# Local Search

CSCI 2400 "Part 8"
Instructor: David Byrd

*Disclaimer:* Lecture notes can't and won't cover everything I say in class. You should attend class each day and use these for review or reinforcement.

# 11 Local Search

Earlier in the course, we studied a class of problems and algorithms we called Search, and subdivided into Uninformed Search (DFS, BFS, UCS) and Informed Search (GreedyBestFS, BestFS, A*). Those algorithms worked well for a certain class of problem: fully observable, with discrete states, and of "reasonable" size. They can give us a complete path (list of actions) as a solution, *but* they can also be quite slow (potentially exhaustive) and use a lot of memory (tracking many candidate paths).

We explored how reformulation as a Constraint Satisfaction Problem can help us find solutions faster *if* the state is factored. Remember, a *factored* state consists of separate variables that we can inspect, so instead of just $state = 42$, we might have $state =$ {V: blue, SA: green, T: red} as in the Australia map-coloring problem. But CSP can still be very slow if the state space is large enough.

Worse, what if we have an *infinite* state space? That might seem unreasonable, but it happens all the time – any problem with continuous inputs, or in which the factored state contains real numbers, has an infinite state space.

Finally, what if we have a severe memory constraint, such as running the algorithm on a smart watch or other small wearable device?

## 11.1 A New Class of Problem

Let's assume we have such a problem that cannot be reasonably solved with Search or CSP techniques. There is an infinite (or at least *very* large) state space *and* we have only a small, fixed amount of memory.

### 11.1.1 Introducing Local Search

It's pretty clear we can no longer store all possible "paths" we explore. In fact we cannot rely on keeping *any* sort of history. When we do find a goal state, then, we do not know the path that was followed. Given that we cannot evaluate all the solutions, we cannot guarantee to have found an optimal solution. We now have to consider a **goal** to be any state from which we can no longer find an improvement to our situation.

So what can we do? We can identify a goal state as a *potential (but not necessarily optimal) solution*.

This result is less certain, and perhaps less useful, than our precise and guaranteed solutions to Search and CSP, but it allows us to find approximate solutions to NP-Complete problems (traveling salesman, vertex cover, boolean satisfiability) and general optimization problems (given some parameters and a function, find the parameter values that make the function result as large or small as possible).

Why can't we just use calculus for the optimization problems? We know how to optimize a function! 1) Find the points at which the first derivative is zero. These are local optima. 2) Evaluate the second derivative at those points to check if they are minima or maxima.

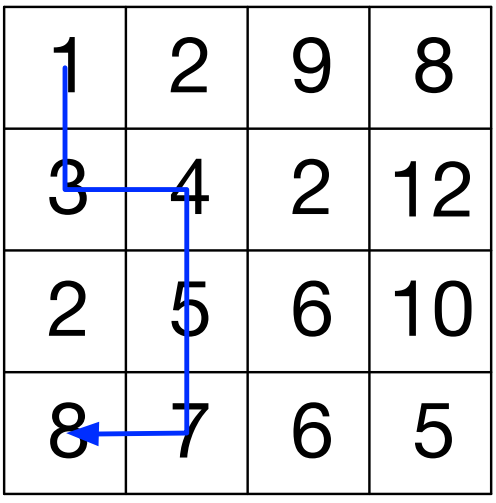
You *could* do this *in theory*, but the type of problems we approach in AI tend to have *millions* of parameters. That makes exactly computing those optima unrealistic in terms of time and memory, and so we need approximation methods instead.

The field of **local search** techniques solves this class of problem. With no memory, a typical local search algorithm selects the best neighboring state to its current state (by some metric), repeating until no neighboring state is better. It then returns the current state as the solution.

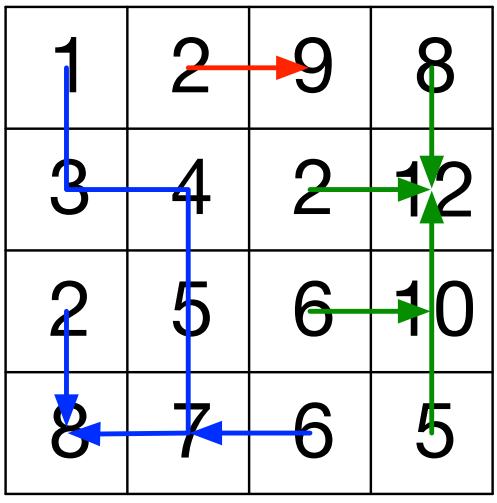
### 11.1.2 The Issue of Local Optima

You have probably realized it by now, but the major problem we will face in local search is that of **local optima**. That is, states we may reach from which no further improvement is possible using local search, but which are *not* equal in value (goodness) to the best possible solution, or the **global optimum**.

A simple example: we wish to find the largest number in an infinitely (or at least *very*) large grid. I have yet to figure out how to draw an infinite grid in LaTeX, so you will have to use your imagination while considering this 4x4 grid. Local search might start in the top-left corner, and hope to find the highest number by moving up, down, left, or right to the highest number it can "see" in an adjacent space.




(a) Path taken from top left corner.




(b) All possible paths to solutions.

The example search terminates at the lower-left square containing the number 8, from which no further improvement can be made, because all neighboring squares have lower values. Unfortunately, the problem we were trying to solve was "find the highest number", and the 8 in the lower-left corner is a sub-optimal solution from which our algorithm cannot escape. That is, it is a *local optimum*.

There are three local optima in this problem: the bottom-left 8, the top row 9, and the right column 12. They are local optima because it is not possible to improve the algorithm's situation once we reach those states – they are better than all their neighbors. One of the local optima also happens to be the *global optimum*, because it is the overall best state the algorithm could have found.

## 11.2 Hill Climbing

The first local search algorithm we consider in this class is **hill climbing**. It is also called *greedy local search* for reasons that will become obvious.

Visualize the state space being explored as a two-dimensional landscape in which you must walk along the ground. You like high places (the view is nice!), so you consider elevation as the measure of "goodness" of a location. Your goal is to reach the highest elevation you can find. The algorithm can then be described as: "Climb the steepest hill nearby." This intuitive image is the inspiration for the name.

Hill Climbing is a simple technique that requires just three things: a description of the problem state, a function to identify possible successor states, and a cost function or performance metric that assigns a given state a numeric value to be minimized or maximized. The algorithm itself is trivial: 1) initialize with a random current state, 2) generate all possible successor states and their evaluation metrics, 3) move the current state to the successor with the lowest or highest evaluation metric (problem dependent), 4) repeat 2 and 3 until no successor state has a better value than the current state, 5) return the current state as the solution.

Here is a small but complete example, using the classic 4-Queens problem. We wish to place four queens on a standard chessboard (here a 4x4 board) such that no two queens are positioned to attack each other. Recall that in chess, queens attack along rows, columns, or diagonals at any distance.

We assume an arbitrary starting state. For simplicity, we will restrict ourselves to one queen per column, and will always have all four queens on the board. These restrictions eliminate no solutions, because neither a sub-four-queen configuration nor one with two queens in a column can be a proper solution to the problem posed. Here is the starting state:

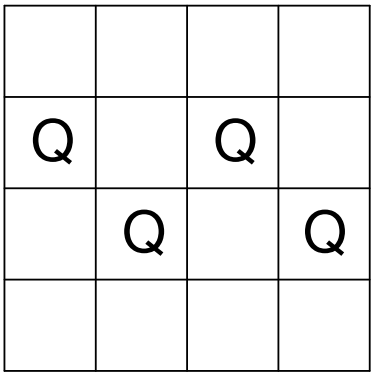



Figure 2: An arbitrary starting configuration for the 4-Queens problem.

This state description can be represented as a simple 1-D vector due to the constraint of placing only one queen per column. Each position in the vector corresponds to a column, and the number in that position is the row occupied by the queen. For example, the above starting state could be represented as $< 2, 3, 2, 3 >$, presuming we label the top row as row one.

The second requirement for Hill Climbing is a successor function. Given a current state, it must generate all possible successor states. Again for simplicity, we will move only one queen at a time. Thus our successor states will have exactly one value different from the predecessor: a queen that has been moved to a new row. With 12 empty squares on the board, there are 12 successors for any given state. Two of the successor states of the starting state are: $< 2, 1, 2, 3 >$ and $< 2, 3, 2, 4 >$. Note that $< 2, 1, 2, 4 >$ would *not* be a successor state, because it requires moving two queens from the starting configuration.

The third and final requirement is a performance metric. Here we will construct a cost that we wish to minimize: the number of queen pairs that are positioned to attack each other. In 4-queens, the maximum value of this cost function is 6, because there are six pairs of queens to consider. Our ideal solution, with no queens attacking, occurs at the minimum value of 0. The cost of the starting state is 5, because there is only one queen pair not attacking each other (the queens in column one and four).

In the diagram below, we list the cost of each successor state. The costs listed in column $i$ represent the state in which the column $i$ queen has been moved to that position.

| | | | |
|---|---|---|---|
| 3 | 5 | 2 | 5 |
| Q | 4 | Q | 5 |
| 5 | Q | 4 | Q |
| 5 | 2 | 5 | 3 |

Figure 3: Successor state costs for the starting position.

For the 4-Queens problem, we could of course perform a global search or solve a CSP to find an optimal solution where $cost = 0$. However, we would like an approach that can scale to millions of queens, so we will use Hill Climbing. In this case, we simply select the successor state with the lowest cost, breaking ties randomly. This will be one of the $cost = 2$ successors. Let us arbitrarily choose to move the column two queen to the bottom row.

Here is the new state configuration, with costs listed for *its* successor states:

| | | | |
|---|---|---|---|
| 1 | 5 | 0 | 1 |
| Q | 4 | Q | 2 |
| 3 | 5 | 2 | Q |
| 3 | Q | 3 | 1 |

Figure 4: Successor state costs after the first move.

We can now see that our second move will provide an optimal solution: the queen in the third column moving to the first row creates a $cost = 0$ state, and that will be our output solution $< 2, 4, 1, 3 >$.

Notice the significant benefit of a local search algorithm: at each step we remember *nothing* from the previous step(s) and we *never* need to see the entire state space. We only require the current state, the immediate successor states, and some evaluation metric to optimize. The algorithm cannot enter the same state twice because it only moves to a successor with a lower cost, so even lacking memory, there is no risk of an infinite loop.

However, the *local optima problem* we mentioned in the introduction is alive and well. Consider a state in which we have not been so lucky:

| | | | |
|---|---|---|---|
| 4 | Q | 2 | 2 |
| Q | 4 | 3 | 3 |
| 2 | 3 | Q | 3 |
| 2 | 3 | 3 | Q |

Figure 5: Stuck in a local minimum.

In the diagram above, the current state has $cost = 2$ because two pairs of queens are attacking. There is no successor state that improves the cost, and so the algorithm terminates in a local optimum with this solution. This is not what we wanted, but it was the best Hill Climbing could achieve on this run.

You may notice that the state shown is actually a *plateau*, because we could move "sideways" to a different $cost = 2$ solution, and from there perhaps we would find a way to proceed. It *might* work, however as soon as we allow such "sideways" moves, we create the risk of an infinite loop, because our algorithm has no memory of previously-visited states!

There are a few common techniques to reduce the problem of sub-optimal solutions in Hill Climbing: allow limited memory use (for example a fixed-size open list of states to visit), allow a limited number of sideways movements to escape plateaus, or allow a certain number of random restarts in an attempt to find a better solution. The last solution particularly works well for the N-Queens problem. In 8-Queens, Hill Climbing finds a sub-optimal solution 86% of the time, but on average requires only 6 random restarts and 22 total moves to find an optimal $cost = 0$ solution.

## 11.3 Simulated Annealing

Hill Climbing works well for problems without too many local optima. When there are *very many* local optima, we may have greater success with a different local search algorithm. Simulated Annealing is one such algorithm that takes its name from metallurgy (annealing is the process of alternately heating and cooling metal to strengthen it).

The intuitive explanation for Simulated Annealing is this: flip your Hill Climbing surface over and consider *lower* elevation to be better. If you drop a ball somewhere and let it roll (downward with gravity), that is equivalent to Hill Climbing, and the ball will likely come to rest in a local minimum.

What if we grab this landscape and *shake* it occasionally? We might knock the ball out of a local minimum and allow it to continue rolling to a better (lower) position. Ideally, we would shake just hard enough to knock the ball out of a local minimum, but not out of the global minimum. We can't exactly do that, but we can simulate it by shaking hard initially, and less as time passes, and *that* is the idea of Simulated Annealing.

We will require a "temperature" $T$ (again, nomenclature from metallurgy), which corresponds to how hard we are "shaking" the landscape right now, possibly allowing the marble

to briefly roll uphill. We also require a neighborhood: all states (locations) within some distance of the current state, which means we need some concept of "distance" for our problem as well.

Here is the basic Simulated Annealing algorithm:

```
T = some high starting temperature
current = a random state

loop:
    reduce T   # according to some schedule
    next = a random neighbor of current state

    delta = cost(next) - cost(current)
    if delta < 0 then
        current = next   # good, make the move
        continue         # start next loop

    if rand() < e^(-delta/T) then
        current = next   # bad move, but do it anyway

until T == 0
```

We assume our `rand()` function returns a real number over the interval $[0, 1)$. The expression $e^{-delta/T}$ can thus be thought of as a *probability* that we move to a neighboring state despite it being worse than the current state. This probability decreases the "worse" the neighbor is. It also decreases as $T$ decreases over time. When $T = 0$, the landscape no longer "shakes" and the marble comes to rest in a final location: the solution. If we reduce $T$ quickly, the algorithm runs quickly but can still come to rest in a local optimum. If we reduce $T$ slowly enough, we can approach a guaranteed globally-optimal solution, but the algorithm may run for quite a long time. This trade-off of runtime versus solution quality is a common feature of local search techniques.

In general, Simulated Annealing will run more slowly than Hill Climbing, but it is better suited for problems with many local optima. It is also better suited for problems with large neighborhoods (many successor states for each current state), because Hill Climbing must evaluate *all* neighbors at each step, while Simulated Annealing simply selects *one* neighbor at random.

## 11.4 Genetic Algorithms

The third class of local search algorithms we consider are Genetic Algorithms (GA) inspired by the idea of "survival of the fittest". As usual, we will require an objective function to optimize, here called a *fitness function*. The current state is now a set of candidate solutions to the problem, some better than others. A candidate solution must be represented as a string (as if it were DNA).

Our earlier N-Queens example already expressed potential solutions as a vector, so it is easy to translate to a string instead. For example, `47167248` is a candidate solution to the

8-Queens problem, with the first-column queen in row 4, the second-column queen in row 7, and so forth.

In GA, the fitness function is typically maximized, so we might reformulate our cost function to: $f(s) = N - A$, where $f(s)$ is the fitness of candidate solution $s$, $N$ is the pairwise count of queens on the board, and $A$ is the pairwise count of attacking queens. For example, with eight queens on the board, there are 28 distinct pairs of queens. If none of them are attacking each other, $f(s) = 28$, an optimal solution. If twelve pairs are in attack position, $f(s) = 28 - 12 = 16$. Many other equally-reasonable fitness functions are possible.

The basic structure of GA is this:

```
Start with n randomly-generated ancestor solution candidates

loop:
    Evaluate fitness of population
    Select k fittest solutions to "breed"

    Generate m >= k "children":
        cross-over:
            select two parents
            select a character index at which to "cut"
            create a child by splicing the pieces together
        mutate: randomly change one character in the child

    Cull the population:
        evaluate all n+m parents and children
        keep only the best n

until the fitness stops improving
```

Note that there are many variations on the GA theme. Generally you will select parents with likelihood based on fitness and choose the "cut" point to take more DNA (characters) from the higher-fitness parent.

For example, if we select parent solution candidates `47167248` and `73281121`, we may calculate their fitness as 14 and 11 respectively. (Warning: I did not calculate the actual fitness.) In that case, the "cut point" for DNA splicing could be after index `5`, favoring the higher-fitness parent.

Then our DNA strands from the first parent are `47167` and `248`, and our strands from the second parent are `73281` and `121`. We take the first piece of parent one and the second piece of parent two: `47167121`. Then we randomly change (mutate) a single character, perhaps producing `47165121` as the child solution candidate.

In this way, we maintain a *population* of solution candidates. Over time, we mix and mutate the best candidates, while eliminating low-fitness candidates. Thus the mean fitness of the population should increase over time. When the rate of increase becomes too low to be worth continuing, we stop and return our highest-fitness candidate as the solution.

## 11.5 Gradient Descent

The final class of local search algorithms discussed in our course are varieties of Gradient Descent, primarily used to optimize *continuous* objective functions. As the name implies, gradient descent operates by calculating the gradient (or first derivative) of the function output with respect to its inputs. If you recall basic calculus, a partial derivative will answer the question: "if we change this variable by a small amount in some direction, what happens to this other variable?" In our case, we want to know how changes to the input parameters will affect the output. We can then use that information to minimize the error of our output predictions.

It is helpful to have an example problem on which to describe gradient descent, so first we will briefly describe linear regression as a learning algorithm.

### 11.5.1 Multivariate Linear Regression

Multivariate linear regression can be thought of as a parametric, generative, batch regression learner. (Refer to our introduction to machine learning lecture if you do not remember the significance of those terms.) In the multivariate case, we are trying to find parameters $m_0 \dots m_n$ to best fit a training data set with an equation that looks like this:

$$y = m_0 + m_1 x_1 + m_2 x_2 + \cdots + m_n x_n$$

Notice that each input feature $x_1 \dots x_n$ gets its own coefficient parameter that can be adjusted during training. We additionally supply one parameter that is not tied to any particular $x$ feature. This function describes some hyperplane ("flat" ($n$-1)-dimensional surface in an $n$-dimensional space). Without the extra $m_0$ parameter, the hyperplane would always have to pass through the origin (think what happens when all $x_i$ are zero), eliminating many potential options because we can only rotate the hyperplane about the origin, not translate (slide) it. With the $m_0$ parameter, we can translate it as well.

Given $(X, y)$ training tuples, where $X$ is a feature vector $< x_1, x_2, \dots, x_n >$, the linear regression learner therefore learns a best fit hyperplane in $n$-dimensional space. By "best fit", we mean that it minimizes some loss function relating $y$ and $y_{pred}$, the desired output and our prediction of it. We then query the trained learner with new $X$ feature vectors to obtain a prediction of the correct $y$ values, by simply plugging the $x_1 \dots x_n$ into our equation and reading off the $y$.

What does it learn? The values for $m_0 \dots m_n$ that produce the best fit hyperplane for this training data. *How* does it learn? In the linear case, there is a known, closed-form solution, but the general technique we would use is *gradient descent*. We will find the direction of the gradient from $y_{pred}$ to $y$ (the gradient of the loss function) and shift the $m$ parameters slightly in the direction that shrinks the loss, repeating this until the loss is very small.

### 11.5.2 Example of Gradient Descent

*Univariate* linear regression is identical to the multivariate case above, except we now have only a single $x$ feature, and so are trying to find the best fit line (1-D) in a two-dimensional space: $y = m_0 + m_1 x_1$, or in its more familiar form $y = mx + b$.

It is straightforward to describe a small Python class that implements a univariate linear regression learner:

```
class LinRegLearner:

    def train(self, trainX, trainY):
        self.m, self.b = favoritelib.linreg(trainX, trainY)

    def query(self, testX):
        return self.m * testX + self.b
```

Note that with an appropriate Python library like `numpy`, the method parameters can all be vectors or matrices, permitting all of the training data to be given at once, or a large number of queries to be computed at once.

If we have a favorite statistics library `favoritelib` that can do linear regression, wrapping a learning algorithm around it is trivial, as you can see. What if we want to write our own `linreg` function using gradient descent?

We must first select the specific loss function that we will try to minimize. Here we will use the sum of squared errors (SSE). It is popular first because squaring eliminates any sign difference, so opposing errors will not offset each other, and second because large errors are penalized much more heavily than small errors.

It is customary to label the objective function for optimization as $J$, so our SSE loss function can be:

$$J = \frac{1}{2}(y_{pred} - y)^2$$

We need the gradient of the loss function to adjust our parameters, so we compute the first partial derivative with respect to each input parameter. Remember that $y_{pred} = mx + b$, and our training tuples are $(x, y)$, so $m$ and $b$ are the parameters we will try to learn.

$$\begin{aligned}
\frac{dJ}{dm} &= \frac{1}{2} \cdot \frac{d}{du}\left[u^2\right]\frac{du}{dm} \\
&= \frac{1}{2} \cdot 2u \cdot \frac{d}{dm}\left[y_{pred} - y\right] \\
&= u \cdot \frac{d}{dm}\left[mx + b - y\right] \\
&= ux \\
&= x(y_{pred} - y)
\end{aligned}$$

We have used substitution to simplify calculation of the derivative, with $u = y_{pred} - y$. Then $\frac{dJ}{dm} = \frac{dJ}{du}\frac{du}{dm}$ and the calculation proceeds as shown above. Calculating $\frac{dJ}{db}$ is identical except that $\frac{d}{db}\left[mx + b - y\right] = 1$, so the result is simply $\frac{dJ}{db} = (y_{pred} - y)$.

We now know how to alter $m$ and $b$ to produce the desired change in the loss function $J$ (raising or lowering it), and we can proceed with some simple code:

```
start with random m,b

loop:
    sse = 0
    m_new = m
    b_new = b

    for x,y in zip(trainX,trainY):
        ypred = m*x + b
        sse += (ypred - y) ** 2

        m_new -= (ypred - y) * x * learningRate
        b_new -= (ypred - y) * learningRate

    J = 0.5 * sse
    m = m_new
    b = b_new

until stopping condition (see below)
```

Each iteration of gradient descent, we calculate the "badness" of our estimates. We accumulate the SSE to calculate $J$, but also for each training point, we make a small change to $m$ and $b$ so the error will be smaller in the future. We do this by taking a "step" against the direction of increasing loss. (Instead of `-= (ypred-y)`, we could equally do `+= (y-ypred)`.) Note that we do not take a step in the full size of the gradient. We multiply it by a fractional learning rate over the interval $(0, 1)$. The learning rate parameter can be adjusted to take small enough steps that we do not overshoot the parameter values that produce minimum error (loss). In fact, the learning rate is often decreased on some schedule while the algorithm is running, to promote smooth convergence.

If properly implemented, $J$ should decrease every loop (if learning rate is small enough to avoid overshoot) and for a convex loss function is guaranteed to converge to a global optimum. It is likely to require infinite time to reach $J = 0$, however, and so we typically stop when $J < \epsilon$ or $\Delta J < \epsilon$. That is, we stop when the loss, or the change in loss from one iteration to the next, drops below some very small threshold.

The presented algorithm is batch gradient descent. We have all of our data, use it all each iteration, and if new data arrives later, we must retrain from the beginning. In **stochastic gradient descent**, we operate in an online manner, selecting data points one at a time and computing the parameter update for an entire iteration from just the single data point. This is much faster, and does not require full retraining when new data arrives, but it is *not* guaranteed to converge to a global optimum even for a convex loss function. (This is because the algorithm selects data points for training at random, so with unlucky choices it could "wander" the loss function or orbit around the optimum indefinitely. Learning rate decay can help shrink this orbit and thus solve the problem.)

One final note: the most common form of gradient descent used to train neural networks is **mini-batch** gradient descent. It lies between batch and stochastic, processing the largest

batch of training data that can fit in the GPU's memory (of the training computer) at one time. GPU training is preferable because these batch calculations can be performed in parallel.

*Acknowledgements and thanks to Professors Mark Riedl and Jim Rehg of the Georgia Tech School of Interactive Computing.*