Algorithms (XV)

Yijia Chen
Shanghai Jiaotong University

The simplex algorithm

General description

```
let v be any vertex of the feasible region
while there is a neighbor v' of v with better objective value:
    set v = v'
```

Vertices and neighbors in n-dimensional space

Definition

Each vertex is the unique point at which some subset of hyperplanes meet.

Pick a subset of the inequalities. If there is a unique point that satisfies them with equality, and this point happens to be feasible, then it is a vertex.

Each vertex is specified by a set of n inequalities.

Definition

Two vertices are neighbors if they have $n - 1$ defining inequalities in common.

The algorithm

On each iteration, simplex has two tasks:

1. Check whether the current vertex is optimal (and if so, halt).
2. Determine where to move next.

The convenience for the origin

Suppose we have some generic LP:

$$\begin{gathered}\max \mathbf{c}^T \mathbf{x} \\ \mathbf{A}\mathbf{x} \leq \mathbf{b} \\ \mathbf{x} \geq 0\end{gathered}$$

where $\mathbf{x}$ is the vector of variables, $\mathbf{x} = (x_1, \ldots, x_n)$.

Suppose the origin is feasible. Then it is certainly a vertex, since it is the unique point at which the n inequalities

$$\{x_1 \geq 0, \ldots, x_n \geq 0\}$$

are *tight*.

Task 1 in the origin

Lemma

The origin is optimal if and only if all $c_i \geq 0$. (We need $\mathbf{b} > 0$)

Task 2 in the origin

We can move by increasing some x_i for which $c_i > 0$.

Until we hit some other constraint. That is, we release the tight constraint $x_i \geq 0$ and increase x_i until some other inequality, previously loose, now becomes tight.

At that point, we again have exactly n tight inequalities, so we are at a *new* vertex.

What if our current vertex $\mathbf{u}$ is elsewhere?

The trick is to transform $\mathbf{u}$ into the origin, by shifting the coordinate system from the usual $(x_1, \ldots, x_n)$ to the "local view" from $\mathbf{u}$.

These local coordinates consist of (appropriately scaled) distances $y_1, \ldots, y_n$ to the n hyperplanes (inequalities) that define and enclose u.

Specifically, if one of these enclosing inequalities is $\mathbf{a}_i \cdot \mathbf{x} \leq b_i$, then the distance from a point x to that particular "wall" is

$$y_i = b_i - \mathbf{a}_i \cdot \mathbf{x}.$$

The n equations of this type, one per wall, define the y_i's as linear functions of the x_i's, and this relationship can be inverted to express the x_i's as a linear function of the y_i's.

Rewriting the LP

Thus we can rewrite the entire LP in terms of the y's.

This doesn't fundamentally change it (for instance, the optimal value stays the same), but expresses it in a different coordinate frame.

The revised "local" LP has the following three properties:

1. It includes the inequalities $\mathbf{y} \geq 0$, which are simply the transformed versions of the inequalities defining $\mathbf{u}$.
2. $\mathbf{u}$ itself is the origin in $\mathbf{y}$-space.
3. The cost function becomes $\max c_{\mathbf{u}} + \tilde{\mathbf{c}}^T \mathbf{y}$, where $c_{\mathbf{u}}$ is the value of the objective function at $\mathbf{u}$ and $\tilde{\mathbf{c}}$ is a transformed cost vector.

The starting vertex

In a general LP, the origin might not be feasible and thus not a vertex at all.

Start with any linear program in *standard form*:

$$\min \mathbf{c}^T \mathbf{x} \text{ such that } \mathbb{A}\mathbf{x} = \mathbf{b} \text{ and } \mathbf{x} \geq 0.$$

We first make sure that the right-hand sides of the equations are all nonnegative: if $b_i < 0$, just multiply both sides of the ith equation by -1.

Then we create a new LP as follows:

- Create m new artificial variables $z_1, \ldots, z_m \geq 0$, where m is the number of equations.
- Add z_i to the left-hand side of the ith equation.
- Let the objective, to be *minimized*, be $z_1 + z_2 + \cdots + z_m$.

The starting vertex (cont'd)

For this new LP, it's easy to come up with a starting vertex, namely, the one with $z_i = b_i$ for all i and all other variables zero.

Therefore we can solve it by simplex, to obtain the optimum solution.

There are two cases:

1. If the optimum value of $z_1 + \cdots + z_m$ is zero, then all z_i's obtained by simplex are zero, and hence from the optimum vertex of the new LP we get a starting feasible vertex of the original LP, *just by ignoring the z_i's*.
2. If the optimum objective turns out to be positive: We tried to minimize the sum of the z_i's, but simplex decided that it cannot be zero. But this means that the original linear program is *infeasible*:
   it needs some nonzero z_i's to become feasible.

Degeneracy

A vertex is degenerate if it is the intersection of more than n faces of the polyhedron, say $n + 1$.
Algebraically, it means that if we choose any one of n sets of $n + 1$ inequalities and solve the corresponding system of three linear equations in n unknowns, we'll get the *same* solution in all $n + 1$ cases.
This is a *serious problem*: simplex may return a *suboptimal* degenerate vertex simply because all its neighbors are identical to it and thus have no better objective.

And if we modify simplex so that it detects degeneracy and continues to hop from vertex to vertex despite lack of any improvement in the cost, it may end up looping forever.

Degeneracy (cont'd)

One way to fix this is by a *perturbation*:

change each b_i by a tiny random amount to $b_i \pm \varepsilon_i$.

This doesn't change the essence of the LP since the ε_i's are tiny, but it has the effect of differentiating between the solutions of the linear systems.

Unboundedness

In some cases an LP is *unbounded*, in that its objective function can be made arbitrarily large (or small, if it's a minimization problem).

If this is the case, simplex will discover it: in exploring the neighborhood of a vertex, it will notice that taking out an inequality and adding another leads to an underdetermined system of equations that has an infinity of solutions.

And in fact (this is an easy test) the space of solutions contains a whole line across which the objective can become larger and larger, all the way to ∞.

In this case simplex halts and complains.

The running time of simplex

How many iterations could there be?

At most

$$\binom{m+n}{n},$$

i.e., the number of vertices.

It is **exponential** in n.
And in fact, there are examples of LPs for which simplex does indeed take an exponential number of iterations.

Simplex is an exponential-time algorithm.

However, such exponential examples do not occur in practice, and it is this fact that makes simplex so valuable and so widely used.

Postscript: circuit evaluation

An ultimate application

We are given a Boolean circuit, that is, a dag of gates of the following types.

- **Input** gates have indegree zero, with value `true` or `false`.
- **AND** gates and OR gates have indegree 2.
- **NOT** gates have indegree 1.

In addition, one of the gates is designated as the **output**.

The CIRCUIT VALUE problem is the following: when the laws of Boolean logic are applied to the gates in topological order, does the output evaluate to `true`?

LP formulation

Create a variable x_g for each gate g, with constraints $0 \leq x_g \leq 1$.

Add additional constraints for each type of gate:

- `true` gate: $x_g = 1$.
- `false` gate: $x_g = 0$.
- OR gate with inputs h and h': $x_g \geq x_h$, $x_g \geq x_{h'}$, $x_g \leq x_h + x_{h'}$.
- AND gate with inputs h and h': $x_g \leq x_h$, $x_g \leq x_{h'}$, $x_g \geq x_h + x_{h'} - 1$.
- NOT gate with input h: $x_g = 1 - x_h$.

These constraints force all the gates to take on exactly the right values – 0 for `false`, and 1 for `true`.
We don't need to maximize or minimize anything, and we can read the answer off from the variable x_o corresponding to the **output** gate.

The generality

The CIRCUIT VALUE problem is in a sense the most general problem solvable in polynomial time!
After all, any algorithm will eventually run on a computer, and the computer is ultimately a Boolean combinational circuit implemented on a chip.

If the algorithm runs in polynomial time, it can be rendered as a Boolean circuit consisting of polynomially many copies of the computer's circuit, one per unit of time,
with the values of the gates in one layer used to compute the values for the next.
Hence, the fact that CIRCUIT VALUE reduces to linear programming means that *all problems that can be solved in polynomial time do!*

Chapter 8. NP-complete problems

Search problems

Efficient algorithms

We have developed algorithms for

- FINDING SHORTEST PATHS IN GRAPHS,
- MINIMUM SPANNING TREES IN GRAPHS,
- MATCHINGS IN BIPARTITE GRAPHS,
- MAXIMUM INCREASING SUBSEQUENCES,
- MAXIMUM FLOWS IN NETWORKS,
- .....

All these algorithms are efficient, because in each case their time requirement grows as a polynomial function (such as n, n^2, or n^3) of the size of the input.

Exponential search space

In all these problems we are searching for a **solution** (path, tree, matching, etc.) from among an exponential population of possibilities.

All these problems could in principle be solved in exponential time by checking through all candidate solutions, one by one.

But an algorithm whose running time is 2^n, or worse, is all but useless in practice

The quest for efficient algorithms is about finding clever ways to bypass this process of exhaustive search, using clues from the input in order to dramatically narrow down the search space.

Now will see some other "search problems" in which again we are seeking a solution with particular properties among an exponential chaos of alternatives. The fastest algorithms we know for them are all exponential.

Satisfiability

The instances of SATISFIABILITY or SAT:

$$(x \vee y \vee \bar{z})(x \vee \bar{y})(y \vee \bar{z})(z \vee \bar{x})(\bar{x} \vee \bar{y} \vee \bar{z})$$

That is, a Boolean formula in conjunctive normal form (CNF).
It is a collection of *clauses* (the parentheses), each consisting of the **disjunction** (logical or, denoted $\vee$) of several literals, where a literal is either a **Boolean variable** (such as x) or the **negation** of one (such as $\bar{x}$).

A *satisfying truth assignment* is an assignment of `false` or `true` to each variable so that *every clause contains a literal whose value is* `true`.

The SAT problem is the following: given a Boolean formula in conjunctive normal form, either find a satisfying truth assignment or else report that none exists.

Satisfiability (cont'd)

SAT is a typical search problem.

We are given an instance I (that is, some input data specifying the problem at hand, in this case a Boolean formula in conjunctive normal form), and we are asked to find a solution S (an object that meets a particular specification, in this case an assignment that satisfies each clause).
If no such solution exists, we must say so.

Search problems

A search problem is specified by an algorithm $\mathcal{C}$ that takes two inputs, an instance I and a proposed solution S, and runs in time polynomial in $|I|$.

We say S is a **solution** to I if and only if $C(I, S) = \texttt{true}$.

The traveling salesman problem

In the traveling salesman problem (TSP) we are given n vertices $1, \ldots, n$ and all $n(n-1)/2$ distances between them, as well as a *budget* b.
We are asked to find a **tour**, a cycle that passes through every vertex exactly once, of total cost b or less – or to report that no such tour exists.

That is, we seek a permutation $\tau(1), \ldots, \tau(n)$ of the vertices such that when they are toured in this order, the total distance covered is at most b:

$$d_{\tau(1),\tau(2)} + d_{\tau(2),\tau(3)} + \cdots + d_{\tau(n),\tau(1)} \leq b.$$

We have defined the TSP as a search problem: given an instance, find a tour within the budget (or report that none exists).

But why are we expressing the traveling salesman problem in this way, when in reality it is an *optimization* problem, in which the shortest possible tour is sought?

Search vs. optimization

Turning an optimization problem into a search problem does not change its difficulty at all, because the two versions *reduce to one another*.

Any algorithm that solves the optimization TSP also readily solves the search problem: find the optimum tour and if it is within budget, return it; if not, there is no solution.

Conversely, an algorithm for the search problem can also be used to solve the optimization problem:

- First suppose that we somehow knew the cost of the optimum tour; then we could find this tour by calling the algorithm for the search problem, using the optimum cost as the budget.
- We can find the optimum cost by **binary search**.

Then why search instead of optimization?

Isn't any optimization problem also a search problem in the sense that we are searching for a solution that has the property of being *optimal*?

The solution to a search problem should be easy to recognize, or as we put it earlier, polynomial-time checkable.

Given a potential solution to the TSP, it is easy to check the properties "is a tour" (just check that each vertex is visited exactly once) and "has total length $\leq b$."

But how could one check the property "is optimal"?

Euler Path

EULER PATH: Given a graph, find a path that *contains each edge exactly once*.

The answer is yes if and only if

(a) the graph is connected and

(b) every vertex, with the possible exception of two vertices (the start and final vertices of the walk), has even degree.

Using above, it is easy to see that there is a polynomial time algorithm for EULER PATH.

Rudrata Cycle

RUDRATA CYCLE: Given a graph, find a cycle that *visits each vertex exactly once*.

In the literature this problem is known as the *Hamilton cycle problem*.

Minimum Cut

A cut is a set of edges whose removal leaves a graph disconnected.

MINIMUM CUT: given a graph and a budget b, find a cut with at most b edges.

This problem can be solved in polynomial time by $n-1$ max-flow computations: give each edge a capacity of 1, and find the maximum flow between some fixed node and every single other node.
The smallest such flow will correspond (via the max-flow min-cut theorem) to the smallest cut.

Balanced Cut

In many graphs, the smallest cut leaves just a singleton vertex on one side – it consists of all edges adjacent to this vertex.

Far more interesting are *small cuts that partition the vertices of the graph into nearly equal-sized sets.*

BALANCED CUT: Given a graph with n vertices and a budget b, partition the vertices into two sets S and T such that $|S|, |T| \geq n/3$ and such that there are at most b edges between S and T.

Integer linear programming

INTEGER LINEAR PROGRAMMING (ILP): We are given a set of linear inequalities $\mathbf{Ax} \leq \mathbf{b}$, where $\mathbf{A}$ is an $m \times n$ matrix and $\mathbf{b}$ is an m-vector; an objective function specified by an n-vector $\mathbf{c}$; and finally, a goal g (the counterpart of a budget in maximization problems).

We want to find a nonnegative integer n-vector $\mathbf{x}$ such that $\mathbf{Ax} \leq \mathbf{b}$ and $\mathbf{c} \cdot \mathbf{x} \geq g$.

But there is a *redundancy* here: the last constraint $\mathbf{c} \cdot \mathbf{x} \geq g$ is itself a linear inequality and can be absorbed into $\mathbf{Ax} \leq \mathbf{b}$.

So, we define ILP to be following search problem: given $\mathbf{A}$ and $\mathbf{b}$, find a nonnegative integer vector $\mathbf{x}$ satisfying the inequalities $\mathbf{Ax} \leq \mathbf{b}$.

Three-dimensional matching

3D MATCHING: There are n boys and n girls, but also n pets, and the compatibilities among them are specified by a set of **triples**, *each containing a boy, a girl, and a pet*.

Intuitively, a triple (b, g, p) means that boy b, girl g, and pet p get along well together.

We want to find n disjoint triples and thereby create n harmonious households.

Independent set, vertex cover, and clique

INDEPENDENT SET: Given a graph and an integer g, find g vertices, no two of which have an edge between them.

VERTEX COVER: Given a graph and an integer b, find b vertices cover (touch) every edge.

CLIQUE: Given a graph and an integer g, find g vertices such that all possible edges between them are present.

Longest path

LONGEST PATH: Given a graph G with nonnegative edge weights and two distinguished vertices s and t, along with a goal g.

We are asked to find a path from s to t *with total weight at least* g.
To avoid trivial solutions we require that the path be *simple*, containing no repeated vertices.

Knapsack

KNAPSACK: We are given integer weights $w_1, \ldots, w_n$ and integer values $v_1, \ldots, v_n$ for n items. We are also given a weight capacity W and a goal g

We seek a set of items whose total weight is at most W and whose total value is at least g.

The problem is solvable in time $O(nW)$ by dynamic programming.

Subset sum

SUBSET SUM: Find a subset of a given set of integers that adds up to exactly W.