Derek Thomas and Derek Tran (joined 04/30)
Stephen Hutt
30 April 2019
CSCI 3202

Practicum Report

Problem Description

Square foot gardens are a method of gardening in a very compact and space efficient way[1]. Unlike more traditional methods where plants are planted in seperate rows, in square foot gardening plants are planted together in densely packed grids. While this method has it's advantages it also introduces a new set of issues. Shorter plants will be shaded by taller plants if they are behind it with respect to the sun, stunting their growth. Additionally some plants should not be planted next to others because they compete for nutrients or release chemicals harmful to those plants. Conversely some plants will benefit by being planted adjacently, either by repelling pests attracted to the other plants or anecdotally improving growth, flavor, or other aspects.

While an average gardener may not give much consideration to the layout of the garden bed, it would be best if the layout satisfied certain constraints. Chiefly:

1. No plant should be shaded by another plant (i.e. they are arranged shortest to tallest in the direction the sun faces the garden)
2. Plants harmful to one another should not be planted adjacently

Exhaustively searching for such garden plans is impractical. A single 4'x8' garden bed with 6 different types of plants can have upwards of 10 million arrangements. The number of arrangements grows factorially with both the size of the bed and number of types of plants to be planted. However, finding these arrangements are tractable as a constraint satisfaction problem.

Description of Methods

Constraint satisfaction problems (CSP) consist of three components[4].

1. A set of variables
2. A set of domains
3. A set of constraints

In the context of finding satisfactory garden plans, the set of variables are the squares in which plants can be planted. The set of domains are the subsets of plants that can be planted in those squares of the set of plants that the gardener wants to put into the entire garden. The constraints are the relative height between the plants occupying two adjacent squares and whether or not those two plants are allowed to be planted adjacently, both of which are binary constraints.

Naïvely solving the CSP as laid out above quickly reveals a problem: the resulting garden plan will be a pseudo-random selection of the plants the gardener wants to plant in the garden. There are no guarantees about the number of each type that will be selected. This may result in several of the plants selected to be planted not being assigned in the garden plan or that a large (and unsatisfactory) portion of the garden plan is assigned one type of plant.

So we will implement an additional global constraint:

3. The number of plants for each type must be equal to the number specified by the gardener.

The variables for this CSP are the square feet of planting area, with each square being assigned one plant in the solution. This is stored as a 2d numpy array and can be of an arbitrary size. The array is filled with '0' to represent plantable space and '-1' to represent non plantable space, e.g. walking paths between beds. While '-1' was not implemented in the examples, the code to remove cells with '-1' from the list of variables and from the set of adjacent variables exists. This can be considered a unary constraint on the variable, that the '-1' type variables be assigned only an empty value. In addition to the variables a second set, adjancent_vars was created to represent to the CSP solver how the variables are connected to each other for determining arc-consistency. The adjacent_vars are the directly adjacent variables (up, down, left, right, *not* diagonal), within the bounds of the plant bed.

The domain is a dictionary in the form {(x,y) : [Plants]}. In order to get the data for the Plant objects, we scraped data from Burpee.com[2] and Wikipedia[5] respectively. Burpee.com was the source of the attributes of each plant such as name, time needed in the sun, height range, etc. from a predefined list of urls. The data was then exported to a json file to be used later. The only data that is used by the CSP is the name and height, but we were unsure what additional data might be used at a later point and opted to grab all of it at once. The Wikipedia page detailing the list of companion plants was also scraped in a similar fashion and exported to a json file. This was the source of data for which plants could not be planted adjacently. Again there was extra data gathered that was not used in the practicum but may be useful later.

With the data comes the constraints. Mentioned briefly before, we constrain each variable in the garden bed space to order the plants from shortest to tallest. Another constraint come with the companion plant relationships where we must identify what plants should not grow next to each other. Typically in a normal CSP, once we reach a goal state where the constraints are satisfied, we are "done" however that is not the case with this particular problem. Derek has a list of plants needed to be planted. Using those in a different dictionary with the name as a key and the amount needed to plant as the value, we fully satisfy this constraint if all plants from here are planted at least; meaning if we get a fully filled garden bed but not all the plants are utilized, we have failed to satisfy our constraints.

The two core components of a CSP solver are inference over constraints, and search. Inference is used to remove values from variable domains when those values would violate arc-consistency. For some CSP, like simple sudoku problems, repeated applications of inference is sufficient to find a solution. Harder problems such as our garden plans are not so lucky. Even after repeated applications of inference there are still many possible values for each variable.

A recursive backtracking search is required to find a solution. This search is a recursive depth first search. It assigns a value to a variable and then checks consistency, if it is consistent it moves on to the next variable and assigns another value either until all variables have been assigned values or until an assignment is not consistent. In the case of the later the search "backtracks" to a previous variable and assigns a different value.

It has with two key features though which set it apart from standard depth first search: variable and value selection order. Rather than simply iterating on the next variable or value in the list, a heuristic is used. In our solver we implemented minimum-remaining-values and degree heuristic for selecting the next variable and least constraining value for selecting the next value.

Minimum-remaining-values selects the variable with the fewest values left in its domain, leading to a "fail-fast" search behavior. If a variable has no legal values remaining then that variable is selected first and immediately returns failure. This prunes off invalid search branches very quickly. Since minimum-remaining-values is not always useful, e.g. when many variables have the same number of values (such as the beginning of the search), a degree heuristic is used in conjunction. This heuristic selects the variables with the most unassigned neighboring variables. This can significantly reduce the branching factor of the search tree.

For values a different approach to "fail-fast" is taken: the least-constraining-value. This heuristic selects the value that imposes the fewest constraints on the neighboring unassigned variables. This leaves maximum flexibility for future variable assignments. Our implementation of backtrack search uses all three heuristics and the performance improvements can be seen in our results.

Results

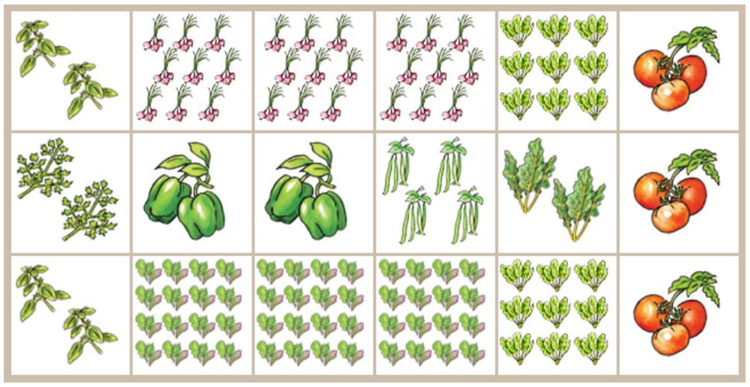
Our solver successfully finds solution to a wide variety of garden layouts and plant selection with both different constraints of sun facing and avoiding certain plant type adjacencies. Upon finding a solution the solver outputs a dictionary of the coordinates of each square foot and their respective assigned plant, along with a 2d array of the garden bed showing the maximum heights per square for easy human verification of the height layout.

```
[[20 14 20 14 20 14 14 20]
 [14  8 14  8 14  8  8 14]]
```

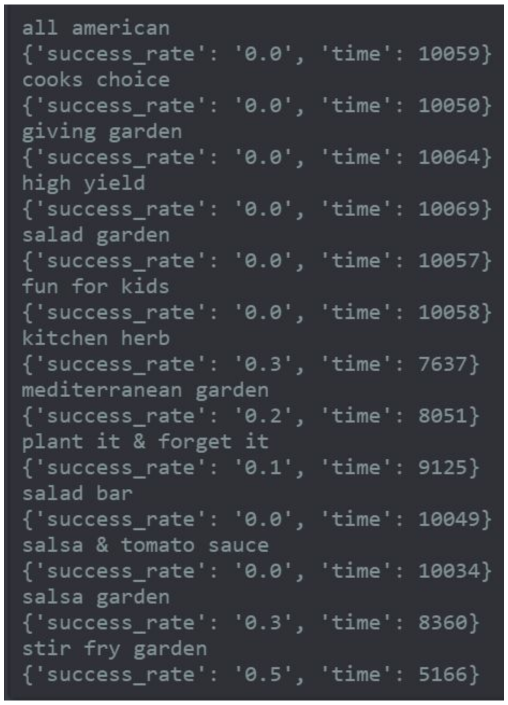
An example output array satisfying a south facing height constraint

In order to verify if this was correct, we examined the output array and the dictionary. We checked if the plant at each respective coordinate in the dictionary matched the maximum height expected. We also checked if the array output correctly arranged the heights of the plants based on what the sun direction was set to be. For instance, the default direction ran would be to have the sun shine from the south because we are in the Northern Hemisphere, thus making the sun always coming from the south in some degree. Thus we expect hour heights (only from south -> north) to be from shortest to tallest.

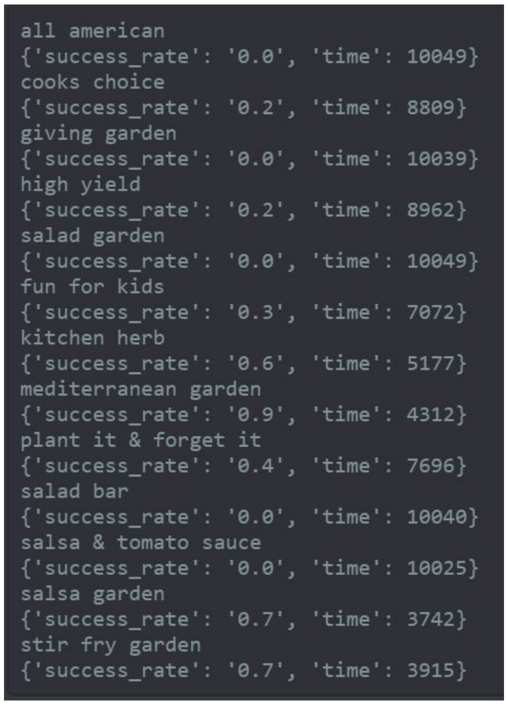
To exhaustively test our solver we grabbed some pre-planned square foot garden beds from gardeners.com[3]. We found that this would end up being a useful way to test out how well the garden bed CSP solver would perform with different initializations of garden bed spaces and required plant types. For each scenario given, we would go a simulate what happens to the garden if the current scenario of plants is initialized to be a certain way. For instance, if the garden bed were to be initialized with several locations with 3 corn, 2 peas, 5 tomatoes, would the AI be able to use the CSP problem given to determine if a garden bed would be possible to optimize with? Not all of these given scenarios would work out though with our given constraints. For example, the website shows this as the "All American" pre-planned garden:



Based on this picture, we could go into the dictionary containing the type and number of plants, edit that to have to requirement of plants given the picture i.e. 3 tomatoes, 2 bell peppers, etc., and see if the CSP solver would be able to produce an optimized garden based on the necessary plants. With the "All American" set up however, with 1 garden bed it was not possible to do so. This is not surprising as these pre-planned were most likely meant to be aesthetically pleasing rather than optimized for growth with sunlight. A fix for this would be including multiple garden beds to optimize the number of plants needed to be planted. The more garden beds added in, the more likely we would be able to create an optimized garden because we would essentially be loosening the constraints. Rather than focus on constraining one garden bed a much as possible, there is more wiggle room with multiple beds to get the necessary plants put down; the main issue is that they wouldn't be in the same bed obviously but the CSP solver is designed to just satisfy based on height and neighbor relations rather than account for any aesthetically nice to look at bed. The CSP solver was able to optimize some scenarios with a single garden bed by default however. Implementing minimum remaining values, degree heuristic, least constraining value significantly reduced the search time for solutions to these scenarios. Allowing the CSP solver to find some solutions it was unable to given the time constraints in place.



```
all american
{'success_rate': '0.0', 'time': 10059}
cooks choice
{'success_rate': '0.0', 'time': 10050}
giving garden
{'success_rate': '0.0', 'time': 10064}
high yield
{'success_rate': '0.0', 'time': 10069}
salad garden
{'success_rate': '0.0', 'time': 10057}
fun for kids
{'success_rate': '0.0', 'time': 10058}
kitchen herb
{'success_rate': '0.3', 'time': 7637}
mediterranean garden
{'success_rate': '0.2', 'time': 8051}
plant it & forget it
{'success_rate': '0.1', 'time': 9125}
salad bar
{'success_rate': '0.0', 'time': 10049}
salsa & tomato sauce
{'success_rate': '0.0', 'time': 10034}
salsa garden
{'success_rate': '0.3', 'time': 8360}
stir fry garden
{'success_rate': '0.5', 'time': 5166}
```



```
all american
{'success_rate': '0.0', 'time': 10049}
cooks choice
{'success_rate': '0.2', 'time': 8809}
giving garden
{'success_rate': '0.0', 'time': 10039}
high yield
{'success_rate': '0.2', 'time': 8962}
salad garden
{'success_rate': '0.0', 'time': 10049}
fun for kids
{'success_rate': '0.3', 'time': 7072}
kitchen herb
{'success_rate': '0.6', 'time': 5177}
mediterranean garden
{'success_rate': '0.9', 'time': 4312}
plant it & forget it
{'success_rate': '0.4', 'time': 7696}
salad bar
{'success_rate': '0.0', 'time': 10040}
salsa & tomato sauce
{'success_rate': '0.0', 'time': 10025}
salsa garden
{'success_rate': '0.7', 'time': 3742}
stir fry garden
{'success_rate': '0.7', 'time': 3915}
```

On the left here we have the results of just running scenarios with random variable and value selection. The right shows adding in minimum remaining values and degree heuristic to further speed up the optimization of the garden bed. The success rate was how many times we were able to find a solution in 10,000 calls to backtrack(), demonstrating we are getting expected performance improvements with these heuristics.

From our testing it seems that the key to a garden bed plan than can be arranged to satisfy the conditions is to have a larger variety of plants the larger the bed, especially if the height constraint is in two directions (e.g. "South & East"). Several of the beds that had solutions with one directional constraint did not with a two-directional constraint. The beds that did have solutions with the two-directional constraint either had a large variety of plants or were smaller, e.g. 2x8 instead of 3x6. Note that while the 2x8 is only 2 sq. ft. smaller than the 3x6, is it 50% smaller along one dimension.

Conclusions

We believe this project was successful on multiple fronts. The solver successfully finds solution garden plans (when they exist) for any given garden bed given sufficient variety of plant types and with multiple different constraints.

There are many ways this can be expanded. We did not touch on placing plants adjacently for beneficial companion planting, mainly because enforcing beneficial plants to be placed adjacently would have imposed overly harsh constraints on the overall problem and only very very few combinations of specific plant types, quantities, and bed sizes would have had even a single solution. However, the problem can be modified from one of constraint satisfaction to constraint optimization, whereby each solution has an associated score and the search does not stop when all variables are assigned while satisfying constraints but when a sufficiently high score or length of time searching has been achieved. Going even further the problem can be expanded to one of scheduling, the garden plan needs to satisfy constraints not just at one moment but throughout the growing season, with some plants being harvested and others being reseeded.

Derek Thomas intends to keep expanding on this project over the summer to help himself plan and manage his own garden by implementing the above constraint optimization and scheduling solvers.

References:

1. Bartholomew, Mel. *All New Square Foot Gardening: the Revolutionary Way to Grow More in Less Space*. Cool Springs Press, 2013.
2. Burpee. 2019. Retrieved from https://www.burpee.com/.
3. Gardener's Supply Company. 2019. Pre-Planned Gardens. Retrieved from https://www.gardeners.com/kitchen-garden-planner/preplanned-gardens.
4. Russell, Stuart J., and Peter Norvig. *Artificial Intelligence: a Modern Approach*. Prentice Hall, 2010.
5. Wikipedia. 2019. List of Companion Plants. Retrieved from https://en.wikipedia.org/wiki/List_of_companion_plants/.

Note: Derek Tran joined Derek Thomas's project after Derek Tran realized his own project scope was way too large to complete. Derek Tran joined on Tuesday 4/30 and at that point Derek Thomas had a working prototype of the code and solver and no report done.