Final Project Technical Specification

Sam Kim[1], Nick Merrill[2], Crystal Stowell[3]
[1]samuelkim@college.harvard.edu
[2]merrill@college.harvard.edu
[3]cstowell@college.harvard.edu

Overview

Much of the world surrounding us, whether it be the physics of a computer chip or the economic trends of a complex society, can be represented by mathematical equations and models. The class of optimization problems is popular as most entities and individuals struggle to maximize utility. These often involve dozens of variables that have numerous constraints such that a brute-force approach is either infeasible or flat out impossible. Thus, optimization algorithms for complex, non-linear spaces must focus on approximating the solution, which can be a difficult task.

One such optimization problem is the nurse scheduling problem. This problem comprises the difficulties that many round-the-clock companies and organizations face - how to best spread their finite resources of employees - and is also reflective of more general situations in industry. Several constraints arise in all scheduling problems. Namely, no individual can work more than two back-to-back shifts. Also, most employees do not want to have more than a certain number of night shifts in a given week. There may also be personal preferences of vacation time and other factors. In order to maintain the functionality of the hospital while meeting the requests of the most employees possible, the employee schedule must be optimized.

To solve this optimization problem, we will be focusing on using the Cuckoo Search algorithm, which is a meta-heuristic optimization algorithm published in 2009. This algorithm attempts to find the optimal solution using a genetic algorithm approach, in which new solutions are randomly generated from the previous generation of solutions and the worse solutions are discarded. Several modifications exist, such as the use of Levy flights in place of the random walk, as well as multi-objectivity. Additionally, we will also focus on other more well-known and verified optimization algorithms such as the Particle Swarm Optimization (PSO), and compare the performance of this versus the Cuckoo Search.

Our goal for the project is to implement the cuckoo search, the PSO, and further modifications to optimize the nurses' schedules. The design will be constructed such that other problems and other optimization algorithms can be easily swapped in and out to allow for easily collecting performance data.

Feature List

Core features

- Optimize an objective function using Cuckoo Search.
- Provide a set of optimized solutions (See next section for implementation details). (Allows for analysis of solution sets with multiple peaks)
- Interface the problem to be solved with the back-end optimization algorithm.

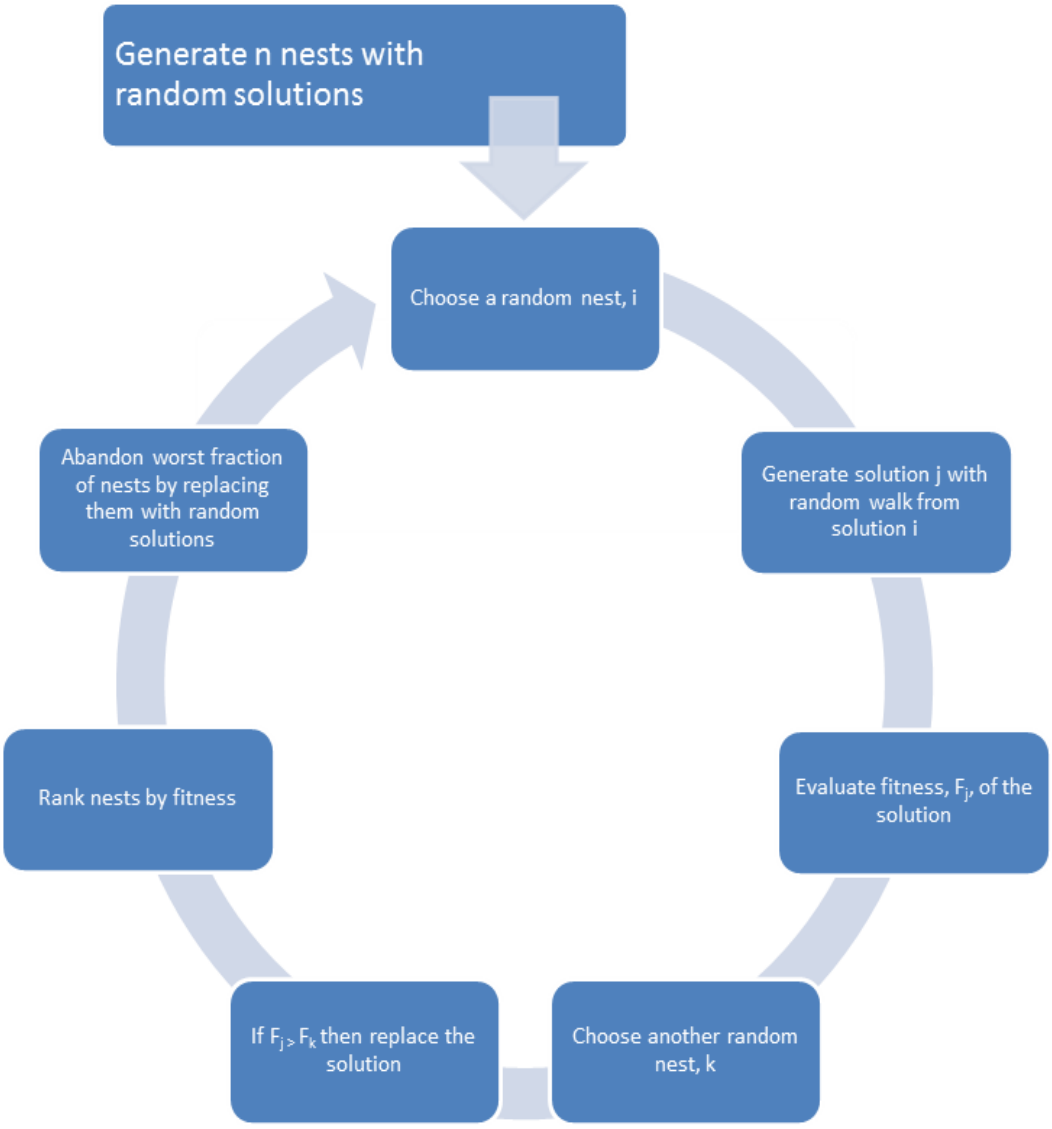
Cool extensions

- Optimize an objective function using other algorithms (e.g. Particle Swarm Optimization).
- Create a user-friendly frontend for users to enter a problem and fitness function of choice.
- Create an intuitive way to visualize the results.

Modularization

1. *Initial phase*
    - Back End:
        i. Objective/fitness function
        ii. Solution
        iii. Walking - Generating new solutions
            1. Random walk
            2. Levy flights
        iv. Overall algorithm (which includes above functions)
            1. Cuckoo Search (CS)
    - Front End:
        i. Defining the problem to be optimized
2. *Modification phase*
    - Multiobjective
    - Variants of Cuckoo Search: Quantum-inspired, modified CS
    - Compare to different optimization algorithms
3. *Cool features phase*
    - Graphical frontend interface
        i. allow manipulation of objective(s)
        ii. visualization of results
    - Build Particle Swarm Optimization (PSO) and compare the results to CS

Flow Diagram for Cuckoo Search



Testing

We will test the algorithms through several stages of tests.

For the more basic optimization problems, traditional testing can be performed. As a common introductory calculus problem, the fence problem can be solved in just a few lines of algebra. We can take advantage of this fact when implementing tests.

For more difficult problems, however, such as multivariable problems or simply problems that cannot be solved with a cookie-cutter process, more complex testing is required. In order to test the algorithm, we will use problems from papers that supply an expected range of answers. We will then assert that our found solution falls within this range.

These tests will utilize jUnit's assertTrue and assertEquals functions. Depending on the width of the solution range for the more difficult problems, the assertEquals with tolerance may also be utilized.

Code Framework (by package)

To see the current status of our code, visit https://github.com/NicholasMerrill/Optimizer.

algorithms

public class **CuckooSearchOpt**

`public void` **`solve`**`(OptimizationProblem` *`optProb`*`)`

Solves *optProb* for maximum fitness (as determined by *optProb*.**fitness**) using the Cuckoo Search algorithm, making a number of optimizations, and stores the solutions privately. This is the highest level code for Cuckoo Search, and it makes use of other methods such as **randWalk** and CSSolutionSet.**abandonWorstSolutions**. A more holistic understanding of the Cuckoo Search algorithm can be gained by viewing our Cuckoo Search flowchart.

`public void` **`randWalk`**`(CSSolution` *`seed`*`)`

Returns a new solution that is a random walk away from *seed*. This serves a core function in the overall Cuckoo Search algorithm.

`public SolutionSet` **`getSolutions`**`()`

Returns all the current solutions, sorted by fitness in descending order.

problems

This package contains all the optimization problems to be solved, which include the constraints and fitness functions.

public *abstract* class **OptimizationProblem**

Each problem to be optimized extends from `OptimizationProblem`

public abstract int **getNumVar**()

Returns the number of variables in the problem.

public abstract double **fitness**(Solution *s*)

Returns the quality of a given solution.

public abstract boolean **withinConstraints**(Solution *s*)

Returns true if the current solution is within the logical bounds of the problem.

public class **FenceProblem**

The fence problem serves as an example application for any optimization algorithm we implement. Three sides of a fence and a river together form a closed area. Representation rule: The coefficient of the single variable in the solution for this problem is the length of a side that is adjacent to the river. The goal is to maximize that area, given a certain length of available fence. It implements the **OptimizationProblem** methods explicitly.

public abstract int **getNumVar**()

Returns 1, as the fence problem is a univariate problem.

public abstract double **fitness**(Solution *s*)

Returns the area of the fence.

public abstract boolean **withinConstraints**(Solution *s*)

Returns true if all sides of the fence will be of length greater than zero with the given solution *s*.

solutions

This package contains the classes for the variables that are to be optimized. The data structures of the variables and other information are abstracted away to allow for modularity.

`public class` **`CSSolution`**

Solution set specifically for the Cuckoo Search algorithm - in the terminology of the algorithm, this represents a nest.

`public void` **`newSolution`**`(CSSolution` *`i`*`)`

Generates new coefficients for the private arraylist structure of coefficients based on the coefficients for CSSolution *i* by taking a random walk from *i*'s coefficients.

`public void` **`setAsRandSol`**`()`

Sets the coefficients of the solution to random numbers.

`public ArrayList<Double>` **`getCoefs`**`()`

`public void` **`setCoefs`**`(ArrayList<Double>` *`coefs`*`)`

In order to allow different OptimizationProblem classes to operate on solutions (e.g. making a random walk, etc.), we have to expose the coefficients of a solution.

`public class` **`CSSolutionSet`**

Contains all the solutions of the current generation. Abstraction allows for techniques specific to genetic algorithms (finding new solutions, discarding bad ones).

`public CSSolution` **`getRandSol`**`()`

Returns a random solution from the solution set.

`public CSSolution` **`getSol`**`(int` *`i`*`)`

Returns a specific solution by index *i*.

`private void` **`sortSolsByFitness`**`()`

Sorts all solutions in the private arraylist by their fitness, in descending order.

`public void replace(int j, CSSolution sol)`

Replaces the CSSolution at index *j* of the private arraylist with CSSolution *sol*.

`public int getNumSols()`

Returns the number of solutions in the solution set.

Timeline

Week 1 (4/14-4/21) Alpha Version

1. Put together a working version of Cuckoo search.
    a. Implement a random walk for two variables.
2. Finalize and test Fence Problem in order to test basic Cuckoo algorithm.
3. Implement jUnit tests for Cuckoo Search.

Week 2 (4/21-4/28) Beta Version

1. PSO - choose specific algorithm and implement
2. Finalize and test Box Problem (including random walk for three variables)
3. Implement random walk for N variables (ideally, normalize by distance--step size--from current solution).
4. Possibly implement other random walks (e.g. Levy flights, diff distributions, etc.).
5. Possibly implement modified versions of Cuckoo Search.

Week 3 (4/28-5/5) Official Release

1. Find/create a real-world, complex (i.e. many-variable) problem that can be expressed as an optimization problem.
2. Compare results of PSO to CS.
3. Prepare video and paper.
4. Submit final project!

Progress Report

As indicated in the Code Framework above, from a design perspective, we chose to break our project up into problems, solutions, and algorithms, each of which inherits or implements a core set of features required of its function in the overall optimization process.

Currently, we have coded the basic Cuckoo Search algorithm, though some of its functionality (e.g. random walk) remains to be implemented. This segment of the code can be viewed in `CuckooSearchOpt.java` under the `algorithms` package. Additionally, we have implemented two simple optimization problems to test the basic functionality of the code.