CS 325 Recitation Session 5

Chirag Kaudan

May 2025

1 Introduction

- Survey of how many people have grading questions/complaints
- Are people confused about some conceptual stuff from lecture?
- Goal for today: Introduce greedy algorithms, compare to dynamic programming, proof techniques

2 Frog Jumping Problem

Problem Setup: We want to help our friend get home, but our friend is a frog, and it has to cross the river to get there. There are various lily pads scattered throughout this river, and the frog starts at location 0. There are always lily pads at locations 0 and n, which is the frog's destination. The frog can jump at most r units at a time. We want the frog to reach location n using the least number of jumps, if possible. Let's formalize the problem.

> **Problem Statement (this is perhaps an overly formal way to state the problem)** Given a set of lily pad locations L which includes location 0 and location n, return the path $0, l_1, l_2, \ldots, n$ of lily pad locations that starts at 0 and ends at n which has the smallest number of elements and for every consecutive pair l_i, l_j of locations in the path, we have that $|l_j - l_i| \leq r$, if such a path is possible.

There's a very natural, intuitive idea that probably pops in your mind immediately when seeing this problem; try to take the biggest leap you possibly can every time! Right away this seems reasonable since, for example, if it is possible to make progress, we always do (we never go back with this strategy), and its easy to formally design an algorithm around this idea.

Algorithm 1: GreedyJump

Input : Set L of lily pad locations and the maximum frog jump distance r per move

Create an empty set J of jumps
Let our current position $x = 0$
while $x < n$ **do**
    Let l be the furthest lily pad reachable from x that is not after position n
    Add a jump j to J from x to the location of l
    Set current location x to location of l
Return J

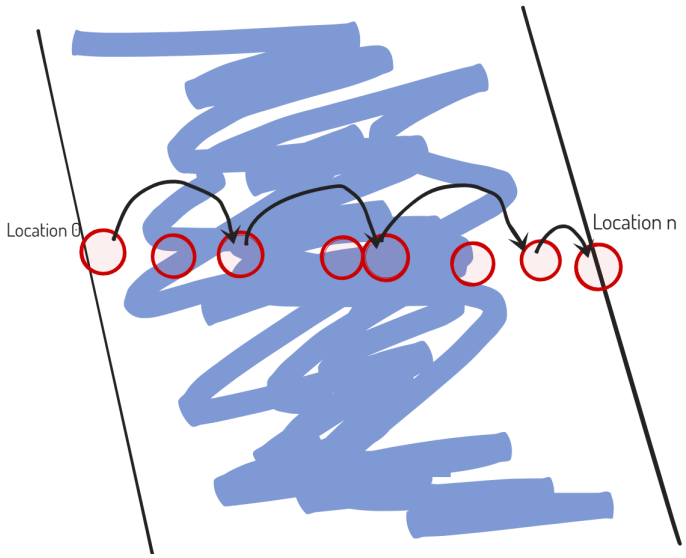

Figure 1: Depiction of the frog jumping problem with a river and lily pads spread throughout

This is a **greedy algorithm**. Greedy algorithms construct a solution step by step, at each step choosing the decision that seems best in the moment. Like in our frog jumping problem; at each possible step, the frog is at location x and we find the farthest lily pad to which the frog can reach from its location, as that is what seems best in the moment. Sometimes greedy algorithms work, but certainly not always.

We refer to this idea of making a decision that "seems best in the moment" as making a **locally optimal** decision, and our hope is that for the problem we are facing, a greedy strategy builds up a **globally optimal** solution.

2.1 Proving Our Algorithm Works

So we have this simple algorithm which seems correct intuitively, but we need to justify its correctness formally. We need to prove two properties about this algorithm.

- (Legality) Prove that it builds a "legal" solution i.e. that the frog doesn't get stuck somehow
- (Optimality) Prove that it is actually optimal i.e. no better path exists for the frog to take

Proving legality may seem a bit strange, but for many greedy algorithms that actually work, it's not at all clear that it builds a feasible solution at all (a valid object to return that is of the form the problem is asking), much less an optimal one.

Lemma 1. *The greedy algorithm always finds a path from the starting lily pad (at position 0) and the destination lily pad (at position n) if one exists.*

Proof. We prove this by contradiction: suppose that there exists some set of lily pad locations for which the greedy algorithm fails to find a path from position 0 to position n (but such a path does exist). Let the positions of the lily pads be $x_1 \leq x_2 \leq \cdots \leq x_m$. Now consider the sequence of lily pad locations that the frog jumps on using the greedy algorithm. Since the greedy algorithm failed, we know that it ends at some lilypad at location x_k and got stuck, not being able to take another jump to a future lily pad. Therefore, we know that $x_k + r < x_{k+1}$. But by assumption, there is some path that exists which starts at location 0 and ends at location n. In particular, there is some path that starts to the left of x_{k+1} and ends at x_{k+1} or

to the right of it. Since $x_k + r < x_{k+1}$, this path cannot use this lily pad k, so it must use some other lily pad s which is at location $x_s \leq x_k$. However, this is impossible, since $x_s + r \leq x_k + r < x_{k+1}$, so we cannot make this jump from lily pad s either, a contradiction. Thus, our assumption that the greedy algorithm can get stuck if there exists some path was incorrect and we conclude that the greedy algorithm always finds a path if one exists. □

How would we go about showing that the greedy algorithm is optimal, taking the least amount of jumps to get to location n? Here's a series of suggestions that you should think carefully about, and then we show a proof of the fact that the greedy algorithm is in fact optimal using these ideas.

- What if somebody magically gave us an optimal solution to this problem i.e. a series of jumps that gets the frog to location n legally?
- Call that series of jumps J'; we don't know what J' looks like, nor do we know whether our algorithm is optimal at this point in time
- The idea is to show that the greedy algorithm always produces a series of jumps J no worse than J' and conclude the optimality of the greedy algorithm

Theorem 1. *The greedy algorithm for frog jumping described above is optimal. That is, if a path from location 0 to n exists, the greedy algorithm finds a path taking the least number of jumps.*

Proof. There is nothing to prove if no such path to location n exists, so assume one does. Let J be the series of jumps the frog takes using the greedy algorithm. By lemma 1, we know that the greedy algorithm returns some path from 0 to n. Let J' be an optimal series of jumps to location n (note that there may be multiple optimal series of jumps). Let $|J|$ and $|J'|$ denote the number of jumps in J and J' respectively. By their definitions, we know that $|J| \geq |J'|$.

Let $p(i, J)$ denote the location of the frog after taking i jumps in series J. We prove the following lemma, which will help us finish this proof.

Lemma 2. *For all $0 \leq i \leq |J'|$ we have $p(i, J) \geq p(i, J')$.*

Proof. We prove this by induction. For the basis step, $p(0, J) = p(0, J') = 0$, since the frog has not moved from position 0, so the claim is true. For the induction step, let $0 \leq i \leq |J'|$ and suppose the claim is true for i. We will prove that the claim is true for $i + 1$ in two cases.

Case 1. *$p(i, J) \geq p(i + 1, J')$. Since the greedy algorithm never goes backwards, we know that $p(i + 1, J) \geq p(i, J)$, so it follows that $p(i + 1, J) \geq p(i + 1, J')$ and we have proven the inductive step in this case.*

Case 2. *$p(i, J) < p(i + 1, J')$. Since the frog can jump at most r units forward at each step and we know by our inductive hypothesis that $p(i, J) \geq p(i, J')$, it follows that $p(i, J) + r \geq p(i, J') + r \geq p(i+1, J')$. We have shown that the greedy algorithm can jump to position at least $p(i + 1, J')$, therefore $p(i + 1, J) \geq p(i + 1, J')$ completing the inductive step in this case as well.*

Therefore, the claim is true for all $0 \leq i \leq |J'|$. □

To finish our proof, for the sake of contradiction, assume that the greedy algorithm does not produce an optimal series of jumps i.e. assume $|J| > |J'|$. Let $k = |J'|$. Then $k < |J|$. It follows from lemma 2 that $p(k, J) \geq p(k, J') = n$. But the greedy algorithm explicitly never goes past location n (while loop terminates at location n), therefore we always have that $p(k, J) \leq n$. Together, we conclude that $p(k, J) = n$. But since $k < |J|$, this contradicts the behavior of the greedy algorithm, so our assumption that the greedy algorithm does not produce an optimal series of jumps is incorrect. □

This concludes our proof that our algorithm is optimal. That was a fair bit of effort to show something that was perhaps somewhat obvious, but this was a simple problem and a simple algorithm. When things get more involved, we require showing the details as above since the arguments may get increasingly more complex. We recap what we've done so far in the next subsection.

2.2 What Did We Just Do? Greedy Stays Ahead Technique

The technique that we just used to prove the optimality of the greedy algorithm for the frog jumping problem is called the "Greedy Stays Ahead" technique. This is roughly how this technique goes:

- Find some set of measurements $M_1, M_2, \ldots, M_k$ that you can apply on any solution
- Compare the solution your greedy algorithm builds to that of an optimal solution
- Prove that according to your measure, the greedy algorithm stays ahead of the optimal solution every iteration. More precisely, show that greedy never falls behind (usually done inductively, though not always)
- Using that greedy "stays ahead", show that the greedy algorithm solution must be optimal (usually done by contradiction, though not always).

You don't need to do this step by step by the book; the proof showing that our greedy frog jumping algorithm is optimal did not. But this is usually what is involved in making a "Greedy Stays Ahead" argument. Usually, the hardest part about this technique is coming up with a set of measurements. This requires creativity and cleverness!

3 Advantages of Greedy Algorithms

Why would somebody try to solve a problem using a greedy algorithm?

- Easy to suggest **a** greedy algorithm
- Usually simple to understand and efficient algorithms
- Usually not difficult to analyze run time, though this depends on your data structures involved
- Usually not difficult to implement (again depends on data structures)

4 Challenges with Greedy Algorithms

- Difficult to suggest a **correct** greedy algorithm i.e. difficult to prove correctness
- Therefore, designing a correct greedy algorithm is difficult, precisely because it is so easy to suggest **a** greedy algorithm and most of them fail!

5 Principles of Greed: Greedy Algorithms vs Dynamic Programming

How do greedy algorithms differ from Dynamic Programming? Both paradigms solve a problem by making a sequence of decisions, but greedy algorithms **cannot** use the answers of future subproblems to aid in making decisions. At each step, it makes the decision that is locally optimal/best in the moment. It proceeds iteration by iteration in its quest to solve the problem, at each step making a myopic decision.

Dynamic programming, on the otherhand, may solve subproblems before it ever makes the first "decision" (thinking about it from a bottom-up iterative viewpoint). It oftentimes amounts to doing brute force to find the optimal solution to some set of subproblems, and then using this information. This is the heart of dynamic programming; brute force done carefully.

You should think of greedy algorithms in the following "top-down" sense:

- The algorithm makes a greedy choice for the original problem, leaving a remaining subproblem.

- To solve this subproblem, the algorithm makes another greedy choice, leaving a remaining subproblem
- And so on...

So greedy algorithms make local decisions in the hope that stringing together several locally optimal decisions lead to a globally optimal solution. They are not guaranteed to work; in fact, almost every greedy algorithm to solve a problem is wrong. But when they work, they are elegant and efficient. These two algorithmic paradigms are not mutually exclusive; there are algorithms that possess both elements of dynamic programming and greedy algorithms (for example, Dijkstra's Algorithm for the single source shortest path problem with non-negative weights).

6 Conclusion

We looked at what greedy algorithms are, giving an informal "definition" of what characterizes them and contrasted this to dynamic programming. We saw an example of a problem in which a natural greedy algorithm solves it optimally, and we formally proved that our greedy algorithm was indeed optimal. This led to a discussion on one common technique used to prove that greedy algorithms, the "greedy stays ahead" technique.