1 Weighted Caching

In the *weighted caching* problem, we are given a cache of size k, and a main memory of n pages; let $[n] := \{1, 2, \ldots, n\}$ be their names. We are given a request sequence $p_1, p_2, \ldots, p_T$, where each request $p_t \in [n]$. Pages can be requested multiple times over the request sequence. For example

$$1, 4, 5, 2, 1, 3, 1, 1, 5, 2, \ldots$$

When a page is requested, it must be loaded into the cache. (The CPU can only access pages via the cache.) The cache holds at most k pages, so once it is full, and a page is requested that is not in the cache, we must evict some page in it. The choice of page to evict is called an *page eviction policy*. E.g., if $k = 2$, one solution for the above request sequence results in the following cache contents:

$$(1), (1,4), (4,5), (2,5), (1,5), (3,5), (1,3), (1,3), (1,5), (2,5), \ldots$$

Note that the red ones denote states where an eviction has just been done. Every page $i \in [n]$ has an eviction cost c_i, and we want to minimize the sum of eviction costs over the whole sequence.

Typically we study the problem in the *online* setting, where the request sequence is revealed over time. However, today let's study it when the entire request sequence is given up-front — how do we figure out the optimal cost?

1.1 Unit Costs

When all page eviction costs are equal, the problem can be solved optimally using a greedy algorithm: each time we need to evict a page, we evict one that is going to be requested *furthest in the future*. Note this requires knowledge of the entire request sequence. (This is often called Bélády's Rule.) In the above example with $k = 2$ we would get

$$(1), (1,4), (1,5), (1,2), (1,2), (1,3), (1,3), (1,3), (1,5), (2,5), \ldots$$

> *Exercise:* prove that Bélády's rule incurs the least number of cache evictions. Hint: suppose there is some eviction which is not for the element furthest in the future...

1.2 General Costs

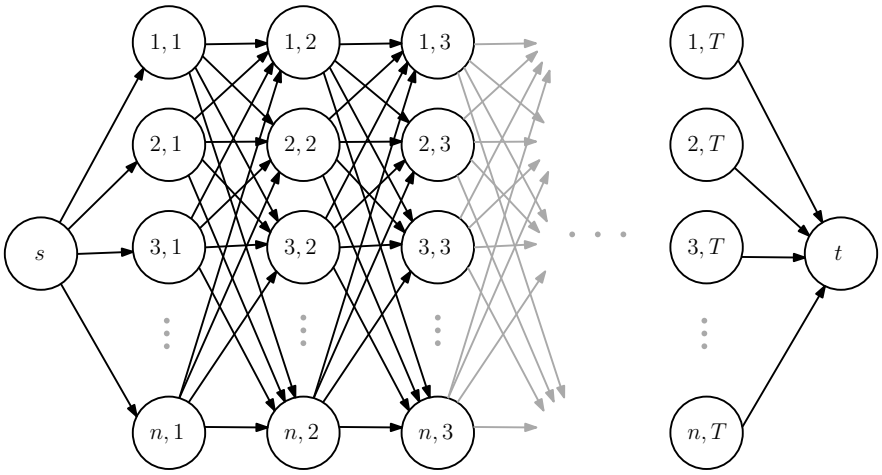
When costs are general, no greedy strategy is known to exist. However, we can solve the general cost case using min-cost max-flows, where we want to send k units of flow through a networks having $O(nT)$ nodes. (Think about how you may solve it, given this hint.) In fact, we will use the version of flows where we are allowed to specify a lower bound on the flow through an edge.

Consider the following network:

- We have T nodes each of the n pages, one node for each (page,time) pair. Imagine them being arranged in a grid. Also, you have an edge from each node (i, t) to each node $(j, t+1)$ in the immediately following layer. The edge cost for such an edge is c_i if $j \neq i$, and 0 if $j = 1$.

- There's also a source vertex s connected to all nodes $\{(1,1),(2,1),\ldots,(n,1)\}$, and then a sink t with edges from all nodes $\{(1,T),(2,T),\ldots,(n,T)\}$ going to it. These edges have zero cost.
- All edges have unit capacity.

This is what the network looks like:



Now if we send k units of flow from s to t, and this flow is integral, then it will trace out k paths from s to t. You can think of the i^{th} such path as telling us the contents of the i^{th} location in the cache. And the total cost of the flow paths is precisely the total eviction cost.

here

So it makes sense to ask: *what is the min-cost flow that sends k units of flow from s to t?* We'll solve this in the next few sections.

1.2.1 Capacities on Nodes

Let's enforce that two different flow paths don't pass through the same node (p,t). In other words, the page p can be present in only location in the cache at time t. (We can afford to ignore this issue, because this would just mean the cache could have fewer than k pages in it at some time, but it illustrates a useful idea.)

To enforce an edge capacity of x on node v, split it two copies v^{in} and v^{out}. All edges (u,v) going into v now turn into (u,v^{in}), and edges (v,w) leaving v now become (v^{out},w). Put an edge (v^{in},v^{out}) of capacity x, and zero cost.

So the network now has almost twice as many nodes, and we've converted the node capacities to edge capacities.

1.2.2 Flows with Lower Bounds

But still, we're not done: we did not enforce that page p_t must be in the cache at time t! Let's fix this. For each time t, let us place a *lower bound* on the node (p_t, t): we want at least one unit of flow to go through this node. Since we already split each such node (p_t, t) in two, we place the lower bound on that special edge $e_t := ((p_t, t)^{in}, (p_t, t)^{out})$.

> *Exercise:* Convince yourself that each integer flow of value k from s to t in this graph that sends unit flow through the special edge e_t for each time t (and having total cost C) corresponds to a sequence of caches with total eviction cost C, and vice versa.

How do handle lower bounds on the flow through an edge? We saw this in the last lecture: let's do it again.

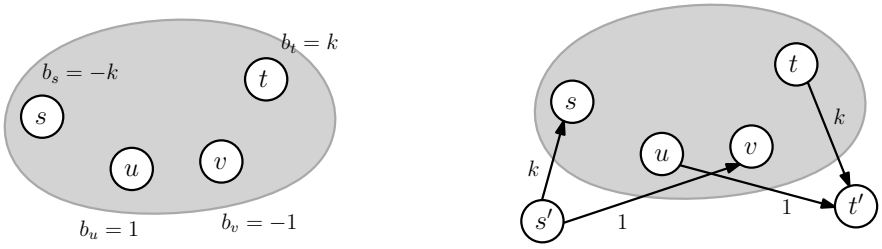
Suppose we want to send k units of flow from s to t, and ℓ units of this flow to go through some edge $e = (u, v)$. (For the moment assume there is a lower bound on a single edge, the same process extends to the case where multiple edges have lower bounds.)

Define the supply of s to be $b_s = -k$, the demand of t to be $b_t = k$. All other nodes x have $b_x = 0$. Let e have capacity $cap_e \geq \ell$ and cost $cost_e$. So now "pre-push" that ℓ units of flow through $e = (u, v)$ as follows: reduce the capacity of e to $cap_e - \ell$, let v have a extra supply and hence $b_v = b_v - \ell$, and u have a demand of $b_u = b_u + \ell$. The cost for e remains unchanged.

(If there are multiple edges with lower bounds, we can similarly pre-push that flow, resulting in more supplies and demands being created/changed.)

> *Exercise:* Suppose we pre-push the flow over some edge e. Convince yourself that the original graph has a flow with some cost C that satisfies the lower bound on e if and only if this new graph has a flow of the same cost C satisfying all the demands, having cost $C - \ell \cdot cost_e$. (This loss in cost is because we pre-pushed the flow.)

The above process gives us an instance with multiple sources/supplies and sinks/demands. So finally, we create a new source s^{new} and sink t^{new}, attach s^{new} to each source x (i.e., to each node x with $b_x < 0$) with capacity $|b_x|$, and attach each source y (i.e., each node y with $b_y > 0$) to t^{new} with capacity b_y. These new edges have cost zero.



> *Exercise:* Convince yourself that the original graph has a flow with some cost C that satisfies all demands if and only if this new graph has an s^{new}-t^{new} flow of value $\sum_{y:b_y>0} b_y$ of the same cost C.