15-780: GRADUATE AI (SPRING 2018)

Homework 1: Search and Optimization

Release: February 4, 2018,
Last Updated: February 20, 2018, 10:30pm,
Due: February 19, 2018, 11:59pm

1 Bidirectional A* tree search [20 points]

Search problems appear in a variety of contexts in computer science and artificial intelligence. In class, we studied unidirectional strategies for solving search problems, in which we begin at a start state s and search forward until we expand a node containing some goal state t. However, *bidirectional* strategies may yield faster running times in certain contexts. Given a graph G with start state s and set of goal states $\mathcal{T}$, bidirectional search finds a path from s to some goal $t \in \mathcal{T}$ by running two simultaneous searches: one starting at s and searching forward along edges towards the goal states, and one starting at the goal states and searching *backwards* towards s. In this problem, we will consider a *bidirectional A^* tree search strategy*. For simplicity, we will be considering undirected graphs only, with only a single goal state t.

More formally, for a node x, let $h_s(x)$ be a "forward-looking" heuristic that estimates the distance from x to a goal, and let $h_t(x)$ be a "backwards-looking" heuristic that estimates the distance from x to s. (As in unidirectional A^* tree search, these heuristics must be admissible for the algorithm to be optimal.) Then, assuming x is in the frontier of the forward search, the priority of x in the forward search is $f_s(x) = g_s(x) + h_s(x)$, where $g_s(x)$ is the cost of the forward path used to reach x from s. Likewise, the priority of x in the backward search (assuming it is in the frontier) is $f_t(x) = g_t(x) + h_t(x)$, where $g_t(x)$ is the cost of the backward path used to reach x from t.

Let F_s and F_t denote the frontiers of the forward and backward searches, respectively. The strategy begins by expanding s and t and inserting their neighbors into F_s and F_t, respectively. Then, at each timestep, the priorities of the next nodes from F_s and F_t are compared, and the one with lower priority is dequeued and expanded, with its neighbors being added to the frontier that the node was drawn from.

1.1 Naive stopping criterion [3 points]

When one search expands a node m that was already expanded by the other, a full path from s to a goal t is obtained by concatenating the $s \to m$ and $m \to t$ paths. One might be tempted to terminate the search as soon as the first complete path is found. Unfortunately, this path is not guaranteed to be optimal.

Construct a graph in which the first complete path found by bidirectional A* tree search is not optimal, even under an admissible heuristic. Indicate source and target nodes s and t, and values for admissible heuristics h_s and h_t for all vertices in your graph. Describe the order in which nodes will be expanded, and which search (forward or backward) they will be expanded by. (You should only need a very small number of vertices.)

1.2 Correct stopping criterion [7 points]

Luckily, a correct stopping criterion is not much more complicated. At any time, let $L_{\min}$ denote the length of a shortest complete path that has been discovered by our algorithm so far. If no complete paths have been found, let $L_{\min} = \infty$. At any iteration, if we have

$$\max \left\{ \min_{x \in F_s} f_s(x),\ \min_{x \in F_t} f_t(x) \right\} \geq L_{\min},$$

then we terminate and return an already-discovered path with length $L_{\min}$.

For this and the next problem, you may use this fact pertaining to *unidirectional* A* search (under an admissible heuristic)[1] without proof:

Fact 1 (for *unidirectional* A*): For any optimal path $P^\star_{s,t}$, at any point before termination, there exists $v' \in P^\star_{s,t}$ such that v' is in the frontier, and $f(v') \leq \ell(P^\star_{s,t}) = L^\star$, where $\ell(\cdot)$ denotes the length of a path and $L^\star$ is the cost of the globally optimal path.

Prove that if bidirectional A* tree search (with admissible heuristics) terminates under this stopping criterion, then it returns an optimal path from s to the goal t.

1.3 Early termination with bounded heuristics [10 points]

The correct stopping condition unfortunately leads to most of the work in the bidirectional search occurring after the first path is found, since one of the frontiers must be expanded to the length of that path. However, if we are willing to accept only an *approximate* shortest path, then early termination is possible. This does, however, require certain quality guarantees on the heuristics.

Define a heuristic h to be ϵ-*bounded* if, for all vertices x:

$$c(x,t) - \epsilon \leq h(x) \leq c(x,t),$$

where $c(x,y)$ denotes the shortest-cost path from x to y. In other words, h is always within ϵ of the true cost $c(x,t)$ from x to the goal. In this case, we can bound the additive error incurred by terminating at the first path.

[1] An earlier version of this writeup did not explicitly mention admissibility, which is necessary for Fact 1 to hold true.

Prove that if bidirectional A* uses ϵ-bounded heuristic functions in both directions, and P is the first completed path (i.e. the path formed when the two searches meet, as in problem 1.1), then $\ell(P) \leq L^\star + 2\epsilon$.

Hints:

- Consider how having an ϵ-bounded heuristic affects *unidirectional* A* search. For any node v that has been expanded, can you relate the found path cost $g(v)$ to the true cost $c(s, v)$ from the start state to that node?
- Let $k_s = \min_{x \in F_s} f_s(x)$, and $k_t = \min_{x \in F_t} f_t(x)$. Note that $\max(k_s, k_t)$ is the left-hand side of the stopping criterion in problem 1.2. Can you relate this quantity to $\ell(P)$ when the first path P is completed?
- Suppose that v was the node expanded to complete P. How does its f-value compare to k_s and k_t? And how does the length of P relate to path costs (i.e. g-values) from the two searches?

2 Gradient descent [25 points]

In class, we showed that for a differentiable objective function $f : \mathbb{R}^n \to \mathbb{R}$, taking a sufficiently small step in the direction opposite of the gradient is guaranteed to decrease the objective. This is however only a *local* guarantee; when and how can we show that iteratively taking gradient descent steps will get us to the global minimum of the function? In this section, we will explore different assumptions that guarantee global convergence and examine the *rate of convergence* under those assumptions.

2.1 Case A: Convexity and Lipschitz continuity [2+2+4+4 = 12 points]

We will make two assumptions about f in this case.

Assumption 1 is that f is convex, i.e., for all $x, y \in \mathbb{R}^n$,

$$f(y) \geq f(x) + \nabla f(x)^T (y - x).$$

Note that this definition is equivalent to what is presented in the slides (you do not have to prove this equivalence). One way of interpreting this definition is that, for a convex function, the first order Taylor expansion of the function underestimates the function everywhere.

Assumption 2 is that the gradient of f is *Lipschitz continuous with constant* $L > 0$, i.e., for any $x, y \in \mathbb{R}^n$, we have:

$$\|\nabla f(x) - \nabla f(y)\|_2 \leq L \|x - y\|_2.$$

Informally, this means that the gradient of f does not vary too much as we move from one point to another in $\mathbb{R}^n$.

1) Do each of the convex functions $f(x) = x^2$ and $f(x) = x^4$ satisfy **Assumption 2**, for some $L > 0$? If yes, give the minimum L such that the inequality is satisfied. If not, why?

2) If f satisfies **Assumption 2**, Taylor's remainder theorem guarantees that for any $x, y \in \mathbb{R}^n$,

$$f(y) \leq f(x) + \nabla f(x)^T (y - x) + \frac{L}{2}\|x - y\|_2^2. \tag{1}$$

Now, let us fix a step size of $1/L$. That is, from any x, we move to

$$x' = x - \frac{1}{L}\nabla f(x) \tag{2}$$

during gradient descent. Given that Equation (1) holds, derive the following inequality for the amount of progress gradient descent makes in locally decreasing the value of f:

$$f(x') \leq f(x) - \frac{1}{2L}\|\nabla f(x)\|_2^2. \tag{3}$$

3) Let $x^\star$ be the global minimum of the function f. To analyze the convergence of gradient descent, we want to bound the difference between the current value of the objective $f(x')$ and the global minimum $f(x^\star)$. In this question, show that under **Assumptions 1** and **2**, the following upper bound on $f(x') - f(x^\star)$ holds:

$$f(x') - f(x^\star) \leq \frac{L}{2}(\|x - x^\star\|_2^2 - \|x' - x^\star\|_2^2). \tag{4}$$

The right side of this inequality can be thought of as a measure of how far we move towards the minimum $x^\star$ in the input space with each update (we will see how this helps us analyze global convergence in the next question!).

You may use the other equations presented above without (re-)deriving them.

Hints:

- Use the given definition of convexity to upper bound $f(x)$ in terms of $f(x^\star)$ in Equation (3).
- $2a^T b + \|b\|_2^2 = \|a\|_2^2 + 2a^T b + \|b\|^2 - \|a\|_2^2 = \|a + b\|_2^2 - \|a\|_2^2$.

4) Let $x^{(0)}, x^{(1)}, x^{(2)}, \ldots, x^{(\tau)}$ be a sequence of gradient descent updates starting from some initial point $x^{(0)}$, with $x^{(t+1)} = x^{(t)} - \frac{1}{L}\nabla f(x^{(t)})$. Using Equations (3) and (4), prove the following two inequalities:

$$f(x^{(\tau)}) - f(x^\star) \leq \frac{1}{\tau}\sum_{t=1}^{\tau}(f(x^{(t)}) - f(x^\star)) \leq \underbrace{L\frac{\|x^{(0)} - x^\star\|_2^2}{2}}_{\text{constant}} \times \frac{1}{\tau}. \tag{5}$$

Thus, we have shown that gradient descent converges at the rate $O(1/\tau)$, where τ is the number of iterations of gradient descent.

2.2 Case B: Strong convexity [4+2+4+2+1 = 13 points]

In this section, in addition to **Assumptions 1** and **2**, we will add **Assumption 3**, which is that f is ℓ-*strongly convex* for some $\ell > 0$. This means that for all $x, y \in \mathbb{R}^n$:

$$f(y) \geq f(x) + \nabla f(x)^T (y - x) + \frac{\ell}{2}\|x - y\|_2^2. \tag{6}$$

Note that plugging $\ell = 0$ into this inequality recovers the definition for (weak) convexity presented in **Assumption 1**. Thus, Equation (6) is a *strong* notion of convexity because if the equation holds for some $\ell > 0$, then it also holds for $\ell = 0$. Also note that while this assumption lower bounds $f(y)$, **Assumption 2** upper bounds $f(y)$.

1) Show that the right side of Equation (6) can be lower bounded purely in terms of x (i.e., independent of y) as:

$$f(x) + \nabla f(x)^T(y - x) + \frac{\ell}{2}\|x - y\|_2^2 \geq f(x) - \frac{1}{2\ell}\|\nabla f(x)\|_2^2. \tag{7}$$

(**Hint:** Show that the expression we want to lower bound is a convex quadratic function in y; then the minimum is attained where its gradient with respect to y is zero.)

2) Using Equations (6) and (7), show that:

$$f(x) - f(x^\star) \leq \frac{1}{2\ell}\|\nabla f(x)\|_2^2. \tag{8}$$

3) Using Equations (3) and (8), show that:

$$f(x') - f(x^\star) \leq \left(1 - \frac{\ell}{L}\right)(f(x) - f(x^\star)). \tag{9}$$

Observe that this equation is similar to Equation (4) in that it upper bounds how far we are from the global minimum value.[2]

4) Again, let $x^{(0)}, x^{(1)}, x^{(2)}, \ldots, x^{(\tau)}$ be a sequence of gradient descent updates starting from some initial point $x^{(0)}$, where $x^{(t+1)} = x^{(t)} - \frac{1}{L}\nabla f(x^{(t)})$. Using Equation (9), show that:[3]

$$f(x^{(\tau)}) - f(x^\star) \leq \left(1 - \frac{\ell}{L}\right)^\tau (f(x^{(0)}) - f(x^\star)). \tag{10}$$

5) Is this a faster or slower rate of convergence than in Case A? Why?

NOTE: In this section, we were able to prove the convergence of gradient descent under a fixed step size based on a global Lipschitz constant which is known to us. However, if **Assumptions 1** and **2** do not hold or if we do not know the Lipschitz constant, it may be necessary to select a different step size at each iteration using methods such as *backtracking line search*.

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.

[2] The right hand side of this equation has been corrected; specifically, $f(x') - f(x^\star)$ has been changed to $f(x) - f(x^\star)$.

[3] The right hand side of this equation has been corrected; specifically, $f(x^{(0)} - x^\star)$ has been changed to $f(x^{(0)}) - f(x^\star)$.

3.1 Basic simplex [15 points]

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:

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

Your output should be a tuple consisting of:

- v: the value of the optimal solution.
- 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. (Your code does not need to deal with infeasible or unbounded problems.) 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 are 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 is $< -10^{-12}$ and whenever you want to check if a number is ≥ 0 you should check if it is $> -10^{-12}$.

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 our method works.

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

$$\begin{aligned} a_{1,1}x_1 + a_{1,2}x_2 + \cdots + a_{1,n}x_n &= b_1 \\ a_{2,1}x_1 + a_{2,2}x_2 + \cdots + a_{2,n}x_n &= b_2 \\ &\vdots \\ a_{m,1}x_1 + a_{m,2}x_2 + \cdots + a_{m,n}x_n &= b_m. \end{aligned}$$

For each equation where b_i is negative, we multiply both sides of the equation by -1. 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:

$$\begin{aligned}
\text{sign}(b_1)(a_{1,1}x_1 + a_{1,2}x_2 + \cdots + a_{1,n}x_n) + z_1 &= b_1^+ \\
\text{sign}(b_2)(a_{2,1}x_1 + a_{2,2}x_2 + \cdots + a_{2,n}x_n) + z_2 &= b_2^+ \\
&\vdots \\
\text{sign}(b_m)(a_{m,1}x_1 + a_{m,2}x_2 + \cdots + a_{m,n}x_n) + z_m &= b_m^+.
\end{aligned}$$

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:

$$\begin{aligned}
&\underset{x,z}{\text{minimize}} \quad \sum_{i=1}^{m} z_i \\
&\text{subject to} \quad A^+x = b^+,\ x \geq 0,\ z \geq 0.
\end{aligned}$$

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 this problem 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 $1 \leq i \leq m$; then, $Ax = b$ and $x_i \geq 0$ for $1 \leq i \leq n$, meaning that the remaining non-zero x_i form a feasible basis for our original solution.[4]

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`.

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],
          [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]],
```

[4] 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.

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],[5]
                 [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 starting at 1, it may be necessary to add/subtract 1 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[6] :

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

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

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

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

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

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

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

[5] This row was missing in an earlier version of the writeup.

[6] The text description for the row and the column constraints were incorrectly mapped.

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 to split on (in this case, $\{(x_{(i,j)})_k : i, j, k = 1, ..., 9\}$), 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
simplex.py  sudoku.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  simplex.py  sudoku.py  writeup.pdf
```