AI Intro (TDT-4136) Homework # 3:
Local Search Using Simulated Annealing

Purpose: Gain a detailed understanding of local search by implementing and using simulated annealing (SA) to solve a few puzzle problems.

1 The Simulated Annealing Algorithm

The local search procedure known as simulated annealing (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.

Being a local search method, SA works with **complete solutions** (a.k.a. *attempts*) that are gradually modified in the search for an optimal solution. This differs 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 SA, the search can be stopped at any time to yield a viable (though probably not optimal) solution.

Consider the problem of finding an optimal combination of settings for the dials on a control panel that regulates a complex industrial process. Let V 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. 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.

1.1 Designing the Solution Representation for Local Search

Although the 8 Puzzle (see your textbook for details) is traditionally used to illustrate best-first search using A*, it is also amenable to local search methods such as SA. Here, we use it simply to illustrate the representational issues of using local search.

When formalizing a problem to be solved by a local-search method, the data structure(s) chosen for the search-state representation are often critical to success. This structure should be as easy as possible to evaluate via the objective function, and it should be easy to copy and modify when generating neighbor solutions in search space.

When solving the 8 puzzle with A*, one typically uses a simple array of numbers (the tiles and the hole) as the state representation; new states are generated by moving the hole one spot to the east (0), north (1), west (2) or south (3). The complete solution (if found) is then a path through the search graph from the goal to the start state.

In local search, each state represents an entire solution, not a single board configuration. In reality, a solution represents the entire **sequence** of board configurations, from start to finish. However, it behoves us to compress this sequence down to its essential elements, and those are the **moves** themselves. Given a starting board configuration and a sequence of moves, anyone (or any computer) can generate the sequence of board states. They are completely determined by the starting configuration and moves.

So a compact representation of an entire 8-puzzle solution might be a simple data structure (i.e. a list or 1-dimensional array) comprised of the integers 0 - 3:

2 3 0 1 1 1 2 3 0 0 0 3 1 3 1 2 2 2 0 .....

This represents a series of moves (of the hole) beginning with west (2) then south (3) then east (0) then north(1), north(1) and north(1). Since these solutions are generated and modified somewhat randomly, they may be far from **good** solutions. In fact, the term *attempt* often seems more appropriate. However, *solution* is common in the search literature, with *optimal solutions* being the desired ones.

The solution above may not even be legal, since the 3 consecutive northerly moves of the hole could not be performed on the classic 8-tile version of the puzzle. The objective function would need to penalize it in some way for this impossible/wasted move.

To evaluate this representation, a board with 8 tiles must be modeled and the moves simulated on it. The board configuration at the end of the move sequence can then be compared to the goal configuration to give a quantitative measure of the solution's quality. In this case, the objective function would probably use metrics such as the Manhattan distance, just as is in the common heuristics for A* approaches to this puzzle. Clearly, heuristics and objective functions have a lot in common, since both express the quality of a partial or whole solution, respectively.

In general, the first key specialization needed to apply SA (and other local search methods) to a problem such as the 8 puzzle is to code up a simulator of these changing board configurations so that move sequences can be evaluated. To generate neighbors for the 8 puzzle is much simpler: just change one or more of the integers in the move sequence. So the second key specialization for using SA, neighbor generation, is quite trivial, given the proper representation (i.e. data structure).

1.2 The Objective Function

The success of local search is strongly dependent upon the objective function.

Local search algorithms generally have little knowledge about *how* a good solution/hypothesis should be designed - as discussed above, neighbors are normally generated quite randomly. Lacking this *intelligent-design* knowledge, they require the ability to **recognize** a good design/solution. This not only entails recognizing optimal solutions, but knowing the difference between horrible, mediocre, promising, excellent and optimal solutions.

The objective function needs to put a number on quality so that different solutions can be compared. If optimal solutions receive a score of 1, and horrible solutions get 0, then promising variants might get 0.5, while excellence but non-optimality would garner a 0.9. If your objective function is not able to give partial credit, i.e., if it only can differentiate between *the best* and *the rest*, then it will be of little assistance to local search.

The reasonably good solutions need an evaluation that reflects their promise so that search can move in their direction. Otherwise, local search becomes a process of fumbling around in the dark in search of a midnight snack wrapped in cellophane. By adding partial credit to the objective function, you unwrap the cellophane, allowing the food odors to drift through the room and give clues as to the distance and direction of the snack.

1.3 Generating Neighbors

In theory, the solutions to many problems, especially those involving real numbers, have an infinite number of neighbors. If the solution representation is a vector of real numbers, then one, some or all of those values could be changed by any real-number increment. Methodical restrictions are therefore needed in the neighbor-generating procedure; these constraints essentially define the concept of neighbor for the search process.

One possibility is to consider a neighbor to be a vector that differs from the current vector, V, in just one feature. You may even define a standard difference for each feature. For example, if your current point in the search (i.e. the current solution) corresponds to the 3-feature vector V = [3, 0.5, 0.1], and if the *standard difference vector* is dV = [1, 0.2, 0.04], then the following would be the complete set of neighbors for V:

[2, 0.5, 0.1]

[4, 0.5, 0.1]

[3, 0.3, 0.1]

[3, 0.7, 0.1]

[3, 0.5, 0.06]

[3, 0.5, 0.14]

You may choose to use a more stochastic neighborhood generator in which randomly-generated increments are applied to the current vector, possibly with several features being modified at once. The main point is that you want neighbors to be relatively close to the original, V. Otherwise, if they are vastly different, then your search process will essentially be taking very long jumps in the search space, often hopping far away from the area around V. If V has a reasonably good evaluation, then you want search to continue in that area, since V might be close to the optimal solution. Constantly taking long jumps produces more of a random than a methodical search, so *choose your neighbors wisely.*

2 The Assignment

Your task is to implement the SA algorithm (either the version above or that found in your AI textbook) and use it to attempt to solve the two puzzles below. **Do not download someone else's code. Doing so can result in a failing mark not only on this homework, but on the complete homework set, making you ineligible to take the final exam.** AI is all about programming, not cutting and pasting.

For both puzzles, you must document:

1. The representation of solutions that you chose/designed. Use text and a diagram or 2.
2. The objective function - described using a combination of mathematical expressions and text.
3. The neighbor-generating procedure - described briefly, using an example.

Each puzzle may involve several variants, as shown below. You must attempt them all. For each variant, show the best solution that SA found, preferably with a picture (either hand-drawn or computer-generated).

2.1 The 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 K eggs.

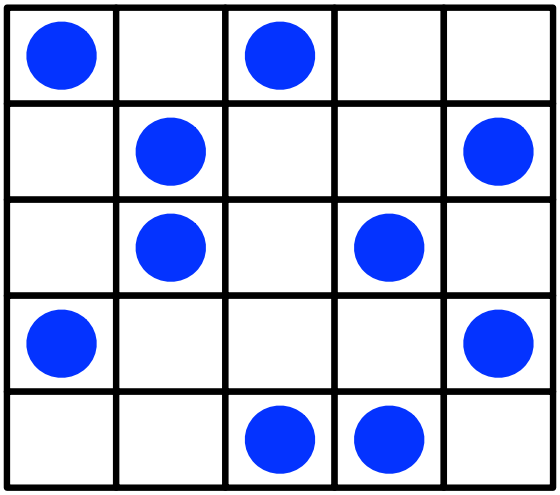

Figure 1: A solution to the egg carton puzzle where M=N=5 and K=2.

Here, ALL diagonals need to be considered, not just the main diagonals.

A common version of the problem is M = N = 6 and K = 2, which has an optimal solution involving 12 (6 x 2) eggs. Figure 1 shows a solution for the case where M=N=5 and K=2.

Puzzle Variants (all of which you must attempt):

1. M = N = 5 and K = 2
2. M = N = 6 and K = 2
3. M = N = 8 and K = 1 (Does this problem look familiar?)
4. M = N = 10 and K = 3

2.2 The Switchboard Puzzle

An electrician encounters a rectangular switchboard consisting of M rows and N columns of pegs. Given a pre-defined starting and ending peg, the task is to connect all of the pegs with one wire of the shortest possible length. The key constraints are:

1. The wire cannot cross any peg more than once.
2. The distance between neighboring pegs in the vertical or horizontal direction is D.
3. Anytime the wire turns at a peg P, it needs to be wound completely around P before turning. This uses W additional units of wire.
4. The wire is wound around the first (but not the last) peg of the sequence.

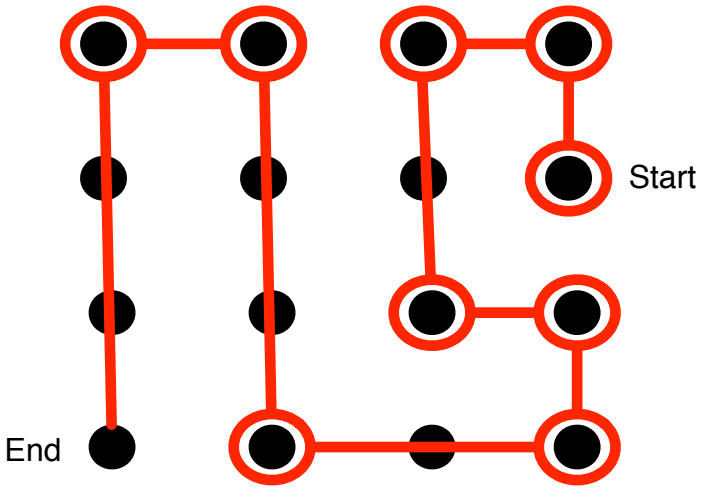

Figure 2: An optimal solution to the switchboard problem with M = N = 4, D=3, and W = 2. The total length of wire is 45 + 18 = 63 units: 3 x 15 = 45 to connect all 16 pegs, and 9 x 2 = 18 for wrapping around the start peg and 8 different pegs where turns occur.

Figure 2 shows the solution to a version of the problem where M = N = 4, D=3, and W = 2; and where the start and end points are as shown in the diagram. Clearly, the locations of these two points have a huge effect upon the logical complexity of the task. For example, if either M or N is an odd number, then the problem is simple for a start point in one corner and an end point is the diagonally opposite corner.

Puzzle Variants (all of which you must attempt:

1. M = N = 4, D=3 and W = 2. Start position = 1 peg below the upper right corner. End position = bottom left peg.
2. M = 6, N = 5, D=3 and W = 2. Start position = upper right peg. End position = bottom left peg.
3. M = N = 8, D=3 and W = 2. Start position = 1 peg below the upper right corner. End position = bottom left peg.

Note: This is not an easy puzzle, so your SA might have problems with one or more of the 3 variants. If you are really ambitious, try using your A* algorithm to solve the problem instead. This is not a requirement, just a suggestion for those who either a) enjoy puzzles or b) hate puzzles and like to get their computers to do the work for them.

3 Deliverables

You must deliver the following items:

1. A well-commented version of your SA code, both the general algorithm and the specialization code for at least one of the two puzzles.
2. Descriptions of the 3 key aspects involved in the specialization of your SA code to one or both of the puzzles above: representation, objective function and neighbor generation.
3. Diagrams (hand-drawn or computer generated) of the solutions found by SA for each of the puzzle variants listed above, for either one or both puzzles.
4. An answer to the following question:
   (a) Discuss the similarities and differences between heuristics and objective functions.