {
		int v = q.front();
		q.pop();
		for (int id : adj[v]) {
			if (edges[id].cap - edges[id].flow < 1)
				continue;
			if (level[edges[id].u] != -1)
				continue;
			level[edges[id].u] = level[v] + 1;
			q.push(edges[id].u);
		}
	}
	return level[t] != -1;
}
```

Implementations in C++: *dfs* function

```
long long dfs(int v, long long pushed) {
	if (pushed == 0)
		return 0;
	if (v == t)
		return pushed;
	for (int& cid = ptr[v]; cid < (int)adj[v].size(); cid++) {
		int id = adj[v][cid];
		int u = edges[id].u;
		if (level[v] + 1 != level[u] || edges[id].cap - edges[id].flow < 1)
			continue;
		long long tr = dfs(u, min(pushed, edges[id].cap - edges[id].flow));
		if (tr == 0)
			continue;
		edges[id].flow += tr;
		edges[id ^ 1].flow -= tr;
		return tr;
	}
	return 0;
}
```

Implementations in C++: *add_edge* function

```
void add_edge(int v, int u, long long cap) {
      edges.emplace_back(v, u, cap);
      edges.emplace_back(u, v, 0);
      adj[v].push_back(m);
      adj[u].push_back(m + 1);
      m += 2;
}
```

Looking ahead!

- Nearly-linear(!) time algorithm (Li Chen, Rasmus Kyng, Yang P. Liu, Richard Peng, Maximilian Probst Gutenberg, Sushant Sachdeva)
  - Video: https://www.youtube.com/watch?v=7ivZ0_vx4Nc
  - Try to watch the video before Professor Richard Peng's lecture next week! :)
- Dynamic graphs! (Professor Richard Peng's lecture, scheduled on July 14)

References

- https://cp-algorithms.com/graph/dinic.html
- https://people.orie.cornell.edu/dpw/orie633/LectureNotes/lecture9.pdf
- http://www.cs.cmu.edu/afs/cs/academic/class/15451-f14/www/lectures/lec11/dinic.pdf
- http://www.cs.toronto.edu/~bor/375s06/dinic-sketch.pdf
- Introduction to Algorithms (CLRS), 4th Edition
- Wikipedia

Contact Information

- yongwhan@mit.edu or yongwhan.lim@columbia.edu
- https://cs.columbia.edu/~yongwhan
- https://www.linkedin.com/in/yongwhan/

- For those who are interested in **quantitative analysis in finance**, feel free to send me an email to schedule 1:1 zoom meeting.