Iterative Improvement

Miles Jones. (help from Neil Rhodes)

August 25, 2016

In iterative improvement, we start with a potential solution, and then try making a change to the solution that gives a better result. We continue until we can no longer improve the result.

There are a couple of things that are needed:

1. We need some sort of evaluation metric that describes how well we are doing. This lets us know what is a better vs. worse solution.

2. We need a way to change an existing solution.

There are iterative improvement algorithms which find optimal solutions, as well as those which are used as heuristics: an approach that will find a good, but not necessarily optimal solution.

0.1 Optimal Solutions

In this section, we'll look at how one can use iterative improvement to find optimal solutions to problems

0.1.1 Using Invariants

An invariant is something which doesn't change. Examples from physics include all the laws of conservation: momentum, angular momentum, spin, charge, and energy. In computer science, we use invariants to represent unchanging quantities or relationships. One way they are used is in loop invariants: a relationship that continues to hold in each iteration of a loop.

A *Monotonic Invariant* (or *Monovariant*) is a quantity that:

- changes only in a desired direction,

- has only a finite number of possible values,
- when it reaches the final value, the problem is solved.

As an example, if the problem is in walking to the door, a possible monovariant would be the current distance to the door. As one moves towards the door, that quantity decreases. When the quantity reqaches zero, the problem is solved. If movement is always at least some minimum amount, then the distance has at most a finite number of possible values. Thus, this is a valid monovariant.

Note that if the distance moved had no lower bound (for example, imaging moving 1/2 the distance, then 1/4, then 1/8, and so on), then the quantity would not be a valid monovariant since there wouldn't be a finite number of possible values.

0.1.2 Linear Programming

The most widely used iterative improvement technique is *Linear Programming*, a method of solving a system of linear inequalities maximizing an objective function.

Here's an example Linear Programming problem: [1]

> A calculator company produces a scientific calculator and a graphing calculator. Long-term projections indicate an expected demand of at least 100 scientific and 80 graphing calculators each day. Because of limitations on production capacity, no more than 200 scientific and 170 graphing calculators can be made daily. To satisfy a shipping contract, a total of at least 200 calculators much be shipped each day.
>
> If each scientific calculator sold results in a \$2 loss, but each graphing calculator produces a \$5 profit, how many of each type should be made daily to maximize net profits?

This can be written as a series of inequalities (letting x be the number of scientific calculators produced, and y be the number of graphing calculators produced):

[1] courtesy of http://www.purplemath.com/modules/linprog3.htm

$$\begin{aligned} 100 < x &< 200 \\ 80 < y &< 170 \\ y &> x + 200 \end{aligned}$$

where we're trying to maximize:

$$P = -2x + 5y$$

Linear programming is designed to solve such problems, but can handle problems with (currently) up to tens of thousand inequalities and variables.

0.2 Heuristics

Iterative improvement is also used in heuristic problem solving. In all these cases, imagine, that we have some measure of goodness (some objective we're trying to reach).

0.2.1 Hill Climbing

In hill climbing, we start with a potential solution (either random, or one that is thought to be close to optimal), and then search through the solution space by trying a single change to the current solution. if this change yields an improvement, the changed solution is kept. This continues until no further improvement can be made.

Hill climbing solutions can easily be stuck in a local maximum. This can be ameliorated by trying hill climbing multiple times, each time starting with a diferent random starting solution.

0.2.2 Simulated Annealing

Simulated annealing improves on hill climbing by avoiding getting stuck in a local maximum. This is done by adding a non-zero chance of choosing a solution that is worse than the current solution. This chance is reduced over time.

This is modeled after the annealing of metal: heating it, and then slowly cooling to remove disclocations and reduce internal stresses.

0.2.3 Genetic Algorithms

Genetic algorithms use the process of evolution to generate solutions.

A solution is represented by a bit string which we'll call a chromosome.

A population contains a population of solutions. At each step, selection occurs by measuring the *fitness* of each solution (fitness is our measure of goodness). A fraction of the best solutions are retained, while the others are discarded. The retained solutions are bred to produce solutions for the next cycle.

The breeding consists of taking two of the parent bitsprings and randomly choosing a crossover point. Bits before the crossover point come from parent 1, bits after the crossover point come from the corresponding section from parent 2. Then, with low but not zero probability, a mutation is chosen (a bit is flipped).

This genetic algorithm searches the sample space. The mutations allow not getting stuck in local optima.

1 Problems

1. Parliament Pacificiation[2]

   In a parliament, each member has at most three enemies. (We assume that enmity is always mutual.) True or false: one can always divide the parliament into two chambers in such a way that no parliamentarian has more than one enemy in his or her chamber?

   Solution: Create an arbitrary division of the parliament into two chambers. Let p (a monovariant) be the number of enemy pairs in the same chamber. If any parliamentarian has more than one enemy in his or her chamber, move that parliamentarian into the other chamber. This will reduce p by at least one (since there is at most one enemy parliamentarian in the new chamber, and there are at least two in the old chamber: we're either reducing p by two and adding one, or reducing p by three and adding zero). Keep going in this fashion until there is at most one enemy in his or her chamber for each parliamentarian. There can be no more than p iterations so this process eventually completes.

[2] *Algorithmic Puzzles*, Levitin, Problem 122

2 Homework

1. Heads Up[3]

   There are n coins in a line, heads and tails in random order. On each move, one can turn over any number of coins laying in succession. Design an algorithm to turn all the coins heads up in the minimum number of moves. How many moves are required in the worst case?

2. Candy Sharing[4]

   In a kindergarten, there are n children sitting in a circle facing their teacher in the center. Each child initially has an even number of candy pieces. When the teacher blows a whistle, each child simultaneously gives half of his or her candy pieces to the neighbor on the left. Any child who ends up with an odd number of pieces is given another piece by the teacher. THen the teacher blows her whistle again, unless all the children have the same number of candies, in which case the game stops. Can this game go on forever or will it eventually stop to let the children go on with their lives?

3 Advanced Problems

1. King Arthur's Round Table[5]

   King Arthur wants to seat $n > 2$ knights around his Round Table so that none of the knights is seated next to his enemy. Show how this can be done if the number of friends foreach knight is not smaller than $n/2$. You may assume that friendship and enmity are always mutual.

2. Hitting a Moving Target[6]

   A computer games has a shooter and a moving target. The shooter can hit any of $n > 1$ hiding spots located along a straight line in which the target can hide. The shooter can never see the target; all he knows is that the target moves to an adjacent hiding spot between every two

[3] *Algorithmic Puzzles*, Levitin, Problem 146
[4] *Algorithmic Puzzles*, Levitin, Problem 138
[5] *Algorithmic Puzzles*, Levitin, Problem 147
[6] *Algorithmic Puzzles*, Levitin, Problem 146

consecutive shots. Design an algorithm that guarantees hitting the target or prove that no such algorithm exists.