

1. Investing in NVIDIA

Many people have made a fortune in the past few years by buying NVIDIA stocks. Charlotte borrows \$10,000 from her father on Christmas and wants to see how much she could make in one year by investing in NVIDIA. For simplicity, let's assume that the market opens every day. Let $n = 365$ be the number of days in the following year. Her father has two rules: (i) She must sell all NVIDIA stocks by the next Christmas; (ii) Short selling is not allowed: she cannot sell more stocks than what she has.

For example, suppose Charlotte buys \$10,000 NVIDIA stocks on Monday and sells all of them on Thursday. If the stock price is \$100 on Monday and \$110 on Thursday, then she makes \$1,000 and now has \$11,000 in cash. Buying less than 1 share (like 0.1 or 2.1 share) is allowed.

On Christmas Eve, a mysterious Santa visits Charlotte's home and gives her a surprise. He tells her the stock price of NVIDIA on every morning of the following year. With this oracle information, Charlotte wants to make as much money as possible.

- (a) Let us first assume that trading stocks does not entail any costs. Describe an algorithm that computes the maximum amount of money Charlotte is able to make.
- (b) In the real world however, trading does have costs, such as tax, commission, slippage, etc. For convenience, let us assume that whenever Charlotte buys or sells stocks, she needs to pay 2% of the total amount she trades. Describe an algorithm for this scenario.
- (c) Charlotte learns that the interest rate is very high this year, so she could also make money by putting the money in the bank. She can put some money in the bank for 30 days, and earns 0.5% at maturity. For instance, she earns \$50 by putting \$10,000 in the bank for 30 days. During these 30 days, she cannot take the money out. The bank opens every day. With this new option, describe how you would change your algorithm in (b).

(Disclaimer: We are not suggesting you to invest in NVIDIA.)

2. Forest for the Trees. Here are slight variants on some familiar algorithms.

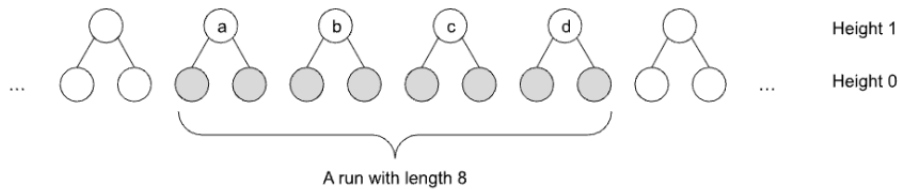
- (a) Given a connected undirected graph with non-negative edge lengths, you want to pick some of its edges so that you have exactly k connected components in the graph. Now among all edges whose endpoints lie in distinct components, consider the one with the shortest length. (Think of the k components as a clustering, and call the length of this edge the "separation" of the clustering.) You want a clustering whose separation is as large as possible. This clustering is often called *single-linkage clustering*. Show how to use the minimum spanning tree in general (or Kruskal's algorithm in particular) to solve this problem.
- (b) Given a directed graph with edge weights $w_e \geq 0$, and two nodes s and t , you want to find a path from s to t whose "bottleneck" weight is as large as possible. (Given a path, the bottleneck weight is the least weight among the edges on the path.) Show how to alter Dijkstra's algorithm slightly to solve this problem in essentially the same amount of time.

3. **(Fast Probing.)** Recall from class that *linear probing* is the following simple scheme for resolving hash collisions: we use a hash function $h : U \rightarrow [M]$. If an element hashes to a cell that already contains something, search to the right (wrapping around at the end) until a free cell is found, and place the element there. Similarly, to query an element q , start with the cell $h(q)$, and scan to the right (wrapping around the end) until you find the element q or an empty cell. We now show that for a *totally random hash function* h , the expected time for a find/insert is $O(1)$ (assuming no deletions).

Suppose we insert a total of n elements into a table of size $M = 4n$. A *run* is a maximal interval of occupied table cells. The insert/find time for element e is bounded by the length of the run containing the location $h(e)$, so we want to understand the probability of long runs occurring.

Purely for sake of the analysis, we think of the cells of the hash table as being the leaves of an imaginary complete binary tree of height $\log_2 M$ (where the leaves are at *height* 0). Define a node v at height k in this binary tree to be *suspicious* if at least $1/2 \cdot 2^k$ elements hash to a leaf in the subtree T_v under v .

- For a node v at height k , how many elements do you expect to hash into the subtree T_v ? Show that probability of v being suspicious is e^{-c2^k} for some constant $c \in (0, 1)$.
- Consider a run R whose length lies in the range $[2^\ell, 2^{\ell+1})$ for $\ell \geq 3$. Now consider the set of nodes v at height $k = \ell - 2$ in the imaginary tree such that the leaves of the subtree rooted at each v intersect with R . For example in the picture below with $\ell = 3$ this is $\{a, b, c, d\}$.



What is the minimum and maximum size of this set? Argue that among these nodes at height $k = \ell - 2$, at least one must be suspicious.

- Use the previous parts to bound the probability that the run containing any given cell x has length in $[2^\ell, 2^{\ell+1})$. Conclude that the expected length of the run containing any given cell x is $O(1)$.

4. Collaborative Dijkstra

Meredith learned Dijkstra in class and she wonders if there is a way to speed up the algorithm by splitting the work. So she designed an algorithm called Collaborative Dijkstra.

The idea is that instead of growing one "search ball" from the source s , grow two search balls simultaneously - one forward from s and one backward from t until they meet. Formally, given a directed graph G :

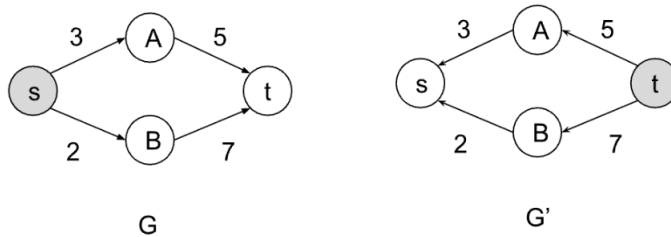
- Forward search from s : run one step of Dijkstra and record $d_f(v)$ as the shortest distance from s to some vertex v
- Backward search from v : on the reversed graph G' (i.e., a graph with all edge directions reversed), run one step of Dijkstra from t and record $d_r(v)$ as the shortest distance from t to some vertex v in G'

- (c) Alternating execution: The algorithm alternates between forward search and backward search, i.e., Dijkstra runs one step of forward search, then one step of backward search, and so on.

For example, in the graph below, the process goes as:

1. Forward: $d_f(B) = 2$
2. Backward: $d_r(A) = 5$
3. Forward: $d_f(A) = 3$

She proposes that the algorithm finds the shortest path from s to t once the searches meet, i.e., $d_f(u) \neq \infty, d_r(u) \neq \infty$ for some u . In the example above, the algorithm terminates at step 3 since $d_f(A) \neq \infty, d_r(A) \neq \infty$ after this step.



- (a) After running her algorithm on some examples, she realized that her stopping condition does not always guarantee an optimal solution. Can you give an example where it fails?
- (b) So she brings her algorithm to Jason, and Jason suggested an alternative stopping condition:
 - let $QueuePeek_f, QueuePeek_r$ be the smallest distance to s and t in the priority queue in Forward and Backward search respectively.
 - keep track of the shortest path length so far μ as the algorithm progresses. That is, initialize $\mu = \infty$, and update $\mu = \min\{\mu, d_f(v) + d_r(v)\}$ whenever a vertex v is visited in either direction.
 - stop when $QueuePeek_f + QueuePeek_r \geq \mu$

This algorithm turns out to be correct. Can you prove correctness of the algorithm by contradiction as done in lecture? You may assume that all edge weights are positive, as done in lecture.

- (c) Derive the time complexity and compare it to traditional Dijkstra. Is the algorithm any faster in the worst case?
- (d) Analyze specific cases where Meredith's collaborative Dijkstra performs better, same, or potentially worse than traditional. When is collaborative Dijkstra useful? Consider search space.