LEWIS UNIVERSITY

The Mold Shop Problem

Automating, Optimizing, and Reacting to the Domain-Specific Challenges of Scheduling Injection Molding Jobs

A Thesis

By

Ryan Palo

Department of Department of Engineering, Computing and Mathematical Sciences

Submitted in partial fulfillment of the requirements

for the degree of
Master of Science in Computer Science,

Concentration in Intelligent Systems

April 28, 2021

The undersigned have examined the thesis entitled '**The Mold Shop Problem**' presented by **Ryan Palo**, a candidate for the degree of **Master of Science in Computer Science (Concentration in Intelligent Systems)** and hereby certify that it is worthy of acceptance.

Advisor	Date
Program Director	Date
Department Chair	Date

ABSTRACT

Scheduling work for injection molding shops is a more domain-specific version of the Flexible Job Shop Problem. Luckily, there are domain-specific assumptions that allow us to simplify the problem. In this paper, I develop an automated method of generating a schedule based on existing jobs with processing times, due dates, and other metadata as well as domain-specific soft and hard constraints including setup and teardown time, material changeover, and machine tonnage limitations. My approach consists of a greedy list-building initialization improved by a simulated annealing local search. A concrete implementation of the solution was tested on real-world data and was able to generate schedules with similar or better fitness (based on the various fitness metrics) to human-generated schedules on datasets.

ACKNOWLEDGMENTS

Thank you to my parents for pushing me and always removing obstacles to success. If I'm even half the parent that you've both been, I will consider it a victory.

Thank you to my wife, for supporting me, taking on extra parenting while I work on classwork, listening to me talk about linked lists, XORs, and scheduling algorithms and pretending like I'm the most interesting man alive, and dealing with grumpy burnt-out academic Ryan when he shows up.

Thank you to Elle for grabbing my face and forcing me to focus on what's important.

Thank you to all my professors and advisors at Lewis for providing me with this opportunity to learn, grow, and prepare to teach the next generation of computer scientists.

Thank you to God for helping me get all the way done with this. And thank you, Lord, that I survived.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I: Introduction

Injection molding is an industry that is based upon a large up-front investment in precision tooling in order to be able to produce thousands of identical custom parts per day for minimal cost. This minimal cost is based on the fact that each part takes only a few seconds to create from raw plastic resin. At this time-scale, every second of machine time is precious, delays can be extremely expensive, and proper scheduling is the difference between success and failure for the business.

While the mold tools are able to create consistent and uniform parts, the differences between setups required to make different parts can be vastly different: different materials, processing conditions, mold tools, and set-up and cycle times, to name a few. Being able to put together a schedule of which parts to work on in which order is relatively straightforward. Generating a schedule that optimizes for minimal setup time and maximal run time while still meeting all due dates is more complicated. Additionally, like any other industry, nothing ever seems to go as planned. Tools break, operators get sick, and material runs short. It's not uncommon to completely throw out a schedule and build a new one from scratch several times a week. With each iteration of the schedule taking a few hours to build manually, it's clear that an automated system able to generate even a mediocre schedule quickly would be able to save a business hundreds of hours of scheduling labor a year.

The purpose of this research is to develop an algorithm that is able to do just that: build a passable schedule based on the job requirements in a work queue that is able to adapt to constraints and optimize for on-time deliveries as well as aid in future work estimation.

This paper is structured as follows: Chapter 2 provides a literature review of approaches taken to similar problems as well as how those problems differ from the mold shop problem. Chapter 3 is a discussion of the methodology used to build this solution, including the assumptions made, complications or simplifications taken into account, and the final algorithm developed. Chapter 4 is a discussion of the performance of that algorithm using real-world examples. Finally, Chapter 5 is a conclusion and summary of what was developed, as well as a discussion of future work to be done.

CHAPTER II: Background and Literature Review

In its simplest form, the Mold Shop Problem is a more specific, domain-focused version of the Job Shop Problem. In the Job Shop Problem, you have some finite number of jobs *J* that each have a set of operations to perform *O*. You also have a set of different machines *M* that can perform these operations. The goal is to decide on an assignment of the order of which operations for which jobs should occur on which machines when (Meeran, 2012). In general, for any non-trivial example, the Job Shop Problem is NP-complete (Vázquez-rodríguez, 2010). This means that the state-space is so large that any naïve or "brute force" solution is essentially unfeasible, even at the smallest of scales.

There are several common methods to approaching this problem, but they generally seem to boil down to some sort of relaxation of the problem, a simplification using domain-specific assumptions, machine learning, a simulation approach, genetic algorithms, some sort of local search algorithm, or some hybrid of two or more of those. Without the ability to generate many good example schedules, and with "learn by failure" a potentially costly endeavor, we will avoid the usage of machine learning for our purposes. There are a few different simulation approaches in use, including the use of Petri Nets (Wongwiwat, 2013) (Caballero-Villalobos, 2013), but those are typically used where there are multiple operations on multiple work stations for each job. That is not typically the case for the mold shop. The remainder of this chapter will look at applications of the genetic algorithm to similar problems as well as a few different local search algorithms.

The Genetic Algorithm

The genetic algorithm is an algorithm that takes its inspiration from how gene pools evolve in nature through natural selection and survival of the fittest. It is made up of six essential parts: the

definition of what the "DNA" of a solution looks like, the fitness function, initial gene pool generation, selection, crossover, and mutation (Meeran, 2012). Typically, this ends up being a list of numbers defining the order in which the jobs are worked on (e.g. [1, 5, 3] would denote completing Job 1, then Job 5, followed by Job 3). There are two orderings that exist in parallel and, together, fully define the system (Rahmati, 2012). First, there are the operation precedence rules that define each job. For example, on Job 1, first it must go to Machine 3, then to Machine 1, and then to Machine 2. These are fixed and cannot (usually) be done out of order. The other ordering is that from the machines' perspectives. For a given machine, in what order does it process the jobs from the global queue? This ordering will more closely affect the final result of how long it takes to complete all jobs (the "makespan"). A poorly scheduled machine becomes a bottleneck and holds up jobs that could be being worked on at other machines in parallel.

A common DNA model improvement has the dual benefit of solving the per-job operation precedence issue while actually dramatically simplifying the size and shape of the DNA: repeated job numbers. Assuming there is only one of each machine that can perform some operation, you can specify the work queue as a simple list of job numbers, which repetitions of the job number specifying which operation is supposed to be worked on next. For example, a genome of [1, 3, 2, 1, 2, 1, 3, 3] would be processed as follows. Operation 1 of Job 1 should be placed in the queue of the machine that can perform that operation (e.g. a lathe). Next, Operation 1 of Job 3 should be placed in the queue of *that* machine. If Operation 1 of Job 3 uses the lathe, then it'll get placed behind Job 1 in that machine's queue. Otherwise, it gets placed in the front of the queue for whatever machine it requires. Then Operation 1 of Job 2, followed by Operation 2 of Job 1, Operation 2 of Job 2, Operation 3 of Job 1, and so on. In this way, it's not possible to get the

operations out of order for any specific job because they are looked up on-demand as they are needed and as their prerequisites are finished up (at least, per the schedule) (Meeran, 2012).

The fitness function is a measure of how well a particular solution performs. It takes a solution vector as input, and outputs a value based on problem-specific criteria. Often, in the case of Job Shop problems, this will include consideration of possibly the makespan, total tardiness vs. due dates, as well as any other constraints that would generate invalid schedules (Gökan Maya*, 2015).

The next part of the genetic algorithm is the initial gene pool generation. Some sources recommend using heuristics or priority dispatch rules (Vázquez-rodríguez, 2010), some recommend a process called a mid-cut (Meeran, 2012), and some declare that too much work with too little benefit and state that a completely random initial population is sufficient (Gökan Maya*, 2015).

The third element required is a method of selection for which solutions will "procreate," passing their "gene structure" forward to future generations of solutions. The most common selection criteria is a weighted-average "roulette"-style random selection. Mimicking real life, the solutions that are the fittest will be more likely to procreate, but, in order to maintain diversity in the solution pool (so as not to get hung up in a local minimum) sometimes less fit solutions might be selected and might contribute something different to the future gene pools.

Crossover is the algorithm for how the two "parents'" DNA is combined to make a child. Options might include selecting a random splitting point and using the front of one parent up to that index and the back of the other parent after that index, picking which parent contributes each index at random by generating a vector of 0's and 1's, or, as (Meeran, 2012) do, choose a hybrid solution,

randomly select each index, and then perform list rotations, and then combine the inverse of the selected items to form a second child.

The final step is mutation. It has been shown that mutation can help keep the diversity level of a solution pool up which can actually increase how quickly high-quality solutions are reached (Meeran, 2012). This involves changing a strand of DNA just a small amount using randomness. (Meeran, 2012) chose to perform a relocation of specific job's operations within the list. (Trilling, 2012) swapped the location of two assignments and then used a greedy algorithm to patch any violated constraint issues this might cause. As with mutation in real life, the mutation can introduce a sudden significant improvement in fitness, but it can also cause a change for the worse.

Local Search Options

A local search is a searching algorithm that is non-exhaustive and cannot see the entire search space. It can see the state it is in and possibly a bit of the neighborhood of solutions around it, and it knows how to get from one solution to one of the others in its neighborhood. It also has some idea of how well it's doing via a fitness function, which it is usually trying to minimize. The algorithm used dictates how it make its way towards a final solution and what does it do to avoid local minima.

Stochastic Descent

Stochastic descent is the simplest form of optimization where a state moves from neighbor to neighbor ensuring that every step is in the direction of steepest decrease. In this way, the solver essentially "follows the slope and rolls downhill" until it reaches a minimum. But therein lies the problems with a simple unaided gradient descent solver: local minima and plateaus. It's possible for a solver to get trapped, because, if it gets to a point where any step it takes in any direction will

either increase the fitness function (a local minimum) or all steps in all directions have the same fitness (a plateau), it doesn't know what do do. There are varying methods for how to deal with plateaus, such as continuing on in the same direction until the edge of the plateau is found. However, to deal with local minima, something more is needed. The following few algorithms will provide solutions to this problem in different ways.

Simulated Annealing

Simulated annealing is another algorithm with roots in the physical world; this time, materials science. In order to reduce stress in a metal, the material is heated, and the grain structure grows and loosens. Then, the temperature is reduced, but slowly. As the metal cools, the grains contract, but the slowness with which they contract allows them the time to shift into a more favorable, low-stress position than the original (Klement, 2021). It's similar to shaking a mixture of sand and rocks to settle the bigger rocks to the bottom.

In the simulated annealing algorithm, a "temperature" coefficient is initialized at a high number. The algorithm will generate a neighbor state. If the neighbor's fitness is better, that state will be selected as the next iteration's current state. If it's worse, but a randomly selected value is less than the temperature coefficient, the neighbor state will be selected anyway. The algorithm repeats again and again, with every step decreasing the temperature, usually on a negative exponential curve (i.e. rapid initial cooling which slows as it reaches "room temperature") (Hernández-Ramírez, 2019). In this way, the basis of the algorithm is a gradient descent, but the high levels of initial randomness are included in the hopes that the solution will settle into the "right" minimum "well" as the temperature drops, before following the more traditional gradient descent towards the global minimum.

Branch and Bound

Branch and Bound is another strategy that combines a myopic gradient descent with a more traditional and time-intensive bounded exhaustive search to find its way out of local minima. It starts by searching and creating a small amount of "sons" or neighbors. However, if it gets stuck, it temporarily expands the number of "sons" it generates for a given state. It keeps growing its neighborhood, but the second it finds an improved solution, it shrinks the neighborhood back down as small as possible once again to continue its descent (Ladhari, 2003). It only stops if computation time reaches some limit or if its neighborhood size reaches some maximum (meaning the minimum it found itself in is suitable deep enough that it may be the global minimum).

Particle Swarm Optimization

Particle Swarm Optimization is a neat algorithm that draws inspiration from flocking/swarming mechanics. It initializes several "drones" or searchers, and it keeps track of which of them has the best fitness and what that fitness is. Each drone performs a local search, but with some amount of randomness scaled by how far away from the global minimum they are, their solution might get "course-corrected," bringing them a little closer to the global best solution. In this way, as the swarm in general finds better and better solutions, those drones that got stuck in local minima will get "respawned" or "pulled" out of their minima and put to work closer to the true best solution (Trilling, 2012).

Tabu Search

Tabu Search is actually a more general form of Simulated Annealing in that it is a gradient descent, but if no neighbor states are present that are an improvement on the current fitness, the algorithm will allow for a move to a worse fitness state. The only difference is that a history of previously

visited states is kept and forbidden (hence, “tabu” or, taboo). This is to prevent the algorithm from going around in circles around the same local minimum, but, instead, heading out of a local minima and into new areas of exploration (Meeran, 2012).

Fitness Functions

Regardless of what sort of searching or randomness is included in the final solution, all of them require some way of telling how good of a schedule has been created because there is no “one right answer” to the question of how a schedule should be designed. And that requires a heuristic, or fitness function. Typically, in the types of Job Shop problems we are concerned with, the main factor in the fitness function will be either the total maketime (Meeran, 2012), which is the total time it takes to complete all jobs, or some measure of tardiness as compared to due dates (Klement, 2021). That being said, there are many other factors that could feasibly be optimized for, including reducing how early jobs are finished (Vázquez-rodríguez, 2010) or even how energy-efficient your shop is (Gökan Maya*, 2015).

In addition to this main optimization goal, one might also want to consider the ability to add additional constraints to the solution, including domain-specific requirements that would invalidate certain solutions (i.e. hard constraints) or preferences where, all else being equal, one alternative is preferred over another (i.e. soft constraints). The challenge is deciding how and when to incorporate these constraints into the algorithm.

In general, it has been shown to be efficient to incorporate hard constraints as early as possible, using them to limit the size and shape of your search space. This cuts down on wasted effort generating solutions that would get thrown out later for not meeting constraints (Trilling, 2012). It *is* possible to emulate hard constraints by creating a soft constraint with an outrageously large

penalty, but (Lorenz, 2016) discuss how this can still allow a small fraction of invalid solutions to poison your search space or gene pool, leading to lost work chasing generations of solutions all based on a flawed assumption.

Soft constraints can be implemented in a few different ways. By far the most common is to create some sort of penalty to the objective function, although it has been shown that rewarding positive features of solutions in addition to punishing negative features improves performance (Lorenz, 2016). Another approach is to convert the entire problem to a multi-objective optimization and create a hierarchy of weighted objective functions all based on different constraints, thus removing the dichotomy between a soft and hard constraint (Nieße, 2016). It is possible to emulate soft constraints using only hard constraints, but, because soft constraints aren't always guaranteed to be independent of each other, all combinations of constraints must be checked for the final fitness function, which leads to an excessive amount of simulation and computation (Lorenz, 2016).

Domain-Specific Assumptions

So far the only other research found on scheduling for production injection molding, (Klement, 2021) discusses a few alterations to the traditional Job Shop format.

- Firstly, most—if not all—jobs contain only one operation, so operation precedence can either be ignored completely or dealt with as a separate/special case. This simplifies the problem.
- Second, there are usually multiple machines that could potentially run a particular job. This shifts the problem into a subset of the Flexible Job Shop Problem, where the additional question of which of multiple machines should be used must be incorporated into the solution. This complicates the problem dramatically.
- Setup time is considered between jobs as a cost, which is an additional complication on top of the traditional problem. This is because the setup cost can vary depending on which job is following which, and that needs to be accounted for. However, this actually opens up a

nice avenue into which we may be able to slip other soft constraints and/or hidden costs, which is helpful.

The complication of moving from a standard Job Shop Problem to a Flexible Job Shop Problem means that we now have to solve two separate sub-problems: the machine assignment problem and the per-machine operation scheduling problem. Some research notes that solving these two sub-problems entirely separately can simplify things once again, but that solving them both at once in a hybrid solution usually creates algorithms that perform better. It means that solution models will need to take the form of two vectors: one for the order of operations worked on and one for which job is assigned to which machine (Rahmati, 2012). Luckily, the fact that each job typically only has one operation takes a large portion of the additional complexity out. We may be able to re-simplify our solution "DNA." (Klement, 2021) solves this by greedily always assigning jobs to the first open machine that can run them, removing the need for the second strand of DNA, as it can be derived from the first strand and problem input information.

CHAPTER III: METHODOLOGY

The shape of my solution to this problem is very similar to that found in (Klement, 2021): a two layer approach. A local search algorithm will start with an initial solution (or set of solutions) and iterate, searching through the total solution space for the best solution it can find. The solution "DNA" the local search uses will be a linear list of jobs in the order they should be worked on. At each iteration, as well as at the end, the current solution will be converted into a shop schedule, which is a set of queues for each molding machine in operation. This shop schedule will be evaluated for effectiveness based on a fitness function, and this fitness score will be fed back into the local search. Appendix A contains the source code for the reference implementation. The following subsections will describe each piece of this approach in more detail.

Formalizing the Problem

The first task is to formalize the variables in the problem. The first input to the problem are a set of *k* machines to be used ($m_0 \ldots m_k$). It is sufficient to simply identify them via natural numbers, and thus sufficient to simply specify how many machines are available as *k*. Next, there are *n* jobs that need performed ($j_0 \ldots j_n$). Each job will be assumed to provide the following info at a minimum:

- Some sort of unique identifier
- Which mold it uses
- What plastic resin material is used
- Quantity of parts required
- Number of cavities the mold has
- Cycle time to shoot the mold once and eject finished parts from all cavities

- Amount of time to set up the mold
- Amount of time to tear the mold down
- A list of which machines the job can run on. Which job can run on which machine is determined by machine tonnage, barrel size, and other factors which can be defined once offline for each mold and assumed as given for this problem.

The final solution will come out as assignments ($a_0 \ldots a_n$) where each assignment specifies which machine a job will run on and when it will start (e.g. $a_6 = m_{3,5}$) might denote Job 6 starting on Machine 3, 5 days after the schedule's start date). From this, all other features of the solution can be calculated, including job end dates, lateness vs. due date, workspan on each machine and maximum overall workspan.

Modelling Solutions

In order to describe how to find the best solution, we must first describe what a solution looks like. Solutions will take two forms.

In the "outer loop" local search, solutions will be identified by a simple list of jobs, sorted by the order in which they should be assigned to machines.

While assigning jobs to machines both in order to calculate fitness and to present the final solution, the linear list of jobs will be broken out into separate queues for each available machine. Assuming the data is input with an existing list of numbered machines in mind, this can be simplified to a "list of lists," with the first list representing the order of jobs to be run on Machine 1, and so on.

Calculating Fitness

It is impossible to tell whether the solution this algorithm comes up with is good or not without some way to quantify how "good" a solution is. Coming up with each domain-specific (and likely shop-specific) factor and how heavily to weight it, whether or not to calculate multiple tiers of

fitness to break ties, and how heavily to weight separate tiers is likely an entirely separate thesis of its own. Only the final fitness function I arrived at will be discussed here.

The primary metric that should certainly be used in any approach is the total number of days late summed across all jobs.

$$lateness = \sum_{j=0}^{n} \max(finish\ date(j) - due\ date(j), 0) \quad (1)$$

Lateness cannot be negative. While some solutions may want to minimize how early jobs are completed as well to prefer leaner schedules with less stock in inventory, I'm leaving that out of my solution as that has historically never been a problem at our shop.

The other factor that will be used is a penalty for changing plastic resins. Any time there is a changeover from one resin to another, there must be a purging cycle and there is always a higher risk of material cross-contamination leading to rejected parts. For this reason, a penalty equivalent to a half-day of delay is added to the fitness score for every material change between jobs.

$$material\ penalty = \sum_{j=1}^{n} int\big(material(j) \neq material(j-1)\big) * f_m \quad (2)$$

$where: f_m \equiv user - defined\ material\ change\ penalty\ factor\ (0.5\ here)$

The fitness score is calculated by combining these factors, and the goal of the entire algorithm is to find the schedule with them minimum possible fitness score.

$$fitness = material\ penalty + lateness \quad (3)$$

Generating Initial Solutions

Some approaches simply generate initial schedules randomly (Ghiath Al Aqel, 2019), but I believe that it is likely that the optimal or near-optimal solution is not far off from a simple greedy approach based on which jobs are due first. For this reason, in local searches where only one initial solution is required, this solution is created by sorting all jobs, ordered first by increasing due date (i.e. start on the ones that are due soonest first), with ties broken by increasing duration (i.e. work on the shorter jobs first to avoid tying up machines early on). There are a multitude of other heuristics that could be used alternatively—some of which are described in the literature review above—but this is a close match to what we use in our fitness function, so it should be a fairly helpful initial heuristic.

```
sort(jobs, key=(job.due_date, job.duration))
```

Listing 1: Creating the initial solution by sorting by due date and duration

This mention of job "duration" is new and should be defined as the following:

$$duration = \frac{quantity * cycle\ time}{\#\ cavities} + setup + teardown \tag{4}$$

It should be clear that cycle time, setup time, and teardown time should have equivalent units.

Generating Neighbor States

The definition of a "local search" is a search in which the shape and size of the global search space is too large to enumerate so transitions are made from one state to another through small moves. The set of states that can be generated by performing these small moves are referred to as "neighbor

states" (Hoos, 2005). For our solution, this is as simple as selecting two jobs and switching their positions in the list. For local searches where it is undesirable to enumerate all of the possible neighbor states, it is sufficient to randomly select two integers *a* and *b* which are in *[0, j)* and where $a \neq b$, and switch the jobs at those two indices.

```
indices = range(0, n)
shuffle(indices)
a, b = indices[0], indices[1]  # Ensure a != b
neighbor = copy(current)
neighbor[a], neighbor[b] = neighbor[b], neighbor[a]
```

Listing 2: Generating neighbor states via swapping

It is valid to note that this is essentially a "shot in the dark" technique, and there may be room for some sort of informed selection of jobs that, when switched are the most likely to yield good results. But it is just as valid to note that many algorithms (e.g. Genetic Algorithms) make use of such randomness to improve diversity in the solution pool and keep solutions from being trapped in a local suboptimal minimum (Yang, 2014).

The Outer Loop: Local Search Algorithm

There are a number of algorithms that would all fit relatively well into this part of the algorithm: stochastic descent (Klement, 2021), branch-and-bound (Ladhari, 2003), tabu search (Meeran, 2012), particle swarm optimization (Trilling, 2012), or a genetic algorithm (Vázquez-rodríguez, 2010). The would all be able to use the same list-based model, generate neighbor states, evaluate fitness functions, and output selected candidates. The question then becomes a balance of which provides the best fitness, which runs the fastest, and how each of the variable parameters each require are set (e.g. mutation rate for genetic algorithms or initial temperature for simulated

annealing). For this research, stochastic descent and simulated annealing were used. Listing 3 below sketches the Stochastic Descent algorithm's structure.

```
current = generate_initial_solution()
current_fitness = fitness(current)

while not stopping_condition():
    neighbor = generate_neighbor(current)
    neighbor_fitness = fitness(neighbor)
    if neighbor_fitness < current_fitness:
        current = neighbor
        current_fitness = neighbor_fitness

return assign(current)
```

Listing 3: Stochastic Descent

In essence, every iteration, a single neighbor is randomly generated. If the neighbor is more fit than the current state, the neighbor is selected as the next current state. The benefits of this algorithm are mostly in its simplicity. It is easy to write, easy to describe, and hard to incorporate bugs. It's main drawback is that it has no way of escaping local minima that are suboptimal. It drives down to the very best solution that can be created in a given neighborhood and can go no further.

Listing 4 below shows how the simulated annealing algorithm works. It generally functions the same as stochastic descent, but, if the generated neighbor is not as good as the current state, it gets a second, probability-based chance anyways. This probability is driven by a "temperature" function, which is usually some exponential function dependent on the difference in fitness between the two states as well as an ever-decreasing "temperature." This constantly decreasing temperature ensures that the algorithm is likely to accept almost any solution early in the iterations,

but become more picky as time goes on. This allows the search function to "anneal," settling into the deepest, best well of fitness before operating like the stochastic descent and plunging for the minimum.

```
    current = generate_initial_solution()
    current_fitness = fitness(current)
    t = t_initial

    while not stopping_condition():
        neighbor = generate_neighbor(current)
        neighbor_fitness = fitness(neighbor)
        if neighbor_fitness < current_fitness:
            current = neighbor
            current_fitness = neighbor_fitness
        elif random() < f(current_fitness, neighbor_fitness, t):
            current = neighbor
            current_fitness = neighbor_fitness

        t *= alpha

    return assign(current)
```

Listing 4: Simulated Annealing

The temperature-based acceptance function ("f" in Listing 4) was defined as follows in Listing 5 for this analysis:

```
    define f(fitness_0, fitness_1, t):
        return exp(-(fitness_0 - fitness_1)/t)
```

Listing 5: Temperature-based acceptance function used for this analysis

Note that the top term inside the exponential will always be negative as this function only gets run in the case that the neighbor fitness is higher/worse than the current fitness. Because of this, the value returned will always be between 0 and 1. The likelihood of going with a worse neighbor goes up as the neighbor gets less bad or when temperature is very high.

Stopping Conditions

No matter what local search is used, because there is no way of knowing whether or not a "goal" state has been found, some sort of stopping condition must be artificially applied. A more thorough, longer-running option might be setting a max number of iterations without seeing any global improvement (i.e. stopping when it's been a while since you've seen any better solutions), but the simplest alternative is to simply pick a fixed number of iterations and run tests to identify the point of diminishing returns for what that fixed number should be. That is the strategy used for this research.

Generating Schedules from Job Lists

The fitness cannot be calculated—nor can the final solution be useful—without some secondary algorithm for decoding the list of jobs into a schedule of assignments of jobs to machines in a certain order. (Klement, 2021) recommend a simple, greedy algorithm where the soonest available machine is used to run a job, and so the list is distributed amongst the machines linearly. Intuition agrees with this decision, as this is the approach that is often taken during manual scheduling: decide which jobs are the highest priority and then load them onto compatible machines one-by-one. Even with the simplicity of this algorithm, it allows for a rough leveling of machine loading, making sure that the machines are as evenly loaded as possible. This is the approach taken here.

The main difference between (Klement, 2021) and this research for this portion of the algorithm is that predetermined machine compatibility is considered. Usually, a mold shop is not composed entirely of one type and size of machine. There will be different machines of different tonnages, some electric, some hydraulic, some with side-action capabilities, some with rotating platens, some vertical presses, and some horizontal. This is handled by searching for the soonest available

machine *only* in the machines that are compatible with the job being considered. It works out to essentially doing a *min(filter(machines))* instead of a *min(machines)* computation.

```
    for each job in jobs:
        compatible = filter(machines, machine in job.machines)
        machine = min(compatible, key=end_date)
        add_job(machine, job)
```

Listing 6: Assignment algorithm

CHAPTER IV: RESULTS

For this analysis, four different test cases were evaluated. 56 different jobs were derived from a real-world molding schedule, including different molds, materials, quantities, and machine compatibilities, and their details were changed to tune the differences between the test-cases. 8 machines were used, although one machine was incompatible with all jobs as a sanity check that the algorithm was respecting compatibility limitations.

- The baseline workload was designed such that it should be reasonably achievable to complete all jobs by their due dates, but all machines should be loaded most or all of the total makespan.
- The heavy workload was designed as a sort of Kobayashi Maru; all solutions should have at least a few late jobs.
- The light workload was designed to see how the algorithms handled easily achievable due dates and open machine capacity, shifting focus instead toward minimizing material changeovers.
- The late workload is similar to the baseline workload, but with a late start such that most jobs are late at the start of the schedule.

Looking at these four test cases together should give us enough information to evaluate the performance of the different local searches overall.

Local Search Optimization

Before any of the algorithms could be compared to each other, each individual local search algorithm had to be internally optimized. They all have their own unique set of problem-specific tuning variables, and these can only be set through experimentation.

Stochastic Descent

Luckily, with Stochastic Descent being as simple of an algorithm as it is, the only variable to define is the number of iterations after which to stop. Figure 1 below shows how the fitness for each test case evolves over time.

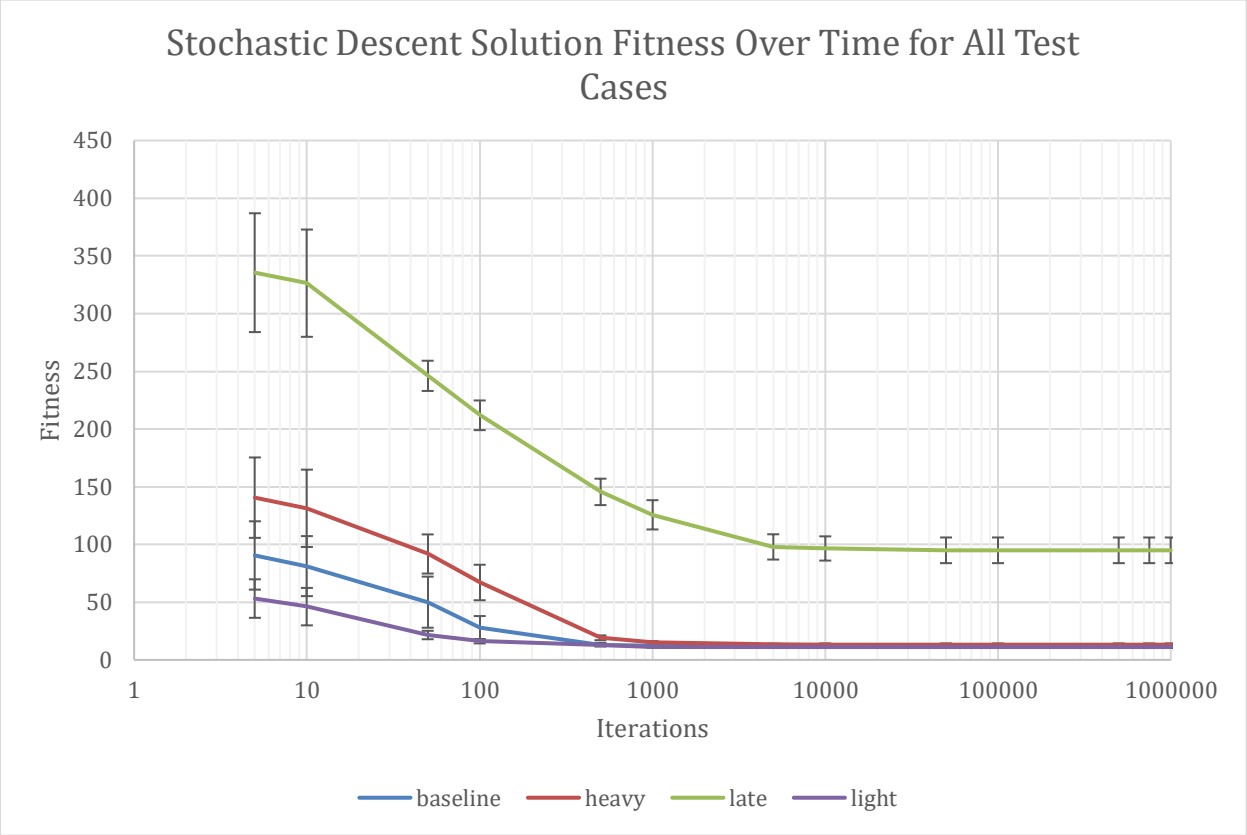

Figure 1: Determining the point of diminishing returns for the Stochastic Descent local search (trials=5).

For instances where it is easy to find acceptable non-late solutions, the algorithm converges as early as 100 iterations. In more normal cases, things appear to settle in around 1000. In cases where there are no real good solutions and only less bad ones, it can take up to 50,000 iterations. To be conservative, 50,000 iterations will be used as the ideal stopping condition for Stochastic Descent for the remainder of the tests here.

Simulated Annealing

Simulated Annealing has a few different variables to consider. In the simplest case, at least initial temperature, the reduction constant, and the total number of iterations must be considered. It should be noted that completely optimizing every parameter of the Simulated Annealing algorithm for this problem (and dataset) could potentially be a rabbit hole of research all its own. (Kim, 1998) presents a "Simplex algorithm" for doing just that for a similar problem set, but that task is outside the scope of this research. I believe the end parameters settled on after some experimentation, detailed below, to be *good enough*.

Initial Temperature (T_0)

It is recommended that the initial temperature be high enough that initial probabilities of transition are very close to 1 while not being too high that convergence takes too long. 1000 neighbors were generated each from both random and sorted initial solutions. The absolute worst neighbor generated was 53.44 fitness points worse and was generated by the random initial case. Figure 2 below shows the distribution. An overwhelming majority of neighbors were no more than 20 points worse.

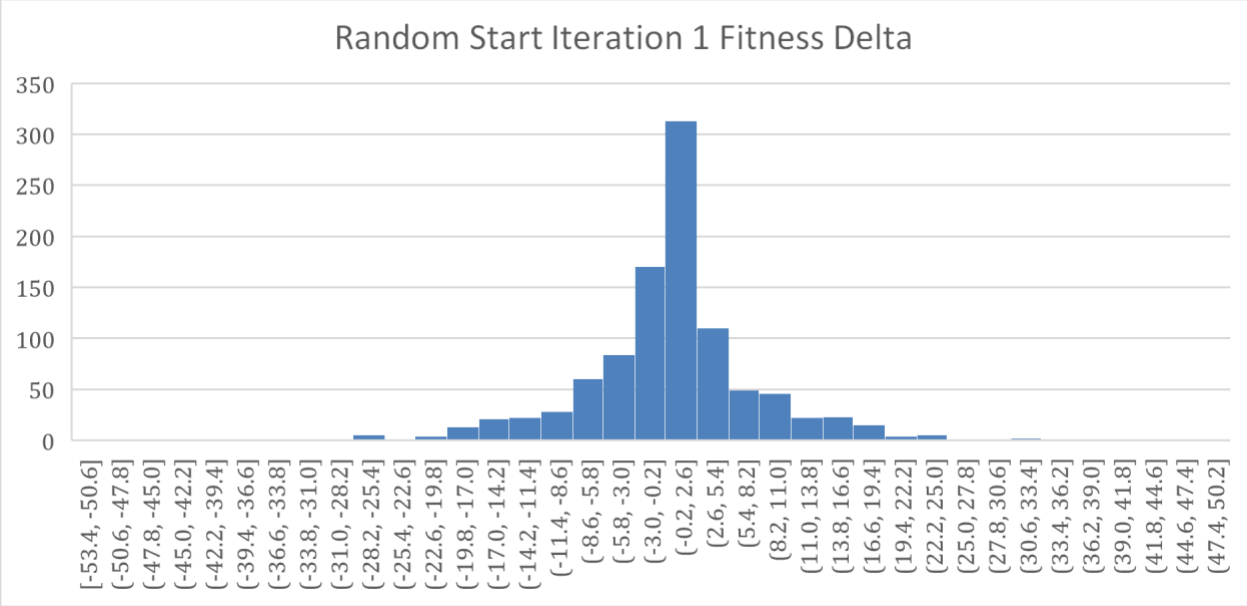

Figure 2: Distribution of fitness changes after one iteration from a random initial solution

Table 1 shows what the acceptance probability would be in both the "extreme" worst-case scenario and the "conservative" worst-case scenario provided by the samples. With the goal of getting a probability reasonably close to 1 using as low as possible of an initial temperature, selecting a T_0 of 10,000 appears to offer the best compromise. It's the lowest where even the extreme worst case probability is higher than 99.0%.

Table 1: Acceptance probability at various initial temperatures based on extreme worst-case and conservative worst-case fitness changes

Temperature	$e^{-\frac{53.4}{t}}$	$e^{-\frac{20}{t}}$
1	6.157E-24	2.061E-9
10	.0047	.1353
100	.5859	.8187
1000	.9480	.9802
10,000	.9947	.9980
100,000	.9995	.9998
1,000,000	.9999	.9999

Reduction Constant (α)

While there are many different approaches to controlling the temperature function over time (Kim, 1998), perhaps the simplest method is a geometric progression achieved by multiplying the current temperature by the same less-than-one factor every iteration. The main guideline for this factor is that, depending on how many iterations are performed as well as the initial temperature, the final temperature should be very close to zero. This would force the algorithm to reject almost all worse neighbors at high iterations, performing essentially identically to Stochastic Descent.

Testing was performed on α every .01 between 0.60 and 0.99 using 10,000 iterations, a 10,000 initial temperature, and the baseline testcase. 30 trials per datapoint were run. Figure 3 shows the results.

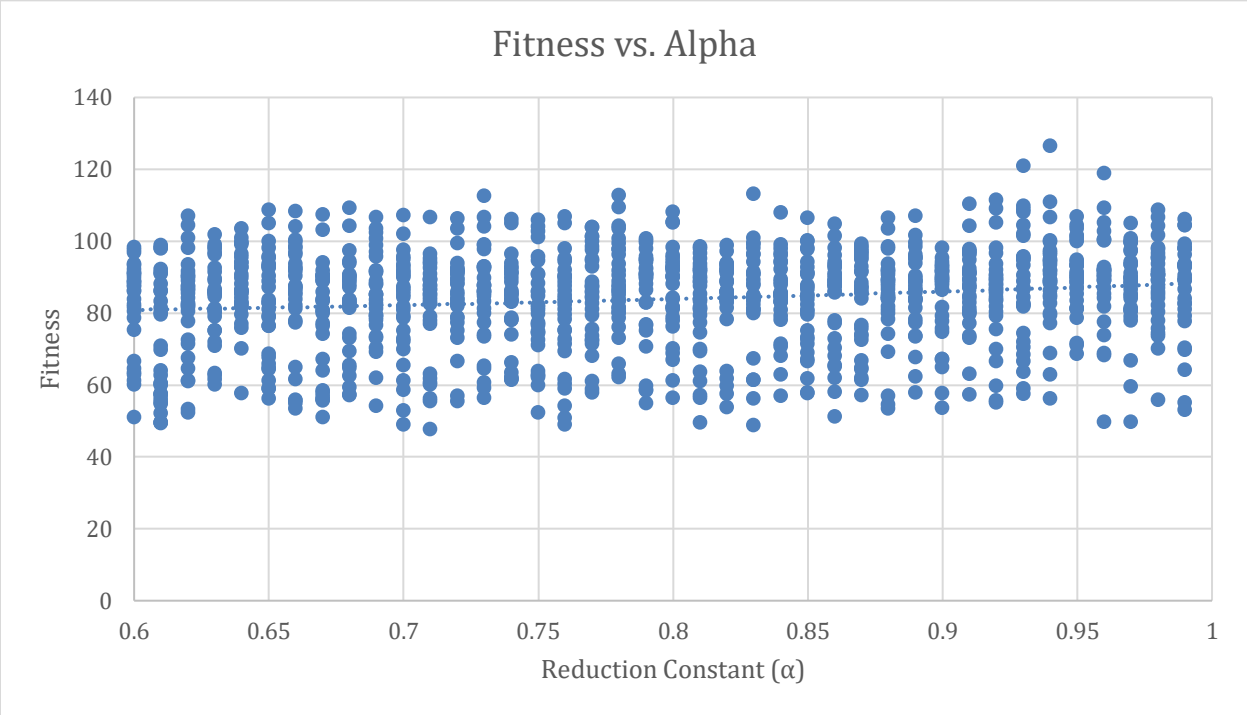

Figure 3: Plotting fitness at a range of alpha values

It does not appear that the value for α strongly effects the results. Even with a reduction constant of 0.99, the final temperature was on the order of 2E-40, showing that all selections of α would meet the "T_{Final} close to zero" requirement. Based on research, a middling value of 0.89 will be selected.

Initial Solution Generation Algorithm

Absolutely the most shocking result of the entire analysis comes from the area of initial solution generation. Intuition would suggest that the ideal solution would be very close to one where the jobs were worked on in the order that they were due. However, in the spirit of questioning as many assumptions as possible, a test was run against every testcase with both a randomized starting list as well as a list sorted first by due date and then by job duration. The following figures detail the results.

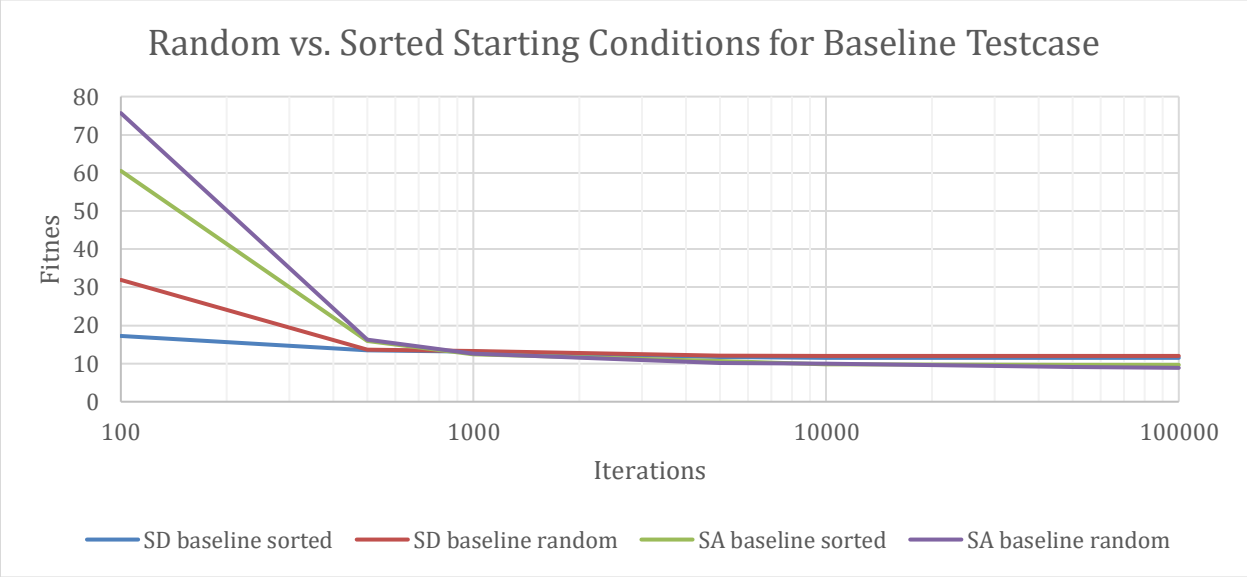

Figure 4: Comparing random and sorted starting states for the baseline case

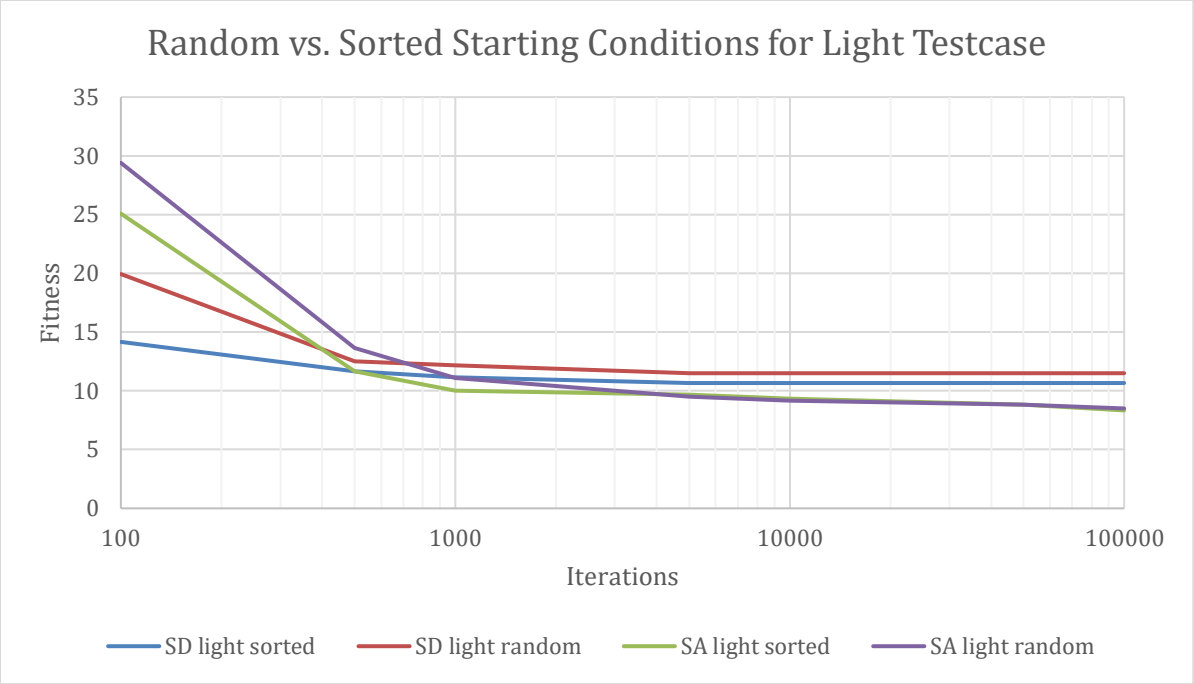

Figure 5: Comparing random vs. sorted starting states for the light testcase

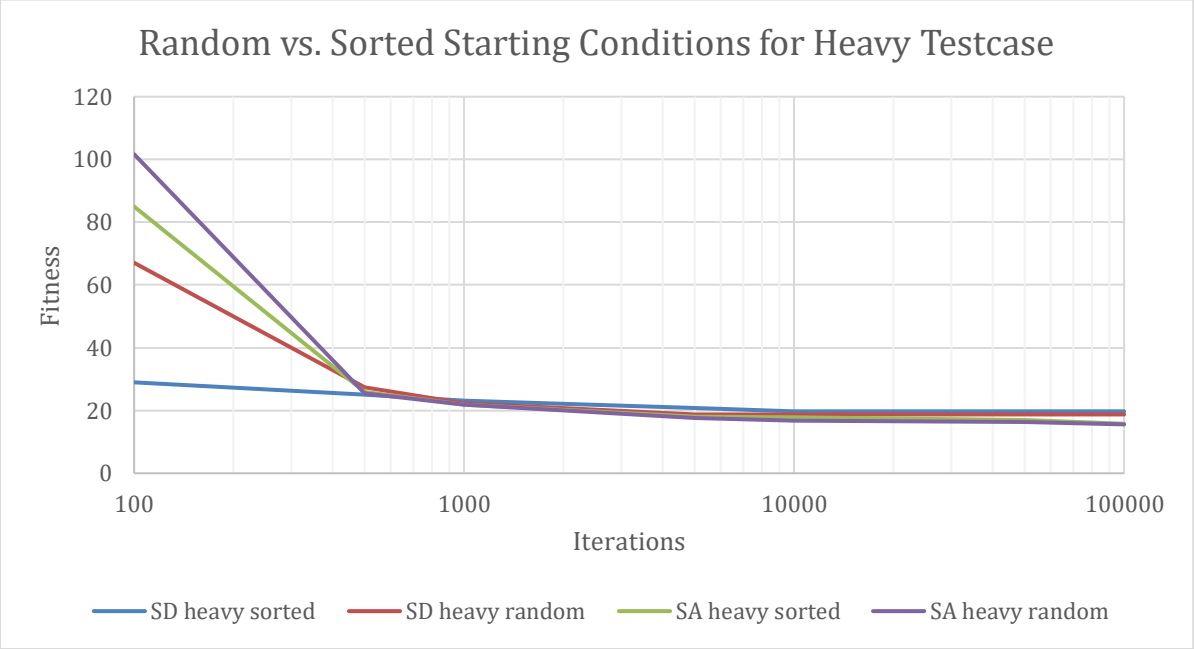

Figure 6: Comparing random vs. sorted starting states for the heavy testcase

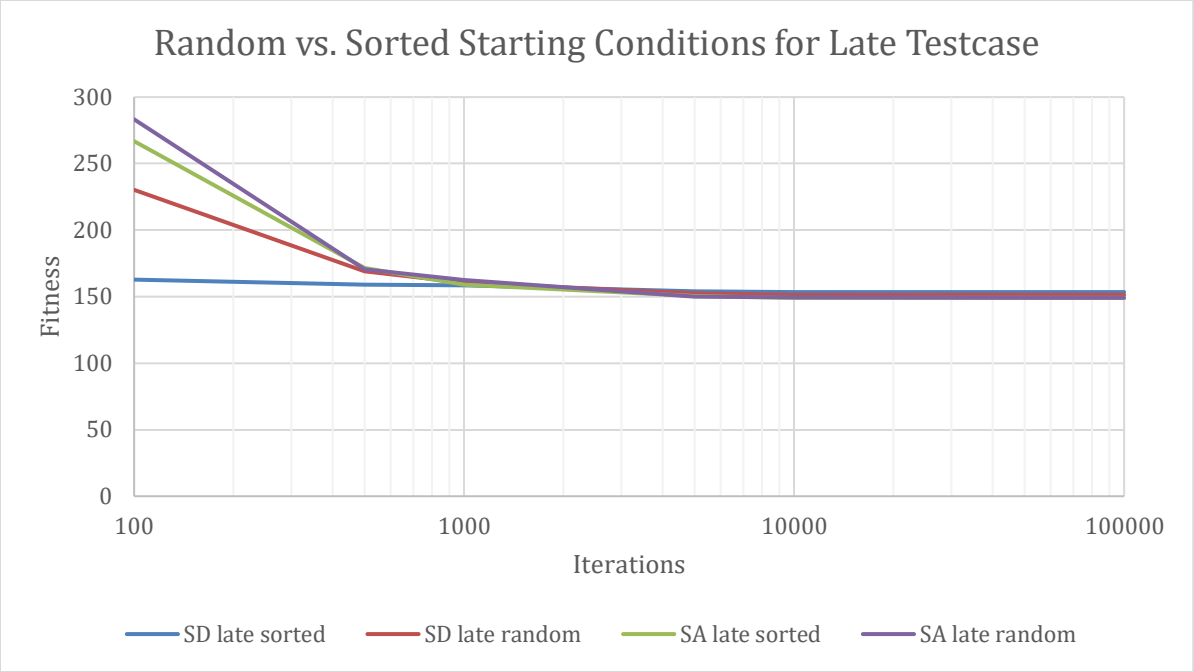

Figure 7: Comparing random vs. sorted starting states for the late testcase

In Figure 4 and Figure 5, the test cases where a near-ideal schedule was achievable, the sorted starting state performed slightly better. However, when the schedule got more impacted, as in the heavy and late testcases in Figure 6 and Figure 7, the random starting point actually performed better. Using a sorted starting state was a main difference between my approach and the approach detailed by (Klement, 2021), and I fully expected a sorted state to perform significantly better. Seeing this dependence on schedule density is an unexpected result.

Manual Scheduling

The manual scheduling process was performed as a baseline. Obviously the run time for the manual scheduling process is orders of magnitude longer than any of the algorithms, but it is shown for completeness. The fitness is the real metric to compare all of the other approaches to. This was done in a spreadsheet by hand with the goal of minimizing the fitness function, just like the automated algorithms would do.

Overall Results

Table 2 below shows the results of testing the various local search methods against four different workloads using the ideal initial generation algorithm, parameters, and settings uncovered from the above experiments.

Table 2: Overall Results

	Method	Fitness score	Runtime
Baseline workload	Manual	11.5	2692.18
	Stochastic Descent	11.3	15.19
	Simulated Annealing	9.2	26.23
Heavy workload	Manual	48.3	1352.78
	Stochastic Descent	18.4	12.92
	Simulated Annealing	16.1	28.78
Light workload	Manual	8.0	1085.91
	Stochastic Descent	11.2	13.38
	Simulated Annealing	8.7	31.53
Late workload	Manual	215.6	1217.26
	Stochastic Descent	150.6	13.53
	Simulated Annealing	149.5	29.52

With the exception of the light test case, the algorithms proposed here performed significantly better than even a skilled human scheduler. On top of that, they completed 100-200x faster (~30 seconds rather than 25 minutes). Even the performance in the light testcase was not more than a few days worse than the human schedule. The goal of the project was to develop an algorithm that could prepare a passable schedule and save some human labor time. This appears to be a success, although there is still plenty of work to be done.

CHAPTER V: CONCLUSION

In almost all cases, when being tested on real-world data, all local search algorithms outperformed human-built schedules in terms of our defined fitness metrics including total tardiness and material changeovers. All generated schedules were at least acceptably good, and runtime was two orders of magnitude better than doing the task manually. Simulated annealing performed better overall than stochastic descent. Surprisingly, randomized initial schedules perform better for busy workloads, though the initially sorted starting state worked better for lighter workloads.

The next steps are to improve its real-life usability via tasks like improving the user interface, incorporating past data to help estimate per-job stochastic risk possibly with machine learning, better metric reporting, and incorporating more domain-specific constraints to better match how human schedulers approach the problem with additional secondary preferences. Additional model considerations might include subpar operator efficiencies, per-mold catastrophic crash risks, family molds that run multiple parts at once, day/night shift run constraints, and more.

REFERENCES

Caballero-Villalobos, J. P.-D.-C. (2013). Scheduling of complex manufacturing systems with Petri nets and genetic algorithms: a case on plastic injection moulds. *International Journal of Advanced Manufacturing Technology, 69*(9-12), 2773 - 2786.

Gökan Maya*, B. S. (2015). Multi-objective genetic algorithm for energy-efficient job shop scheduling. *International Journal of Production Research, 53*(23), 7071–708.

Ghiath Al Aqel, X. L. (2019, 3 12). A Modifed Iterated Greedy Algorithm for Flexible Job Shop Scheduling Problem. *Chinese Journal of Mechanical Engineering, 32*(1), 21-32.

Hernández-Ramírez, L. a.-V.-B.-V.-R. (2019, 1). A Hybrid Simulated Annealing for Job Shop Scheduling Problem. *International Journal of Combinatorial Optimization Problems and Informatics, 10*(1), 6-15.

Hoos, H. H. (2005). *Stochastic Local Search : Foundations & Applications.* San Fransisco, CA, USA: Morgan Kaufmann.

Kim, M.-W. P.-D. (1998). A systematic procedure for setting parameters in simulated annealing algorithms. *Computers Ops Res.*, 207-217.

Klement, N. a. (2021). Lot-Sizing and Scheduling for the Plastic Injection Molding Industry—A Hybrid Optimization Approach. *Applied Sciences, 11*(3).

Ladhari, M. H. (2003). A Branch-and-Bound-Based Local Search Method for the Flow Shop Problem. *The Journal of the Operational Research Society, 43*(10), 1076-1084.

Li, Y. W. (2020). An improved simulated annealing algorithm based on residual network for permutation flow shop scheduling. *Complex Intell. Syst*, 1-11.

Lorenz, R. H. (2016). RNA folding with hard and soft constraints. *Algorithms for Molecular Biology, 11*(8), 1-13.

Meeran, S. a. (2012, 8). A hybrid genetic tabu search algorithm for solving job shop scheduling problems: a case study. *Journal of Intelligent Manufacturing, 23*(4), 1063-1078.

Nieße, A. S. (2016). Local Soft Constraints in Distributed Energy Scheduling. *Proceedings of the Federated Conference on Computer Science and Information Systems, 8*, 1517–1525.

Rahmati, S. a. (2012). A new biogeography-based optimization (BBO) algorithm for the flexible job shop scheduling problem. *International Journal of Advanced Manufacturing Technology, 58*(9-12), 1115-1129.

Stawowy, A. a. (2017). Coordinated Production Planning and Scheduling Problem in a Foundry. *Archives of Foundry Engineering, 17*(3), 133-138.

Trilling, L. A. (2012). Anesthesiology Nurse Scheduling using Particle Swarm Optimization. *International Journal of Computational Intelligence Systems, 5*(1), 111-125.

Vázquez-rodríguez, J. A. (2010, 12). A new dispatching rule based genetic algorithm for the multi-objective job shop problem. *Journal of Heuristics, 16*(6), 771-793.

Wirsansky, E. (2020). *Hands-On Genetic Algorithms with Python.* Birmingham : Packt Publishing.

Wongwiwat, A. a. (2013). Production scheduling for injection molding manufacture using Petri Net model. *Assembly Automation, 33*(3), 282-293.

Xingjuan Cai, Z. C. (2009). Individual Parameter Selection Strategy for Particle Swarm Optimization. In A. L. (Ed.), *Particle Swarm Optimization* (pp. 89-112). Rijeka: InTech.

Yang, X.-S. (2014). *Nature-Inspired Optimization Algorithms.* London: Elsevier.

Appendix A

```
"""Mold Shop Problem: A Solution Implementation

Main module.  Imports analysis operations and runs them.

Author: Ryan Palo
Date: 4/28/2021
Lewis University
Master's in Computer Science, Intelligent Systems
"""

from analysis import *

if __name__ == "__main__":
    main()


"""Data I/O: Import and export data to other useable formats."""
import csv
from pathlib import Path

from model import Job


def import_csv(filename, base_dir=Path("data/")):
    """Converts CSV files with the relevant data (see columns below) to
    a list of Jobs.
    """
    datafile = base_dir / filename
    with open(datafile, "r", newline="", encoding="utf-8-sig") as csvfile:
        reader = csv.DictReader(csvfile)
        return [
            Job(
                line["part number"],
                int(line["quantity"]),
                float(line["cycle"]),
                int(line["cavities"]),
                float(line["due date"]),
                line["mold"],
                line["material"],
                [int(num) for num in line["machines"].split(",")],
                float(line["setup"]),
                float(line["teardown"])
            ) for i, line in enumerate(reader, start=2)
        ]


def export_csv(schedule, fitness, time_elapsed, filename, base_dir=Path("results/")):
    """Exports a generated schedule to CSV in a format where each machine
    has its jobs listed with start and end dates in order of operation.
    Each machine separated by a blank line.
    """
    outfile = base_dir / filename
    with open(outfile, "w") as csvfile:
        fieldnames = ["part number", "due date", "material", "start", "end"]
        writer = csv.DictWriter(csvfile, fieldnames=fieldnames)
        writer.writeheader()

        for machine in schedule:
            writer.writerow({"part number": f"Machine {machine.number}"})
            for assignment in machine.queue:
                writer.writerow({
                    "part number": assignment.job.number,
                    "due date": assignment.job.due_date,
                    "material": assignment.job.material,
                    "start": assignment.start,
                    "end": assignment.end,
```

```
                })
            writer.writerow({})
        writer.writerow({})
        writer.writerow({
            "part number": "Total fitness:",
            "due date": fitness
        })
        writer.writerow({
            "part number": "Time elapsed:",
            "due date": time_elapsed
        })


"""Scheduler: All of the main logic for generating initial solutions,
generating machine schedules, and the local search algorithms can be
found here.
"""

import random
from math import exp

from model import Job, Assignment, Machine

def stochastic_descent(initial, num_machines, iterations=100, datapoints=None):
    """Stochastic Descent local search algorithm.  Randomly generate a
    neighbor state, and, if it's better than our original, update to that
    state.  Essentially greedy.  Prone to getting stuck in local minima.
    """
    current = initial
    current_fitness = dna_fitness(current, num_machines)

    for i in range(iterations):
        a, b = random.sample(range(len(initial)), 2)
        current[a], current[b] = current[b], current[a]
        new_fitness = dna_fitness(current, num_machines)
        if new_fitness < current_fitness:
            current_fitness = new_fitness
        else:
            current[a], current[b] = current[b], current[a]

        if datapoints is not None and i in datapoints:
            datapoints[i] = current_fitness

    return generate_schedule(current, num_machines), current_fitness


def simulated_annealing(initial, num_machines, t0, alpha, iterations, datapoints=None):
    """Simulated annealing local search.  Stochastic descent but with a geometrically
    reducing probability of sometimes accepting worse neighbor solutions.
    Allows the algorithm to escape bad wells early and have a better chance
    of finding the global minimum before diving down towards it.
    """
    current = initial
    current_fitness = dna_fitness(initial, num_machines)
    t = t0
    for i in range(iterations):
        a, b = random.sample(range(len(initial)), 2)
        current[a], current[b] = current[b], current[a]
        new_fitness = dna_fitness(current, num_machines)
        if new_fitness < current_fitness:
            current_fitness = new_fitness
        elif random.random() < exp((current_fitness - new_fitness)/t):
            current_fitness = new_fitness
        else:
            current[a], current[b] = current[b], current[a]

        t *= alpha

        if datapoints is not None and i in datapoints:
            datapoints[i] = current_fitness
```

```
    return generate_schedule(current, num_machines), current_fitness


def generate_initial_solution(jobs):
    """Generates an initial solution that is sorted by due date and duration."""
    return sorted(jobs, key=initial_sort_key)


def generate_random_solution(jobs):
    """Generates an initial solution that is a random shuffling of jobs."""
    result = jobs[:]
    random.shuffle(result)
    return result


def initial_sort_key(job):
    return (job.due_date, job.duration)


def schedule_fitness(schedule, num_machines):
    """Calculate the fitness of a given schedule.

    Every day late for any job is +1 to fitness (big fitness is bad,
    this is a minimizing problem).  Every material switch is +0.5 days penalty.
    """
    penalty = 0
    for machine in schedule:
        loaded_material = None
        for assignment in machine:
            penalty += max(assignment.end - assignment.job.due_date, 0)
            if assignment.job.material != loaded_material:
                if loaded_material is not None:
                    penalty += 0.5
                loaded_material = assignment.job.material
    return penalty

def dna_fitness(solution, num_machines):
    """Given a DNA strand of jobs to work on, generate a schedule and
    return the fitness score of that schedule.
    """
    schedule = generate_schedule(solution, num_machines)
    return schedule_fitness(schedule, num_machines)


def generate_schedule(solution, num_machines):
    """Greedy algorithm.  Go through the list and assign each job to the
    first available machine that can work on it.

    Returns:
        list(Machine): Each machine with its queue (Machine 1 is 0th index)
    """
    machines = [Machine(i + 1) for i in range(num_machines)]
    for job in solution:
        machine = min(machines[i - 1] for i in job.machines)
        if not machine.queue or machine.queue[-1].job.mold != job.mold:
            changeover = True
            start_date = machine.open_date
        else:
            start_date = machine.open_date - machine.queue[-1].job.teardown
            changeover = False
        machine.add(Assignment(job, machine.number, machine.open_date, changeover))
    return machines


"""Model: the data model for the analysis.

A Job holds all of the input data from the user, including what is required
and when.  (NOTE: Base units are *days* so all hourly values must be divided
by 8.)
```

```
An Assignment takes a job and includes info about which machine that job
should run on and when.

A machine keeps track of its own queue as well as the date that it comes
open.
"""


class Job:
    def __init__(self, number, qty, cycle, cavities, due_date, mold, material, machines,
setup, teardown):
        self.number = number
        self.due_date = due_date
        self.mold = mold
        self.material = material
        self.machines = machines
        self.duration = qty * cycle / (3600 * 8 * cavities) + setup / 8 + teardown / 8
        self.setup = setup / 8
        self.teardown = teardown / 8

    def __repr__(self):
        return            f"Job(number={self.number},            due_date={self.due_date},
material={self.material})"


class Assignment:
    def __init__(self, job, machine, start, changeover):
        self.job = job
        self.machine = machine
        self.start = start
        self.end = start + job.duration
        if not changeover:
            self.end -= job.setup

    def __repr__(self):
        return            f"Assignment(job={self.job},            machine={self.machine},
start={self.start:.2f}, end={self.end:.2f})"


class Machine:
    def __init__(self, number):
        self.open_date = 0
        self.number = number
        self.queue = []

    def add(self, assignment):
        self.queue.append(assignment)
        self.open_date = assignment.end

    def __lt__(self, other):
        return self.open_date < other.open_date

    def __iter__(self):
        return iter(self.queue)

    def __repr__(self):
        return f"Machine(number={self.number},assignments={self.queue})"


"""Analysis: a quickly changing module containing analysis functions
for outputting data for plotting.  The goal was to enable me to reuse
functionality during multiple tests, but a lot of the tests required
small tweaks to the functions as I went, so not all analysis cases
are correct or well-preserved all the time.
"""

import csv
import time
from pathlib import Path
import random
```

```
from data_io import import_csv, export_csv
from     scheduler     import     generate_initial_solution,     generate_random_solution,
stochastic_descent, dna_fitness, simulated_annealing

TESTCASES = ["baseline", "heavy", "late", "light"]


def run_csv_testcase(name, algorithm):
    """Helper function to run a single test case using default i=100"""
    if name not in TESTCASES:
        raise ValueError(f"{name} not in {TESTCASES}")
    start_time = time.perf_counter()
    jobs = import_csv(f"{name}-jobs.csv")
    initial = generate_random_solution(jobs)
    result, fitness = algorithm(initial, 8)
    export_csv(result, fitness,  time.perf_counter() - start_time, f"{name}-output.csv")


def stochastic_descent_stop_iter_experiment():
    """Test out what stop iteration value works best by trying a bunch and
    seeing where the benefits stop paying off.
    """
    resultfile = Path("results/stochastic-descent-stop-iters.csv")
    with open(resultfile, "w") as csvfile:
        fieldnames = ["testcase", "iteration", "fitness"]
        writer = csv.DictWriter(csvfile, fieldnames=fieldnames)
        writer.writeheader()
        iterations = [5, 10, 50, 100, 500, 1000, 5000, 10000, 50000, 100000, 500000, 750000,
1000000]
        for testcase in TESTCASES:
            for trial in range(5):
                print(f"Running trial {trial + 1} of '{testcase}' case.")
                datapoints = {it: 0 for it in iterations}
                start_time = time.perf_counter()
                jobs = import_csv(f"{testcase}-jobs.csv")
                initial = generate_random_solution(jobs)
                _result, _fitness = stochastic_descent(initial, 8, 1000001, datapoints)
                stop_time = time.perf_counter()

                for iteration, fitness in datapoints.items():
                    writer.writerow({
                        "testcase": testcase,
                        "iteration": iteration,
                        "fitness": fitness,
                    })


def simulated_annealing_stop_iter_experiment():
    """Test out what stop iteration value works best by trying a bunch and
    seeing where the benefits stop paying off.
    """
    resultfile = Path("results/simulated-annealing-stop-iters.csv")
    with open(resultfile, "w") as csvfile:
        fieldnames = ["testcase", "iteration", "fitness"]
        writer = csv.DictWriter(csvfile, fieldnames=fieldnames)
        writer.writeheader()
        iterations = [5, 10, 50, 100, 500, 1000, 5000, 10000, 50000, 100000, 500000, 750000,
1000000]
        for testcase in TESTCASES:
            for trial in range(5):
                print(f"Running trial {trial + 1} of '{testcase}' case.")
                datapoints = {it: 0 for it in iterations}
                start_time = time.perf_counter()
                jobs = import_csv(f"{testcase}-jobs.csv")
                initial = generate_random_solution(jobs)
                _result, _fitness = simulated_annealing(initial, 8, 10000, 0.89, 1000001,
datapoints)
                stop_time = time.perf_counter()
```

```
                for iteration, fitness in datapoints.items():
                    writer.writerow({
                        "testcase": testcase,
                        "iteration": iteration,
                        "fitness": fitness,
                    })


def simulated_annealing_t0_calculation():
    """Try to find a good initial temperature for SA."""
    resultfile = Path("results/sa-t0.csv")
    with open(resultfile, "w") as csvfile:
        fieldnames = ["random delta", "sorted delta"]
        writer = csv.DictWriter(csvfile, fieldnames=fieldnames)
        writer.writeheader()

        jobs = import_csv("baseline-jobs.csv")
        for _ in range(1000):
            random_initial = generate_random_solution(jobs)
            sorted_initial = generate_initial_solution(jobs)
            random_f0 = dna_fitness(random_initial, 8)
            sorted_f0 = dna_fitness(sorted_initial, 8)
            a, b = random.sample(range(len(random_initial)), 2)
            random_initial[a], random_initial[b] = random_initial[b], random_initial[a]
            sorted_initial[a], sorted_initial[b] = sorted_initial[b], sorted_initial[a]
            random_f1 = dna_fitness(random_initial, 8)
            sorted_f1 = dna_fitness(sorted_initial, 8)
            writer.writerow({
                "random delta": random_f0 - random_f1,
                "sorted delta": sorted_f0 - sorted_f1,
            })


def simulated_annealing_alpha():
    """Try to find a good alpha for SA"""
    jobs = import_csv("late-jobs.csv")
    outfile = Path("results/sa-alpha-late.csv")
    with open(outfile, "w") as csvfile:
        writer = csv.DictWriter(csvfile, fieldnames=["alpha", "fitness", "final temp"])
        writer.writeheader()

        for alpha in [val / 100 for val in range(60, 100)]:
            print("testing alpha:", alpha)
            initial = generate_initial_solution(jobs)
            for _ in range(30):
                schedule, fitness, t_final = simulated_annealing(initial, 8, 10000, alpha,
10000)
                writer.writerow({"alpha": alpha, "fitness": fitness, "final temp":
t_final})


def test_greedy_only():
    """Check and see if the initial solution is noticably worse that
    the SD solution after iterations."""
    for testcase in TESTCASES:
        jobs = import_csv(f"{testcase}-jobs.csv")
        initial = generate_initial_solution(jobs)
        print(f"'{testcase}' initial solution fitness: {dna_fitness(initial, 8)}")


def run_simulated_annealing():
    """Quick run of SA to see if it works."""
    for testcase in TESTCASES:
        jobs = import_csv(f"{testcase}-jobs.csv")
        initial = generate_initial_solution(jobs)
        schedule, fitness, _t = simulated_annealing(initial, 8, 10000, .89, 750)
        print(testcase, fitness)
```

```
def compare_sa_sd_with_random_and_sorted_initials():
    """Seeing whether random or sorted starting conditions work better."""
    iteration_values = [100, 500, 1000, 5000, 10000, 50000, 100000]
    resultfile = Path("results/sa-sd-random-and-sorted-baseline.csv")
    with open(resultfile, "w") as csvfile:
        fieldnames = ["algorithm", "testcase", "start type", "iteration", "fitness"]
        writer = csv.DictWriter(csvfile, fieldnames=fieldnames)
        writer.writeheader()

        sd = lambda initial, datapoints: stochastic_descent(initial, 8, 100001, datapoints)
        sd.name = "SD"
        sa = lambda initial, datapoints: simulated_annealing(initial, 8, 10000, .89, 100001,
datapoints)
        sa.name = "SA"
        generate_initial_solution.name = "sorted"
        generate_random_solution.name = "random"
        for testcase in TESTCASES:
            if testcase != "baseline":
                continue
            for algo in [sd, sa]:
                for startcase in [generate_initial_solution, generate_random_solution]:
                    print(f"Starting    '{testcase}'    with    '{startcase.name}'    and
'{algo.name}'...")
                    averages = {it: 0 for it in iteration_values}
                    row = {
                        "algorithm": "SD" if algo == sd else "SA",
                        "testcase": testcase,
                        "start type": "random" if startcase == generate_random_solution
else "sorted"
                    }
                    for sample in range(3):
                        fitnesses = {it: 0 for it in iteration_values}
                        jobs = import_csv(f"{testcase}-jobs.csv")
                        initial = startcase(jobs)
                        result, fitness = algo(initial, fitnesses)
                        for it, score in fitnesses.items():
                            averages[it] += score
                        print(sample)

                    for it, fitness in averages.items():
                        row["iteration"] = it
                        row["fitness"] = fitness/3
                        writer.writerow(row)


def final_results():
    """Generating the final results data for all testcases."""
    resultfile = Path("results/final_results.csv")
    with open(resultfile, "w") as csvfile:
        fieldnames = ["algorithm", "testcase", "fitness", "duration"]
        writer = csv.DictWriter(csvfile, fieldnames=fieldnames)
        writer.writeheader()

        sd = lambda initial: stochastic_descent(initial, 8, 50000)
        sa = lambda initial: simulated_annealing(initial, 8, 10000, .89, 100000)
        for algo in [sd, sa]:
            for testcase in TESTCASES:
                jobs = import_csv(f"{testcase}-jobs.csv")
                fitnesses = []
                durations = []
                row = {"algorithm": "SD" if algo == sd else "SA", "testcase": testcase}
                for trial in range(5):
                    print(f"'{row}' trial {trial + 1}...")
                    start_time = time.perf_counter()
                    if testcase == "light":
                        initial = generate_initial_solution(jobs)
                    else:
                        initial = generate_random_solution(jobs)
                    _result, fitness = algo(initial)
                    row["duration"] = time.perf_counter() - start_time
```

```
row["fitness"] = fitness
writer.writerow(row)
```

BIBLIOGRAPHY[1]

Ryan Palo

Candidate for the Degree of

Master of Science

Thesis: TYPE FULL TITLE HERE IN ALL CAPS

Major Field: Computer Science

Biographical:

Personal Data:

Education: (prior degrees)

"Full Name" has completed the requirements for the Master of Science in Computer Science at Lewis University, Romeoville, Illinois, in "Month", "Year".

ADVISER'S APPROVAL: Type Adviser's Name Here

[1] IF NECESSARY (should not exceed one page except for PhDs)