

Lecture Notes on Constraint Satisfaction

*Alexander Rush***Contents**

1	Introduction	1
2	A Running Example: Map Coloring	2
3	Constraint Satisfaction Problems	3
3.1	Formulation	3
3.2	Factor Graph	4
3.3	Graph Coloring	4
3.4	Example 2: Part-of-Speech Tagging	5
4	Consistency Checks	6
4.1	Arc Consistency	7
4.2	Forward Checking	8
5	Search for CSP	8
5.1	Ordering Heuristics	9
6	Local Search	10
6.1	Complete Assignment Formalism	10
6.2	Local Search Algorithms	11
6.3	Hill Climbing/Gradient Descent	12
6.4	Random Restarts	12
6.5	Simulated Annealing	13

1 Introduction

In the first set of lectures we have looked at two types of models, generic search models and models for game playing. Both utilize a state representation that evolves through the result of actions. In this section we consider a different model formulation, known as a constraint satisfaction problem (CSP). This model will use a more restrictive representation that will require us to specify specific constraints about the world. Instead of actions, CSPs require us to specify a fixed set of variables and constraints. The goal will be to find an assignment to the variables that satisfies all of the constraints.

After defining CSPs we will look at algorithms for solving them. We will see that CSPs can be solved using similar search algorithms as we saw in the previous classes. However the main benefit of this model is it allows us to employ general search algorithms without having to define problem specific heuristics. In addition CSPs also give us a framework for utilizing **local search**, a different style of algorithm that can be a very effective means for solving constrained problems.

2 A Running Example: Map Coloring

For these lectures, we will use the running example of map coloring. In particular we will attempt to find a 4-coloring of a map shown below. That this is always possible is the conclusion of famous mathematical theorem first proven 1976 (with the assistance of algorithmic techniques). While we will not go into details of the proof, coloring a specific map fits nicely as an example of the CSP framework. The theorem states that given a map and four distinct colors it is possible to return a coloring such that no two adjacent contiguous areas are assigned the same color.

We will apply this approach to a map of the Cambridge area shown below. For this problem we will define the coloring set as $COLORS = \{R, Y, O, B\}$, i.e. red, yellow, orange, and blue. Specify a valid coloring relationship using set notation:

$$DIFCOLOR = \{c_1, c_2 \in COLOR : c_1 \neq c_2\}$$

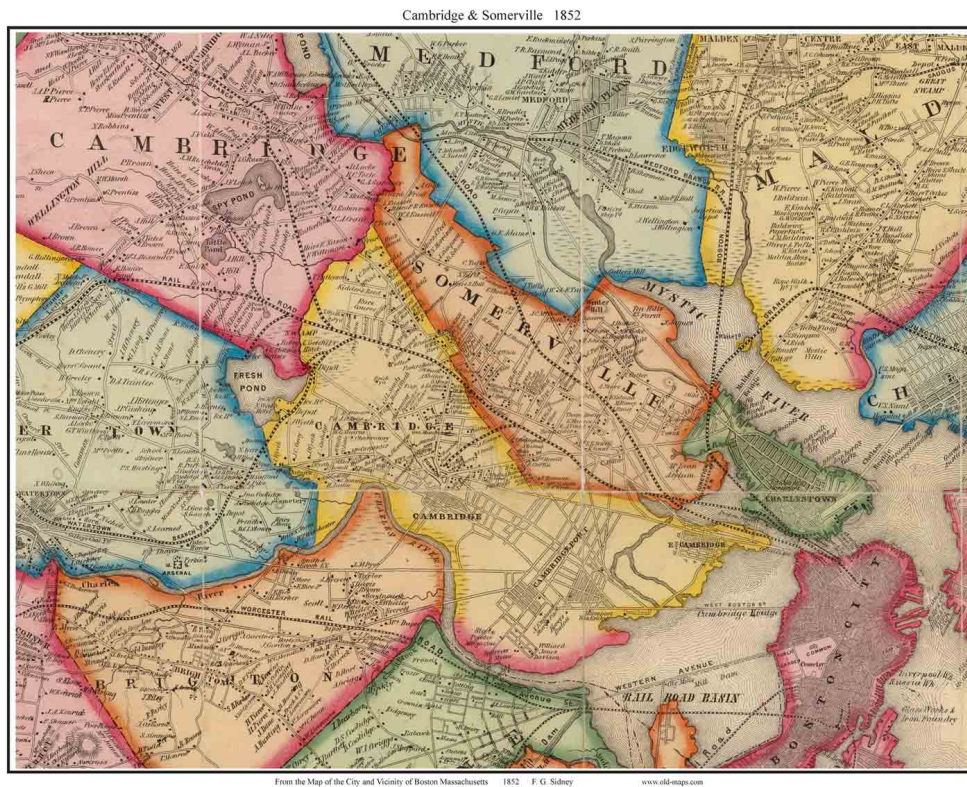


Figure 1: Cambridge-area circa 1852. Best viewed in color.

3 Constraint Satisfaction Problems

3.1 Formulation

We begin by giving a formal model of a constraint satisfaction problem. Note that our definition differs slightly from that given in AIMA in that we try to give a more specific definition of constraints.

A constraint satisfaction problem (CSP) defines a sets of **variables** which are jointly constrained. Formally a constraint satisfaction problem is defined by the following elements:

Name (AIMA)	Type	Description
Variables	X_1, \dots, X_n	Name of each variable in the problem.
Labels/ Variable Domains	$\mathcal{D}_1, \dots, \mathcal{D}_n$	The set of labels that each variable can take on, e.g. domain \mathcal{D}_i corresponds to X_i for all $i \in \{1, \dots, n\}$
Factors	F_1, \dots, F_m	Each factor is a list of variables, constrained by that factor, for example $F_2 = (X_1, X_4, X_{10})$
Constraints/Factor Domains	$\mathcal{R}_1, \dots, \mathcal{R}_m$	The set of labels each variable constrained by a factor can simultaneously take on. \mathcal{R}_i corresponds to factor F_i for all $i \in \{1, \dots, m\}$.

As an example, say we have three variables X_1, X_2, X_3 all with the same domains $\mathcal{D}_1 = \mathcal{D}_2 = \mathcal{D}_3 = \{A, B, C\}$. We could then have a factor $F_1 = (X_1, X_3)$ with constraint $\mathcal{R}_1 = \{(A, B), (A, C)\}$. This says that the only two valid labels for X_1 and X_3 are $X_1 = A, X_3 = B$ and $X_1 = A, X_3 = C$. It says nothing about X_2 . While a single factor can constrain many variables, we will mainly consider pairwise factors, also known as **arcs**.

Instead of paths as in search, we will be interested in assignments for CSP. In particular:

Definition 1 A **assignment** $A \in (\mathcal{D}_1 \cup \{\epsilon\}) \times \dots \times (\mathcal{D}_n \cup \{\epsilon\})$ consists of a label (or blank) for each variable. We say the assignment is **partial** if it includes ϵ , and **complete** if it does not.

Definition 2 A partial/complete assignment is **consistent** if it does not violate any constraints. That is for each factor (X_i, X_j) , the assignment is (A_i, A_j) is in the set of valid relations or at least one is ϵ .

A complete, consistent assignments is analogous to a solution path. For CSPs, we will treat all consistent paths as equally acceptable, i.e. there is no costs associated with assignments. (Note, however there are important variants of the CSP formalism that do model this idea of an assignment costs.)

3.2 Factor Graph

We can represent a constraint satisfaction problem using a graphical representation known as a **factor graph** (called a constraint hypergraph in AIMA). A factor graph is an undirected graph with two types of nodes: **variable nodes**, which are drawn as circles and **factor nodes** drawn as squares. There are n variable nodes in the graph, and m factor nodes. For each factor $F_i = (X_{f_1}, \dots, X_{f_o})$ we connect the corresponding factor node F_i to the corresponding variable nodes X_{f_1} through X_{f_o} .

Note that the factor graph makes no attempt to represent labels or constraints at all, it only represents which variables are constrained by which factors.

3.3 Graph Coloring

Question 1 *Let's now define the CSP for this graph coloring problem.*

Variables	⊗ $X = \langle \text{Cambridge, Somerville, Brighton, Boston, Watertown} \rangle$
Labels	⊗ $\mathcal{D}_1 = \text{COLORS}, \dots, \mathcal{D}_n = \text{COLORS}$
Factors	⊗ <div style="margin-left: 200px;"> $F_1 = (\text{Cambridge, Somerville}),$ $F_2 = (\text{Cambridge, Boston}),$ $F_3 = (\text{Cambridge, Watertown}),$ $F_4 = (\text{Cambridge, Brighton}),$ $F_5 = (\text{Brighton, Boston}),$ $F_6 = (\text{Brighton, Watertown})$ </div>
Constraints	⊗ $\mathcal{R}_1 = \text{DIFCOLOR}, \dots, \mathcal{R}_6 = \text{DIFCOLOR}$

So for this problem are $n = 5$ variables and $m = 6$ factors, each is also an arc. The representation is much easier to visualize as a factor graph. Below we shows a factor graph representation of the same CSP. However note it does not explicitly show the constraints of the graph.

Question 2 *Generate:*

1. *A complete, consistent assignment for this problem.*

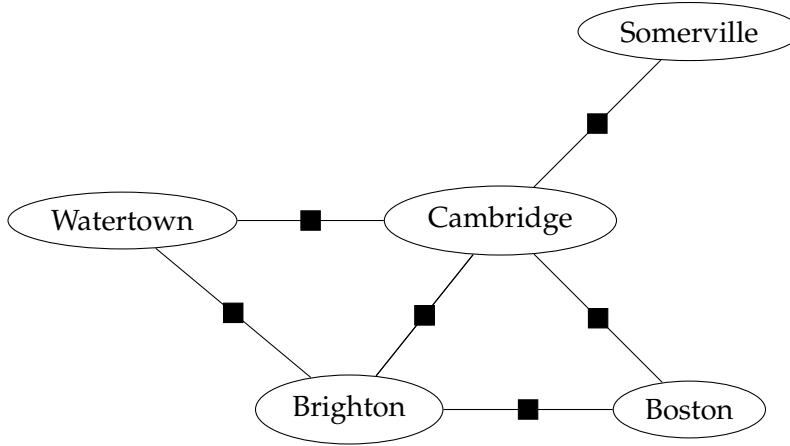


Figure 2: Factor graph showing the constraints for the Cambridge graph coloring problem.

2. An incomplete, consistent assignment for this problem.
3. An complete, inconsistent assignment for this problem.



1. A complete, consistent assignment is show in Figure ???. In our notation this assignment is represented as $\langle Y, O, O, G, B \rangle$.
2. If we had the same graph, but had not yet assigned a color to Cambridge we would write it as $\langle \epsilon, O, O, G, B \rangle$. This assignment is incomplete, but consistent.
3. Alternatively if we flip the color of Cambridge and Somerville we end up with $\langle \epsilon, O, Y, O, G, B \rangle$ which is complete but inconsistent.

3.4 Example 2: Part-of-Speech Tagging

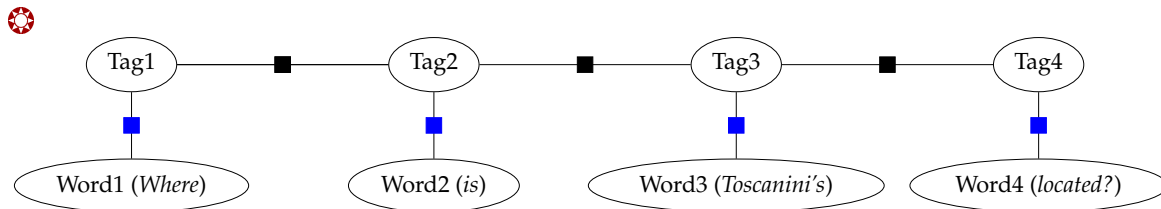
Let's consider a rather different example from the field of natural language processing. Imagine we are implementing a system like SIRI and we are given a crucial question such as "Where is Toscanini's located?". One of the first steps for handling such a question is to label each word with its part-of-speech tag.

We can model this problem using a CSP. The factor graph for this problem is shown in Figure ??. We have a factor connecting each of the part-of-speech tags and a factor connecting each word to its tag. Here the domain of each tag variable consists of a part-of-speech tag $TAGS = \{Noun, Adjective, Verb, \dots\}$, and the word variable domain will consist of single word $\mathcal{D}_1 = \{Where\}$. The constraints will ensure two things:

1. Each tag is a match its word $TAGDICT$
2. Neighboring tags are consistent $TAGCONS$

The first constraint will ensure that the tag of each word is consistent with the tags previously seen for that word. Many words in English are consistent with several different tags. The second constraint will ensure that the tag sequence is consistent with what is common in English.

If we can find a complete consistent assignment here, it is likely to give a good tagging of the sentence.



Question 3 Give some explicit constraints that might be used in this model.

- The word 'is' most always be a verb.
- The Adj/Noun

4 Consistency Checks

One benefit of the CSP formalism is that it allows us to make direct **inferences** about unassigned variables based on a current partial assignment of the CSP. This can be done through **consistency** checks that recalibrate the domains of variables based on neighboring domains.

Consider for example a consistent partial assignment to the map coloring CSP.

$$\langle \epsilon, \epsilon, O, G, B \rangle$$

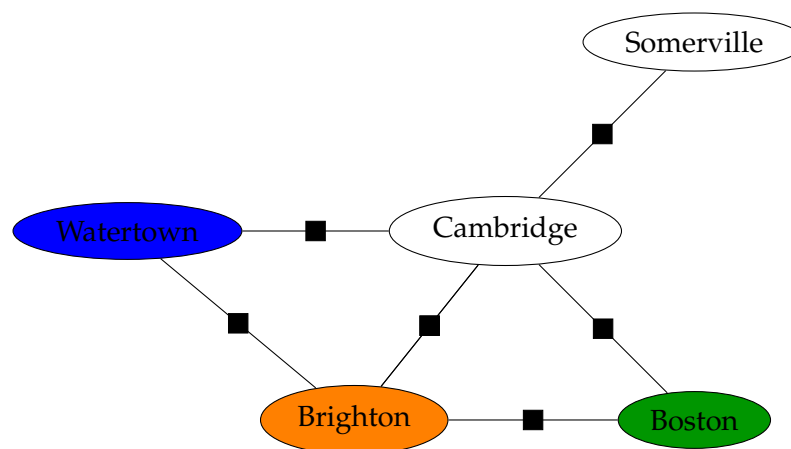


Figure 3: A consistent partial coloring of the Cambridge area.

Our consistency check will ask what is the effective domain of the unassigned variables: Somerville and Cambridge. The algorithm will go through each factor and filter the domains of its constrained variables.

The algorithm proceeds as follows:

1. We start with the Somerville-Cambridge factor and check whether any of it constrains out any of Somerville's assignments. However since the Cambridge variable could at this point be any color, we cannot limit out any of the coloring options for Somerville.
2. Next we check the other factors around Cambridge. In this case it has three factors each with assigned neighbors. Going through the constraints, we see that (1) the factor with Watertown means it cannot be blue, (2) the factor with Brighton means it cannot be orange, (3) the factor with Boston means it cannot be green. This limits the domain \mathcal{D}_1 from $\{R, Y, O, B\}$ to $\{Y\}$
3. Now that we have changed the domain of Cambridge we need to recheck all of its neighboring factors again. This means we return to the Somerville-Cambridge constraint. However now we know the domain of Cambridge is $\{Y\}$ which limits the domain of Somerville \mathcal{D}_2 to $\{R, O, B\}$

For this particular problem after the consistency checks it is now easy to find a complete consistent assignment.

4.1 Arc Consistency

When all factors are of size 2, this technique of enforcing consistent domains is known as **arc consistency**. The algorithm is shown below.

```

1: procedure ARCCONSISTENCY
2:   queue  $\leftarrow [F_1, F_2, \dots, F_m]$ 
3:   while queue is not empty do
4:      $F_i \leftarrow \text{queue.pop}()$ 
5:      $(X_j, X_k) \leftarrow F_i$ 
6:     rev  $\leftarrow$  false
7:     for  $x \in \mathcal{D}_j$  do
8:       if  $\bigotimes$  for all  $y \in \mathcal{D}_k, (x, y) \notin \mathcal{R}_i$  then
9:          $\mathcal{D}_j \leftarrow \bigotimes \mathcal{D}_j \setminus \{x\}$ 
10:        rev  $\leftarrow$  true
11:     if rev then
12:       if  $\mathcal{D}_i$  is empty then return false
13:       for  $\bigotimes$  factors  $F$  neighboring  $X_j$  do
14:         queue.push( $F$ )
15:   return true

```

Note for the purpose of this algorithm we treat the partial assignment as a domain of size one (i.e. Watertown would have domain $\{B\}$). By itself, this algorithm is not guaranteed to find a complete consistent assignment. In fact, if we run the algorithm on a blank assignment for

graph coloring it, will not even limit the domain at all! However it can be very effective for other problems, for instance for the tagging example.

Question 4 *What is the complexity of running this algorithm?*



4.2 Forward Checking

For arc consistency we include a check that the domain \mathcal{D}_i is empty. When run from the initial assignment we would hope that this check is never hit (or else the problem is unsolvable). However, we can also use arc consistency to **forward check** whether consistent, incomplete assignments are on the right path.

To do this, we set $\mathcal{D}_i = \{A_i\}$ for all i such that $A_i \neq \epsilon$. We then run the arc consistency algorithm. If the algorithm returns false, it tells us that there is no consistent, complete assignment that overlaps with A . As we assign, new variables we can repeat this check, starting with the factors that constrain the newly assigned variable. We will see in the homework that this is an important tool for speeding up CSP search.

5 Search for CSP

Arc consistency can effectively limit the valid domains of a CSP, but in general to actually find a complete consistent assignment we may have to return to running a search-like algorithm. (However we will see a different approach in the next section). Here is the search model for solving CSPs:

Name (AIMA)	Type	Description
State space	\mathcal{S}	⚙️ All possible consistent CSP assignments (possible partial).
Action model	ACTIONS	⚙️ All consistent assignments to any unassigned variable.
Transition model	RESULT	⚙️ A new consistent assignment after assigning an unassigned variable.
Initial state	$s_0 \in \mathcal{S}$	⚙️ The partial assignment with all variables blank, i.e. ϵ .
Goal test	$\text{GOAL} : \mathcal{S} \rightarrow \{0, 1\}$	⚙️ Any complete consistent assignment.

As with game playing we will use the recursive, tree-search version of DFS for this problem.

Question 5 Why not use BFS or UCS for this problem? Why not graph search? What are the search properties of this model?



Below is the recursive DFS algorithm. It adds a couple of elements to standard DFS. First it checks if an action is valid by ensuring that the new CSP state is a consistent assignment. Second at each step it runs a round of INFERENCE, which may consists of running arc consistency as above to limit the future domains. AIMA also describes various other inference steps that are specific to special types of factors.

Require: A is consistent assignment

Ensure: true if complete, consistent assignment is found

```
1: procedure BACKTRACK( $A$ )
2:   if  $A$  is complete then return  $A$ 
3:    $X_i \leftarrow$  variable with  $A_i = \epsilon$ 
4:   for  $x \in \mathcal{D}_i$  in pre-selected order do
5:     if  $x$  is consistent with assignment then
6:        $A_i \leftarrow x$ 
7:       check  $\leftarrow$  FORWARDCHECK( $A$ )
8:       if  $\neg$  check then return false
9:       result  $\leftarrow$  BACKTRACK( $A$ )
10:      if result then return true
11:      undo FORWARDCHECK( $A$ )
12:       $A_i \leftarrow \epsilon$ 
13:   return false
```

5.1 Ordering Heuristics

Finally of special concern for CSPs is selecting the order in which DFS proceeds. CSPs often have a very large branching factor $O(n^{D^*})$ where D^* is the size of the largest domain $D^* = \max_i |\mathcal{D}_i|$. Therefore there has been research into selecting the order for expanding future assignments of a given state. In particular in the homework you will explore a few variants of this search algorithm.

Question 6 For the homework you will implement a Sudoku solver. As practice spend, spend time filling out the puzzle below.

		6	8		1	2		
	5	3	4		6	7	8	
4	8						5	1
5	6						9	4
				6				
2	4						7	6
7	9						3	8
	3	1	5		7	9	6	
		5	9		8	4		

Question 7 Given the puzzle, what order should we handle unassigned variables? (line 3)

☛ Min-Remaining Values Degree/ number of Factors heuristic

Question 8 Given the puzzle, what order should we assign values from \mathcal{D}_i ? (line 4)

☛ Least constraining value

6 Local Search

Now we consider very different set of algorithms for solving CSPs. Up into this point we have only looked at search algorithms that keep track of incomplete assignments. We saw in the last section that standard search can be a poor fit for CSPs. For one, search inherently requires a some of bookkeeping that can make it inefficient, particularly with a large branching factor. More importantly when we run DFS we may make a decision early on that makes it impossible to satisfy the rest of the problem. This can lead the algorithm down very costly dead-end paths when forward checking is not helpful.

In **local search**, instead of constructing a search tree using consistent, partial assignments, we work in the inverse space of inconsistent, complete assignments. We look for a consistent assignment by moving in the local area of the current state.

6.1 Complete Assignment Formalism

The complete assignment formalism (complete state formalism in AIMA) assumes that all non-goals states are inconsistent complete assignments. Define the set \mathcal{S} as giving all possible complete assignments for a CSP.

Additionally for any complete assignment, we assume we have a function $\text{NEIGHBORS} : \mathcal{S} \rightarrow 2^{\mathcal{S}}$ that produces similar states. In the case of CSP this will give all complete assignments formed by changing the label of a single variable. For instance in graph coloring this might correspond to changing the color of one city, for tagging this might correspond to changing the tag of one word.

We also assume that we have a random function $\text{SAMPLE} : \mapsto \mathcal{S}$ that can give us a random initial assignment. When the domains are finite we can simply sample uniformly for each variable.

Name	Value	Description
Neighbor	$\text{NEIGHBORS} : \mathcal{S} \mapsto 2^{\mathcal{S}}$	☼ Set of neighbors of each state.
State Score	$f : \mathcal{S} \mapsto \mathbb{R}$	☼ Score a state.
Sample	$\text{SAMPLE} : \mapsto \mathcal{S}$	☼ Produce a random state.

The neighbors and sample function can also in theory incorporate knowledge of the underlying domain. In practice this is often important. We might want to prune out obviously bad neighbors, or sample from distributions that are more likely to give partially consistent assignments.

For instance, in the homework we will return to the Sudoku problem. For this problem our function `SAMPLE` will sample from assignments where all of the row factors are satisfied (but with possibly inconsistent columns and boxes). The `NEIGHBORS` function will consider all possible ways are flipping to elements of a row. Since this operation keeps the rows consistent, local search spends much less time exploring inconsistent assignments.



6.2 Local Search Algorithms

Local search refers to a large class of algorithms that work by iteratively moving along the space of neighboring assignments. The underlying algorithm is remarkably simple:

```

1: procedure LOCALSEARCH
2:    $s \leftarrow \text{SAMPLE}(\mathcal{S})$ 
3:   for  $t = 1 \dots \infty$  do
4:      $s \leftarrow \text{LOCAL}(s, \text{NEIGHBORS}(s), t)$ 
5:     if ☼  $s$  is consistent then return  $s$ 

```

In this case `LOCAL` is one of several possible local transition functions. For each of these, we will use a cost or fitness function $f : \mathcal{S} \mapsto \mathbb{R}$ which will estimate how far we are from a consistent assignment. For CSP there are two natural choices are:

1. Min Conflicts
Count the number of remaining factors that are unsatisfied.
2. Weighted Conflicts
Version of above where the factors are weighted.

6.3 Hill Climbing/Gradient Descent

Let's begin with the simplest strategy. Here we will simply enumerate all neighbors and return the run that minimizes our cost function. This is known as the method of steepest descent.

```
1: procedure STEEPESTDESCENT( $s, \mathcal{N}, t$ )
2:   for  $s' \in \mathcal{N}$  do
3:     if  $f(s') < f(s)$  then  $s \leftarrow s'$ 
4:   return  $s$ 
```

Alternatively we can return the value that first gives a descent direction, known as stochastic descent. This has the advantage of not requiring us to enumerate the full set of neighbors.

```
1: procedure STOCHASTICDESCENT( $s, \mathcal{N}, t$ )
2:   for  $s' \in \mathcal{N}$  at random do
3:     if  $f(s') < f(s)$  then return  $s'$ 
4:   return  $s$ 
```

However the problem with hill climbing/gradient descent methods is that they may quickly find a **local optima**, that is an assignment for which there are no neighbors with lower f .

While the space of assignments is not continuous, this situation is analogous to finding a point with zero-gradient on a non-convex function, as shown in Figure ?? . When we are in this situation there is not local way to “get out” and find a different optimum.

6.4 Random Restarts

One of the simplest way get out of this situation is with a random restart. If we alternate calls to gradient descent and random restart we move around the space hopefully exploring different local optima.

```
1: procedure RANDOMRESTART( $s, \mathcal{N}, t$ )
2:   if RESTARTSCHEDULE( $t$ ) then return Sample( $\mathcal{S}$ )
3:   return  $s$ 
```

One nice benefit of this algorithm is that we can run many random restarts in parallel since they never interact directly. However there are many functions of interest that can have a very large number of local minima, so even with this strategy we may not do much better.

A similar approach is a method called local beam search or sometime Tabu search. This method keeps around the K -best assignments it has seen so far. At each step it checks the neighbors and recalculates the K -best list. This algorithm can be combined with random restarts to refresh the list if all the assignments become too close.

This approach is similar to **genetic algorithms** another popular local search method described in AIMA.

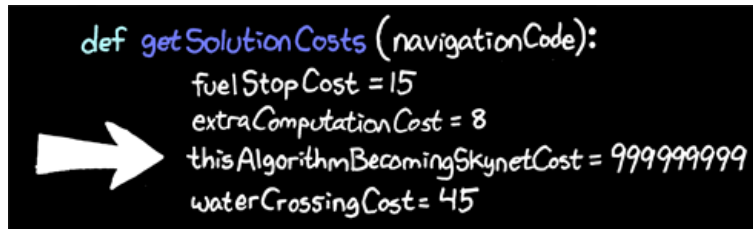
6.5 Simulated Annealing

Finally we consider a local algorithm that does not try to require making improvements at each step. This approach makes the assumption that early on we might want to bounce around a bit, but that later in the process we might fall back to gradient descent.

```

1: procedure LOCALBEAMSEARCH
2:   for  $k = 1 \dots K$  do
3:      $s_k \leftarrow \text{Sample}(\mathcal{S})$ 
4:   for  $t = 1 \dots \infty$  do
5:     for  $k = 1 \dots K$  do
6:       for  $s' \in \text{NEIGHBORS}(s_k)$  do
7:         for  $k' = 1 \dots K$  do
8:           if  $f(s') < f(s_{k'})$  then
9:              $s_j = s'$ 

```



```

def getSolutionCosts (navigationCode):
    fuelStopCost = 15
    extraComputationCost = 8
    thisAlgorithmBecomingSkynetCost = 999999999
    waterCrossingCost = 45

```

GENETIC ALGORITHMS TIP:
ALWAYS INCLUDE THIS IN YOUR FITNESS FUNCTION

Figure 4: Subtext: Just make sure you don't have it maximize instead of minimize.

To do the former, it randomly accepts some neighbors that increase f based on how much worse they are. The likely of doing this decrease as we go, based on a predefined annealing schedule.

```

1: procedure SIMULATEDANNEALING( $s, \mathcal{N}, t$ )
2:    $T \leftarrow \text{ANNEALSCHEDULE}(t)$ 
3:    $s' \leftarrow \text{SAMPLE}(\mathcal{N})$ 
4:    $\Delta E \leftarrow f(s) - f(s')$ 
5:    $R \leftarrow \text{Rand}(0, 1)$ 
6:   if  $\Delta E > 0$  or  $R > e^{\Delta E/T}$  then return  $s'$ 
7:   return  $s$ 

```

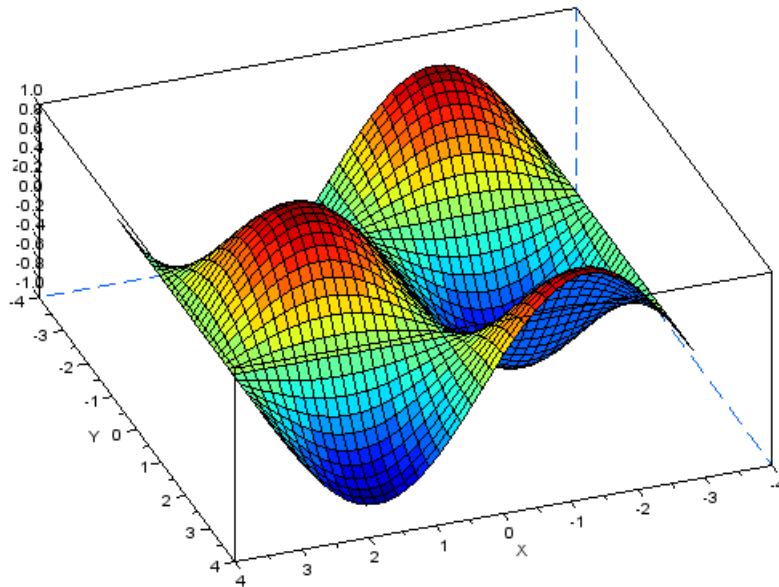


Figure 5: Graph with local minima and maxima.

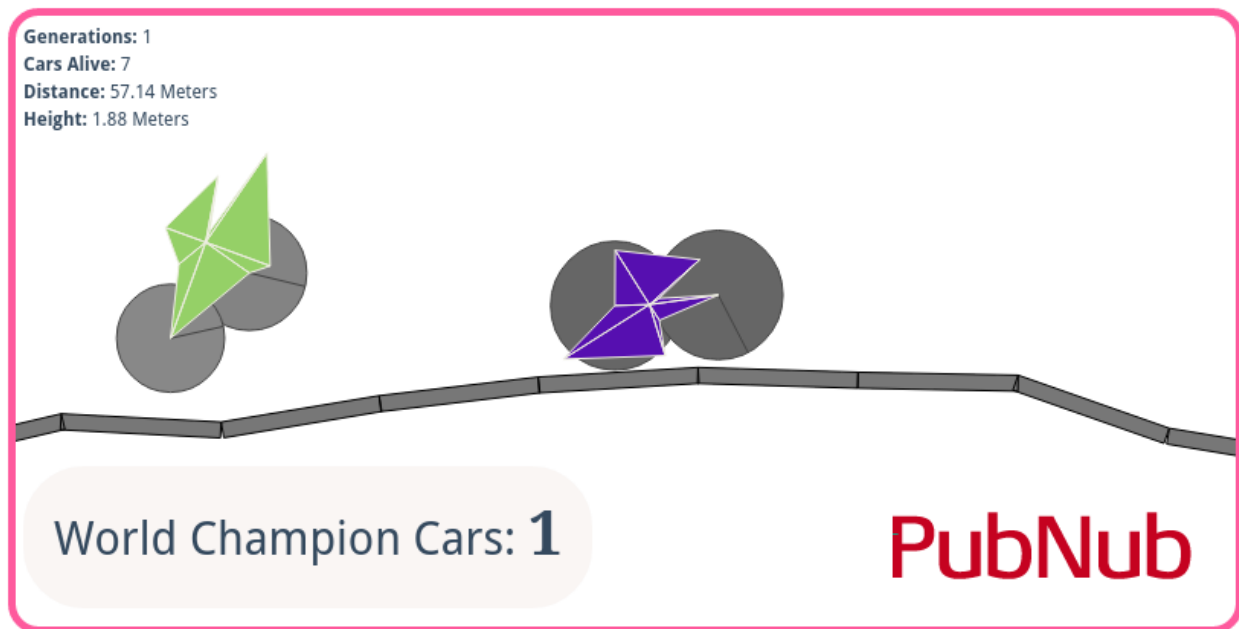


Figure 6: Genetic algorithm car racing (<http://gencar.co/>). Watch out, you can spend hours watching this.