7 OPTIMIZATION

The engineer is continually faced with non-trivial decisions, and discerning the best among alternatives is one of the most useful and general tasks that one can master. Optimization exists because in nearly every endeavor, one is faced with tradeoffs. Here are some examples:

- Contributing to savings versus achieving enjoyment from purchases made now;
- Buying an expensive bicycle from one of many manufacturers - you are faced with choices on accessories, weight, style, warranty, performance, reputation, and so on;
- Writing a very simple piece of code that can solves a particular problem versus developing a more professional and general-use product;
- Size of the column to support a roof load;
- How fast to drive on the highway;
- Design of strength bulkheads inside an airplane wing assembly

The field of optimization is very broad and rich, with literally hundreds of different classes of problems, and many more methods of solution. Central to the subject is the concept of the *parameter space* denoted as X, which describes the region where specific decisions x may lie. For instance, acceptable models of a product off the shelf might be simply indexed as x_i. x can also be a vector of specific or continuous variables, or a mixture of the two. Also critical is the concept of a *cost* $f(x)$ that is associated with a particular parameter set x. We can say that f will be minimized at the optimal set of parameters x^*:

$$f(x^*) = \min_{x \epsilon X} f(x).$$

We will develop in this section some methods for continuous parameters and others for discrete parameters. We will consider some concepts also from planning and multi-objective optimization, e.g., the case where there is more than one cost function.

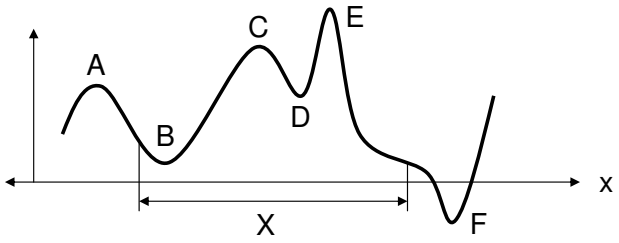
7.1 Single-Dimension Continuous Optimization

Consider the case of only one parameter, and one cost function. When the function is known and is continuous - as in many engineering applications - a very reasonable first method to try is to zero the derivative. In particular,

$$\left[\frac{df(x)}{dx}\right]_{x=x*} = 0.$$

The user has to be aware even in this first problem that there can exist multiple points with zero derivative. These are any locations in X where $f(x)$ is flat, and indeed these could be

at local minima and at local and global maxima. Another important point to note is that if X is a finite domain, then there may be *no* location where the function is flat. In this case, the solution could lie along the boundary of X, or take the minimum within X. In the figure below, points A and C are local maxima, E is the global maxima, B and D are local minima, and F is the global minimum shown. However, the solution domain X does not admit F, so the best solution would be B. In all the cases shown, however, we have at the maxima and minima $f'(x) = 0$. Furthermore, at maxima $f''(x) < 0$, and at minima $f''(x) > 0$.



We say that a function $f(x)$ is *convex* if and only if it has everywhere a nonnegative second derivative, such as $f(x) = x^2$. For a convex function, it should be evident that any minimum is in fact a global minimum. (A function is concave if $-f(x)$ is convex.) Another important fact of convex functions is that the graph always lies above any points on a tangent line, defined by the slope at point of interest:

$$f(x + \delta x) \geq f(x) + f'(x)\delta$$

When δ is near zero, the two sides are approximately equal.

Suppose that the derivative-zeroing value x^* cannot be deduced directly from the derivative. We need another way to move toward the minimum. Here is one approach: move in the downhill direction by a small amount that scales with the inverse of the derivative. Letting $\delta = -\gamma/f'(x)$, makes the above equation

$$f(x + \delta) \approx f(x) - \gamma$$

The idea is to take small steps downhill, e.g., $x_{k+1} = x_k + \delta$ where k indicates the k'th guess, and this algorithm is usually just called a gradient method, or something similarly nondescript! While it is robust, one difficulty with this algorithm is how to stop, because it tries to move a constant decrement in f at each step. It will be unable to do so near a flat minimum, although one can of course to modify γ on the fly to improve convergence.

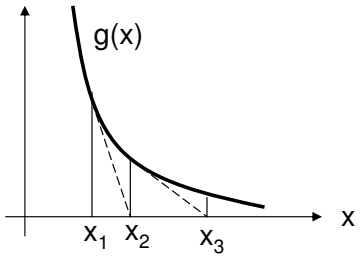
As another method, pose a new problem in which $g(x) = f'(x)$, and we now have to solve $g(x^*) = 0$. Finding the zero of a function is an important problem on its own. Now the convexity inequality above resembles the Taylor series expansion, which is rewritten here in full form:

$$g(x + \delta) = g(x) + g'(x)\delta + \frac{1}{2!}g''(x)\delta^2 + \frac{1}{2!}g'''(x)\delta^3 + \cdots$$

The expansion is theoretically true for any x and any δ (if the function is continuous), and so clearly $g(x+\delta)$ can be at least approximated by the first two terms on the right-hand side. If it is desired to set $g(x+\delta)=0$, then we will have the estimate

$$\delta = -g(x)/g'(x).$$

This is Newton's first-order method for finding the zero of a function. The idea is to shoot down the tangent line to its own zero crossing, and take the new point as the next guess. As shown in the figure, the guesses could go wildly astray if the function is too flat near the zero crossing.



Let us view this another way. Going back to the function f and the Taylor series approximation

$$f(x+\delta) \approx f(x) + f'(x)\delta + \frac{1}{2!}f''(x)\delta^2,$$

we can set to zero the left-hand side derivative with respect to δ, to obtain

$$\begin{aligned} 0 &\approx 0 + f'(x) + f''(x)\delta \longrightarrow \\ \delta &= -f'(x)/f''(x). \end{aligned}$$

This is the same as Newton's method above since $f'(x) = g(x)$. It clearly employs both first and second derivative information, and will hit the minimum in one shot(!) if the function truly is quadratic and the derivatives $f'(x)$ and $f''(x)$ are accurate. In the next section dealing with multiple dimensions, we will develop an analogous and powerful form of this method. Also in the next section, we refer to a line search, which is merely a one-dimensional minimization *along a particular direction*, using a (unspecified) one-dimensional method - such as Newton's method applied to the derivative.

7.2 Multi-Dimensional Continuous Optimization

Now we consider that the cost function f is a function of more than one variable. X is a multi-dimensional space, and x is a vector. We consider again continuous functions. At the

minima, as for the single-dimension case, we have certain conditions for the first and second derivatives:

$$\begin{aligned} \nabla f(x*) &= [f_{x_1}, f_{x_2}, \cdots]_{x=x^*} = [0, 0, \cdots] \\ \nabla^2 f(x) &= \begin{bmatrix} f_{x_1x_1} & f_{x_1x_2} & \\ f_{x_2x_1} & f_{x_2x_2} & \\ & & \cdots \end{bmatrix}_{x=x^*} > 0, \end{aligned}$$

where the notation that a matrix is greater than zero denotes that it is positive definite. We will present three practical methods for finding the minimum. First is the method of steepest descent. The idea here is to find the downhill direction and take a step δ that way:

$$\begin{aligned} e &= -\nabla f(x)/||\nabla f(x)|| \\ \delta &= \gamma e. \end{aligned}$$

Note that, as written, this is a different algorithm than the first method given in one dimension, because here the direction vector e is normalized, and hence the magnitude of the step in x is the same no matter what the steepness of the function. We note also that there exists a value α such that $x+\alpha e$ is the minimizing argument of f, along the e direction and passing through the point x. This is the result of the so-called line search, and a reasonable steepest descent procedure is to perform iteratively a two-step procedure of a gradient calculation followed by a line search in the downhill direction.

The performance of successive line searches can be quite good, or very poor. Poor performance results in some cases because the successive downhill directions are constrained to be orthogonal to each other. This has to be the case because at the minimum on the line, the gradient in the direction of the line is zero by definition. In fact none of these downhill directions may actually point to the minimum, and so many pointless steps might be taken. A solution to this is the conjugate gradient method, wherein we make a useful modification to the downhill directions used for each of the line searches.

We will call the downhill direction vector corresponding with the k'th guess d_k. Letting $g(x) = \nabla f(x)$ and $d_0 = -g(x_0)$, we will let $d_{k+1} = -g(x_{k+1}) + \beta d_k$; this says that we will *deflect* the next search direction from the downhill direction by the term βd_k; the scalar factor β is given by

$$\beta = \frac{g(x_{k+1})^T g(x_{k+1})}{g(x_k)^T g(x_k)}$$

Note here that d is not normalized. The algebra needed to derive this rule is not difficult, and a simple example will illustrate that it is a very powerful one.

Finally, we mention Newton's second-order method in multiple dimensions, using the second derivative. It comes from the multivariable version of the Taylor series expansion:

$$f(x+\delta) \approx f(x) + \nabla f(x)\delta + \frac{1}{2}\delta^T \nabla^2 f(x)\delta.$$

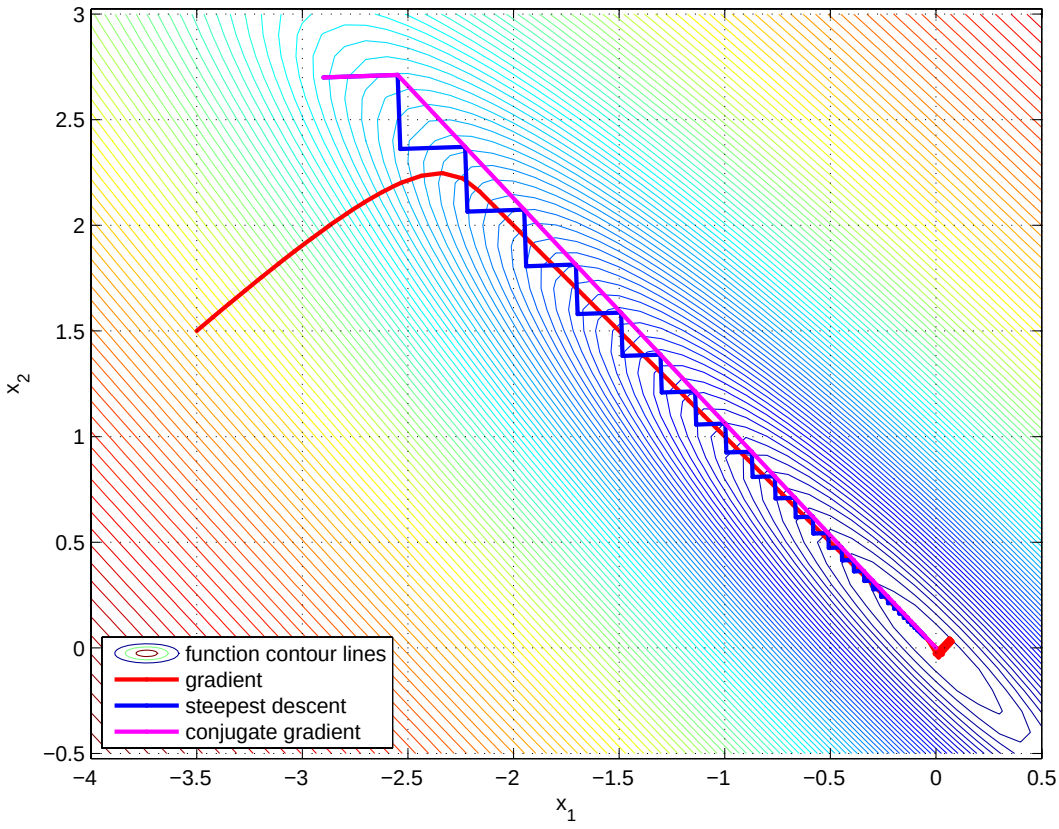
Following from the one-dimensional case, we try to select δ so as to cause $\partial f(x+\delta)/\partial\delta = 0$. This gives

$$\begin{aligned} -\nabla f(x) &= \delta^T \nabla^2 f(x) \longrightarrow \\ \delta &= -[\nabla^2 f(x)]^{-1} \nabla f(x). \end{aligned}$$

In words, Newton's method takes the first and second-derivative information and tries to come up with a specific solution in one step. The algorithm has extremely good convergence properties and is recommended when second derivatives are available.

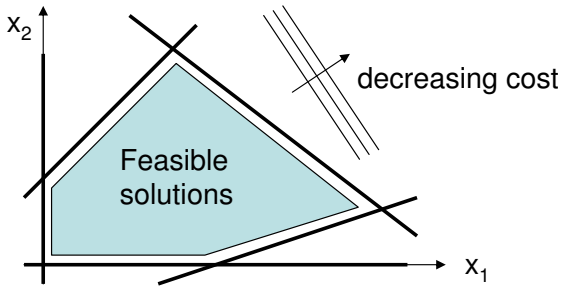
It is important to understand that both the conjugate gradient method and Newton's second-order method get the exact answer in two (conjugate gradient) or one (Newton) tries, when in fact the function is quadratic. Thinking of computational cost, the conjugate gradient algorithm has to have the derivative vector at two points, whereas Newton has to have the gradient plus the Hessian matrix, at the starting point. If these derivatives have to be created numerically and with accuracy, it is clear that the Hessian could be quite expensive. For this reason, the conjugate gradient method may be preferred.

For non-quadratic forms, these algorithms will both do the best they can to approximate and it will take some additional trials. An elegant combination of gradient and Newton methods is found in the Levenberg-Marquardt algorithm.



7.3 Linear Programming

We now consider the case that the cost is a linear function of n parameters. There is clearly no solution unless the parameter space is constrained, and indeed the solution is guaranteed to be on the boundary. The situation is well illustrated in two dimensions ($n = 2$), an example of which is shown below. Here, five linear inequality boundaries are shown; no x are allowed outside of the feasible region. In the general case, both equality and inequality constraints may be present.



The nature of the problem - all linear, and comprising inequality and possibly equality constraints - admits special and powerful algorithms that are effective even in very high dimensions, e.g., thousands. In lower dimensions, we can appeal to intuition gained from the figure to construct a simpler method for small systems, say up to ten unknowns.

Foremost, it is clear from the figure that the solution has to lie on one of the boundaries. The solution in fact lies at a *vertex* of n hypersurfaces of dimension $n - 1$. Such a hypersurface is a line in two dimensional space, a plane in three-space, and so on. We will say that a line in three-space is an intersection of two planes, and hence is equivalent to two hypersurfaces of dimension two. It can be verified that a hypersurface of dimension $n - 1$ is defined with one equation.

If there are no equality constraints, then these n hypersurfaces forming the solution vertex are a subset of the I inequality constraints. We will generally have $I > n$. If there are also $E < n$ equality constraints, then the solution lies at the intersection of these E equality hypersurfaces and $n - E$ other hypersurfaces taken from the I inequality constraints. Of course, if $E = n$, then we have only a linear system to solve. Thus we have a combinatorics problem; consider the case of inequalities only, and then the mixed case.

- I **inequalities, no equalities.** n of the inequalities will define the solution vertex. The number of combinations of n constraint equations among I choices is $I!/(I-n)!n!$. Algorithm: For each combination (indexed k, say) in turn, solve the linear system of equations to find a solution x_k. Check that the solution does not violate any of the other $I - n$ inequality constraints. Of all the solutions that are feasible (that is, they do not violate any constraints), pick the best one - it is optimal.

- I **inequalities,** E **equalities.** The solution involves all the equalities, and $n - E$ inequality constraints. The number of combinations of $n - E$ constraint equations

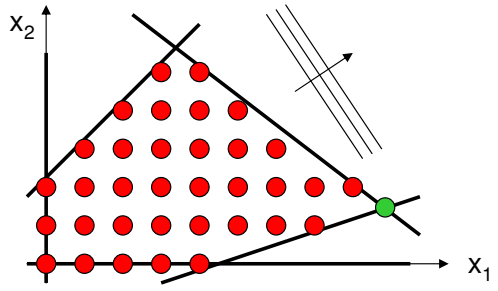
among I choices is $I!/(I-n+E)!(n-E)!$. Algorithm: For each combination (indexed with k) in turn, solve the linear set of equations, to give a candidate solution x_k. Check that the solution does not violate any of the remaining $I-n+E$ inequality constraints. Of all the feasible solutions, pick the best one - it is optimal.

The above rough recipe assumes that none of the hypersurfaces are parallel; parallel constraints will not intersect and no solution exists to the linear set of equations. Luckily such cases can be detected easily (e.g., by checking for singularity of the matrix), and classified as infeasible. In the above figure, $I = 5$, and $n = 2$. Hence there are $5!/(5-2)!2! = 10$ combinations to try: AB, AC, AD, AE, BC, BD, BE, CD, CE, DE. Only five are evident, because some of the intersections are outside of the area indicated (Can you find them all?).

The linear programming approach is extremely valuable in many areas of policy and finance, where costs scale linearly with quantity, and inequalities are commonplace. There are also a great many engineering applications, because of the prevalence of linear analysis.

7.4 Integer Linear Programming

Sometimes the constraint and cost functions are continuous, but only integer solutions are allowed. Such is the case in commodities markets, where it is expected that one will deal in tons of butter, whole automobiles, and so on. The image below shows integer solutions within a feasible domain defined by continuous function inequalities. Note that the requirement of an integer solution makes it far less obvious how to select the optimum point; it is no longer a vertex.

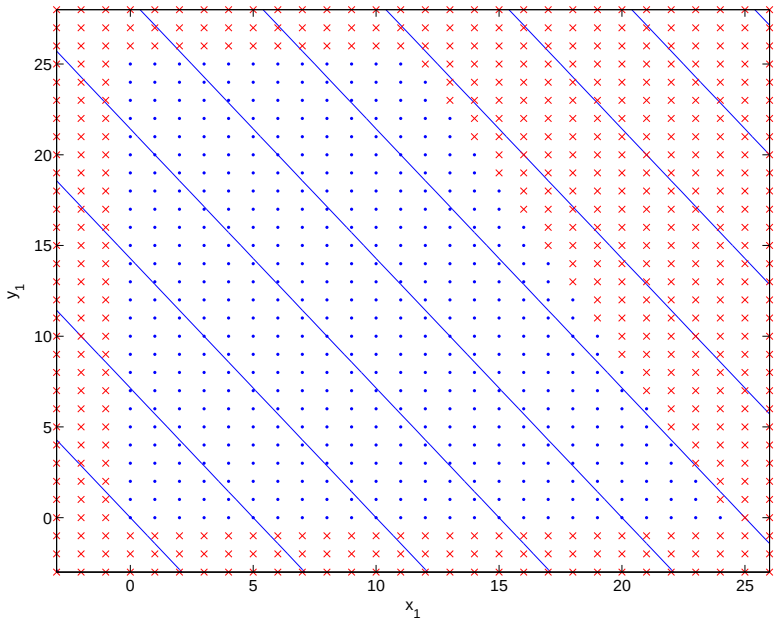


The branch-and-bound method comes to the rescue. In words, what we will do is successively solve continuous linear programming problems, but while imposing new inequality constraints that force the elements into taking integer values.

The method uses two major concepts. The first has to do with bounds and is quite intuitive. Suppose that in a solution domain X_1, the cost has a known *upper bound* $\bar{f}_1$, and that in a different domain X_2, the cost has a known *lower bound* $\underline{f}_2$. Suppose further that $\bar{f}_1 < \underline{f}_2$. If it is desired to minimize the function, then such a comparison clearly suggests we need spend no more time working in X_2. The second concept is that of a branching tree, and an

example is the best way to proceed here. We try to maximize[1]

$$\begin{aligned} J &= 1000x_1 + 700x_2, \text{ subject to} \\ 100x_1 + 50x_2 &\leq 2425 \text{ and} \\ x_2 &\leq 25.5, \\ &\text{with both } x_1, x_2 \text{ positive integers.} \end{aligned}$$



Hence we have four inequality constraints; the problem is not dissimilar to what is shown in the above figure. First, we solve the continuous linear programming problem, finding

$$A : J = 29350 : x_1 = 11.5, x_2 = 25.5.$$

Clearly, because neither of the solution elements is an integer, this solution is not valid. But it does give us a starting point in branch and bound: Branch this solution into two, where we consider the integers x_2 closest to 25.5:

$$\begin{aligned} B : J(x_2 \leq 25) &= 29250 : x_1 = 11.75, x_2 = 25 \text{ and} \\ C : J(x_2 \geq 26) &= X, \end{aligned}$$

where the X indicates we have violated one of our original constraints. So there is nothing more to consider along the lines of C. But we pursue B because it still has non-integer solutions, branching x_1 into

$$\begin{aligned} D : J(x_1 \leq 11, x_2 \leq 25) &= 28500 : x_1 = 11, x_2 = 25 \text{ and} \\ E : J(x_1 \geq 12, x_2 \leq 25) &= 29150 : x_1 = 12, x_2 = 24.5. \end{aligned}$$

[1]This problem is from G. Sierksma, Linear and integer programming, Marcel Dekker, New York, 1996.

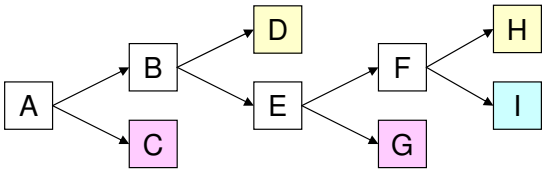
D does not need to be pursued any further, since it is has integer solutions; we store D as a possible optimum. Expanding E in x_2, we get

$$\begin{aligned} F : J(x_1 \geq 12, x_2 \leq 24) &= 29050 : x_1 = 12.25, x_2 = 24, \text{ and} \\ G : J(x_1 \geq 12, x_2 \geq 25) &= X. \end{aligned}$$

G is infeasible because it violates one of our original inequality constraints, so this branch dies. F has non-integer solutions so we branch in x_1:

$$\begin{aligned} H : J(x_1 \leq 12, x_2 \leq 24) &= 28800 : x_1 = 12, x_2 = 24 \text{ and} \\ I : J(x_1 \geq 13, x_2 \leq 24) &= 28750 : x_1 = 13, x_2 = 22.5. \end{aligned}$$

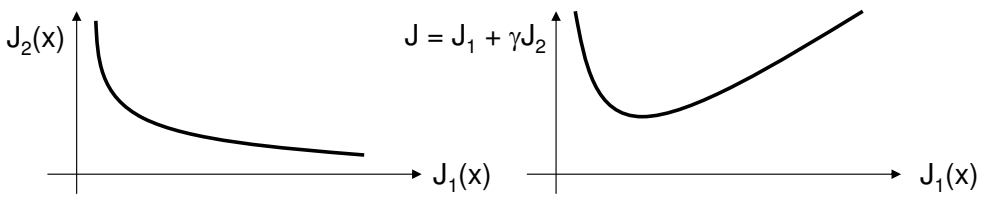
Now I is a non-integer solution, but even so it is not as good as H, which does have integer solution; so there is nothing more to pursue from I. H is better than D, the other available integer solution - so it is the optimal.



There exist many commercial programs for branch-and-bound solutions of integer linear programming problems. We implicitly used the upper-vs.-lower bound concept in terminating at I: if a non-integer solution is dominated by any integer solution, no branches from it can do better either.

7.5 Min-Max Optimization for Discrete Choices

A common dilemma in optimization is the existence of multiple objective functions. For example, in buying a motor, we have power rating, weight, durability and so on to consider. Even if it were a custom job - in which case the variables can be continuously varied - the fact of many objectives makes it messy. Clean optimization problems minimize one function, but if we make a meta-function out of many simpler functions (e.g., a weighted sum), we invariably find that the solution is quite sensitive to how we constructed the meta-function. This is not as it should be! The figure below shows on the left a typical tradeoff of objectives - one is improved at the expense of another. Both are functions of the underlying parameters. Constructing a meta-objective on the right, there is a indeed a minimum (plotted against $J_1(x)$), but its location depends directly on γ.



A very nice resolution to this problem is the min-max method. What we look for is the candidate solution with the smallest normalized deviation from the peak performance across objectives. Here is an example that explains. Four candidates are to be assessed according to three performance metrics; higher is better. They have raw scores as follows:

	Metric I	Metric II	Metric III
Candidate A	3.5	9	80
Candidate B	2.6	10	90
Candidate C	4.0	8.5	65
Candidate D	3.2	7.5	86

Note that each metric is evidently given on different scales. Metric I is perhaps taken out of five, Metric II is out of ten perhaps, and Metric III could be out of one hundred. We make four basic calculations:

- Calculate the range (max minus the min) for each metric: we get [1.4, 2.5, 35].
- Pick out the maximum metric in each metric: we have [4.0, 10, 90].
- Finally, replace the entries in the original table with the normalized deviation from the best:

	Metric I	Metric II	Metric III
Candidate A	(4.0-3.5)/1.4 = 0.36	(10-9)/2.5 = 0.4	(90-80)/35 = 0.29
Candidate B	(4.0-2.6)/1.4 = 1	(10-10)/2.5 = 0	(90-90)/35 = 0
Candidate C	(4.0-4.0)/1.4 = 0	(10-8.5)/2.5 = 0.6	(90-65)/35 = 1
Candidate D	(4.0-3.2)/1.4 = 0.57	(10-7.5)/2.5 = 1	(90-86)/35 = 0.11

- For each candidate, select the worst (highest) deviation: we get [0.4, 1, 1, 1].

The candidate with the lowest worst (min of the max!) deviation is our choice: Candidate A.

The min-max criterion can break a log-jam in the case of multiple objectives, but of course it is not without pitfalls. For one thing, are the metrics all equally important? If not, would a weighted sum of the deviations add any insight? We also notice that the min-max will throw out a candidate who scores at the bottom of the pack in any metric; this may or may not be perceived as fair. In broader terms, such decision-making can have fascinating social aspects.

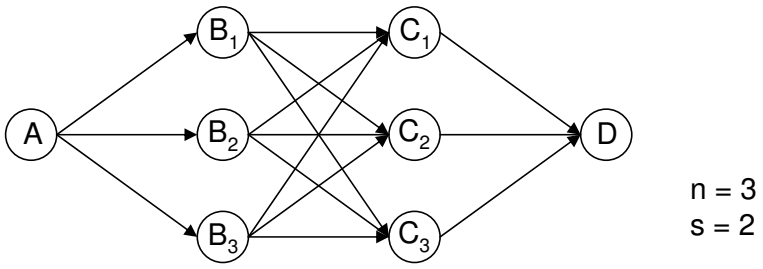
7.6 Dynamic Programming

We introduce a very powerful approach to solving a wide array of complicated optimization problems, especially those where the space of unknowns is very high, e.g., it is a trajectory

itself, or a complex sequence of actions, that is to be optimized. Only an introductory description here is given, focussing on *shortest-path* problems. A great many procedure and planning applications can be cast as shortest-path problems.

We begin with the essential concept. Suppose that we are driving from Point A to Point C, and we ask what is the shortest path in miles. If A and C represent Los Angeles and Boston, for example, there are *many* paths to choose from! Assume that one way or another we have found the best path, and that a Point B lies along this path, say Las Vegas. Let X be an arbitrary point east of Las Vegas. If we were to now solve a new optimization problem for getting only from Las Vegas to Boston, this same arbitrary point X would be along the new optimal path as well.

The point is a subtle one: the optimization problem from Las Vegas to Boston is easier than that from Los Angeles to Boston, and the idea is to use this property *backwards* through time to evolve the optimal path, beginning in Boston.



Example: Nodal Travel. We now add some structure to the above experiment. Consider now traveling from point A (Los Angeles) to Point D (Boston). Suppose there are only three places to cross the Rocky Mountains, B_1, B_2, B_3, and three places to cross the Mississippi River, C_1, C_2, C_3. By way of notation, we say that the path from A to B_1 is AB_1. Suppose that all of the paths (and distances) from A to the B-nodes are known, as are those from the B-nodes to the C-nodes, and the C-nodes to the terminal point D. There are nine unique paths from A to D.

A brute-force approach sums up the total distance for all the possible paths, and picks the shortest one. In terms of computations, we could summarize that this method requires nine additions of three numbers, equivalent to eighteen additions of two numbers. The *comparison* of numbers is relatively cheap.

The dynamic programming approach has two steps. First, from each B-node, pick the best path to D. There are three possible paths from B_1 to D, for example, and nine paths total from the B-level to D. Store the best paths as $B_1D|_{opt}, B_2D|_{opt}, B_3D|_{opt}$. This operation involves nine additions of two numbers. Second, compute the distance for each of the possible paths from A to D, *constrained to the optimal paths from the B-nodes onward*: $AB_1 + B_1D|_{opt}$, $AB_2 + B_2D|_{opt}$, or $AB_3 + B_3D|_{opt}$. The combined path with the shortest distance is the total solution; this second step involves three sums of two numbers, and the total optimization is done in twelve additions of two numbers.

Needless to say, this example gives only a mild advantage to the dynamic programming approach over brute force. The gap widens vastly, however, as one increases the dimensions of the solution space. In general, if there are s layers of nodes (e.g., rivers or mountain ranges), and each has width n (e.g., n river crossing points), the brute force approach will take (sn^s) additions, while the dynamic programming procedure involves only $(n^2(s-1)+n)$ additions. In the case of $n = 5$, $s = 5$, brute force requires 15625 additions; dynamic programming needs only 105!

7.7 Solving Dynamic Programming on a Computer

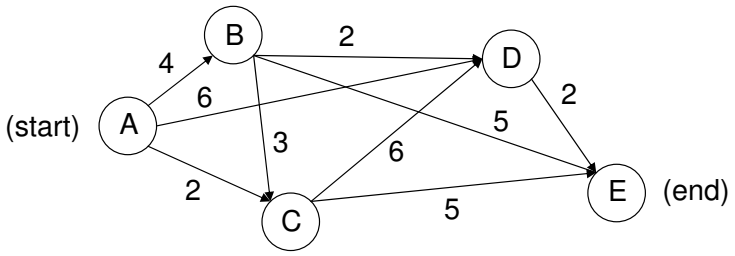
Certainly the above algorithm can be implemented as written - moving backward from the end to the beginning, keeping track at each stage only of the optimal trajectories from that stage forward. This decision will involve some careful recording and indexing. A very simple algorithm called value iteration may be more accessible on your first try. As we will show in an example below, value iteration also allows us to consider problems where distinct stages are not clear.

It goes this way:

1. Index all of the possible configurations, or nodes, of the system (cities).
2. With each configuration, create a list of where we can go to from that node - probably this is a list of indices (cities that are plausibly part of an optimal path). The starting node (Los Angeles) is pointed to by no other nodes, whereas the end node (Boston) points to none.
3. For each of these simple paths defined from node to node, assign a cost of transition (simple driving miles between the cities).
4. Now assign to each of these configurations an *initial guess* for what is the cost from this node to the end state (optimum total miles from each city to Boston). Clearly the costs-to-go for nodes that point to the terminal node are well-known, but none of the others are.
5. Sweep through all the configurations (except the terminal one), picking the best path out, based on the local path and the estimated cost at the next node. At each node, we have only to keep track of the best next node index, and the new estimated cost-to-go.
6. Repeat to convergence!

This algorithm can be shown to converge always, and has a number of variants and enhancements. An example makes things clearer:

Node	Points to Nodes	With Costs	Initial Estimate of Cost to Go
A (initial)	B,C,D	4,2,6	10
B	C,D,E	3,2,5	10
C	D,E	6,5	10
D	E	2	2 (known)
E	(terminal)	NA	NA



And here is the evolution of the value iteration:

	A	B	C	D
iteration	cost-to-go	cost-to-go	cost-to-go	cost-to-go
0	NA	10	10	2(E)
1	min(14,12,8) = 8(D,E)	min(13,4,5) = 4(D,E)	min(8,5) = 5(E)	2(E)
2	min(8,7,8) = 7(C,E)	4(D,E)	5(E)	2(E)

We can end safely after the second iteration because the path from A involves C, which cannot change from its value after the first iteration, because it connects all the way through to E.

2.017J Design of Electromechanical Robotic Systems
Fall 2009