AI Programming Project:
Building a General Constraint-Based Puzzle Solver

Purpose:

- Learn the basics of local search and constraint-based problem solving by coding a few of the basic algorithms.
- Gain experience building a modular, general-purpose puzzle solver.

1 The Basic Assignment

You will implement two basic algorithms for solving constraint-satisfaction puzzles via local search: simulated annealing (SA) and minimum-conflicts (MC). These will be used to solve 3 different types of puzzles: K-Queens, Graph Coloring, and a third puzzle of your choice. Emphasis is placed on the modularity of your code such that the addition of each new puzzle type should not require changes to the core search-algorithm code.

2 The General Puzzle Solver (GPS)

In 1959, Allen Newell and Herbert Simon, two of AI's most famous pioneer researchers, designed a system known as the General Problem Solver (GPS). Our goal in this project is a bit more modest, but we will borrow the GPS acronym for inspiration.

Your GPS will need to handle many different puzzles, all of which can be assumed to have constraint-based representations. Your code **must be modular** such that:

1. The simulated annealing and min-conflicts algorithms are completely general-purpose, with only a few calls to problem-specific methods.
2. Each new puzzle is handled by a separate subclass that supplies the SA and MC algorithms with problem states and evaluations (of states).

Figure 1 gives the high-level organization for the GPS. The class *Constraint-Based Local Search* incorporates most of the key elements of local search with a constraint-based representation. Simulated Annealing and Min-Conflicts are then subclasses, since each will have some specialized properties (such as the temperature

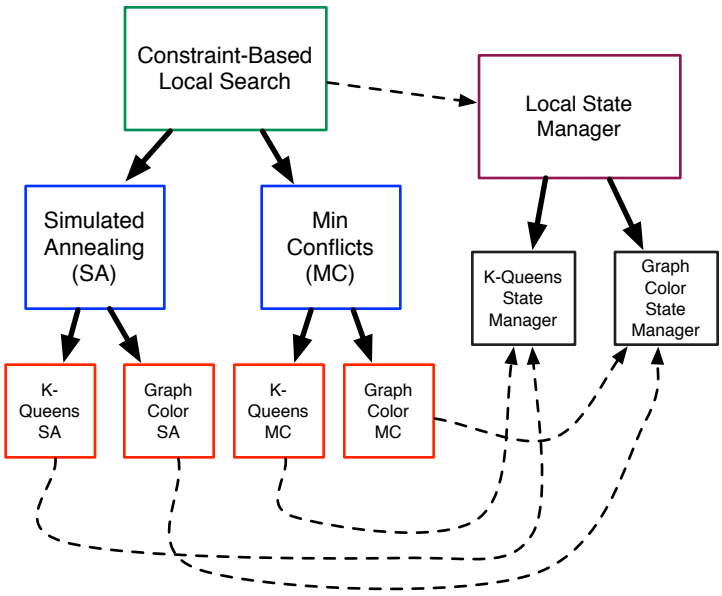

Figure 1: Basic class structure for the General Puzzle Solver. Solid lines indicate subclass relationships, while dashed lines denote properties. Each search object has a state-manager object as a property.

variable used in SA) and methods (such as the check for the variable value that produces the fewest conflicts in Min-Conflicts). None of these 3 classes should have any code that is specialized for a particular puzzle, such as K-Queens. To tailor them to a problem, you will need further specialization, yielding subclasses such as K-Queens SA, and Graph Color MC. These bottom-level classes should have very few methods, and quite possibly no additional properties. In fact, you may choose not to include them at all, although they enhance the generality of your code should you extend it to handle more than the 3 puzzles of this assignment.

The main problem-specific activity should be handled by the state managers for each problem; these must be included. Each problem type will probably require a very specialized state representation along with code for evaluating, modifying and producing states. Checking constraint violations will probably also be a problem-specific exercise, unless you choose to implement a general-purpose constraint language (a rather difficult task for a 4-week project).

The state manager should be able to perform (at least) the following activities when called by one of the search algorithms.

1. Generate the initial state for a local search process.
2. Modify a state in either a random or intelligent manner.
3. Generate a number of successor states to a given state.
4. Evaluate a state, which involves checking constraint violations.
5. Display a state - a very useful activity for both debugging and presenting the final solution to a puzzle.

For Min-Conflicts, you may choose to only modify the current state but never create new successor states. But simulated annealing requires you to produce many successor states, evaluate them, and then decide how

to reassign the current state. In either case, the MC and SA code should call the state manager and let it handle the production and/or modification of states. For each new puzzle that you choose to solve, a new subclass of Local State Manager will be required.

The details of your implementation are up to you, but this basic separation between search algorithm and state manager MUST be incorporated into your design: the search algorithms must be general-purpose, and the state managers must be tailored to particular problem domains.

3 Local Search

Although somewhat of a misnomer, the term *local search* refers to methods that employ **complete solutions** (a.k.a. *attempts*) that are gradually modified in pursuit of an optimal solution. These differ from methods such as depth-first, breadth-first and best-first search, all of which normally deal with **partial solutions** that are gradually extended. In contrast to these, local search can be stopped at any time to yield a viable (though probably not optimal) solution. Common local search methods include hill-climbing, simulated annealing, tabu search and evolutionary algorithms.

Consider the following *Egg Carton* Puzzle: a farmer has a rectangular egg carton consisting of M rows and N columns of cells, where each cell can hold one egg. The task is simply to place as many eggs as possible into the carton without violating the following constraint:

> No row, column nor diagonal can contain more than E eggs.

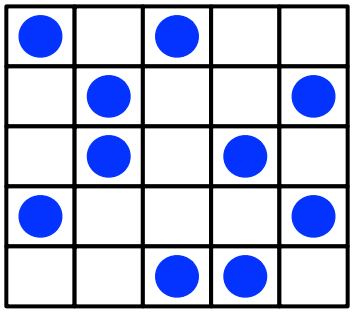

Figure 2: A solution to the egg carton puzzle where M=N=5 and E=2.

Here, ALL diagonals need to be considered, not just the main diagonals. A common version of the problem is M = N = 6 and E = 2, which has an optimal solution involving 12 (6 x 2) eggs. Figure 2 shows a solution for the case where M=N=5 and E=2.

To use local search on this problem, a representation for complete solution attempts is needed. One such representation is a simple binary vector of length MN, which can be folded into an M x N array of cells. Then, each binary vector entry denotes the presence (1) or absence (0) of an egg in that particular cell. Figure 3 illustrates this basic process of conversion from the syntactic bit vector to a representation with semantics (i.e. meaning) with respect to the egg-carton problem: concepts such as row, column and diagonal (and the limit of E eggs in each) make sense in the M x N array.

The *objective function* is one (and possibly the only) source of problem-specific knowledge in local search.

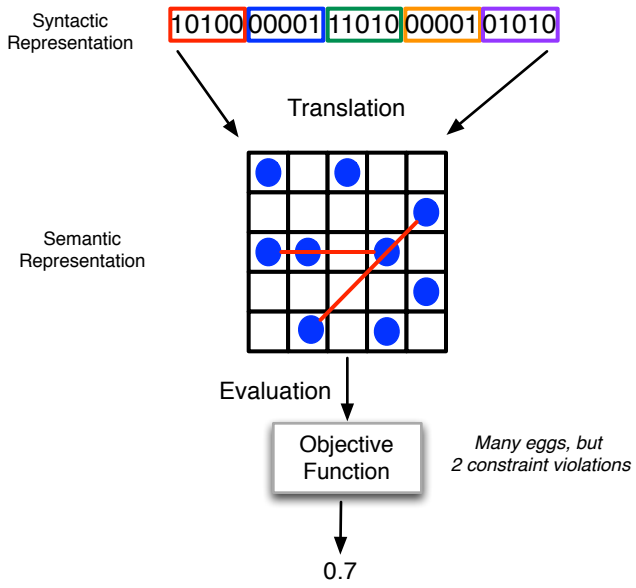

Figure 3: Conversion of a syntactic bit vector into a semantic egg-carton solution attempt, which, in the form of an M x N array, can be evaluated. In this example, E = 2, and thus the two red lines denote violations.

The values that it returns have a strong influence upon the directions that the algorithm pursues in search space. This function should not only give a perfect score for an optimal solution and little or not points for a miserable attempt, but it needs to give **meaningful partial credit** in much the same way that a heuristic does in best-first (e.g. A*) search. For example, the scenario in Figure 3 earns many points for multiple rows, columns and diagonals with the maximum of E=2 eggs; but it loses points for the overloaded row and diagonal. This relatively high evaluation tells the search algorithm to continue investigating in this region of the search space.

4 Constraint Satisfaction Problems

A Constraint Satisfaction Problem (CSP) involves a collection of variables ($\{V_i\}$), each of which can take on values from a particular set, called the domain, D_i, which can be the same or different for each variable. A set of constraints (C) define the legal relationships between the variables, thus restricting the choices of values for each variable.

For example, the egg carton puzzle is easily modeled as a CSP via the following formalization:

1. Each cell/compartment of the carton is a variable. So there are MxN variables for an egg carton with those dimensions.
2. The domain of each variable is simply $\{0, 1\}$, where 0 (1) denotes the absence (presence) of an egg in a cell.
3. There is one constraint for each *line* (i.e., row, column or diagonal) in the carton. Each such constraint says that no more than E eggs shall be present in a line.

Solving the Egg Carton CSP then involves trying different assignments of values to cells in order to find a complete set of assignments that maximizes the number of eggs in the carton while not violating any of the line constraints.

A special case of the Egg Carton problem is the K-Queens problem, an AI classic. In this, K queens must be placed on a chessboard with K rows and K columns so that no queen is attacking another queen. This is equivalent to a K x K egg carton puzzle where E = 1.

Couching K-Queens in terms of CSP is straightforward, since we know that there should be exactly one queen in every row of the chessboard. Use Q_i to denote the queen in the ith row. What we don't immediately know, and what we must figure out to solve the puzzle, is the column for each of the Q_i. This leads to the following CSP formulation of the puzzle:

1. Q_i i = 1..k are the variables.
2. Each Q_i has the domain $\{1..k\}$, with each value indicating a possible column for the queen in the ith row.
3. There is one constraint per line (row, column, or diagonal) indicating that no more than ONE queen can reside in each.

By finding a value assignment for each of the K queens that does not violate any line constraints, you solve the problem. Try it by hand on an 8 x 8 chessboard. When you get tired of that, grab your computer and program it to do the job. Using the appropriate methods, the computer can easily solve the 8-Queens puzzle along with much larger versions involving hundreds, thousands and even millions of queens (given enough computing resources). In this project, your Min-Conflicts algorithm (see below) should easily handle a few hundred queens.

5 Local Search with Constraint Reasoning

In local search, each search state is a complete solution/attempt. For CSPs, this means that each variable has an assigned value, and a state may contain constraint violations, possibly many of them; and these violations tend to be an important factor in the objective function. For example, in K-Queens puzzles, a violation is simply the presence of more than one queen on any line. So violation checking involves either considering every pair of queens to see if they attack each other, or line checking, where every horizontal, vertical and diagonal line is tested for the presence of more than one queen.

There are just under 6N lines to check on an N x N chessboard, while there are N(N-1)/2 pairs to check. However, each line has an average of more than N/2 cells. So both approaches have $O(N^2)$ complexity, and it probably doesn't matter much if you explicitly represent the chessboard as a 2-dimensional array and place queens on it, or if you just check all pairs of queen-column assignments without actually creating the array.

If you do pair-checking instead of line checking, then the violation detector for any pair of queens q_i and q_j (where i and j are row numbers) is the following rule:

> Queens i and j attack one another if and only if column(q_j) $\in$ {column(q_i), column(q_i) + $| i - j |$, column(q_i) - $| i - j |$ }

For example, if i = 5 and j = 8, and if column(q_5) = 4, then q_8 is under attack by q_5 if and only if column(q_8) is a member of the set { 4, 7, 1 }.

When considering the minimum-conflict change (see below) to make to a particular queen (q*), it does speed up the algorithm to do pair checking instead of evaluating all lines of the board, since in these cases, you only need to check all possible violations involving q*. So you can use the simple rule above to compare q* to all other queens, without considering all lines of the chessboard. Alternatively, you can build a data structure that allows quick access to all lines along which a given board cell resides. Then, when q* is placed in cell C, all and only those lines that involve C can be pulled up and checked for other queens.

Once again, the constraint-checking aspect of this approach need not involve general-purpose logical expressions. It can just be a piece of code that goes through all lines or queen pairs and tallies up situations where one queen attacks another.

6 Simulated Annealing

As mentioned above, some forms of local search may only employ problem-specific knowledge at one point: during evaluation using the objective function. This is the case with algorithms such as hill-climbing, evolutionary algorithms (where the objective function is called a *fitness function*), and simulated annealing.

In hill-climbing search, many neighbors of the current solution (P - a point in search space) are randomly generated. If any of these neighbors have a higher evaluation than P, then P becomes the best of these neighbors and search continues. However, if none of the neighbors offers an improvement over P, then search halts and returns P. Hill-climbing is a very weak search algorithm, since it easily gets stuck at local (but not global) high-points in the search space.

Simulated annealing (SA) remedies this early-stagnation problem of hill-climbing by adding a stochastic *jiggle* that allows P to occasionally become a neighbor even if that neighbor has an inferior evaluation. SA draws inspiration from the metallurgic process of annealing, in which metals are heated and then gradually cooled (and sometimes reshaped along the way). Simulated Annealing's main parallel to real annealing involves the temperature parameter, which gradually decreases as the search process continues. This parameter controls the degree to which the search procedure *explores* in random directions rather than *exploiting* directions that have already been proven to yield good results. In other words, SA explores when P becomes an inferior neighbor (thus moving search down a path that holds no immediate promise); but SA exploits when the neighbor is superior (and thus in a direction of guaranteed improvement).

Consider the problem of finding an optimal combination of settings for the dials on a control panel that regulates a complex industrial process. Here, P would be the vector of values for these dial settings. Simulated annealing begins with a random or user-generated vector of values and tries to improve it. Any vector that SA samples needs to be **evaluated** using an **objective function**, which assesses the quality of the solution that the vector represents. In this example, any vector would need to be tested on the industrial process (or a simulation of it): the control-panel dials are set according to the values in the vector, and performance of the system is measured, yielding an evaluation of the vector. These evaluations are then used to help steer search in promising directions, hopefully toward the optimal solution.

A common variant of the SA algorithm is as follows:

1. Begin at a start point P (either user-selected or randomly-generated).
2. Set the temperature, T, to it's starting value: T_{max}

3. Evaluate P with an objective function, F. This yields the value F(P).
4. If $F(P) \geq F_{target}$ then EXIT and return P as the solution; else continue.
5. Generate n neighbors of P in the search space: $(P_1, P_2, ..., P_n)$.
6. Evaluate each neighbor, yielding $(F(P_1), F(P_2), ..., F(P_n))$.
7. Let P_{max} be the neighbor with the highest evaluation.
8. Let q = $\frac{F(P_{max})-F(P)}{F(P)}$
9. Let p = min [1, $e^{\frac{-q}{T}}$]
10. Generate x, a random real number in the closed range [0,1].
11. If $x > p$ then $P \leftarrow P_{max}$;; (Exploiting)
12. else $P \leftarrow$ a random choice among the n neighbors. ;; (Exploring)
13. $T \leftarrow T - dT$
14. GOTO Step 4

A slightly simpler version appears in your AI textbook (*Artificial Intelligence: A Modern Approach*, 3rd Edition). Either one is fine to use for this assignment. Both are general-purpose algorithms applicable to many different problems. However, they also require a few key specializations to tailor search to particular tasks. The most important of these are:

1. The data structure (a.k.a. representation) used for solutions.
2. The objective function - This should give a perfect score to optimal solutions and lower scores to anything less than optimal. It should also give appropriate **partial credit** to good but non-optimal solutions such that SA receives useful hints about the proper directions in which to continue searching.
3. A neighbor-generation procedure - This should produce neighbors in search space to the current solution. These procedures normally make small changes to the original, such as adding (subtracting) 10 % to (from) a vector value.

You will need to experiment with different values of T_{max} and dT in order to find the appropriate time-varying balance between exploration and exploitation. Ideally, simulated annealing begins with a lot of exploration but gradually becomes more exploitative. Feel free to supplement step 4 in order to stop the search after a fixed number of iterations or the attainment of the target value, whichever comes first.

When using simulated annealing on a constraint-satisfaction problem, new neighbors are generated by making **random** changes to the values of some of the variables, e.g., randomly changing the column of a queen. Exactly how many changes can be an important parameter of the system, and one that varies with the problem domain. The important point is that the changes are **random** and do not require any insight into the problem domain. Only the objective function reflects specialized knowledge of the problem. Generally speaking, random choices do not reflect any sophisticated intelligence in an AI system, only trial-and-error search.

7 Min-Conflicts Constraint Satisfaction Search

Though simulated annealing works well on many problems, it can be a disadvantage to **only** employ problem-specific knowledge in the objective function, i.e., in the test. If, in addition, we include problem-related knowledge in the **generation** of new states, search performance can improve dramatically. In constraint-satisfaction problems, this *generator knowledge* typically relates to constraints that will be satisfied or violated when particular variables change value.

The Min-conflicts algorithm is a classic, and very successful, general-purpose procedure that adds just enough knowledge to the generator to bolster performance, often to a surprising level. It is not problem-specific, per se, but it is specific to local-search problems involving variables and constraints.

The basic Min-Conflicts algorithm is trivial:

- Repeat until optimal solution found or maximum number of steps:
  1. Randomly choose any variable, V, that is involved in at least one conflict (i.e., violated constraint)
  2. Assign V the new value a, where a is the value that produces the fewest number of conflicts.

In the K-Queens problem, the use of Min-Conflicts requires you to consider ALL possible columns for a particular queen, counting the number of queens that it would attack if placed in that column. Then, place the queen in the column that produces the fewest attacks. Figure 4 provides a simple example of local search using MIN-CONFLICTS.

Note that page 221 of your AI textbook uses one queen variable per COLUMN, and the problem-solver must choose the proper ROW for each queen. In this document, the opposite (but symmetric) scenario is employed: there is one queen per ROW, and your algorithm must figure out the proper COLUMN placement for that queen. You are free to use either representation in this project.

There are a few important details of MIN-CONFLICTS that should be kept in mind:

1. Assume that column(q_i) is the current column assignment of the ith queen, and it is involved in M queen attacks. Also assume that all other column placements for this queen will produce M or more queen attacks, with Z of these producing exactly M attacks, where $Z \geq 1$. Then, you should move the queen to one of these Z columns even though the change does not immediately improve the state evaluation.
2. If several column placements for q_i would produce the same minimal number of attacks, then RANDOMLY choose among them.

Both of these strategies enhance the exploration performed by local search, and they seem to be critical for success on the K-Queens problem as K becomes large. Without the first rule, for example, the search can stagnate long before a violation-free solution is discovered. But with these 2 rules, the system contains just enough *jiggle* to eventually find a perfect solution.

To summarize, Local Search using MIN-CONFLICTS for the K-Queens problem works as follows:

1. Initialize the board by randomly choosing columns for each of the K queens.

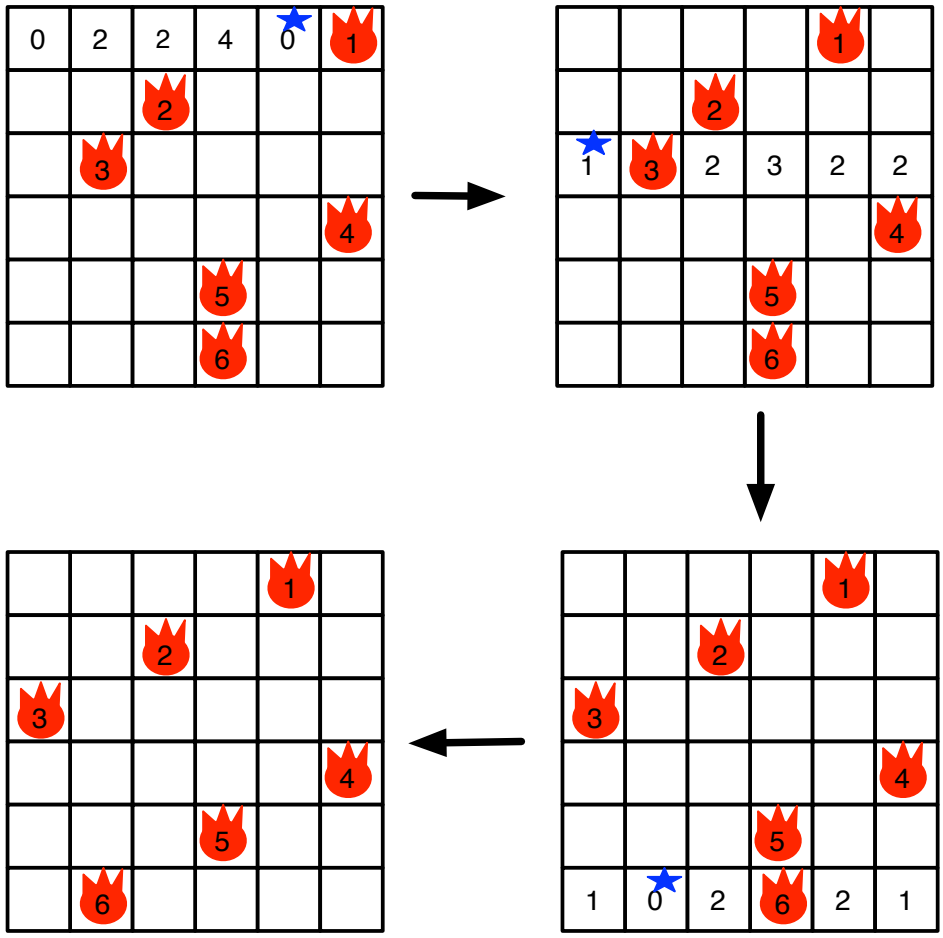

Figure 4: Three applications of MIN-CONFLICTS, to queens 1, 3 and then 6, resulting in a solution to the 6-Queens Puzzle. Stars denote the column with the fewest number of conflicts for the row in question, with ties resolved randomly.

2. Repeat for either a maximum number of steps or until a violation-free solution is found:
   (a) Randomly choose a queen that is involved in at least one conflict.
   (b) Use MIN-CONFLICTS to choose a new column for the queen. Only if ALL columns would produce MORE violations than does the current column, should the queen stay put.

Using this approach, you should be able to solve much larger K-Queens puzzles (in reasonable time, i.e. a few hundred or thousand iterations) than with simulated annealing (or other approaches that use partial solutions, such as backtracking search). K = 100 or 200 should not be a huge chore on a laptop, with K above 1000 also being quite accessible. However, not every run will produce a violation-free solution. This is the nature of local search on very large problems: usually it finds good solutions but cannot guarantee an optimum in any finite amount of time.

8 Graph-Coloring Problems

Graph-coloring, like many constraint-satisfaction problems, is easy to explain but extremely difficult to perform in worst-case scenarios, of which there are many! It is a classic NP-Complete problem.

Given a graph consisting of a set of vertices, V, a set of edges, E, and a restricted number, K, of colors, the problem is to assign exactly one color to each vertex such that no two vertices of the same color are connected by any of the edges in E.

The problem has applications in many areas, including CPU-register allocation, scheduling and (in fact) Sudoku puzzles, but it is often motivated by the map-coloring problem: given the map of a country, divided into different states or regions, can the regions be colored such that no two bordering regions have the same color? Converting a country map into a graph is a simple process of assigning one vertex per region and then connecting two vertices with an edge if and only if their regions share a border. [1]

Figure 5 illustrates this conversion and shows a coloring where K =4. Brief analysis reveals that this map cannot be colored with fewer than 4 colors. In graph theory, this means that the *chromatic number* of the graph is 4. The famous Four-Color Conjecture states that any 2-dimensional map has a chromatic number of 4 or less. However, graphs that do not correspond to 2-d maps can have chromatic numbers as high as the size of the vertex set (in cases where all vertices are connected to all others).

For our purposes, the task is not to determine a graph's chromatic number, but only to come up with a coloring, given a fixed number of colors. To do so, we can treat each vertex as a variable, with its possible values being the K different colors. Then, the edges constitute the constraints, and checking for constraint violations is as simple as testing whether or not the two vertices of an edge have the same or different colors. This constraint-based representation of the problem suffices to use either of the above two local-search methods (simulated annealing or min-conflicts) to attack graph-coloring.

9 The Third Puzzle

You are free to choose the third type of puzzle, but it **cannot** be the Egg Puzzle (shown above). Your chosen puzzle should allow the following:

[1] Note that borders must be more than a single point: they need to meet at more than just a corner. Such corner intersections do not merit edges in the graph.

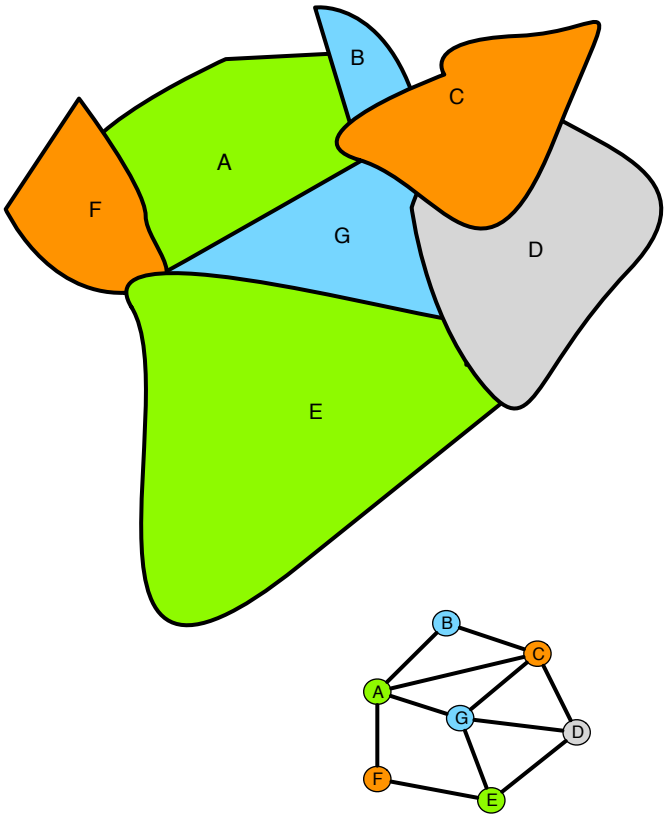

Figure 5: Illustration of a fictitious country map and the corresponding graph, both properly colored such that no two bordering regions (connected vertices) share a color.

1. Representation as a constraint-satisfaction, local-search problem. It should be easy to define variables, their possible values, and the constraints.

2. Scaling, such that easy, medium and hard versions of the puzzle can be generated and tested. For example, K-Queens is scaled by raising or lowering K, and Graph Coloring is scaled by the size and/or connectivity of the graphs. Sudoku and similar puzzles can be scaled by using instances classified as *easy*, *medium* and *hard* by the newspaper or magazine in which they appear.

3. The straightforward definition of an evaluation function (typically in terms of satisfied and violated constraints).

10 Puzzle Testing

For each puzzle, you must test 3 different instances (an easy, medium and hard variant) using both SA and MC; each combination of instance and method must be run a minimum of 20 times, with the results summarized in a table. For each puzzle, you must produce one table with 6 rows, one for each combination of instance and method. Each such row must contain the following:

1. The average and standard deviation of the evaluations of the 20 best-of-run states. The best-of-run state is the state with the highest evaluation found during a run of the algorithm on a puzzle.

2. The best evaluation found over all 20 runs. This is the best of the best-of-run states.

3. The average and standard deviation of the number of steps that the algorithm needed to find an optimal solution on each of the 20 runs. A *step* or *iteration* is a pass through the main loop of the search algorithm. In SA, one such iteration generates many successor states and chooses among them, while in Min-Conflicts, one iteration finds the successor state that minimizes conflicts. You will need to set an upper limit on iterations, particularly for SA. A value of 10,000 is suitable for this exercise.

4. The quickest (in terms of number of steps) that an optimal solution was found among all 20 runs.

Finally, for the **easiest** of the 3 variants of a puzzle, show the best solutions that SA and MC were able to find. Visualize these solutions in a manner that clearly allows the naked eye to verify their correctness.

These are the variants that must be run for the two given puzzles:

- K-Queens: K = 8, 25, and 1000
- Graph Coloring: The 3 files provided on the *Supporting Materials* section of the course web page. You will have to decide yourself which of these are easy, medium and hard. Use K = 4 for all variants.

For the third puzzle, try to choose hard versions that take the best method (normally Min-Conflicts) at least a minute to solve on your machine. This may not be possible for certain puzzles, such as Sudoku, but when possible, make the hard versions quite challenging for your machine!

Be aware that simulated annealing may not be able to find optimal solutions for much more than the easiest versions of each puzzle. Min-Conflicts, however, should be able to solve the medium and hard versions of each puzzle.

11 Deliverables

1. Working GPS code that clearly and cleanly separates the search algorithms from search-state creation, modification and evaluation. (20 points)
2. A diagram or 2 (along with a brief description) showing the main classes and most important methods in your GPS. (10 points)
3. A clear description of your third puzzle, with special focus on the constraint-based representation and the evaluation function. (10 points)
4. Clear demonstration that your system solves K-Queens puzzles for most of the 6 variant-method combinations. (10 points)
5. The results table and description of easy-case solutions (as discussed above) for the K-Queens problem (10 points)
6. Clear demonstration that your system solves Graph-Coloring puzzles for most of the 6 variant-method combinations. (10 points)
7. The results table and description of easy-case solutions for the Graph-Coloring problem (10 points)
8. Clear demonstration that your system solves Puzzle 3 for most of the 6 variant-method combinations. (10 points)
9. The results table and description of easy-case solutions for Puzzle 3 (10 points)

During the demonstration, you should be able to test any of the 3 puzzle types and any of their 3 variants using either of the search methods. This should **not** require any modification of your code, recompilation, etc. A simple interface should handle the different requests for puzzle, variant and method.