December 15, 2021

“Variety Packs with Constraints and Preferences”

CS4100 Artificial Intelligence Final Project Report

James Packard (packard.ja@northeastern.edu)
Hussain Khalil (khalil.h@northeastern.edu)

CS4100: Artificial Intelligence
Professor Stacy Marsella

Northeastern University,
Boston, MA

ABSTRACT

In this project, we attempt to apply an automated approach to the weighted constraint satisfaction problem of inventory assignment. We construct a class-based model in TypeScript to represent the problem including customer allergies and preferences, inventory quantities and assignments, a representation of problem states, a function to evaluate states based on a number of features and a configurable method to automatically generate thousands of sample problems. From this model, we implement and compare several different methods of finding high-scoring assignments of products to customers, including a k-beam hill-climbing local search with simulated annealing, as well as two methods designed to provide baseline performance: a random assignment agent to provide baseline scores, and an exhaustive graph search agent producing an optimal assignment in exponential time. With careful tuning of local search parameters, we achieve near optimal assignments in polynomial time.

1 INTRODUCTION

1.1 Motivation

The motivating problem is as follows: a seafood business sells a variety pack containing perishable food items to customers each week following a subscription-based model. These items are unknown to the customer at the time of ordering, and are decided on a weekly basis by the business based on the following factors:

- Inventory (availability)
- Market cost
- Customer allergies (hard constraints)
- Customer preferences (soft constraints)

The hard constraints of the problem specify that only what exists in inventory can be assigned to customers, and no customer shall receive products they are allergic to. An optimal assignment would be one that minimizes the market cost of the products assigned and maximizes overall customer satisfaction. Therefore, there is a tradeoff when comparing solutions: choosing an expensive assignment may reduce profit in the short-term, but satisfying customer preferences may improve customer retention and increase profit in the long-term. Finding an optimal assignment through brute-force search takes exponential time and is not feasible for complex scenarios (many products in inventory and customer orders) due to the number of possible product combinations.

1.2 Challenges

1. **Designing a model** that is programmable and extensible to represent the problem, including inventory, customer preferences and constraints, and assignments.
2. **Representing state** and a search graph as a unique assignment of products to customers as a distinct state with successors.

3. **Designing an effective evaluation function** over states that considers consumer preferences, market price, remaining inventory, etc.
4. **Implementing a search algorithm** that is capable of efficiently generating high-scoring assignments comparable to an optimal solution.

1.3 Approaches

As a baseline for comparison, we implement a random assignment agent that unintelligently chooses products to assign to customer orders until all orders are fulfilled, while observing the hard constraints of the problem. This requires us to incorporate those constraints into the model of the problem, which is useful for the AI algorithms implemented later. This agent produces assignments quickly but does not consider customer preferences or product cost, and therefore typically generates low-scoring solutions.

Next, we implement an exhaustive graph search agent that performs a depth-first search on the graph of possible solutions. To do this, we define a state for the problem which contains a partial assignment of products to customer orders, remaining unfulfilled orders, and remaining inventory. Then, we define a method for generating successor states where one additional product has been assigned to a customer order while following the hard constraints of the problem. A state has no successors when all orders are fulfilled or valid inventory runs out (including when customers are allergic to what remains in inventory). The graph search agent traverses all nodes of this graph and returns the state with the highest evaluation, which is guaranteed to be optimal. However, given the vast amount of combinations that arise as the problem grows in size, this agent becomes an infeasible option for generating high-scoring assignments. Thus, we explore near-optimal solutions instead.

Finally we consider local search, a set of non-exhaustive, near-optimal search algorithms that operate without keeping track of paths taken. These types of search algorithms are appropriate for our problem because it is under-constrained; realistically only a small fraction of customers will have allergies, of which the allergies eliminate only a portion of the products. Therefore, local search approaches will be able to consider a large number of potential solutions without running into hard constraints. We implement k-beam local search, which keeps track of the k best states while exploring successors. We then test local beam search with varying amounts for k to see how the solutions of each compare on the same sets of problems. We explore achieving additional performance through the implementation of stochastic randomness to simulate the annealing process.

1.4 Summary of Results

After evaluating graph search and variations of local beam search based on runtime and a solution evaluation function that considers customer satisfaction, assignment cost, and number of unfilled orders, we conclude that

using a narrow-scope (1- or 10-beam) local hill-climbing search with a constant simulated annealing to be best suited for this problem.

2 RELATED WORK

In *Artificial Intelligence: A Modern Approach* by Russel and Norvig, a constraint satisfaction problem (CSP) is defined as a problem that can be factored into a distinct set of variables where a solution has a value assigned to each variable that satisfies all constraints on the variable. In our problem, the items in a customer order are variables and the products are their possible values, or domain. The hard constraints of inventory and customer allergies form the constraints, however the soft constraints of market cost and customer satisfaction result in numerous solutions that differ in optimality. This type of combinatorial problem, where various sets of assignments result in different evaluations, is often described as a Weighted Constraint Satisfaction Problem (WCSP). As the problem of finding an optimal assignment of products to customers can be represented as a WCSP, there is prior research on solving generalized versions of the problem.

Gallaro et al. introduce a technique for solving WCSPs that combines bucket elimination with branch-and-bound in "Solving Weighted Constraint Satisfaction Problems with Memetic/Exact Hybrid Algorithms", approaches from genetic algorithms and local search respectively. Bucket elimination is used to recombine solutions without domain-specific knowledge by having the evolutionary entities cooperate and compete as part of the selection process. This results in a solver that finds optimal solutions to large instances of a known WCSP problem, the Maximum Density Still Life Problem (MDSLP), in reasonable time.

Lee and Leung solve WCSPs by modeling global cost functions as flow networks in "Consistency Techniques for Flow-Based Projection-Safe Global Cost Functions in Weighted Constraint Satisfaction" [2]. By using a minimum cost flow algorithm on the flow network, the researchers achieve polynomial-time solutions that minimize global cost.

3 APPROACH

3.1 Problem Model

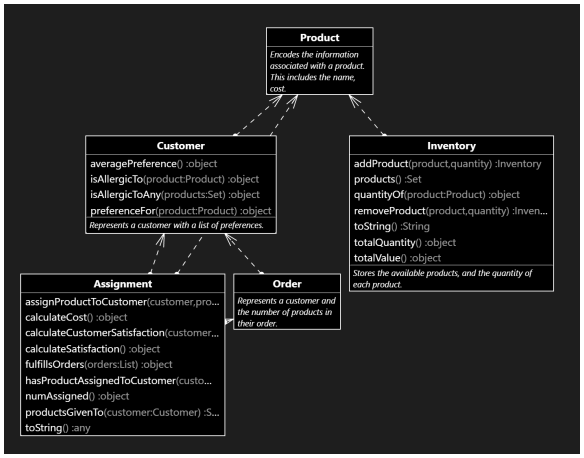
Implementation details

Efficiently modelling the problem to allow generating, evaluating and iterating problem solutions posed a significant challenge during the development phase of the project. Two qualities were identified as being critical to a successful implementation of the problem model:

1. **Efficiency:** the data representation of problems had to allow efficient query operations to facilitate performant search and evaluation operations. Complex problems may include thousands of customers and products, and billions of queries on these instances over the course of solution search.
2. **Tractability and modularity:** in our design of the problem model, we

focused on managing the complexity of the model by using the object-oriented programming paradigm to divide functionality into distinct classes representing cohesive components of the problem model. Instances of each class contained unique representations of relevant data and provided methods to allow queries and write operations. We found that using an object-oriented approach to model our problem increased our efficiency in using, navigating and extending the model during development. Furthermore, we determined that making instances of each class immutable reduced obscure and elusive bugs arising from unintended modification of the problem model.

The problem model was implemented in five classes that encapsulate customer orders, inventory and assignments. TypeScript was selected as an implementation language for the relatively high performance of the V8 JavaScript runtime, rich support for types and interfaces, wealth of libraries that contain solutions for problem and state representation and problem generation, and the authors' own knowledge and familiarity with the language. Each class is described in detail below.



Class diagram showing relationships between problem model classes.

Product class

Instances of the Product class store basic information about unique products in the inventory. This includes the name and cost of the product stored as an immutable dictionary. In a problem instance, Product objects, once instantiated, cannot be modified; state information like quantity and assignments are stored relationally in composite classes. Therefore the Project class, alongside the Customer class, provides an atomic building block for the design of the problem model.

Customer class

The constraints of a problem are encoded in instances of the Customer class, including hard constraints like a customer's allergies to particular Product objects or "soft" constraints including customer preferences of each Product. In particular, customer allergies are stored in a set of products, optimal for binary constraints such as the presence or lack of an allergy to a particular product. Conversely, for weak constraints, a real-number value between

0 and 1 is used to represent the weight of a preference: values greater than 0.5 represent an affinity for a Product instance, 0.5 for the lack of any preference, and values less than 0.5 for a preference to avoid a certain Product. These weak preferences are stored in a hash map keyed on products with real number values. The hash map and set data structures were selected for their read performance characteristics. Improvement to constant or near-constant time lookups for state and problem significantly increases the performance of state evaluations, which require reading the preferences of every Customer for each Product at every state. A single evaluation operation over a large, complex state may require many tens or hundreds of thousands of preferences to be read, limiting the performance of evaluation-based search methods, such as k-beam hill-climbing.

Order class

Instances of the order class contain a tuple of a Customer and a positive integer representing the quantity of the Customer's order. These instances are passed to the state problem and reflect the total remaining quantity of each Customer's order.

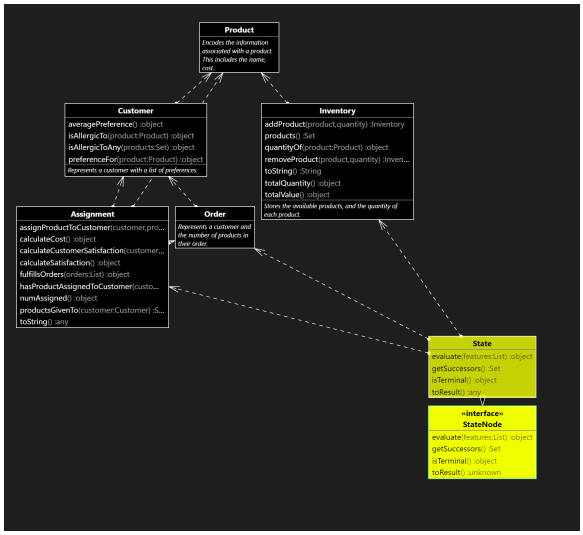
Inventory class

Instances of the Inventory class encode quantity information as a hash map keyed on Products with non-negative integer values. Using a hash map for this purpose results in significantly increased read performance as compared to an Product-Integer tuple array, a critical improvement as both generation and evaluation of each of hundreds of thousands or millions of problem states require reading inventory data from each individual Product. Similarly to the usage of hash maps in the Customer class, read performance is critical for Inventory quantity read performance, as both state generation and evaluation may request billions of map lookups over the course of a solution search on a complex problem.

Assignment class

Instances of the Assignment class represent an assignment of Product instances to Customers, enforcing all hard constraints like Customer allergies, but allowing incomplete assignments with unfilled orders. Assignments are stored as a hash map keyed on Customers with a set of Product as values. As discussed above, this structure increases read performance of Assignments, used for both state generation and evaluation. Assignment instances also support a number of methods which allow queries on assigned quantities (both per-Assignment and per-Customer) and operations such as assigning a Product to a Customer to be carried out. Note that due to the immutable nature of Assignment objects, modifying operations return an updated instance rather than mutating the queried instance.

3.2 State Representation



Extended class diagram including StateNode interface and State implementation

As with the problem model, our state representation was implemented in TypeScript, and is contained in one interface with an implementing class. A StateNode interface and implementing classes contain methods to evaluate the state using a provided set of feature vectors, generate all possible successors to the current state, determine if a state is terminal and extract the problem model from a state.

StateNode implementations are parameterized over a problem model; for the concrete State implementation used for this problem, the problem model is represented as a custom type called ProblemState (shown below) containing an Assignment, Order and Inventory. The principal component of a state is an Assignment instance; however, as the Assignment class only contains a mapping of Customer instances to sets of Product, the concrete state implementation also includes a list of Customer orders and the problem Inventory.

```
export type ProblemState =
RecordOf<{
  assignment: Assignment,
  orders: List<Order>,
  originalOrders: List<Order>,
  inventory: Inventory
}>
```

Components of the ProblemState type, which contains the current Assignment of Products to Customers, the remaining Inventory and a reference to the original Order list.

Generation of successor states is one of the critical functions of the State class. All search strategies select from among the possible successor states, which must follow the rules of valid assignment and assignment order. We carefully designed the successor method of the State implementation to limit the branching factor while eventually permuting over all valid states. In complex problems, there may be thousands of Customer and Product instances, leading to a quadratic number of possible next steps in state generation. We were able to mitigate this problem by limiting the definition of a valid assignment: by adding the additional restriction that all Customer orders must be completely filled, if possible, we can select the first unfilled Order and return an individual state for all possible product assignments to that Order. We also ensure the hard constraint of Customer allergies is satisfied in every assignment. Remaining quantity of Inventory items and Customer orders are maintained by

creating new Inventory and Order instances with updated data.

```
getSuccessors(): Set<State> {
  return this.orders
    .filter(order => order.size > 0)
    .reduce((states, order) => states.concat(
      this.assignableProducts(order.customer)
        .map(product => this.assignProductToOrder(product, order))
      ),
      Set()
    );
}

private assignableProducts(customer: Customer): Set<Product> {
  return this.inventory.products()
    .filter(product =>
      !customer.isAllergicTo(product) &&
      !this.assignment.hasProductAssignedToCustomer(customer, product)
    );
}
```

Implementation of the getSuccessors methods within the State class.

Another central feature of the State interface is the evaluate method. This method is used to score a particular assignment, returning a real number value that can be used to relatively compare solutions to the same problem.

For the purposes of state evaluation, we designed a system of evaluation centered on "feature vectors," which are a set of functions and associated weights and biases. Each feature is represented as a function on states that considers specific aspects of the state and returns a real number value. The value returned is multiplied by an associated weight and added to a bias. The final score of a solution is the sum of all individual feature scores.

```
evaluate(features: List<[(s: ProblemState) => number, number, number]>): number {
  return features.reduce((acc, feature) => {
    return acc + (feature[1] != 0 ? (feature[0](this) * feature[1] + feature[2]) : 0);
  }, 0);
}
```

The evaluate method that returns the sum after computing all features and incorporating weights and biases.

In our usage of the feature vector evaluation system, we found it to provide sufficient flexibility to extract the relevant components of a state and produce useful metrics describing the quality of that state. Since features are described as functions, they allow a nearly unlimited range of possible operations that can be performed using data from a state. The critical flexibility of the system also allowed us to quickly try multiple different evaluation functions, vary weights and experiment with the usefulness of different metrics.

```
export const defaultFeatureVectors:
List<FeatureVector<ProblemState>> =
List([
    [costOfAssignment, -1, 0],
    [valueOfInventory, -1.1, 0],
    [customerSatisfaction, 1, 0],
    [unfilledOrders, -1.1, 0],
]);
```

The features and weights used for scoring states for the purposes of local search and comparison.

However, a significant limitation existing within the feature vector system was in the manual, arbitrary assignment of weights to each feature. This decision, stemming from the nature of the problem and the lack of pre-existing data and results on which to train, is discussed more in the 5.2 “Limitations and Future Work” subsection of the conclusion.

3.4 Problem Generation

To test, optimize and compare the performance of various local search methods versus an exhaustive brute-force search, we needed a large and diverse set of problems on which to run each algorithm.

The goals for problem generation were as follows:

1. Ability to generate tens of thousands of distinct problems to test each algorithm.
2. Ability to adjust parameters of generated problems to ensure all problems were valid, tractable and solvable in a limited amount of time.
3. Ability to reproduce tests to measure relative changes in performance as we iterate and tweak search algorithms.
4. Wide distribution of problems sizes, including number of Customers, customer preferences and allergies, number of Products and size of Orders.

Our solution to this problem was to use the Fast Check library to generate thousands of

random, but reproducible problems with varying inventory sizes, numbers of customers, and customer preferences. Fast Check falls under the category of property-based testing frameworks, which provide utilities to define arbitrary data within given parameters. By building up from primitive data types such as floats, integers, and strings, it became possible to specify the whole problem model as an arbitrary type and utilize the Fast Check library to generate the example problems we needed.

As an illustrative example, allergies are generated by taking a random subset of Product instances while Customer preferences are generated using a random float between -1 and 1 for each Product in the problem. Using these types, arbitrary Customer and Customer Orders can be generated. A similar method is used to define an arbitrary Product, Inventory and finally Problem instance which consists of an arbitrary Inventory and list of Order instances.

During the testing phase, we specified the number of problems for Fast Check to generate using this specification and ran each agent against each problem, recording the following key metrics:

1. The size of the problem, measured by multiplying the number of products and customer orders.
2. The amount of time it took to find a solution to each problem using each agent.
3. The final evaluation of the solution.

Through this method, we generated hundreds of thousands of data points from each of the tested agents allowing us to comprehensively compare the performance of each.

3.4 Random Assignment

The simplest of the types of agents that we tested in our project was the Random Assignment Agent, which simply assigned each Product in a problem Inventory to a random Customer without considering the evaluation of each state, including the preferences of each Customer, the cost of assigned items and remaining inventory. However, hard constraints, such as Customer allergies, are respected.

The mode of assignment is useful for two important reasons. Firstly, it provides a performance baseline for solution search. Due to the simple operation of the Random Assignment Agent, it is expected to return a complete, valid assignment faster than a local or graph search. Secondly, since this agent makes assignments without respect to evaluations, it provides a baseline for solution scores that should be beaten by both an effective local search and a correctly implemented graph search.

3.5 Graph Search

The Graph Search Agent comprises another greedy, yet significantly more complex search strategy: a comprehensive search through all possible assignments. This is achieved by performing a Depth-First Search (DFS) while completely expanding all states in the stack and

storing all successor states. To avoid cycles, the Graph Search Agent stores all previously explored nodes and avoids re-visiting those nodes. Since the Graph Search Agent comprehensively generates all possible assignments over the course of the search, and retains the highest scoring assignment, the algorithm always returns the optimal solution to a problem, and thus provides an optimal upper-bound to compare relative performance against the other agents.

However, as a consequence of the exhaustive, brute-force nature of the Graph Search Agent, it is extremely inefficient, both in memory usage, as it must keep track of all states encountered, and processing time, as it must determine if a new state exists in an expanding set of visited states and may encounter an exponential number of states over the course of a depth-first search.

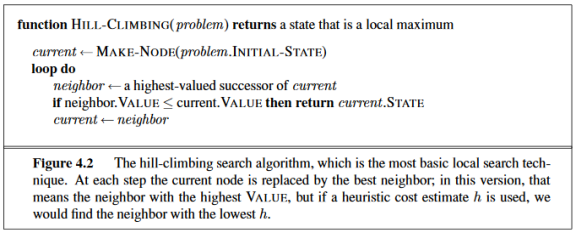
3.6 Local Search

The central focus of our investigation was into the performance of a local search algorithm in finding near-optimal solutions to the weighted constraint-satisfaction problem much faster than the exhaustive brute force search.

Specifically, our objective was to produce scores similar to the exponential-time brute force graph search in only polynomial time, similar to the random assignment algorithm, even in large, complex problems. An additional objective was memory efficiency; an effective local search would use polynomial space in memory in addition to polynomial time.

There are many algorithms that can be used to find solutions to weighted CSP problems; for a complete discussion, see Section 1.3 "Approaches." For the purposes of this project, we focused on the use of local search algorithms, which come in a variety of flavors and allow a significant degree of flexibility, to explore and evaluate as a solution to the problem.

Hill-climbing local search was the specific form of local search that we selected during the investigation. In this local search, at each step, the highest scoring successor is returned. The pseudo-code below, taken from *Russell and Norvig*, illustrates this simple mode of local search.



```
function HILL-CLIMBING(problem) returns a state that is a local maximum
  current ← MAKE-NODE(problem.INITIAL-STATE)
  loop do
    neighbor ← a highest-valued successor of current
    if neighbor.VALUE ≤ current.VALUE then return current.STATE
    current ← neighbor
```

Figure 4.2 The hill-climbing search algorithm, which is the most basic local search technique. At each step the current node is replaced by the best neighbor; in this version, that means the neighbor with the highest VALUE, but if a heuristic cost estimate h is used, we would find the neighbor with the lowest h.

Hill-climbing local search improves on graph search in one key respect: instead of wastefully expanding all possible successors, hill-climbing intelligently selects the one that offers the greatest increase in a solution score. This significantly reduces the time and memory cost of finding solutions by reducing the branching factor to a linear constant.

However, the naïve operation of hill-climbing presents a significant limitation. As the algorithm only selects the single best successor state to the current one, it is possible for the

search to get stuck in a local maxima. This is when, in one particular state, all possible neighbors would lead to a reduction in the evaluation, even if going down a particular path would ultimately lead to an improved evaluation, potentially the global maxima. This is illustrated in the following diagram, taken from *Russell and Norvig*.

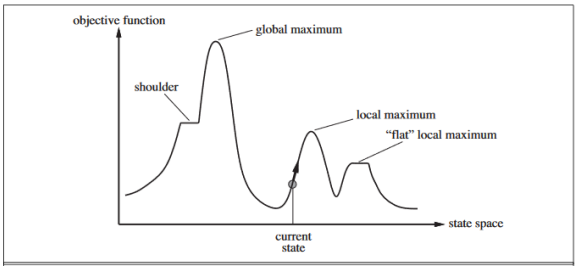


Figure 4.1 A one-dimensional state-space landscape in which elevation corresponds to the objective function. The aim is to find the global maximum. Hill-climbing search modifies the current state to try to improve it, as shown by the arrow. The various topographic features are defined in the text.

k-beam Local Search

We experimented with the implementation of two distinct modifications to hill-climbing local search to overcome this limitation. The first of the two is *k*-beam local search, in which the *k* best successors to the current state are selected, rather than the single one from before. At each step, all successor states are generated from the *k* successors and the new generation is selected by pruning that list. This technique can potentially overcome the local-maxima limitation of naïve hill-climbing as it considers less-than-best neighbors throughout the search, potentially allowing the algorithm to descend from local maxima when it is optimal to do so.

In our project, this step is accomplished using a max-heap priority queue that maintains the frontier of unexplored neighbors. At each step, the successors of the *k* states in the priority queue are generated, evaluated and enqueued, and the queue is pruned to the best *k* states. At all times, the best state encountered during the search is stored, and after each evaluation, this is updated if a new state exceeds the evaluation of the previous best. Finally, when there are no more states remaining in the frontier, the best state is returned as the solution.

Simulated Annealing (*i.e., heat*)

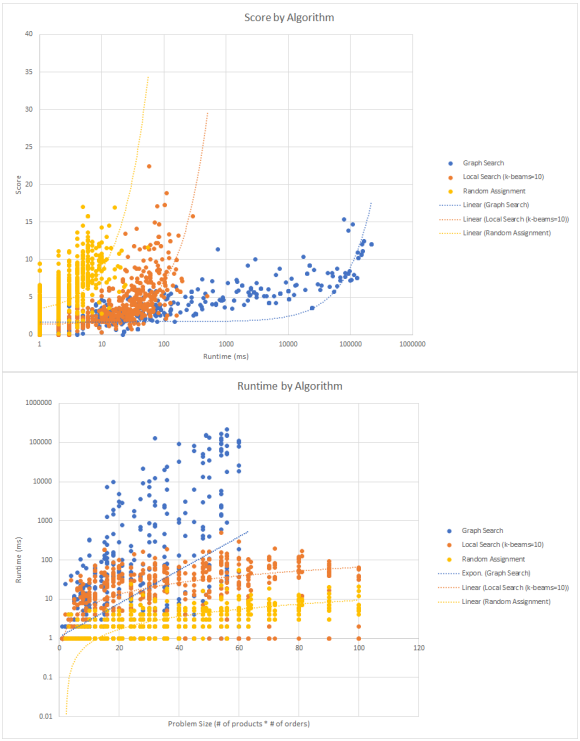
Another modification to hill-climbing we explored was the introduction of stochastic randomness into the evaluation of states. This process, known as simulated annealing, or heat, aims to balance the inefficient but potentially more optimal method of randomly sampling states with the efficiency of hill-climbing by introducing a random perturbation to the evaluations of states. Doing so can allow local search to discover global maxima faster by rapidly "jumping" around the state space and discovering global trends rather than copiously focusing on a local section of the space.

The simplest form of simulated annealing involves a constant "heat" that is maintained throughout the search. For each state, a value is randomly generated with an absolute value bounded by the heat and is added to the state's evaluation. Alternatively, a high amount of heat can be used in the beginning of the search and, as more states are explored, can be reduced as the search approaches a global maxima. Yet another approach involves keeping track of the time since the best solution was improved and varying the heat directly with this

measurement. This approach aims to stimulate the search when it gets stuck at a local maxima, potentially helping it to discover better solutions. All three strategies can be implemented alongside *k*-beam local search; we implement each and compare the performance in Section 4, "Results."

4 RESULTS

To compare the performance of random, graph search, local search, and hill-climbing, we generated a set of problems for every agent to solve (see section 3.4 "Problem Generation"). For each problem, we recorded the number of products and orders, the runtime duration of each agent, and the evaluation of each agent's outcome.



Early results from test runs showed Random Assignment Agent outperforming Graph Search Agent in some problems (note that runtime is displayed using a logarithmic scale).

In our initial run, we discovered a bug in state successor generation that was leading to some possibilities not being explored by local and graph search agents. In some cases, this led to the unexpected result of having the exhaustive and supposedly optimal Graph Search Agent producing lower-scoring solutions than Random Assignment Agent despite taking exponentially longer to run.

The problem existed with state successor generation, which did not create successor states for each order that a product could be assigned to. Instead, it worked through each order iteratively until all were fulfilled. This bug was resolved for future runs.

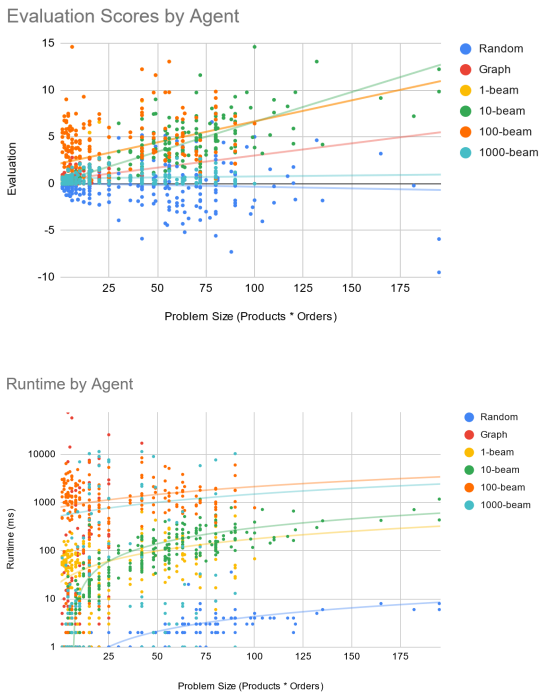
In a second test run, we ran six algorithms and compared the performance of each:

1. Random Assignment Agent
2. Graph Search Agent
3. naïve hill-climbing Local Search Agent (1-beam)
4. 10-beam Local Search Agent
5. 100-beam Local Search Agent
6. 1000-beam Local Search Agent

Data from this run supported our hypothesis that an optimal graph search requires an exponentially increasing runtime with respect to problem size (measured by multiplying the number of products with the number of customer orders). In fact, for all except trivial problems with single-digit complexity, Graph

Search Agent terminated without producing an optimal solution due to memory and time constraints.

We also observed that Local Search Agent performed very well in all configurations. In problems where we were able to compute optimal solutions using Graph Search (note, this is very difficult to see on the graph; this problem is discussed in 5.2 “Limitations”), Local Search performed nearly as well, and in some cases, achieved identical performance. This performance came at a significantly reduced time cost. While the time to run Graph Search grew exponentially with problem complexity, Local Search in all configurations grew linearly. Lower-order, including 1- and 10-beam local search in particular, achieved exceptional performance, finding a near-optimal solution in seconds even for very complex problems (tens of thousands of visited states).



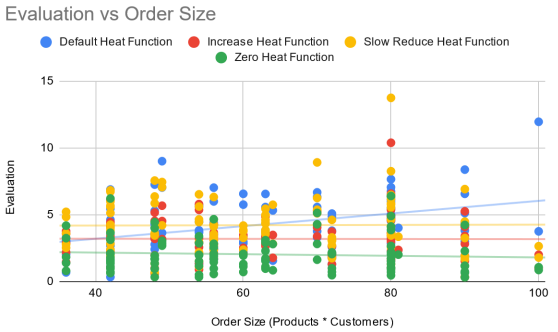
Results from the second test run indicate Local Search performs similarly to Graph Search in linear time.

Unexpectedly, 100- and 1000-beam local search showed some, but insignificant improvement over naïve (*i.e.,* 1-beam) local search, possibly due to the nature of the problem, despite a significant increase in runtime (though still linear to problem size). This could stem from a lack of local maxima in our state space as adding an item to a customer’s order nearly always results in an increase in the evaluation of a state. A key factor in this may be that our method of problem generation produces “under-constrained” problems. In a more constrained problem, a naïve hill-climbing local search may greedily select high-satisfaction products for a customer but have nothing to give to a customer with allergies later on. In

many of the problems this condition does not occur, so naïve hill-climbing appears to perform nearly as well as *k*-beam local search.

Additionally, as discussed in section 3.6 "Local Search," we explored the additional optimization of simulated annealing, or "heat", which may improve the performance of hill-climbing by rapidly exploring the state space.

Implementation of this optimization yielded consistent improvements relative to a zero-heat local search, which uses evaluations directly without perturbation. Over a hundred test problems, we saw significant differences between each method of simulated annealing, with close to a 100% increase in evaluation score from zero-heat. Among the heat functions used, the Default Heat Function, which applied a constant heat throughout the local search, performed best on average, followed closely by Slow Reduce Heat Function, which starts problems with a high heat that is gradually reduced as the search progresses. In implementing the Increasing and Decreasing heat functions, we faced a problem of assessing the progress of a search: for example, some searches conclude after a few dozen states, while some may explore up to a hundred thousand states before returning a solution; as it is difficult to assess the difficulty of a problem while solving it, the performance of these heat functions may have been limited by reducing or increasing heat too quickly or slowly. Further investigation is required to identify optimal heat functions as they may offer significant improvements in local search performance.



Results from the simulated annealing test show relative increases in performance from a zero-heat baseline, with a constant heat function performing best.

5 CONCLUSIONS AND FUTURE WORK

In our results, we discover that using graph search to find an optimal solution to the problem of assigning products to variety packs is infeasible due to time and memory constraints. Instead, performing local beam search with a small number of beams is optimal or nearly optimal with a far shorter runtime. We believe that the effectiveness of local beam search in this problem results from the large space of valid solutions given few constraints. Considering the problem as defined and approached in our investigation, we find that using 1- or 10-beam local hill-climbing search with constant simulated annealing achieves the most effective balance of runtime, memory, and near-optimality of solutions. However, we recognize several limitations in this project and propose some improvements for future work.

5.2 Limitations and Future Work

Due to runtime limitations, we had to terminate search agents early that exceeded 100,000 explored states. This mostly affected Graph Search, which would often encounter this limitation on larger problems. Thus, data points are omitted where agents could not completely solve a problem.

We also believe that a lack of historical data restricts the relevance and applicability of our results to real-world scenarios since our randomly generated problems may not be representative of the actual distribution of customer preferences, allergies, and inventory sizes. The use of historical order, inventory, and customer satisfaction data would benefit both the accuracy of the problem representation and the evaluation of solutions. Currently, solutions are evaluated based on the sum of satisfied customer preferences, assignment cost, and number of unfulfilled orders, each with arbitrary weight. Historical data could be used to adjust these weights for greater customer retention, which is a better indicator of long-term profitability.

To explore additional solutions, the problem could be transformed into a format accepted by existing WCSP solvers, such as a cost function network. The performance of these solvers could then be directly compared against those demonstrated in this paper to determine whether the problem warrants more advanced optimization.

6 ACKNOWLEDGEMENTS

The authors of this paper would like to thank Professor Stacy Marsella, who guided us towards research in weighted constraint satisfaction problems. We also extend our gratitude to Nutchanon Yongsatianchot and Tanmay Khokle, who helped to refine the problem formulation and approach, and to Sven Olson, a professor of business at Northeastern who introduced us to this problem.

REFERENCES

J. E. Gallardo, C. Cotta, and A. J. Fernández, "Solving Weighted Constraint Satisfaction Problems with Memetic/Exact Hybrid Algorithms," *Journal of Artificial Intelligence Research*, vol. 35, pp. 533–555, Jul. 2009, doi: 10.1613/jair.2770.

J. H. M. Lee and K. L. Leung, "Consistency Techniques for Flow-Based Projection-Safe Global Cost Functions in Weighted Constraint Satisfaction," *Journal of Artificial Intelligence Research*, vol. 43, pp. 257–292, Feb. 2012, doi: 10.1613/jair.3476.

R. J. Stuart and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, 2010.

APPENDIX

1. Project Source Code Repository: github.com/packard-j/ai-project
   (.zip file attached)
2. Variety Packs: Test Data Spreadsheet
   (.xlsx file attached)