15780: GRADUATE AI (SPRING 2017)

Homework 1: Motion Planning and Optimization

Release: February 8, 2017,
Due: February 21, 2017, 11:59pm

1 Theta* [20 points]

In class, the Theta* algorithm was presented where the world was discretized into cells and the center of each cell was a vertex in our search space. In this problem, we will consider the Theta* algorithm as it was described in the original paper that presented it. Here the world is discretized into cells, but there is a vertex at the corner of each cell, and there is an edge between every two vertices on an unblocked cell. If a cell is blocked, there are still vertices on the corners and edges on the sides, but there are no edges that go into the cell. An example of a four-by-four grid with one blocked cell is shown in Figure 1. For this problem, assume that if a cell is blocked, the entire cell is actually blocked, so the optimal path in a graph cannot enter a blocked cell. In any of the problems below, if you provide a counterexample you must formally describe why it is a valid counterexample. Note that for this problem, the solution to some of these may be easily found online if you go digging for material on Theta*; this is **not** allowed. If you have questions about the Theta* algorithm, please ask on Piazza or come to office hours.

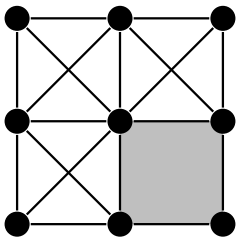

Figure 1: Example grid with one blocked cell.

1.1 Vertices that can('t) handle optimal paths [5 points]

Clearly on the graph described above, the optimal path between two nodes cannot always be represented (e.g., in Figure 1, the optimal path between the bottom left vertex and the the middle vertex on the top would be the straight line between them, which cannot be represented by the edges on the graph). Therefore we know that an algorithm that only considers edges on this graph cannot be optimal. But recall that Theta* does not only consider direct edges in the graph when constructing it; however we know that the path it finds will always consist of unblocked straight lines from one vertex to another.

So let us consider a new graph where every two vertices on our original graph that have an unblocked straight line between them have an edge in this new graph. Can the optimal path always be represented by edges on this new graph? What if we consider the same kind of graph but where the vertices are in the center of each unblocked cell (as described in class) rather than on the corners of the cells—i.e. a graph that has all straight line unblocked paths between these vertices as edges? Prove your answers.

1.2 Theta* Optimality [15 points]

Prove or disprove the following statement: In the setting described above, Theta* will always find the optimal path.

2 Convexity [25 points]

2.1 Sets [10 points] [1]

Identify if the following sets are convex or not and provide a proof or counterexample for each.

(a) (3 points) The set $\{x \in \mathbb{R}^n | \alpha \leq a^T x \leq \beta\}$ for some $\alpha, \beta \in \mathbb{R}$.

(b) (3 points) The set of points closer to a set than another, i.e., $\{x | \text{dist}(x, S) \leq \text{dist}(x, T)\}$ for some distance function (that is a metric) and $S, T \subseteq \mathbb{R}^n$.

(c) (4 points) The set $\{x > 0 | \prod x_i \geq 1\}$.
(Hint: If $a, b \geq 0$ and $0 \leq t \leq 1$, then $a^t b^{1-t} \leq ta + (1-t)b$.
You must prove the hint if you use it. You may use the fact that $\log x$ is concave without proof.)

2.2 Optimization Problems [15 points]

Which of the following mathematical programming problems are convex? Prove your statements. It might be helpful to try sketching the objective functions or the sets we are optimizing over.

(a) (5 points) The optimization variables are $\boldsymbol{x} = (x_1, x_2) \in \mathbb{R}^2$.

$$\begin{array}{ll} \text{minimize} & 3x_1 - 5x_2 \\ \text{subject to} & x_1^2 + x_2^2 \leq 1 \end{array}$$

(b) (5 points) The optimization variables are $\boldsymbol{x} = (x_1, x_2) \in \mathbb{R}^2$.

$$\begin{array}{ll} \text{minimize} & 3x_1 - 5x_2 \\ \text{subject to} & x_1^2 - x_2^2 \leq 1 \end{array}$$

(c) (5 points) The optimization variables are $\boldsymbol{x} = (x_1, \ldots, x_n) \in \mathbb{R}^n$. Also A is an $m \times n$ matrix and $\boldsymbol{b} \in \mathbb{R}^m$.

$$\begin{array}{ll} \text{minimize} & \exp\left(\sqrt{\sum_{i=1}^n x_i^2}\right) \\ \text{subject to} & A\boldsymbol{x} \leq \boldsymbol{b} \end{array}$$

3 Simplex [25 points]

For this problem, you will be implementing the simplex algorithm in Python. Your code should go in `simplex.py`. You are **not** allowed to use `cvxpy` or any other convex optimization library for this problem.

[1] These are exercises from Boyd & Vandenberghe's book and are copied here for convenience.

3.1 Basic Simplex [15 points]

First you should implement the basic simplex algorithm as described in Lecture 5, Slide 31. You should use the Numpy package for matrix manipulations. If you haven't used Numpy before, some examples of common operations are provided for you in `numpy_examples.py`.

The inputs to your simplex method will be in the form of numpy arrays and matrices:

1. I: an array consisting of a feasible basis index set
2. c: a cost vector
3. A, b: a matrix-vector pair, your solution x should satisfy $Ax = b$.

Your output should be a tuple consisting of:

1. v: the value of the optimal solution.
2. x: the optimal solution.

We will grade your code based on whether it is able to obtain the optimal solution to a number of different linear programs. You have been given a set of test cases in the directory `test_cases` as well as a module called `test.py`, which you can use to test your code. Notice that because we're doing numerical computations, computations that should actually yield 0 will often be approximately 0 (e.g., in $[-10^{-15}, 10^{-15}]$). To avoid problems with this, whenever you want to check if a number is strictly negative, you should check if it's $< -10^{-12}$ and whenever you want to check if a number is ≥ 0 you should check if it's $> -10^{-12}$.

3.2 Two-Phase Simplex [10 points]

In the problem above, you implemented the simplex algorithm assuming you have an initial feasible basis to start with; this is how the algorithm was presented in class. In this problem, we will implement the Two-Phase Simplex algorithm, which first finds a feasible basis (phase one), and then proceeds with the basic simplex algorithm (phase two). To find a feasible basis, the algorithm will first solve another linear program whose solution will yield a feasible basis to the original linear program, which we can then pass to the simplex algorithm. We now describe how to find a feasible basis and why it works.

Recall that the constraints $Ax = b$ represents a set of equations:

$$a_{1,1}x_1 + a_{1,2}x_2 + \cdots + a_{1,n}x_n = b_1$$

$$a_{2,1}x_1 + a_{2,2}x_2 + \cdots + a_{2,n}x_n = b_2$$

$$\cdots$$

$$a_{m,1}x_1 + a_{m,2}x_2 + \cdots + a_{m,n}x_n = b_m$$

For each equation where b_i is negative, we multiply both sides of the equation by negative one. Thus the right side of the equations can now be represented by $b^+ = |b|$. To the left side of each equation, we also add an artificial variable z_i, so that we have the following:

$$\text{sign}(b_1)(a_{1,1}x_1 + a_{1,2}x_2 + \cdots + a_{1,n}x_n) + z_1 \qquad = b_1^+$$

$$\begin{aligned} \text{sign}(b_2)(a_{2,1}x_1 + a_{2,2}x_2 + \cdots + a_{2,n}x_n) \quad & +z_2 \quad = b_2^+ \\ & \cdots \\ \text{sign}(b_m)(a_{m,1}x_1 + a_{m,2}x_2 + \cdots + a_{m,n}x_n) \quad & + z_m = b_m^+ \end{aligned}$$

Let A^+ be the matrix that results from negating the equations where b_i is negative and adding the artificial variables. We will now solve the following linear program:

$$\min_{x,z} \sum_{i=1}^{m} z_i$$

subject to

$$A^+x = b^+, x \geq 0, z \geq 0$$

Notice that a feasible point for this linear program is to set $x_i = 0$ for $1 \leq i \leq n$ and $z_i = b_i^+$ for $1 \leq i \leq m$ (since all $b_i^+ \geq 0$). Thus the set of artificial variables can serve as a feasible basis for this new problem, which we can use to solve it with the basic simplex algorithm. Furthermore, notice that the minimum of the objective function can at best be 0, since it must be that all $z_i \geq 0$. If we find a solution such that $\sum_{i=1}^{m} z_i = 0$, then we have that $z_i = 0$ for $1 \leq i \leq m$, so

$$Ax = b$$

and $x_i \geq 0$ for $1 \leq i \leq n$, meaning the remaining non-zero x_i form a feasible basis for our original solution[2].

Write a function that implements the two-phase simplex algorithm. The inputs and outputs of the function should be the same as your simplex function, *except* that you will not be given I, the initial feasible basis set, as an input. Your code should call your simplex algorithm twice (once in each phase).

4 Integer Programming [30 points]

Develop an integer programming algorithm, based upon branch and bound, to solve Sudoku puzzles. Specifically, implement the function solve_sudoku in `sudoku.py` as follows:

```
def solve_sudoku(puzzle):
    # ...
    return (solved_puzzle, constraints)
```

The input of the function is a Sudoku puzzle, represented as a 9×9 "list of lists" of integers, e.g.,

```
puzzle = [[4, 8, 0, 3, 0, 0, 0, 0, 0],
          [0, 0, 0, 0, 0, 0, 0, 7, 1],
          [0, 2, 0, 0, 0, 0, 0, 0, 0],
          [7, 0, 5, 0, 0, 0, 0, 6, 0],
          [0, 0, 0, 2, 0, 0, 8, 0, 0],
          [0, 0, 0, 0, 0, 0, 0, 0, 0],
```

[2]There will possibly be a degenerate solution where more than m variables are zero, but you will not have to deal with any such cases for this problem.

```
[0, 0, 1, 0, 7, 6, 0, 0, 0],
[3, 0, 0, 0, 0, 0, 4, 0, 0],
[0, 0, 0, 0, 5, 0, 0, 0, 0]],
```

where zero entries indicate missing entries that should be assigned by your algorithm, and all other positive integers (between 1 and 9) indicate known assignments. The function solve_sudoku should return a tuple (solved_puzzle, constraints), where solved_puzzle is the input puzzle with all the missing entries assigned to their correct values. For instance, for the above puzzle, solved_puzzle should be

```
solved_puzzle = [[4, 8, 7, 3, 1, 2, 6, 9, 5],
                 [5, 9, 3, 6, 8, 4, 2, 7, 1],
                 [1, 2, 6, 5, 9, 7, 3, 8, 4],
                 [7, 3, 5, 8, 4, 9, 1, 6, 2],
                 [9, 1, 4, 2, 6, 5, 8, 3, 7],
                 [2, 6, 8, 7, 3, 1, 5, 4, 9],
                 [8, 5, 1, 4, 7, 6, 9, 2, 3],
                 [3, 7, 9, 1, 2, 8, 4, 5, 6],
                 [6, 4, 2, 9, 5, 3, 7, 1, 8]].
```

and the constraints value could be `constraints = [(8, 5, 2, 1), (5, 3, 4, 1)]`.

This implies that if we add two constraints $(x_{8,5})_2 = 1$ and $(x_{5,3})_4 = 1$ to the linear program for Sudoku (described in detail below) then the solution has only integer values and returns the solution above. Note that, since these values are all indexed staring at 1, adding/subtracting 1 can be needed to convert to and from Python indices. Also note that there can be more than one set of constraints that lead to the same integer solution. Thus, we will check your solution by directly plugging the constraints your algorithm returns in, not by comparing with our solution. Specifically, you need to do two things to solve this problem:

1. Write code to solve the linear programming relaxation of a Sudoku puzzle. Let $x_{i,j} \in [0,1]^9$ be the indicator of the (i,j)-th square in a Sudoku board, then solve the following optimization problem:

   minimize $\sum_{i,j}^{9} \max_k (x_{i,j})_k$

   subject to $x_{i,j} \in [0,1]^9 \;\; i,j = 1...9$ (i.e., all variables must be between zero and one)

   $\sum_{k=1}^{9} (x_{i,j})_k = 1, \;\; i,j = 1...9$ (i.e., each grid must have only one assigned number)

   $\sum_{j=1}^{9} x_{i,j} = \mathbf{1}, \;\; i = 1...9$ (i.e., each column must contain each number)

   $\sum_{i=1}^{9} x_{i,j} = \mathbf{1}, \;\; j = 1...9$ (i.e., each row must contain each number)

   $\sum_{m,l=1}^{3} x_{i+m,j+l} = \mathbf{1}, \;\; i,j \in \{0,3,6\}$ (i.e., each 3×3 box must contain each number)

   $(x_{ij})_k = 1$ if (i,j)-th square is $k \geq 1$ (i.e., assignments of the entries fixed by the puzzle).

You should write code to solve this problem using `cvxpy`. You can find additional documentation about `cvxpy` at its website `http://cvxpy.org`. You can also try to use the two-phase simplex algorithm you implemented instead of using `cvxpy`; however, in that case you will have to deal with the A matrix possibly not having full row rank (which requires some additional steps of removing linearly dependent rows of the matrix), possibly deal with degenerate initial solutions, and encode the max function above using linear equalities in standard form. To test this part of the assignment, you can use the following puzzle, where the linear programming relaxation happens to give an integer solution without any additional constraints:

```
puzzle = [[8, 5, 0, 0, 0, 2, 4, 0, 0],
          [7, 2, 0, 0, 0, 0, 0, 0, 9],
          [0, 0, 4, 0, 0, 0, 0, 0, 0],
          [0, 0, 0, 1, 0, 7, 0, 0, 2],
          [3, 0, 5, 0, 0, 0, 9, 0, 0],
          [0, 4, 0, 0, 0, 0, 0, 0, 0],
          [0, 0, 0, 0, 8, 0, 0, 7, 0],
          [0, 1, 7, 0, 0, 0, 0, 0, 0],
          [0, 0, 0, 0, 3, 6, 0, 4, 0]],
```

whose linear programming relaxation should give the following solution:

```
solved_puzzle = [[8, 5, 9, 6, 1, 2, 4, 3, 7],
                 [7, 2, 3, 8, 5, 4, 1, 6, 9],
                 [1, 6, 4, 3, 7, 9, 5, 2, 8],
                 [9, 8, 6, 1, 4, 7, 3, 5, 2],
                 [3, 7, 5, 2, 6, 8, 9, 1, 4],
                 [2, 4, 1, 5, 9, 3, 7, 8, 6],
                 [4, 3, 2, 9, 8, 1, 6, 7, 5],
                 [6, 1, 7, 4, 2, 5, 8, 9, 3],
                 [5, 9, 8, 7, 3, 6, 2, 4, 1]].
```

Important: The results of simplex may not be exactly integers because of numerical approximation. For this part of the assignment, you can simply round any value within 0.005 of 0 or 1 to the nearest integer.

2. Next, write a branch and bound algorithm solving a Sudoku Puzzle even when its linear programming relaxation is not tight. That is, implement the algorithm in the integer programming slides (implement the "simpler" algorithm instead of the version generating feasible upper bounds). To select variables (in this case, $\{(x_{(i,j)})_k : i, j, k = 1, ..., 9\}$) to split on, a simple rule is to pick the variable with the "most undetermined" value closest to 0.5 and split on this variable. Then, solve the two subproblems where we constrain the variable to be either 0 or 1.

5 Submitting to Autolab

Create a tar file containing your writeup for the first two problems and the completed `simplex.py` and `sudoku.py` modules for the programming problems. Make sure that your tar has these files at the root and not in a subdirectory. Use the following commands from a directory with your files to create a `handin.tgz` file for submission.

```
$ ls
solution.py  writeup.pdf
$ tar cvzf handin.tgz writeup.pdf simplex.py sudoku.py
a writeup.pdf
a simplex.py
a sudoku.py
$ ls
handin.tgz  solution.py  writeup.pdf
```