NP-Hard Optimization of Wizards through Simulated Annealing

Jun Hui Zhou, David Luo, Raymond Ly

December 2, 2017

1 The Problem

Given a set of constraints, each constraint holding exactly three wizard names with the format ["A", "B", "C"] where C's age is not in between the age of A and C's, and a set of wizards, find an ordering of the wizards such that it does not violate any constraint.

2 Simulated Annealing

NP-Hard problems can typically be challenged by reduction-approaches as we have so often encountered in this class; however, our group decided take a non-deterministic approach through simulated annealing [1]. The analogous real-world phenomenon this algorithm draws inspiration from is annealing in metallurgy, a technique involving controlled heating and cooling to control the form of crystals. The general approach of the algorithm is to use neighboring solutions and a state heuristic to climb to a global optimum through local optimum while also allowing and accepting neighbors of a lower state heuristic than the current one. The intuition behind this is that by following only the best local optimum, the solution converges to a solution that is not the global optimum because of the limited proximity of the greedy approach. However, by accepting neighbors of a lower heuristic from the current state with a probability proportional to the global temperature, this alleviates the local-optimum sink of hill climbing. Furthermore, the cooling of the temperature should exponentially decay such that when the temperature is high, the acceptance probability of a worse local optimum should be exponentially higher than that if the temperature is low. Analogous to a metal being more malleable at higher temperatures while being heated, then becoming more rigid when it is cooling: simulated annealing jumps around states before converging to an approximate global optimum.

3 Our Algorithm

Our base code was inspired by Katrina Ellison Geltman's article on simulated annealing [1]. Initialize a global temperature t of 1, an α of 0.99, which is the rate at which the system cools, a global temperature minimum m 0.000001, and a random ordering of the given set of wizards. Then, find a neighboring solution by randomly finding a wizard in the ordering and inserting it into a random new location in the ordering. If the neighboring solution has a higher heuristic, defined as the number of constraints that the ordering satisfies, then the current solution state should be set to the neighbor. Also, if the acceptance probability, which is defined as $e^{\frac{old\_heuristic - new\_heuristic}{temperature}}$ is higher than a randomly generated number in between 0 and 1 inclusive, then the current solution should be set to the neighbor even if it is a worse neighbor. Then, we set the next iterative temperature, if the temperature is t, to $t \cdot \alpha$. Repeat until either the global optimum is found or the global temperature minimum is reached.

4 Further Optimization

1. Find a neighbor i (in our case i=1250) times at a single temperature instance of just a single time. This permits more flexible solution-seeking at the higher temperature state which undermines the local optimum sink of hill climbing.

2. Instead of checking a heuristic by going through all the constraints, we only go through the constraints that involve the wizards for which the heuristic applies to. We keep a global hashmap that maps, (key-value), each wizard to the set of constraints that contains it respectively.

3. Originally finding a neighbor involved shifting two random wizards, but then we found that randomly placing a wizard within a set ordering had more success because swapping was more likely to find deceptively higher heuristics while not necessarily approaching the global optimum.

5 Code

```python
import argparse
import math
from random import shuffle, sample, random

conWiz = {} # A dictionary that maps each wizard to the list of constraints that contain its names

def acceptance_probability(energy, newEnergy, temperature):
"""
    Input:
        energy: The current heuristic
        newEnergy: The new heuristic of a neighbor's
        temperature: The current temperature of the system
    Output:
        A number in between zero and one that determines the probability
        that the system will accept the neighbor's ordering
    """
    if (newEnergy < energy):
        return 1
    else:
        return math.exp((energy - newEnergy) / temperature)

def oneWizCost(num_wiz_in_input, wizards, wizard, solution):
"""
    Input:
        num_wiz_in_input: The number of wizards in the input
        newEnergy: The new heuristic of a neighbor's
        temperature: The current temperature of the system
    Output:
        The new heuristic of inserting the random wizard somewhere random in the
        ordering
    """
    output_ordering = solution
    output_ordering_set = set(output_ordering)
    output_ordering_map = {k: v for v, k in enumerate(output_ordering)}

    if (len(output_ordering_set) != len(output_ordering)):
        return "The output ordering contains repeated wizards."

    # Counts how many constraints are satisfied.
    constraints_satisfied = 0
    constraints_failed = []
    wiz_constraints = conWiz[wizard]
    for i in range(len(wiz_constraints)):
        constraint = wiz_constraints[i]

        c = constraint # Creating an alias for easy reference
        m = output_ordering_map # Creating an alias for easy reference

        wiz_a = m[c[0]]
        wiz_b = m[c[1]]
        wiz_mid = m[c[2]]

        if (wiz_a < wiz_mid < wiz_b) or (wiz_b < wiz_mid < wiz_a):
            constraints_failed.append(c)
        else:
```

```
            constraints_satisfied += 1

    return num_constraints - constraints_satisfied


def cost(num_wiz_in_input, num_constraints, wizards, constraints):
"""
    Input:
        num_wiz_in_input: The number of wizards in the input
        newEnergy: The new heuristic of a neighbor's
        temperature: The current temperature of the system
    Output:
        The new heuristic of inserting the random wizard somewhere random in the
        ordering
    """
    output_ordering = wizards
    output_ordering_set = set(output_ordering)
    output_ordering_map = {k: v for v, k in enumerate(output_ordering)}

    if (len(output_ordering_set) != num_wiz_in_input):
        return "Input file has unique {} wizards, but output file has {}".format(num_wiz_in_input, len(output_ordering_set))

    if (len(output_ordering_set) != len(output_ordering)):
        return "The output ordering contains repeated wizards."

    # Counts how many constraints are satisfied.
    constraints_satisfied = 0
    constraints_failed = []
    for i in range(num_constraints):
        constraint = constraints[i]

        c = constraint # Creating an alias for easy reference
        m = output_ordering_map # Creating an alias for easy reference

        wiz_a = m[c[0]]
        wiz_b = m[c[1]]
        wiz_mid = m[c[2]]

        if (wiz_a < wiz_mid < wiz_b) or (wiz_b < wiz_mid < wiz_a):
            constraints_failed.append(c)
        else:
            constraints_satisfied += 1

    return num_constraints - constraints_satisfied

def neighbor(wizards):
    index1 = sample(range(len(wizards)), 1)[0]
    randWizard = wizards[index1]
    newWiz = wizards[:index1] + wizards[index1 + 1:]
    newWiz.insert(sample(range(len(wizards)), 1)[0], randWizard)
    return newWiz, randWizard

def solve(num_wizards, num_constraints, wizards, constraints):
    """
    Write your algorithm here.
    Input:
        num_wizards: Number of wizards
        num_constraints: Number of constraints
```

```
        wizards: An array of wizard names, in no particular order
        constraints: A 2D-array of constraints,
                     where constraints[0] may take the form ['A', 'B', 'C']

    Output:
        An array of wizard names in the ordering your algorithm returns
    """
    shuffle(wizards)
    solution = wizards
    old_cost = cost(num_wizards, num_constraints, solution, constraints)
    T = 1.0
    T_min = 0.000001
    alpha = 0.99
    new_cost = num_wizards

    while T > T_min:
        i = 1
        while i <= 1250:

            neighborRet = neighbor(solution)
            new_solution = neighborRet[0]
            changed_wiz = neighborRet[1]
            oldSolCost = oneWizCost(num_wizards, wizards, changed_wiz, solution)
            # print oldSolCost
            newSolCost = oneWizCost(num_wizards, wizards, changed_wiz, new_solution)
            # print newSolCost
            new_cost = (old_cost - oldSolCost) + newSolCost
            # print "HALP" + str(newSolCost)
            new_cost = cost(num_wizards, num_constraints, new_solution, constraints)
            ap = acceptance_probability(old_cost, new_cost, T)

            if ap > random():
                solution = new_solution
                old_cost = new_cost
            i += 1
            if new_cost == 0:
                print solution
                return solution
        T = T*alpha
    return solution

"""
======================================================================
   No need to change any code below this line
======================================================================
"""

def read_input(filename):
    with open(filename) as f:
        num_wizards = int(f.readline())
        num_constraints = int(f.readline())
        constraints = []
        wizards = set()
        for _ in range(num_constraints):
```

```
            c = f.readline().split()
            constraints.append(c)
            for w in c:
                wizards.add(w)

    wizards = list(wizards)
    return num_wizards, num_constraints, wizards, constraints

def write_output(filename, solution):
    with open(filename, "w") as f:
        for wizard in solution:
            f.write("{0} ".format(wizard))

def addToDict(wizDict, constraints):
    for i in constraints:
        wiz1 = i[0]
        wiz2 = i[1]
        wiz3 = i[2]
        if wizDict.get(wiz1) == None:
            wizDict[wiz1] = [i]
        else:
            wizDict[wiz1].append(i)

        if wizDict.get(wiz2) == None:
            wizDict[wiz2] = [i]
        else:
            wizDict[wiz2].append(i)

        if wizDict.get(wiz3) == None:
            wizDict[wiz3] = [i]
        else:
            wizDict[wiz3].append(i)



if __name__=="__main__":
    parser = argparse.ArgumentParser(description = "Constraint Solver.")
    parser.add_argument("input_file", type=str, help = "___.in")
    parser.add_argument("output_file", type=str, help = "___.out")
    args = parser.parse_args()



    num_wizards, num_constraints, wizards, constraints = read_input(args.input_file)



    addToDict(conWiz, constraints)


    solution = solve(num_wizards, num_constraints, wizards, constraints)
    write_output(args.output_file, solution)
```

References

[1] Katrina Ellison Geltman. The simulated annealing algorithm.