# CSE257 Winter 2025: Assignment 2

Deadline: Feb-16 11:59pm. You will need to make the following two submissions.

1. Submit a pdf file of your (written or typed) answers on Gradescope (if you are enrolled, you should have already been added to the course on Gradescope).
2. Submit your Python code or Jupyter notebook files for problems that have the "Implementation Involved" label, via the Dropbox link: `https://www.dropbox.com/request/XzChzoa7FtsJ7x4e0yR5`. You can import any library that does not already implement the algorithms we ask for. Submit all the code in one zip file. For these questions that have implementation involved, you need to still put whatever is asked in the questions, such as plots and explanations, in the pdf file that you submit to Gradescope. Otherwise the question will be considered unanswered for grading purposes.
3. Again, use of ChatGPT or other language models is strictly forbidden for anything you write in the answer pdf file (even for editing grammar), but they are allowed for the implementation and the code you submit.

**Question 1** (Implementation Involved)**.** Consider the following functions:

- $f_1(x) = x_1^2 + x_2^2$
- $f_2(x) = x_1^2 + x_2^2 + \sin(10x_1)\sin(10x_2)$
- $f_3(x) = \sum_{i=1}^{50} x_i^2$ (yes it's a 50-dimensional function, don't freak out)

You will need to implement the following algorithms to answer the questions below.

- (SA) Simulated Annealing with initial point $(1, ..., 1)$ (vector of all ones of the right dimensionality), initial temperature $T_0 = 1$ and the annealing schedule $T_k = T_0 \cdot (0.95)^k$ for each $k$-th iteration. In each step, the movement should be sampled from a Gaussian $\mathcal{N}(\mu = 0, \Sigma = 0.1I)$ where $I$ is the identify matrix of the appropriate dimension.
- (CE) Cross-Entropy Methods with $50$ samples in each iteration, and the elite set is formed by the top $20\%$ of all samples in each iteration. The initial distribution start from Gaussian distribution with mean $\mu$ being $(1, .., 1)$ (vector of all ones), and $\Sigma = I$, both of the appropriate dimensions.
- (SG) Search Gradient method with $50$ samples in each iteration, and use a fixed step size of $\alpha = 0.02$ in each iteration. The initial Gaussian distribution should have mean $\mu = (1, ..., 1)$ and $\Sigma = 0.1I$, where $I$ is the identity matrix of appropriate dimensions.

  Sometimes your samples may cause numerical issues in the Gaussian and give you exploding gradients. If that happens you should put an upper bound the Euclidean norm of the gradient to be less than 10. Also if you see problems where the covariance matrix fails to be positive semi-definite, you can skip the update on the covariance matrix in that iteration (there are various ways to avoid this but we don't need to worry about it here), but still update the mean in that iteration.
- (GD) Standard gradient descent with initial point $(1, ..., 1)$ with constant step size 0.02. Do not normalize the gradients.

All algorithms will be run for **100 iterations**.

For each function, our goal is to **minimize the function**, so make sure your updates are in the right direction, which can be different from the formulas in the slides.

For SA, CE, SG, because the algorithm is stochastic, you should repeat each experiment on them using 3 different random seeds. So for each function, you should produce 3 plots, each with a different random seed.

1. (0 Point, meaning you don't have to do it but it's interesting to see it) Plot the 3D shape of the function $f_2$.

2. (3 Points) For $f_1$ and $f_2$, in the $x_1$-$x_2$ plane, plot the following within the range of $x_1, x_2 \in [-2, 2]$:

   - 20 level sets of the function of values $\{0.1, 0.2, ..., 2\}$. No need to separate these different levels with different colors, basically just grid up the space finely and put down a solid point when the function value hits one of these levels.
   - The sequence of points (for CE and SG, just plot the mean of the Gaussian in each iteration) you get from each algorithm above, after 100 iterations. Use different colors for each algorithm so that we can see their different behaviors.

   All of these on the same plot. In each plot the coordinates range from $-2$ to 2. Again for each function, repeat 3 times with different random seeds to generate 3 plots. So in total there should be 6 plots.

3. (1 Point) For $f_3$, since it's high-dimensional, only plot how the function value (for CE and SG, plot the average function values of samples) changes over the number of iterations (100 iterations in total), under each of algorithm. Repeat for 3 random seeds, so show 3 different plots. Again if the covariance matrix is not positive definite in an update, you can keep it unchanged and only update the mean. And just for updating the mean, if gradient in SG gets too large, then normalize it to be unit vector before multiplying with the step size.

4. (1 Point) Compare the performance all algorithms on the functions above and give some guidelines about when to use which method and why (in terms of dimensions, convexity, randomness, etc.). There is no single correct answer, just focus on explaining your own understanding.

5. (Extra 2 Points) Do more experiments by changing the hyperparameters of CE and SG, such as sample size, elite set size, step size, etc. and explain how these different choices affect performance on the same three functions. Again no single correct answer, but make it a useful guideline for people who are fresh users of these algorithms and do not know how to choose the parameters.

**Question 2** (Implementation Involved)**.** Using the starter code here for the 2048 game:

`https://github.com/ucsdcsegao/expectimax`

Implement two versions of Expectimax with vertical cut-off: one with Expectimax on just a depth-1 tree (we refer to it as "Exp-1") and one with a depth-3 tree ("Exp-3").

The Exp-1 tree is trivial of course: it simply has the current game state as the root, and its children nodes correspond to the four actions as the leaf nodes, with payoff defined by the game score (the number displayed on the upper left corner in the game interface) immediately annotated for these leaf nodes.

The Exp-3 tree is the game tree that contains all paths following the sequence of "player-move, computer-move, player-move." Use the score provide by the game engine as the payoff value at any leaf node in these trees (i.e. the evaluation function here).

If any action becomes infeasible (i.e. it becomes impossible to move the tiles in a particular direction), you can annotate the corresponding child state as terminal with some very negative payoff such as -100 (you can feel free to design anything, as you'll only need to plot the actual game scores later).

*Warning: The minimax/expectimax algorithm is conceptually simple, but to make it work in an actual game it may take longer than you expected, especially if you are not very familiar with Python (Hint: "deep copy"). Start early.*

1. (2 Points) Plot the performance of Exp-1 and Exp-3 in the following way. For each version, plot 5 runs of the algorithm starting from the beginning of the game for 1000 moves (i.e., call the algorithm 1000 times), and show how the **actual** game score changes (i.e. what the game engines returns after you make each move. Annotate the y-axis in the plot with this value.) over the number of iterations (x-axis). In any run, if the agent fails the game early (can't move anywhere) then just stop the sequence there.

2. (2 Points) Design a different evaluation function for Exp-3 (use something different as pay-off values at the leaf nodes of the depth-3 tree), so that it performs better than the Exp-3 with the previous evaluation function that simply uses the original game score. You can use any information from the game state, such as highest tile, pattern of the existing tiles, etc. Design your plots to show that the new algorithm (i.e. Exp-3 with the evaluation function you designed) is better than the original Exp-3.

3. (1 Point) Explain how you would model the entire 2048 game as a Markov Decision Process. That is, describe the states, actions, transitions, reward, etc. so that the optimal policy to that MDP corresponds to the optimal way of solving 2048. Now how big is the state space, and how long will a single value iteration take in this MDP? Give a rough estimate and feel free to make assumptions on any part you need for giving this estimate.

**Question 3.** Consider the blackjack game as modeled by the starter code here:

`https://github.com/ucsdcsegao/blackjack`

You don't have to implement anything for now, since I don't want you to freak out for having two major programs to write in one assignment. (We will continue to ask more questions about this in the next assignment that involves implementation, so this is just to get you started to get familiar with code right now.)

1. (1 Point) Roughly describe the MDP for blackjack as implemented in the code, i.e., states, actions, rewards, etc. (no need to give the full table of the probabilities of transitions, of course).

   Write out one arbitrary trajectory that starts from an initial state and takes more than one step to reach the terminal state. The trajectory should be a sequence of tuples $(s_k, a_k, s_{k+1}, r_{k+1})$ with $k \geq 0$.

2. (1 Point) Define a good policy $\pi_g$ and a bad policy $\pi_b$, intuitively justify why they are good or bad. Then give one example trajectory that may be generated under the good policy $\pi_g$ that leads to losing the game. And then give one example trajectory that may be generated under the bad policy $\pi_b$ that leads to winning the game. Again, a trajectory is just a sequence of $(s_k, a_k, s_{k+1}, r_{k+1})$ that starts from some possible initial state and ends at a terminal state.

**Question 4 (Canceled and moved to next assignment).** For each of the following claims about value iteration for MDPs, determine whether they are true or false. If a statement is true, give a proof (doesn't need to be entirely formal but convincing enough with no logical loopholes) and if it's false, give an example (explicitly give an MDP) that shows why you say that.

1. (0 Point) After each round of Bellman update, the value on each state either increases or stays the same, but can not decrease since we are maximizing values.

2. (0 Point) The values of all states (as a vector of $n$ values for $n$ states) converge to the optimal value vector in L2 norm. (you may want to look up something called equivalence of norms)

3. (Extra 0 Point) The Bellman update in value iteration defines a contraction in L2 norm.

4. (0 Point) Suppose in one Bellman update $V' \leftarrow B(V)$, the distance of movement $\|V - V'\|$ is $\delta$. With just that information, we know how far at most $V'$ is from the optimal $V^*$. (Give some calculation if you think it's true.)