Computer Networks - Lecture 11 & 12

- Author: Ruben Schenk
- Date: 07.06.2021
- Contact: ruben.schenk@inf.ethz.ch

4.2.2 Previous algorithms as linear programs

Max-Flow

We can state the max-flow problem as a linear program in the following way:

- Objective: Maximize flow from s to t.
- Constraint: Obey edge capacities $c_{u,\,v}$.

In the linear-programming language, our constraints could look as follows:

- Variables: Flow $s \to t$: f, flow on edge $u \to v$: $f_{u,\,v}$
- Objective: Maximize f
- Constraints: Capacity $f_{u,\,v} \leq c_{u,\,v}$, flow conservation $\sum_{s \to v} f_{s,\,v} = f = \sum_{v \to t} f_{v,\,t}$ where s is the source and t is the sink, $\sum_{w \to u} f_{w,\,u} = \sum_{u \to v} f_{u,\,v} \quad \forall u \in V \setminus \{s,\,t\}$.

Max-Flow with multiple commodities

We are trying to maximize different flow f_x on the same graph/network. Our objective is therefore to maximize/minimize $\sum_x f_x$. We have the following constraints:

- Capacity: $\sum_x f_{x,\,uv} \leq c_{u\,v}$
- Flow conservation: in-flow = out-flow
  - $\sum_{w \to u} f_{x,\,wu} = \sum_{u \to v} f_{x,\,uv} \quad \forall u \in V \setminus \{u,\,d(x)\}$, where $d(x)$ is the destination of x (this constraint makes sure that *flows don't mix*).
  - $\sum_{x \to v} f_{x,\,xv} = f_x$
  - $\sum_{v \to d(x)} f_{x,\,vd(x)} = f_x$

Shortest Path

Our goal is to minimize an $s - t$ path length with given edge lengths $w_{u,\,v}$, i.e. minimize $\sum_{u,\,v} x_{u,\,v} w_{u,\,v}$.

We have the following constraints:

- Path must be connected: $\sum_{u \to v} x_{u,\,v} - \sum_{v \to w} x_{v,\,w} = 0 \quad \forall v \in V \setminus \{s,\,t\}$ ($= 1$ if $v = t$ and $= -1$ if $v = s$).
- $x_{u,\,v} \in \{0,\,1\}$, however in this case $x_{u,\,v} \in [0,\,1]$ is enough.

4.2.3 Integer Linear Programs

Constraints like $x \in \{0, 1\}$ are not linear anymore. Whereas LPs are solvable in polynomial time, the general class if ILPs is NP-hard!

4.3 Exploiting randomness for networking

4.3.1 Balls-into-Bins Problem

Assume you run a popular network application. How do we distribute requests? Our goal is to keep response time uniformly low and we wish to have a uniform load at the servers.

- *round-robin*: Might not work since requests might follow a pattern such as every second request is a complex one. This way, every even numbered server is fucked.
- *send to the least loaded server*: Keeping track of the different loads of each server generates a big overhead and is quite complex to implement.

Our solution is to pick a server *uniformly at random*, similar to the `balls-into-bins` problem.

Recap: Let there be m balls and n bins, then $(P[X_{i,\,j}] = 1) = \frac{1}{n}$, which is the probability that two balls collide.

The expected number of collisions is therefore:

$$\mathbb{E}\Big[\sum X_{i,\,j}\Big] = \frac{1}{n}\binom{m}{2}$$

The expected maximum load on any bin is therefore:

$$O\Big(\frac{\ln n}{\ln \ln n}\Big)$$

which is actually not that good. We can improve this by choosing two bins at random and then selecting the onw with the smaller load. This improvement leads to the following expected load:

$$O\Big(\frac{\ln \ln n}{\ln 2}\Big)$$

which is exponentially better. We can extend this idea by choosing k instead of 2 bins and then choose the one with the smallest load.

We normally only use random distribution once for each session. We determine the server by `hashing` the user-id and taking the hash $\%\, n$. However this means, that upon failure of a server, we will reassign most sessions (since now we calculate $\%\, (n-1)$). We can prevent this with `consistent hashing`, where we hash the server-id's into the same *hash-space* as we do with the user-id's. Each requests will simply be mapped to the server with the next greater hash-value as the user-id of the request.

4.3.2 Membership testing & counting

With a `bloom filter` we use m bits of memory in table T with k hash functions $h_1,\ h_2, . .,\ h_k$ in the hash range $\{1,\ 2, \ldots,\ m\}$. We are trying to represent n elements.

A `check for membership` occurs when we hash the value with all k functions and check whether all fields are 1, if not we can be certain that it isn't a member. If all fields are 1, then it is very likely that it is a member.

After n insertions, the probability that the t-th bit is still 0 is given by:

$$P[T(i) = 0] = \left(1 - \frac{1}{m}\right)^{kn} \simeq e^{\frac{-kn}{m}}$$

Thus, the probability of a match being a false positive is approximately

$$(1 - e^{\frac{-kn}{m}})^k$$

If we choose $k = \frac{m}{n} \log 2$ we can minimize the false positive probability, which then becomes 2^{-k}.

4.4 Distributed decision-making

Distributed settings are challenging since selfish player can cause large sub-optimalities. The best choice typically depends on the setting and may change over time.