CS 38/138 (Introduction to Algorithms) **Spring 2017**

Midterm Review

The CS38 TAs Week 5 – May 8, 2017

We're going to look at a few important themes from this class.

Algorithmic techniques: Although there is enormous variety in algorithmic problems, many can be solved using several common techniques. When solving a problem, these are usually good places to start. In the class so far, we've discussed two of these: dynamic programming and greedy methods.

Abstractions: Abstractions are useful because they allow us to reuse algorithms (and proofs!) in many different contexts. We've talked about several useful and generic mathematical abstractions for understanding algorithmic problems, such as matroids and metric spaces.

Applications: To demonstrate our algorithmic techniques, we've considered a number of common algorithmic problems, including: sorting, assorted graph problems (shorted path, minimum spanning tree, etc.), covering and packing in metric spaces. Many of these are interesting in their own right, and turn out to be useful in many contexts.

This review is meant to give you a broad overview of the ideas we've talked about; it is not comprehensive and is not necessarily indicative of the material that will appear on the midterm.[1]

1 Algorithmic Techniques

Frequently, algorithmic problems have "formulaic" answers in terms of particular algorithmic techniques. When possible, we try to apply these techniques because certain classes of solutions are well-understood; we shouldn't be creative when we don't have to be! However, it's *extremely* important to understand *why* an algorithmic technique works for certain problems, and to be very clear about the properties of the problem at make it possible to apply a particular general technique.

1.1 Dynamic programming

Dynamic programming is a "tabular" method for computing the results of shared sub-problems and combining them to solve larger problems. Using dynamic programming to solve a problem hinged critically on two properties:

Property 1 (Optimal substructure)**.** Finding the optimal solution for a problem is easy (or at least easier) once we have optimal solutions to problems of smaller sizes. Simply put, there exists a recursive formulation of the problem.

Property 2 (Overlapping sub-problems)**.** Each sub-problem shares many (sub^n-problems) with other sub-problems, so that each computed solution is re-used many times.

In working out a dynamic programming problem, it's often useful to consider the *subproblem tree*, which shows the "shape" of the optimal substructure. In a subproblem tree, each node's children are the subproblems that it depends on directly. When Property 2 holds, our subproblem tree will have many repetitions of each subproblem and (hopefully) not too many *distinct* subproblems. In that case, we can collapse our subproblem tree into a subproblem *graph* (see Fig. 1 for an illustration).

[1] i.e. do not study only what is on this sheet and complain to us later that the midterm covered something else.

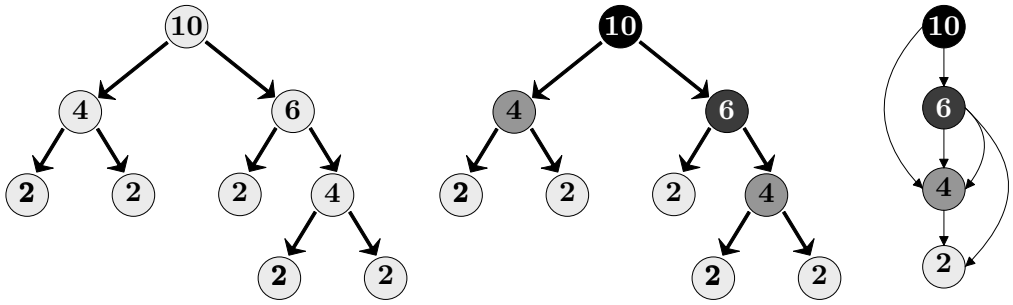

Figure 1: (a) Illustration of a subproblem tree. (b) The subproblem tree has many overlapping subproblems, which are highlighted in colours. (c) The collapsed subproblem graph. Notice that by computing "bottom-up," we can avoid recomputing values many times.

In dynamic programming, we take advantage of the overlapping subproblems by computing the problems in the subproblem graph in a "bottom up" fashion; we generally start with the smallest subproblem(s), and work our way up to the much larger problem that we actually want to solve. Usually, we store the intermediate results in some kind of table or array as we go.

However, we can solve the same problem in a top-down recursive strategy that stores the results of each subproblem so that they aren't recomputed. This "lazy" approach, where we solve the problem using recursion but don't solve the same problem over and over again, is called memoization. In the homework, many students confidently misused the term "memoization" to mean "storing intermediate results in a table," but the term actually refers to something much more specific.

1.1.1 Generic Algorithm and Runtime Analysis

Dynamic programming algorithms all follow a basic structure, which can be described generically as follows:

Generic DP Algorithm:

1. Iterate through subproblems, beginning with the "smallest" and building up to the "biggest". For each:
   (a) Find the optimal value using previously-computed optimal values to smaller subproblems.
   (b) Record the choices made to obtain this optimal value.
2. Reconstruct the optimal solution using the recorded information.

In general we may think of the runtime for a dynamic programming problem by the following formula:

Runtime = (Total number of subproblems) $\times$ (Time per subproblem)

We note further that the latter of these two terms relates to how many previously-computed subproblems we must reference in computing the optimal value for the current subproblem.

1.1.2 Problem Solving Techniques

Many dynamic programming problems can be solved by looking for and precisely defining the following few key things.

1. Define the subproblem (state the inputs and outputs).
2. Find the recursive relationship between a subproblem and smaller subproblems (how to call and use the results of smaller subproblems).
3. Define the base case.
4. Find the desired result of the original problem in terms of the subproblems.

The bulk of your mental effort will be spent figuring out the first two. It may also be useful to think of the tabular structure of the subproblems.

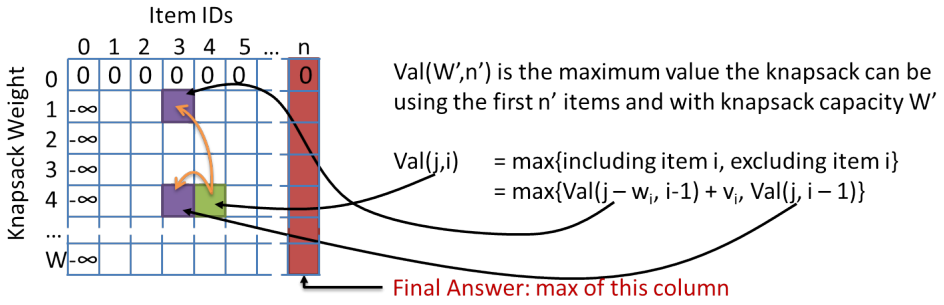

Figure 2: Table of knapsack subproblems. Exercise: pick some set of v_i, w_i. Fill in the table.

A cell at location i, j in the table is filled with the result of a subproblem called with inputs i and j. Now to visualize the recursive relationship between subproblems, pick an arbitrary cell located at i, j and draw arrows to the subproblems it depends on. The base cases will be cells along the sides of the table, while the recursive relation will allow us to successively fill in rows or columns of the table, starting at the base cases. Typically, the final answer is just one of the cells of our table, although it may also involve some computation on a number of the cells. Working through an example and filling in a table by hand is an excellent way to "understand" a dynamic programming solution.

It's one thing to know the ideas behind dynamic programming and another thing to actually be able to solve problems. In many cases, the most difficult part is coming up with a way to define the subproblems that can be efficiently computed. For example, in the 0-1 knapsack problem, you see each subproblem described as "Val$(n^{'}, W^{'})$:= maximal achievable value given the first $n^{'}$ items and a modified maximum capacity of $W^{'}$." In retrospect, it's easy to see why that allows us to solve the problem. But, for example, why couldn't we have instead split it into "Weight$(n^{'}, V^{'})$:= minimal weight needed to have a knapsack of value $V^{'}$ using the first $n^{'}$ items? And, for that matter, why can we just arbitrarily order the items? Why does a subproblem involve the first $n^{'}$ items rather than, for example, items whose index lies between two bounds, or even just all subsets of items?

This is why it's so important to practice solving many different types of dynamic programming problems. As you solve more and more of them, your mind will start to jump to the correct way of defining the subproblem.

1.1.3 Example

Let's examine a simple problem that can be solved with dynamic programming: optimal rod cutting[2]. Given a rod of length n and a table of prices p_i for $i \in \mathbb{Z}^+$, we would like to find the maximum revenue (r_n) achievable by cutting up the rod into integer lengths and selling the pieces.

Suppose we make a cut, we are now faced with the problem of how to cut up the resulting pieces. Here in lies our optimal substructure! The optimal solution for a given length of rod depends on the optimal solution for smaller lengths. We can simplify this even further by just considering that when we cut the rod,

[2]See Chapter 15.1 of CLRS.

we immediately sell one of the pieces and then are allowed to further cut the remaining piece. From this we can express the problem as:

$$r_n = \max_{1 \le i \le n} (p_i + r_{n-i})$$

In terms of overlapping subproblems we note that the naive recursive approach would repeatedly compute the maximum revenue for smaller length rods. Using our optimal substructure we can get a straightforward DP solution:

Algorithm 1 Cut-Rod Algorithm (CLRS page 366)

```
Initialize an array r[0 ··· n]
Set r[0] ← 0                                  // Base case
for j = 1 to n do                             // Fill table
    q ← −∞
    for i = 1 to j do
        q ← max(q, p_i + r[j − i])            // Apply optimal substructure
    r[j] ← q
Return r[n]
```

Applying our generic formula for a DP algorithm's runtime we note that there are n subproblems and that each takes time $O(n)$, so the algorithm has time complexity $O(n^2)$.

We may improve upon this runtime if the price function is concave and monotonically increasing. In this case we need not iterate over all possible cuttings but instead just binary search for the optimal choice (see figure below). Using binary search to select our best cut reduces the runtime on each iteration of the outer for loop to $O(\log n)$, and thus the total runtime becomes $O(n \log n)$.

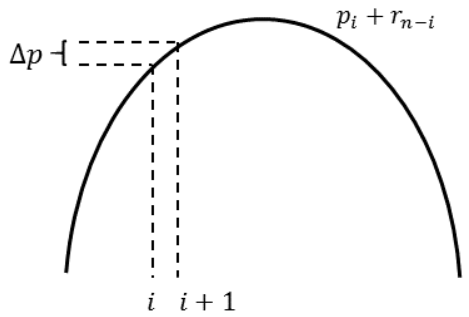

Figure 3: When using binary search to find the maximum we can use the difference Δp between $(p_i + r_{n-i})$ and $(p_{i+1} + r_{n-i-1})$ to determine which side of the maximum we are on.

1.2 Greedy methods

Usually, a problem that can be solved with a greedy algorithm exhibits Property 1, since we start by solving "smaller" versions of the problem. However, greedy methods will *only work* if another property holds, too:

Property 3 (Greedy choice)**.** A globally optimal solution can be assembled by picking locally optimal (i.e. greedy) choices.

Property 3 is related to, but distinct from, Property 2. In dynamic programming, we make a choice at each step that depends on the solutions to smaller subproblems. On the other hand, greedy algorithms make choices that can depend on the solution so far, but not on the solutions to subproblems.

Frequently, you can show that Property 3 holds using a "cut-and-paste" argument: we consider a globally optimal solution to some subproblem, and show that we can "cut out" one of the choices and "paste-in" a greedy choice instead.

1.2.1 Problem Solving Techniques

When designing and proving a greedy algorithm we break down the process into three primary components:

1. Cast the problem such that we make a choice and then have one subproblem to solve.
2. Demonstrate that there exists an optimal solution that makes the greedy choice.
3. Show the optimal substructure. After making the greedy choice, combining with the subproblem's optimal solution should maintain optimality.

Just as was the case for dynamic programming, it can often be difficult to see what measure you need to greedily optimize. Remember the various greedy approaches for activity selection that didn't work? In addition to how difficult it is to choose the right measure to greedily optimize, it can also be difficult to prove that greedy even works at all. In fact it can sometimes be harder to come up with a greedy solution than a dynamic programming one simply because it can be so hard to reason about correctness. For problems in which there are many candidate "measures" we could optimize, it's often important to reason quickly about whether the measure has a chance of being a good greedy choice, without relying on a rigorous proof. Try to quickly find counterexamples that show it can't be greedily optimized. Again, it's important to practice with a lot of different problems.

1.2.2 Examples

There are a handful of good examples of greedy algorithms that we have already covered in the homeworks and lecture, including MST, activity selection, and fractional knapsack. Let us examine another simple example covered in the recitation on greedy algorithms.

We suppose that we are given a set $\{x_1, x_2, \cdots, x_n\}$ of points on the real line, and we are interested in knowing the minimum number of unit length closed intervals that are required to completely contain all the points. This problem nicely illustrates one of the troubles with designing a greedy algorithm - namely, deciding what we should be greedily optimizing. At first glance one might be tempted to greedily optimize the number of points covered by each interval. However, we can easily show by counterexample that this is non-optimal. Instead we will repeatedly choose the left-most point not yet contained in an interval and set that as the left bound of a new interval. The details of the proof can be found in the notes from the recitation on greedy algorithms.

2 Abstractions

Throughout the course, we've found it useful to define abstractions. When we do, we are able to design algorithms to solve problems involving these abstract structures, and show that various problems fit into the framework of the abstraction. Abstractions are useful because they allow us to apply the same algorithms to many different problems.

As a side note, it's important to be able to see an abstract definition and gain an intuition for what it means. It's usually useful to think about concrete cases of the abstraction. Instead of thinking of a metric space, think of standard Euclidean space. Once you've considered enough of these concrete cases, you can being to develop an intuition for the definition, ie, why the definition is what it is. Finally, remember that if you get stuck on a problem that relies on an abstraction, you can always go back to the definition.

2.1 Matroids

We introduced matroids as general structures that support an "exchange property" that allowed us to use an efficient greedy algorithm. Formally:

Definition 1 (Matroid)**.** A *matroid* $(U, \mathcal{F})$ is a finite set U (the "universe") and a non-empty family $\mathcal{F}$ of subsets of U (called the "independent sets"), such that:

1. (Hereditary axiom) If $B \in \mathcal{F}$ and $A \subseteq B$, then $A \in \mathcal{F}$.
2. (Steinitz exchange axiom) If $A, B \in \mathcal{F}$ and $|A| < |B|$, then there exists an $x \in B \setminus A$ such that $A \cup \{x\} \in \mathcal{F}$.

Because of the Steinitz exchange axiom, the *maximum weight basis* problem (given a function $w : U \to \mathbb{R}$, find the maximum weight independent set) can be solved using a greedy algorithm described below.

Matroid Greedy Algorithm

1. Initialize $A = \emptyset$.
2. Sort elements of U by weight.
3. Repeatedly add to A the maximum-weight point $u \in U$ s.t. $A \cup \{u\} \in \mathcal{F}$ until no such u exists.
4. Return A

Runtime: $O(n \log n + nf(n))$, with $f(m)$ being the time to check if $A \cup x \in \mathcal{F}$ is independent given $A \in \mathcal{F}, |A| = m$.

2.1.1 Example

One example of a matroid (which also gives rise to some of the terminology used with it) is that of linearly independent vectors from a given set of m n-vectors. One can verify using tools from linear algebra that the bases from this given set have the same number of elements, and satisfy the exchange and hereditary properties. In fact, the names for independent sets, bases, matroid originate from these terms.

Another example discussed in class and CLRS is the graphic matroid. Given a graph $G = (V, E)$ we can define a matroid $M = (U, \mathcal{F})$ where $U = E$ and $\mathcal{F}$ is comprised of acyclic subgraphs of G. This matroid representation can be used to solve MST with the matroid greedy algorithm.

2.2 Metric spaces

When studying packing and covering problems, our setting was a generalized space that supported a notion of distance:

Definition 2 (Metric Space)**.** A metric space is an ordered pair (M, d) of a set M and a metric $d : M \times M \to \mathbb{R}$ that takes two elements of M and returns a real number—the distance. To be considered a metric, d must satisfy the following:

1. $d(p_1, p_2) \geq 0$ (non-negative), with equality iff $p_1 = p_2$
2. $d(p_1, p_2) = d(p_2, p_1)$ (symmetry)
3. $d(p_1, p_3) \leq d(p_1, p_2) + d(p_2, p_3)$ (triangle inequality)

2.2.1 Example

A simple example is 2-dimensional Euclidean space. Here, our metric space is $M = (\mathbb{R} \times \mathbb{R})$, ie, all pairs of real numbers, and our metric is $d(p_1, p_2) = \sqrt{(y_2 - y_1)^2 + (x_2 - x_1)^2}$. Just the standard Euclidean distance. You should verify that the metric space definition holds.

To see the real power of metric spaces, let's consider something completely different: the Hamming distance between strings of size n. The Hamming distance is a specific case of the edit distance where you can only perform substitutions of characters in the string. Here, the metric space is the set of all strings of size n, and our metric is the Hamming distance.

We will examine one more application of a metric space. Consider an undirected connected graph $G = (V, E)$. We have a corresponding metric space (M, d) where $M = V$ and $d(x, y)$ is defined as the length of the shortest path between $x, y \in V$. (For now just think about this as an unweighted graph). Positivity and symmetry follow easily. To show that it satisfies the triangle inequality $d(x, y) \leq d(x, u) + d(u, y)$, we should consider case where u lies along the shortest path from x to y, and the alternative case where u is off of this path. It is interesting to note that the Hamming distance metric can be considered an instance of the aforementioned graph metric for shortest paths. The graph $G = (V, E)$ is comprised of vertices for each word and edges connect words separated by a single substitution.

Another rather simple example is the discrete metric, where in we take a set M and define the metric:

$$d(x, y) = \begin{cases} 1, & x \neq y \\ 0, & x = y \end{cases}$$

Quick examination of this metric shows that it satisfies all three conditions.

3 Applications

Throughout the class, we've considered a number of example problems to illustrate our algorithmic techniques. Some of these problems are important in their own right and are frequently used when solving algorithmic problems. Throughout our discussion of graph algorithms we will denote $n = |V|$ and $m = |E|$.

3.1 Minimum spanning trees

The minimum spanning tree problem asks: given a connected graph $G = (V, E, w)$, find a spanning tree of minimum weight. In class, we discussed two algorithms for solving the minimum spanning tree problem:

Prim's algorithm: At all times, we maintain a single tree that spans some subset of V. At each iteration, we find the least weight edge that connects a vertex in the current tree and a vertex outside of the current tree. Runs in $O((m + n)\log(n))$ with a binary heap, or $O(m + n \log n)$ with a Fibonacci heap.

Kruskal's algorithm: At all times, we maintain a set of disjoint forests that cover all of V. At each iteration, we find the least weight edge that cross from one disjoint forest to another. Runs in $O(m \log(n))$.

3.2 Single source shortest path

The single source shortest path problem asks: given a connected graph $G = (V, E, w)$ where E contains no self loops (an edge from a vertex to itself) and w is a non-negative weight function, find the paths of minimum weight from a specific vertex $u \in V$ to all other vertices $v \in V$. In class, we discussed two algorithms for solving this problem:

Dijkstra's Algorithm: We assign a tentative distance value to each vertex v representing its distance from the source vertex. Then, starting from the source vertex, we greedily select, then process, the closest unprocessed vertex and update adjacent vertices based on whether a shorter path from the source through the recently processed vertex exists. We continue processing vertices outward until reaching the target vertex. Dijkstra's runs in $O((m + n)\log(n))$ with a binary heap, or $O(m + n \log n)$ with a Fibonacci heap.

Bellman-Ford Algorithm: This algorithm has run time $O(nm)$, but can actually handle negative weight functions and detect negative weight cycles. Initially, a key function is set for all vertices $v \neq u$, equal to ∞. We then iterate: for each $(a, b) \in E$, we set $key(b) = key(a) + w(a, b)$ if it is greater than the latter value. If we ever iterate n times, we report a negative-weight cycle. This algorithm has a relaxation property which is similar to the Floyd-Warshall Algorithm below: it begins with an upper bound on the least weight path, and slowly pushes it down to the actual least weight.

3.3 All-pairs shortest paths

The All-pairs shortest paths problem asks: given a connected graph $G = (V, E, w)$ where E contains no self loops (an edge from a vertex to itself), the least-weight path for every pair of vertices $(u, v) \in V \times V$. We covered two main methods of solving this problem in class:

Floyd-Warshall Algorithm: This is an $O(n^3)$ dynamic programming algorithm, which is the best known for dense graphs. It works by considering the minimal paths with the non-endpoint nodes contained small subsets of the vertices, and using these sub-problems to find the global minimum.

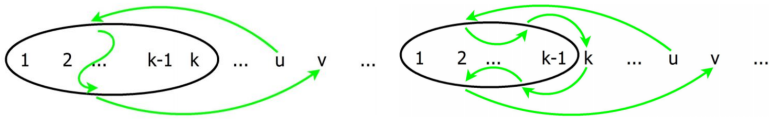

Figure 4: Floyd-Warshall schematic. We consider subsets of non-endpoint vertices.

Johnson's Algorithm: This is an $O(n(m + n \log n))$ algorithm to solve the same problem. In graphs where the feasible number of edges m is less than $O(n^2)$, this algorithm performs like an $O(n^{3-\epsilon} \log n)$ algorithm, which is better than the above.

- Compute a potential function ϕ on each vertex using the $O(nm)$ Bellman-Ford Algorithm, then remapping the weights $w(u, v)$ to $w'(u, v) = w(u, v) + \phi(u) - \phi(v)$.
- The all-source shortest paths are preserved under this new weight function between the two graphs, and the added benefit that w' is a non-negative weight function.
- We now run Dijkstra's Algorithm for all n vertices in V, which solves the problem and gives the desired run time (using Fibonacci heaps.)

3.4 Packing and covering

Recall that for packing and covering, we're working on a finite metric space, ie, a complete graph $G = (V, w)$. We're interested in balls centered at vertices of G. Formally, a ball $B(x, r) = \{v : w(x, v) \leq r\}$. We're interested in the problem of packing these balls and covering using these balls. We say a $(k, r) -$ covering is a set of center vertices $x_1, x_2, \ldots, x_k$ such that $V = \bigcup_{i=1}^{k} B(x_i, r)$ and a $(k, r) -$ packing is a set of center vertices $y_1, y_2, \ldots, y_k$ such that $\varnothing = \bigcap_{i=1}^{k} B(y_i, r)$. It is important to note that for a packing drawn in Euclidean geometry the balls are only disallowed from overlapping in regards to the finite metric space (i.e. they don't contain any common points), but may have overlapping regions in the Euclidean space you draw them in. Table 1 covers the sorts of problems we're interested in.

Table 1: Summary of Problems Related to Covering and Packing

Find	COVER	PACK	Weak Duality
K	$\min\{k : \exists(k,r) - \text{covering}\}$	$\max\{k : \exists(k,r) - \text{packing}\}$	$K_{\text{PACK}}(r) \leq K_{\text{COVER}}(r)$
R	$\inf\{r : \exists(k,r) - \text{covering}\}$	$\sup\{r : \exists(k,r) - \text{packing}\}$	$R_{\text{PACK}}(k+1) \leq R_{\text{COVER}}(k)$

(NB: inf (infimum) and sup (supremum) can be thought of as similar to min and max, except they don't require the minimum and maximum actually exist. Why do we need to use these when finding optimal r?).

3.5 Proof of Dualities

We'll outline the proofs for the weak dualities mentioned in the table above. Let's first consider $K_{\text{PACK}}(r) \leq K_{\text{COVER}}(r)$. Suppose to the contrary that $K_{\text{PACK}}(r) > K_{\text{COVER}}(r)$. We must then have some ball in the covering that contains at least two centers from the packing (by the pigeonhole principle). However since the radius of the balls in the covering and packing is the same, we certainly have that the two packing balls must intersect. By contradiction then, we see that the weak duality holds. We may approach the other duality in a very similar manner. If we again take a proof by contradiction, assuming instead that $R_{\text{PACK}}(k+1) > R_{\text{COVER}}(k)$, there must be a ball in the covering that contains at least two centers from the packing. Since we assumed then that the packing radius was greater, we are assured that the packing balls associated with these centers have an intersection. So by contradiction the duality $R_{\text{PACK}}(k+1) \leq R_{\text{COVER}}(k)$ is shown to hold true.

3.5.1 Example

G/H-S approximation for k-Center (trying to compute $R_{\text{COVER}}(k)$).

Algorithm 2 G/H-S Algorithm

```
Pick any x_1 ∈ V.
Set i ← 1.
while i ≠ k do
    Increment i.
    Set x_i to be the furthest point from x_1, ..., x_{i-1}.
```

Theorem 1 (G/H-S 2-Approximation)**.** $R_C(x_1, x_2, \ldots, x_k) \leq 2 \cdot R_{\text{COVER}}(k)$, where R_C is the radius found via the G/H-S algorithm.

Proof. See lecture 6. □

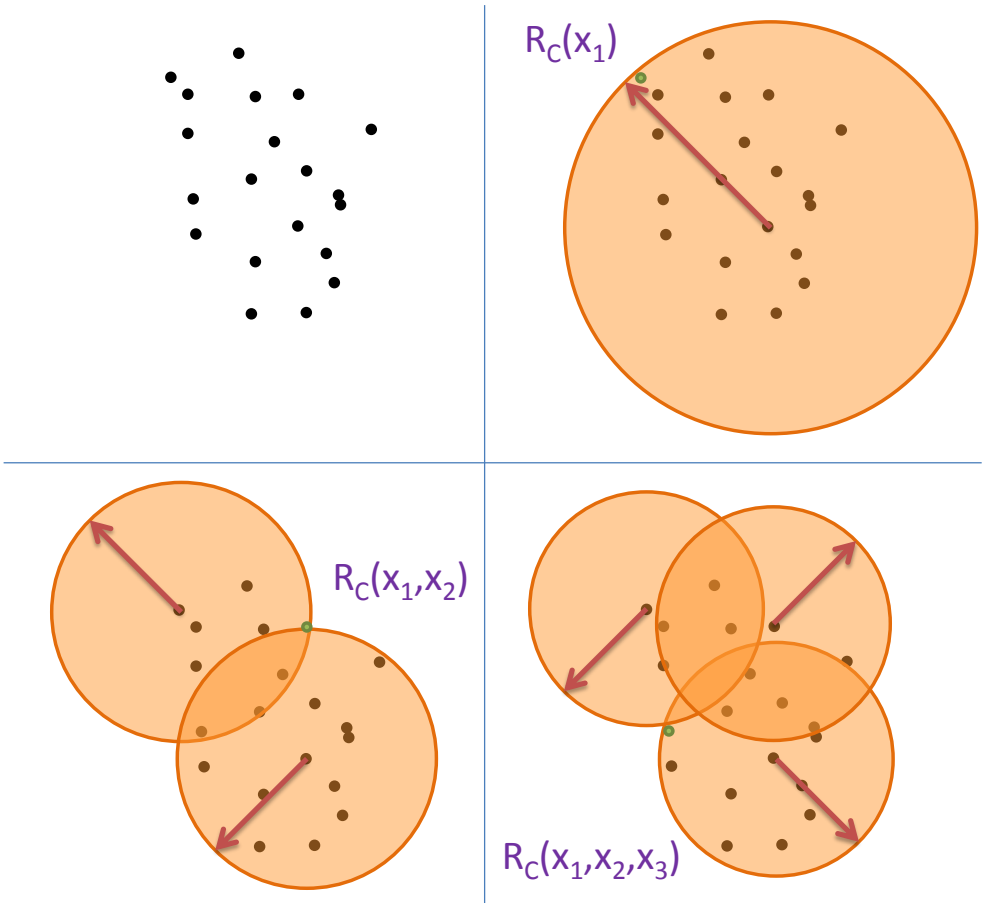

Figure 5: G/H-S algorithm through a few iterations.