15-780: Problem Set #1

February 14, 2014

1. **Convex sets, functions, and optimization problems [20pts]**. This question will ask you to prove or disprove whether certain sets, functions or optimization problems are convex. For the first part, you will need to rigorously prove why each set, function or problem is convex or not, but for the second part you can give a more intuitive or graphical explanation.

   Rigorous proofs Prove whether the follow sets, functions, or optimization problems are convex or not. If you want to claim they are convex, you need to: 1) for a set, show that convex combinations of elements within that set are also within that set, or show that the set is a sub-level set of some convex function (where you can prove it is convex by the same methods as follows); 2) for a function you need to directly verify that convex combinations lie above the function, prove that the second derivative (Hessian, for multivariate function) is positive (positive semidefinite) for all inputs, or show that it is a combination of valid rules for producing new convex functions from the slides; 3) for optimization problems, you need to prove that the the objective and constraint set are convex.

   If you claim that a set, function, or problem is *not* convex, you need to: 1) for a convex set, show that there exist two points x_1 and x_2 that are within the set but that there is a $0 \leq \theta \leq 1$ such that $\theta x_1 + (1-\theta)x_2$ is not in the set. For functions, you need to show the equivalent thing for function values, and for optimization problems you need to show that the objective or one of the constraints is non-convex.

   (a) The set of all points $x \in \mathbb{R}^n$ closer to a point $x_0 \in \mathbb{R}^n$ than another point $x_1 \in \mathbb{R}^n$

   $$\{x | \|x - x_0\|_2 \leq \|x - x_1\|_2\}.$$

   (b) The hyperbolic set

   $$\{x \in \mathbb{R}^2 | x \geq 0, x_1 x_2 \geq 1\}.$$

   Hint: You may want to use the arithmetic-geometric mean inequality, which says that for $a, b > 0$ and $0 \leq \theta \leq 1$ $a^\theta b^{1-\theta} \leq \theta a + (1-\theta)b$.

   (c) An ellipsoid in $\mathbb{R}^n$, defined by

   $$\{x \in \mathbb{R}^n | x^T P x + q^T x \leq r\}$$

   for positive definite $P \in \mathbb{R}^{n \times n}$, $q \in \mathbb{R}^n$, $r \in \mathbb{R}$.

(d) The function $f : \mathbb{R}^2 \to \mathbb{R}$

$$f(x) = \frac{x_1^2}{x_2}$$

for $x_2 \geq 0$. *Hint*: For this and the next problem you can use the fact that a 2x2 symmetric matrix $\begin{bmatrix} a & b \\ b & c \end{bmatrix}$ is positive definitive if and only if $a + c \geq 0$ and $ac \geq b^2$.

(e) The function $f : \mathbb{R}^2 \to \mathbb{R}$

$$f(x) = \frac{x_1}{x_2}$$

for $x_2 \geq 0$.

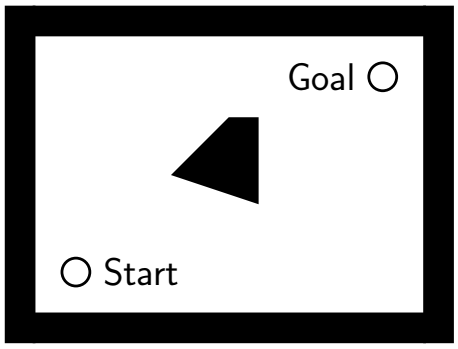
(f) The optimization problem

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & \|Ax - b\|_2^2 + \|x\|_2 \\ \text{subject to} \quad & \|x\|_2 \geq 1 \end{aligned}$$

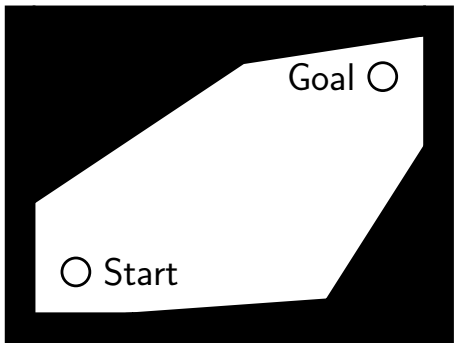
with problem data $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$.

Intuitive arguments Describe in a few sentences why you think the following problems are convex or not. You do not need to provide a rigorous proof, but you need to capture the main intuition that would render these problems convex or not.

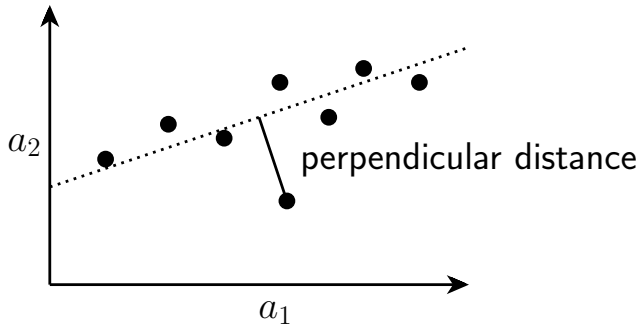
(a) Plan a sequence of "waypoints" for a robot in 2D space that 1) minimize the squared distance between the waypoints and 2) avoid any obstacles (dark regions).



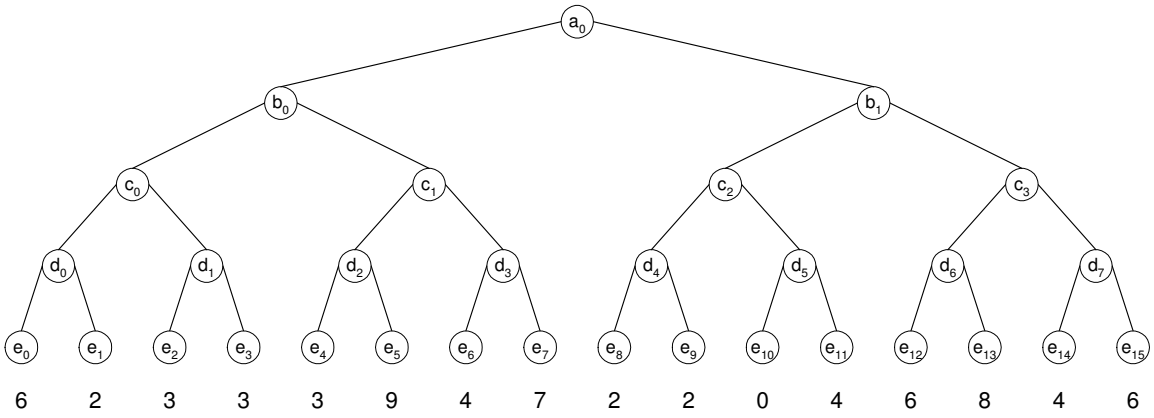
(b) The same question as above for the following room configuration



(c) Given a set of 2D points, find the line the minimizes the sum of the distances from all points to that line, measured perpendicularly from the line. What about if we used squared distances instead of the distances (this question will come up in an upcoming programming problem).



2. **Adversarial search [10pt]** These questions refer to the game tree below. For all questions, assume that the player making the move at the root is a "maximizing" player, i.e. he or she wants to get to a state with a high utility value.



(a) What is the solution? That is, which move should be made next and what is the expected value of that move?

(b) Using $\alpha - \beta$ pruning (and standard left-to-right evaluation of nodes), how many leaves get evaluated? Indicate all parts of the tree that are cut off. Indicate the winning path or paths. Strike out all static evaluation values that do not need to be computed.

(c) How does the answer to the previous problem change if right-to-left evaluation of nodes is used?

3. **Local search [15pt]** For this problem, you are to describe how you would structure a hill-climbing approach to solving the Traveling Salesperson Problem (TSP). Specifically, you should provide descriptions for the following:

- The data structure used to represent a solution

- The method for generating the initial solution(s)
- The method for generating neighbors.
- The method for evaluating solutions.
- The method for selecting the best neighbor.
- The method for terminating the search.

In addition, you are to describe a situation where your hill-climbing approach would not provide an optimal answer and what techniques you could apply to improve the result.

4. **Optimization programming [20pt]**

In this question you will implement several optimization algorithms for different problems in Python. You are encouraged to use cvxpy whenever possible, as this will make your life a lot easier, but it is not required. We are providing the file `optimization.py` as the template for you to fill out for this assignment. Instructors for setting up cvxpy on different architectures (will be available shortly if it is not already) is at `http://www.cs.cmu.edu/~zkolter/course/15-780-s14/cvxpy.html`. All your functions should return values as cvxopt.matrix variables (the Weber point example does this). You will need to write four functions (we also provide the `weber_point` function from class as a way of illustrating the desired format for these functions)

(a) **Least squares** Given a matrix $A \in \mathbb{R}^{m \times n}$ and vector $b \in \mathbb{R}^m$, find the vector x that solve the optimization problem

$$\underset{x}{\text{minimize}} \quad \sum_{i=1}^{m} (a_i^T x_i - b_i)^2 \tag{1}$$

where a_i^T denotes the ith row of A. Write this code in the `least_squares(A,b)` function.

(b) **Least absolute errors** Similar to the above, given a matrix $A \in \mathbb{R}^{m \times n}$ and vector $b \in \mathbb{R}^m$, find the vector x that solves the optimization problem

$$\underset{x}{\text{minimize}} \quad \sum_{i=1}^{m} |a_i^T x_i - b_i| \tag{2}$$

where a_i^T denotes the ith row of A. Write this code in the `least_abs(A,b)` function.

(c) **Linear programming** Write a function that will solve general problems of the form

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & c^T x \\ \text{subject to} \quad & Ax = b \\ & Fx \leq g \end{aligned} \tag{3}$$

where $x \in \mathbb{R}^n$ is the optimization variable, $c \in \mathbb{R}^n$, $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$, $F \in \mathbb{R}^{p \times n}$, and $g \in \mathbb{R}^p$ are the problem data. Write this code in the `linear_program(c,A,b,F,g)` function.

(d) **Minimum distance hyperplane** This problem is a bit trickier and may or may not be convex (we ask this in question 1 above). A *hyperplane* is defined by the variables $x \in \mathbb{R}^n$ and $y \in \mathbb{R}$ and the equation $x^T a + y = 0$, and the *squared distance* from the hyperplane to a point $a_i \in \mathbb{R}^n$ is given by

$$\text{dist}^2(a_i, (x, y)) = \frac{(x^T a_i + y)^2}{\|x\|_2^2}.$$

Now, given a set of points $a_1, \ldots, a_m$, find the hyperplane parameters x and y that minimize the sum of squared distances

$$\underset{x,y}{\text{minimize}} \quad \sum_{i=1}^{m} \frac{(x^T a_i + y)^2}{\|x\|_2^2}.$$

If this is not a convex problem, you can develop alternative methods (perhaps using the least-squares approach above), in order to solve this problem. Write this code in the `min_distance_hyperplane(A)` function, where the *rows* of A will correspond to the different points; this function should return the tuple (x, y).

5. **Search programming [35pt]** For this problem, you will write a program that can discover the series of moves that transform a moving tile puzzle from an initial state into a desired goal state. For example, for a 4x4 puzzle, given the following initial state:

1	2	3	4
5	6	7	8
9	10		12
13	14	11	15

and the following goal:

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

your program should return something like [‘down’, ‘right’], i.e., to solve this puzzle the blank has to be first moved down and then moved to the right. A number of different search algorithms may be used to solve this problem with varying degrees of success. In this assignment, you are required to try to solve it using the following search algorithms: breadth-first, iterative-deepening, A*. For A*, you will implement both the misplaced-tiles and Manhattan-distance heuristics. The possible moves are ‘left’, ‘right’, ‘up’, and ‘down’, and the successor function should generate children by applying those operators in that order.

For this assignment, you will need to implement four functions within the `search.py`: `breadth_first`, `iterative_deepening`, `astar_manhattan` and `astar_misplaced`. These functions each take as input an initial configuration for the 4x4 puzzle as a 16 element list (with the zero represented the empty tile), e.g., [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 0, 12, 13, 14, 11, 15], and will output a tuple where the first elements contains a list of commands to reach the goal, and the second element contains the total number of nodes visted (i.e., the number of times you check if a node is a goal state). You should try your program on a number of puzzles with different initial states. Because the goal state cannot be achieved from all possible states generated by randomly placing the tiles on the board, you should write a function that shuffles a puzzle from the goal state to an initial state by repeatedly moving the blank to a position randomly chosen from the possible moves. The depth of the solution for your shuffled puzzle will be no greater than the number of times the blank is moved.

Here are some general hints on this problem (we may follow up with additional ones depending on any issues that arise):

- Although some search algorithms were presented using recursion in the class slides, Python doesn't handle recursion particularly well, and so it is better to maintain the list of nodes explicitly as a Python list; you can emulate a stack or a queue by appending items either to the beginning or the end of the list.
- For A*, you typically want some form of "priority queue" to be able to find elements with minimum cost; for real implementations, you would want to use a data structure like a heap to implement this queue effectively, but here you can just maintain two lists (one with nodes and one with costs), then find the minimum cost element using the `min` and `index` calls in Python (must less efficient, but simple).
- Your life will be a lot easier if you create a class to represent nodes in the search tree. This should emulate the structure from the initial slides lecture. This way, your node list can consist of Node variables. If you implement the `__eq__` operator for this class to simply check if two nodes represent the same state (that is, they don't need to have the same parents, path costs, etc, to be considered equal), then you can quickly check to see if a node already exists in your list.
- Because states can repeat, you may also want to maintain a separate list of all explored states, and only add nodes to the list if they have not already been explored.