CentraleSupélec

Algorithmes probabilistes et jeux

Mini-projet: *Online* algorithm

This mini-project will allow you to observe in real life the performance degradation of a deterministic or randomized algorithm compared to one that has the complete information (*offline*). The competitivity ratio will be estimated experimentally; [1]

The validation procedures are:

- Teamwork (in pairs),
- Programming (preferably in `python`; this document contains some tips),
- A public defense: **10 minutes max** the presentation[2] of the approach and results (the presentation material due as a `pdf`) + **5 minutes** of questions (everyone will participate in question sessions) on the morning of Thursday, February 23, 2023,
- A short written discussion (one – two page(s)) of the results obtained and observations made.

The problem to be addressed is that of k**-servers** which can be described as follows:

- k mobile technicians (servers), $\{s_1, s_2, \ldots, s_k\}$, provide maintenance service to n customers,
- the technicians treat requests emanating from the customers' sites; these requests arrive in series of length N, $(r_1, r_2, \ldots, r_N)$,
- a request must be processed immediately (it is impossible to put it on hold),
- the movements of the technicians are instantaneous.

The objective is to process all N requests by minimizing the sum of the distances traveled by all k technicians.

We will award algorithms showing a nice competitive ratio for provided instances.

The need to calculate the competitivity ratio implies calculating a solution *offline* beforehand. There is an exact algorithm for doing so [1]. This algorithm is described in the appendix to this statement. We used it to prepare the project instances.

The tasks to be carried out are thus:
design, implementation and performance evaluation of your *online* algorithms compared to the exact algorithm.

We assume that:

- the customer sites are distributed on a "grid" map: $[\![0, 99]\!]$,
- a list of requests is constituted of the sites requesting service,
- algorithm *online* receives requests one by one (it never sees more than one element, a request to be treated at the given time),
- the distance from the Manhattan $||.||_1$ norm,
- initially, all the technicians/servers in the point $(0, 0)$ on the map.

You will find the instances to process in the archive `k-server_instances.zip` in the module directory on `cirrus`, https://cirrus.universite-paris-saclay.fr/s/rGtA9yx4eSniTPB?path=%2FRandom-online%2FMini-projet

The instances are described in four parts:

[1] We are abusing language here; "our" ratio does not conform to the definition of this measure, which is calculated on all instances.

[2] Useless to recall our project's problem definition; get right to the point!

`# opt:` the value of the exact solution *offline*,

`# k:` the number of servers (technicians),

`# sites:` the coordinates of the sites on the map (location of the customers), their sites are numbered in order,

`# requests:` the sequence of the calls from the customers (the number of a site).

You will find the instances to process in the archive `k-server_instances.zip` in the module directory on `cirrus`,

Randomized algorithms require an average performance evaluation (the mean and the confidence interval). For the result to be statistically valid, an algorithm *online* must be run at least 100 times. An example in `python` with `numpy` and `scipy` is provided in the file `if_random.py` which is in the over-mentioned directory `cirrus`.

References

[1] Marek Chrobak, Howard J. Karloff, T. H. Payne, and Sundar Vishwanathan. New results on server problems. In *Proc. of SODA*, 1990.

Appendix

The authors of [1] propose to model the k-server problem by the min-cost max-flow problem which is exactly solvable in polynomial time. The algorithm of the min-cost max-flow problem consists in modifying the Ford-Fulkerson algorithm to find the max-flow: in addition to capacities, arcs of the network are labeled by their cost; the objective is to make the network flow the max-value flow by paying the smallest cost for this service. Its ready implementation is offered by the library `NetworkX`.

The construction of the flow graph $G = (V, E)$:

$$V = \{s, t\} \cup \{s_1, s_2, \ldots, s_k\} \cup \{r_1, r_2, \ldots, r_N\} \cup \{r'_1, r'_2, \ldots, r'_N\},$$

We take two fictitious vertices s (source) and t (*terminus*, sink), all servers and all "duplicate" requests.

The arcs have the associated capacity and cost. **The capacity is unitary everywhere.**

- the arcs (s, s_i), $i \in [\![1, k]\!]$ of zero cost (to irrigate the network),
- the arcs (s_i, r_j), $i \in [\![1, k]\!]$, $j \in [\![1, N]\!]$ of cost equal to the distance between the server i in its initial position and the site of the request j (to be able to send a server to serve its first request)
- the arcs (r_j, r'_j), $j \in [\![1, N]\!]$ of cost $-K$ which is very negative and integer (so that a server can go to serve another request after having finished dealing with the request r_j),
- the arcs (r'_i, r_j), $i, j \in [\![1, N]\!]$ and $i < j$ of cost equal to the distance between the requesting site i and the requesting site j (so that a server can go to serve the request r_j from the place where it performed the intervention r_i),
- the arcs (r'_j, t), $j \in [\![1, N]\!]$ of zero cost to evacuate the flow after serving the requests,
- the arcs (s_i, t), $i \in [\![1, k]\!]$ of zero cost (in case some servers do not serve any clients).

The graph G is very expressive and it is not difficult to notice that there will be an increasing chain passing through each s_i (the max flow is certainly k) and this chain sets the schedule for s_i. Minimizing the cost of transferring the flow must accompany the increase in the value of the flow (`max_flow_min_cost` of `NetworkX`). The algorithm has the time complexity in $\mathcal{O}(kN^2)$.