Proof methods and greedy algorithms

Magnus Lie Hetland

Lecture notes, May 5th 2008*

1 Introduction

This lecture in some ways covers two separate topics: (1) how to prove algorithms correct, in general, using induction; and (2) how to prove greedy algorithms correct. Of course, a thorough understanding of induction is a foundation for the more advanced proof techniques, so the two are related. Note also that even though these techniques are presented more or less as "after the fact" methods of verification, knowing them well can be an important part of coming up with promising algorithm ideas as well.

2 Induction in algorithm design

Summary of method Show that a property (such as correctness) holds for a base case. Assume that the property holds for problems of a given size $n-1$ (or all sizes $< n$), with n greater than the base case, and show that it follows that the problem holds for size n. It then follows that the property holds for all problem sizes greater than the base case.

Induction is assumed to be a known technique (from TDT4120), including its application to proving properties such as correctness on iterative (using invariants) and recursive algorithms. The paper by Manber [7] contains numerous examples of this, as well as several pointers on how to use inductive thinking to construct algorithms. The core of the technique is the idea that if you can construct a solution to any subproblem from solutions to smaller subproblems, then you can solve the problem itself.

Exercise Take any recursive algorithm you know and phrase its correctness proof in terms of induction. Do the same for an iterative algorithm.

In the following, I cover only a single example, which combines induction with the common proof technique of proof by contradiction. This is the technique of proof by maximal counterexample, in this case applied to perfect matchings in very dense graphs.

We want to show that a parameter P can reach a value n. The general idea is as follows:

*These lecture notes for TDT4125 *Algorithm Construction, Advanced Course*, are based on the material listed in the bibliography [1–7]. Of this, the chapter in *Introduction to Algorithms* [1] and the paper by Manber [7] are in the curriculum, in addition to the lecture notes themselves.

1. Prove that P can reach a small value (the base case).
2. Assume that P cannot reach n, and we consider the maximal value $k < n$ that it can reach.
3. Present a contradiction, usually to the maximality assumption.

A *matching* in an undirected graph $G = (V, E)$ is a set of edges that have no nodes in common. A *maximal* matching is one that cannot be extended, and a *maximum* matching is one of maximum cardinality. A matching with n edges in a graph with $2n$ nodes is called *perfect*.

Consider the case in which there are $2n$ nodes in the graph and all of them have degrees of at least n. We want to show that in this case a perfect matching always exists. If $n = 1$ the graph consists of a single edge, and the perfect matching is evident.

Assume that $n > 1$ and that a perfect matching does *not* exist. Consider a maximum matching $M \subset E$. We know that $|M| < n$ by assumption, and clearly $|M| \geq 1$, because any single edge qualifies as a matching. Because M is not perfect, there must be at least two nonadjacent nodes v_1 and v_2 that are not included in M. These two nodes have at least $2n$ distinct edges coming out of them, and all of these lead to nodes in M (because otherwise the edge could be added to M). Because the number of edges in M is less than n and there are $2n$ edges from v_1 and v_2 adjacent to them, at least one edge (u_1, u_2) from M is adjacent to three edges from v_1 and v_2, for example, (u_1, v_2), (u_1, v_2), and (u_2, v_1). By removing (u_1, u_2) and adding (u_1, v_2) and (u_2, v_1) we get a larger matching, which contradicts the assumption of maximality.

Exercise How can this proof be turned into an algorithm for constructing perfect matchings for graphs of this kind? (Manber [7] describes a solution, and the idea applies to other similar proofs as well.)

3 An overview of greedy algorithms

Informally, a greedy algorithm is an algorithm that makes locally optimal decisions, without regard for the global optimum. An important part of designing greedy algorithms is proving that these greedy choices actually lead to a globally optimal solution.

One common way of formally describing greedy algorithms is in terms optimization problems over so-called weighted set systems [5]. A *set system* is a pair $(E, \mathcal{F})$, where U is a nonempty finite set and $\mathcal{F} \subseteq 2^E$ is a family of subsets of E. A *weighted set system* is a set system with an associated weight (or cost) function $c : \mathcal{F} \to \mathbb{R}$. The optimization problem is then to find an element of $\mathcal{F}$ whose cost is minimum or maximum. In the interest of simplicity, we will assume that c is a modular function; that is, $c(X) = \sum_{e \in X} c(e)$.

The set of maximal elements of $\mathcal{F}$ (the *bases* of the set system), that is, the elements that are not subsets of any other elements of $\mathcal{F}$, is denoted by $\mathcal{B}$.

Here are some examples of problems that can be formulated in this way:

Shortest Path Given a digraph $G = (E, V)$, $c : E \to \mathbb{R}$ and $s, t \in V$ such that t is reachable from s, find a shortest s-t-path in G with respect to c. Here $\mathcal{F} = \{F \subseteq E : F \text{ is a subset of an } s\text{-}t\text{-path}\}$.

TSP Given a complete undirected graph $G = (E, V)$ and weights $c : E \to \mathbb{R}_+$, find a minimum weight Hamiltonian circuit in G. Here $\mathcal{F} = \{F \subseteq E : F \text{ is a subset of a Hamiltonian circuit in } G\}$

MST Given a connected undirected graph $G = (E, V)$ and weights $c : E \to \mathbb{R}$, find a minimum weight spanning tree in G. Here $\mathcal{F}$ is the set of forests in G and $\mathcal{B}$ is the set of spanning trees.

In the following, it may be helpful to keep the minimum spanning tree problem in mind as a "prototypical" problem.

Exercise How would you formalize the following problems as optimization problems with modular weight functions over weighted set systems? The knapsack problem; finding a forest of maximum weight; the minimum Steiner tree problem; finding a perfect bipartite matching; finding a maximum weight bipartite matching; finding a maximum weight matching.

Another useful property of these problems is that their set systems are closed under containment. That is, $\emptyset \subseteq \mathcal{F}$, and if $X \subseteq Y \in \mathcal{F}$ then $X \in \mathcal{F}$. We call a set system with this property an *independence system*. The elements of $\mathcal{F}$ are called *independent*, while the elements in $2^E \setminus \mathcal{F}$ are *dependent*.

Given such a formulation of our problems, the greedy approach (or, simply, *the greedy algorithm*) can be characterized as follows (for maximization problems).

Best-In Greedy Algorithm Here we wish to find a set $F \in \mathcal{F}$ of maximum weight.

1. Sort E so that $c(e_1) \geq \ldots \geq c(e_n)$.
2. $F \leftarrow \emptyset$.
3. For $i = 1$ to n: If $F \cup \{e_i\} \in \mathcal{F}$, then $F \leftarrow F \cup \{e_i\}$

It may be useful to compare (and contrast) this with, for example, Kruskal's algorithm for finding minimum spanning trees.

Worst-Out Greedy Algorithm Here we wish to find a basis F of $(E, \mathcal{F})$ of maximum weight.

1. Sort E so that $c(e_1) \leq \ldots \leq c(e_n)$.
2. $F \leftarrow E$.
3. For $i = 1$ to n: If $F \setminus \{e_i\}$ contains a basis, then $F \leftarrow F \setminus \{e_i\}$.

Tip At this point it may be useful to have a look at the discussion of Cormen et al. [1, pp. 379–382] on the topic of the *greedy-choice* property and *optimal substructure* in greedy algorithms, as well as other relevant parts of the TDT4120 curriculum, which is assumed to be known.

4 Two basic greedy correctness proof methods

The material in this section is mainly based on the chapter from *Algorithm Design* [4].

4.1 Staying ahead

Summary of method If one measures the greedy algorithm's progress in a step-by-step fashioin, one sees that it does better than any other algorithm at each step; it then follows that it produces an optimal solution.

To illustrate this method in more detail, let's consider the problem of *interval sceduling*. We have a set of n intervals, where the ith interval has a starting time $s(i)$ and an ending time $f(i)$. We say that a subset of intervals is *compatible* if no two of them overlap in time, and we wish to find as large a compatible subset as possible. (These sets are, in other words, *optimal*.)

Exercise Can you find more than one way of describing the problem structure as a weighted independence system? How does your formulation affect the resulting greedy algorithm? Can you create one formulation where the greedy algorithm is optimal and one where it is not? Also see the following description of a greedy solution. How can it be rephrased in terms of the previous, generic greedy algorithms (best-in and worst-out)?

An optimal, greedy solution for this problem can be formalized as follows:

1. Let I be the set of intervals and let $A = \emptyset$.
2. Choose an interval $i \in I$ that has the smallest finishing time.
3. Add interval i to A.
4. Delete all intervals from I that are not compatible with i.
5. If R is not empty, go to step 2.

It should be obvious that the returned set is a legal solution (that is, all the returned intervals are compatible). We now need to show that A will, indeed, be optimal when this algorithm terminates, and we will do that by showing that the greedy algorithm *stays ahead*.

Let O be an optimal set of intervals. We can't necessarily show that $A = O$ (as there may be more than one optimal set). Instead, we wish to show that $|A| = |O|$ (that is, our solution is as good as any optimal solution). To show that our algorithm stays ahead, we will compare our partial solution at each step with a corresponding initial segment of the solution O, and show that our solution is doing better (or, at least, not worse).

Let $i_1, \ldots, i_k$ be the set of intervals in the order they were added to A. Note that $|A| = k$. Similarly, let the elements of O be $j_1, \ldots, j_m$. We wish to prove that $k = m$. Assume that the elements of O are in sorted order according to their starting and ending times.[1]

[1] Because the elements are compatible, the starting times have the same order as the finishing times.

Formulating this in terms of *staying ahead*, we wish to prove that for all indices $r \leq k$ we have $f(i_r) \leq f(j_r)$. We prove this by induction. The base case, for $r = 1$, is clearly correct: The greedy algorithm selects the interval i_1 with minimum finishing time. Now let $r > 1$ and assume, as induction hypothesis, that the statement is true for $r - 1$. In other words, $f(i_{r-1}) \leq f(j_{r-1})$. The question then becomes: Is it possible for the greedy algorithm to "fall behind" at this step? Is it possible that $f(i_r) > f(j_r)$? The answer is clearly no, as the greedy algorithm could always have chosen j_r (which is compatible with j_{r-1} and therefore must also be compatible with i_{r-1}, which finishes no later).

We now need to show that this staying ahead indeed implies optimality, and we show this by contradiction. If A is not optimal, then an optimal set O must have more intervals, that is, $m > k$. For $r = k$ we know that $f(i_k) \leq f(j_k)$. Since $m > k$ there is a interval j_{k+1} in O. This starts after j_k ends (as all the intervals in O are compatible), and therefore after i_k ends. So after deleting all the intervals that are not compatible with $i_1, \ldots, i_k$, R still contains j_{k+1}. But the greedy algorithm stops with request i_k, and it is only supposed to stop when R is empty. In other words, we have a contradiction.

Exercise Apply the idea of staying ahead to greedy algorithms you already know, such as Prim's or Dijkstra's algorithm.

4.2 Exchange arguments

Summary of method Consider any possible solution to the problem and gradually transform it into the solution found by the greedy algorithm without hurting its quality. It follows that the greedy algorithm must have found a solution that is at least as good as any other solution.

Let's consider a more flexible version of the interval scheduling problem: Instead of a set of intervals with fixed starting and ending times, we have a set of requests, each of which has a deadline d_i and a required length t_i. As before, we disallow overlapping intervals in our solution. Assume that we wish to accomodate all requests, but we are allowed to let some requests run late (that is, we are allowed to place their finish times later than their deadlines). Each interval i is still $[s(i), f(i)]$, this time with $f(i) = s(i) + t_i$.

A request i is *late* if $f(i) > d_i$, and we define its *lateness* as $l_i = f(i) - d_i$. If request i is not late, we say that $l_i = 0$. Our goal is not wo schedule all the requests as non-overlapping intervals so that we *minimize the maximum lateness*, $L = \max_i l_i$.

Exercise How does this problem compare to the generic independence system problems? Can you find a way of formulating the problem such that the generic greedy algorithm gives the right answer? How about formulating it so that you get a wrong answer?

An optimal, greedy solution for this problem (often called *earliest deadline first*) can be formalized as follows:

1. Sort the jobs so that $d_1 \leq \ldots \leq d_n$ and let $f = s$ (the start time).

2. For $i = 1$ to n: Let $[s(i), f(i)] = [f, f + t_i]$; $f \leftarrow f + t_i$.

Considering that this algorithm ignores interval length completely in its scheduling, it may be hard to believe that it is optimal—but it is, and we will show it using an *exchange argument*. The general plan for the proof is as follows: Assume an optimal solution O and gradually modify it, preserving its optimality at each step, until you have a solution identical to the solution A found by the greedy algorithm.

As a first step, observe that there are no gaps, or "idle time," in our schedule. (It should be obvious that there exists an optimal solution with no idle time.) Also, the greedy solution (per definition) has no "inversions," that is, jobs scheduled before other jobs with earlier deadlines. We can show that all schedules without inversions and idle time have the same maximum lateness: Two different schedules without inversions and idle time can only differ in the order in which jobs with identical deadlines are scheduled (and these must be scheduled consecutively). Among these, maximum lateness depends only on the last one, and this lateness does not depend on the order of the jobs.

What we seek to prove, then, is that there is an optimal solution without inversions and idle time (or, simply, one without inversions), as such a solution would be equivalent to the greedy one.

The proof has three parts:

1. If O has an inversion, then there is pair of jobs i and j such that j is scheduled immediately after i and $d_j < d_i$.

2. After swapping i and j we get a schedule with one less inversion.

3. The new swapped schedule has a maximum lateness no larger than that of O.

The first point should be obvious: Between any two inverted jobs there must be one point at which the deadline we see decreases for the first time. This point corresponds to the pair i and j. As for the second point, swapping i and j clearly removes one inversion (that of i and j) and no new inversion is created. The hardest part is showing that this swapping will not increase the maximum lateness (the third part).

Clearly, swapping i and j (so j now comes first) can only potentially increase the lateness of i. No other job (including j) is "at risk." In the new schedule, i finishes where j finished before, and thus its new lateness (if it is late at all) is $f(j) - d_i$. Note that because of the assumption $d_i > d_j$, $f(j) - d_i < f(j) - d_j$, which means that i cannot be *more late* than j was originally. Therefore, the maximum latenes cannot have been increased, and the third part of our proof is complete.

It should be clear that these three parts together show that the schedule produced by the greedy algorithm has optimum lateness.

Exercise See if you can apply the idea of exchange arguments to greedy algorithms you already know.

5 Three advanced proof methods

A discussion of matroids can be found in *An Introduction to Algorithms* [1]. Greedoids are dealt with by, for example, Jungnickel [3] and Korte and Vygen [5] (and even more in-depth by Korte et al. [6]). The concept of matroid embeddings was introduced by Helman et al. [2]. Note that even though matroids and greedoids are fairly managable concepts, the related concept of matroid embeddings are rather complex, and a full treatment is far beyond the scope of this lecture.[2]

The following three sections briefly describe three characterizations of families of problems that can be optimally solved using greedy algorithms (with matroid embeddings exactly characterizing all such problems). The proof method then simply becomes showing that a given problem is, for example, a matroid or a greedoid.

5.1 Matroids

An independence system $(E, \mathcal{F})$ is a *matroid* if it satisfies the so-called *exchange property*: If $X, Y \in \mathcal{F}$ and $|X| > |Y|$, then there is an $x \in X \setminus Y$ with $Y \cup \{x\} \in \mathcal{F}$.

Matroid lemma The greedy algorithm will yield an optimal solution on a weighted matroid [for proof, see 1].

Let's take a look at minimum spanning trees as an example. We clearly have an independence system with a modular cost function: For a graph $G = (V, E)$, the independent in $\mathcal{F}$ are the subsets of E that do not contain cycles, and the cost of any such forest is the sum of the edge weights. To show that this independence system also satisfies the exchange property, let $X, Y \in \mathcal{F}$ and suppose $Y \cup \{x\} \notin \mathcal{F}$ for all $x \in X \setminus Y$. We now want to show that $|X| \leq |Y|$.

For each edge $x = (u, v) \in X$, u and v are in the same connected component of (V, Y), per our assumption. Therefore, for each connected component of (V, X) the nodes will form a subset of a connected component of (V, Y). So the number p of connected components of the forest (V, X) is greater than or equal to the number q of connected components of the forest (V, Y). But then $|V| - |X| = p \geq q = |V| - |Y|$, implying $|X| \leq |Y|$.

Tip For a more in-depth example of showing that a problem is a matroid, see Sect. 16.5 of *An Introduction to Algorithms* [1].

5.2 Greedoids

A greedoid is a set system, but it is not an independence system. Simply put, a greedoid is a matroid that is not necessarily closed under containment (that is, they are not necessarily *subclusive*). In other words: A *greedoid* is a set system $(E, \mathcal{F})$, with $\emptyset \in \mathcal{F}$, which satisfies the exchange property.

[2] Korte and Vygen [5] also discuss another matroid generalization, namely *polymatroids*. They are not covered here.

Note It is interesting to note that instead of subclusiveness, we now have a property called *accessibility*. A set system is accessible if $\emptyset \in \mathcal{F}$ and for any $X \in \mathcal{F} \setminus \{\emptyset\}$ there exists an $x \in X$ with $X \setminus \{x\} \in \mathcal{F}$. All greedoids are *accessible* (this follows directly from their definition).

One interesting example of greedoids that are *not* matroids are undirected branchings. If $G = (V, E)$ is a graph and $r \in V$ is a specified root, and $\mathcal{F}$ is the family of edge-sets of subtrees of G containing r, then $(E, \mathcal{F})$ is a greedoid. This is exactly the situation we have with Prim's algorithm (and this is why matroids can not be used to prove its correctness): We do not wish to include arbitrary subforests of G in $\mathcal{F}$, as we require our solutions to be "growable" from r.

To show that this is a greedoid, we need to show that the exchange property is satisfied: If $X, Y \in \mathcal{F}$ and $|X| > |Y|$, then there is an $x \in X \setminus Y$ with $Y \cup \{x\} \in \mathcal{F}$. This is indeed satisfied, because there must be a node covered by X that is not covered by Y, and that is reachable from Y by the edge x.

Assume that you have a greedoid $(E, \mathcal{F})$ and a (not necessarily modular) function $c : 2^E \to \mathbb{R}$, given by an oracle which for any given $X \subseteq E$ says whether $X \in \mathcal{F}$ and returns $c(X)$. The following is the **greedy algorithm for greedoids**:

1. Let $F = \emptyset$.
2. Let $e \in E \setminus F$ such that $F \cup \{e\} \in \mathcal{F}$ and $c(F \cup \{e\})$ is maximum; if no such e exists then stop.
3. $F \leftarrow F \cup \{e\}$; go to step 2.

Note that even for modular cost functions, this algorithm does not always provide an optimal solution. In fact, optimizing modular cost functions (let alone more complex functions) over general greedoids is NP-hard.[3]

Greedoid lemma The greedy algorithm for greedoids finds a set $F \in \mathcal{F}$ of maximum weight for each modular weight function $c : 2^E \to \mathbb{R}_+$ if and only if $(E, \mathcal{F})$ has the so-called *strong exchange property*: For all $A \in \mathcal{F}$, $B \in \mathcal{B}$, $A \subseteq B$ and $x \in E \setminus B$ with $A \cup \{x\} \in \mathcal{F}$ there exists a $y \in B \setminus A$ such that $A \cup \{y\} \in \mathcal{F}$ and $B \setminus \{y\} \cup \{x\} \in \mathcal{F}$ [for proof, see 5].

The strong exchange property can be rephrased as follows: B is a base for the set system (that is, it is not a subset of any other element of $\mathcal{F}$) and A is a subset of B. The element x is outside B and we can legally add x to A, "branching away from" B. We then require that we can exchange an element y in B (outside A) with x. That is, we are allowed to add y to A, and if we remove y from B, we can subsequently add x to it.

As an example, consider again the case of undirected branchings, in this case with an associated modular cost function (that is, an edge weight function). For this problem, the greedy algorithm for greedoids is simply Prim's algorithm. To show its correctness, we must show that the strong exchange property holds.

[3] On the other hand, there are interesting functions that can be maximized over arbitrary greedoids.

$\mathcal{B}$ is the set of spanning trees and $\mathcal{F}$ is the set of partial spanning trees grown from r.

So, we start out with an arbitrary partial spanning tree A and a full spanning tree B containing A. Now, assume that that A can be grown *outside* B by adding an edge x. We then have to show that it can also be grown *inside* B by adding another edge y, and that if we exchange y with x in B, it is still a spanning tree. This is not really all that hard to show: If A can be grown outside B by adding x, then adding x to B will give us a cycle. As x may be legally added to A, there must be at least one other branch y in this cycle that is not part of A. By exchanging y for x in B, we still have a spanning tree, which means that we have the strong exchange property in place.

5.3 Matroid embeddings

Even more general than matroids and greedoids is the concept of matroid embeddings. Its definition is quite a bit more involved than that of a greedoid.[4] Helman et al. [2] showed that the following three statements are equivalent:

1. For every possible weighted linear objective function, $(E, \mathcal{F})$ has an optimal basis.
2. $(E, \mathcal{F})$ is a matroid embedding.
3. For every linear objective function, the greedy bases of $(E, \mathcal{F})$ are exactly its optimal bases.

Simply put: There exists an explicit, exact characterization of greedy structures (but it is quite complex, and the details go way beyond the scope of this lecture).

Bibliography

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, chapter 16.4, pages 393–404. MIT Press, second edition, 2001.

[2] Paul Helman, Bernard M. E. Moret, and Henry D. Shapiro. An exact characterization of greedy structures. *SIAM Journal on Discrete Mathematics*, 6(2):274–283, 1993.

[3] Dieter Jungnickel. *Graphs, Networks and Algorithms*, chapter 5, pages 123–146. Springer, second edition, 2005.

[4] Jon Kleinberg and Éva Tardos. *Algorithm Design*, chapter 4, pages 115–207. Addison-Wesley, 2006.

[5] Bernhard Korte and Jens Vygen. *Combinatorial Optimization : Theory and Algorithms*, chapter 13 and 14. Springer, second edition, 2002.

[4] A *matroid embedding* is an accessible set system which is extensible, closure-congruent, and the hereditary closure of which is a matroid [for details, see 2].

[6] Bernhard Korte, Lázló Korte, and Rainer Schrader. *Greedoids*. Springer-Verlag, 1991.

[7] Udi Manber. Using induction to design algorithms. *Communications of the ACM*, 31(11):1300–1313, 1988.