# 15780: GRADUATE AI (SPRING 2019)

# Homework 1: $A^*$ Search and Linear Programming

Release: January 24, 2019,
Due: February 14, 2019, 5:00pm

We will use scientific Python for the implementation portions in this course. If you have not used Python before, we recommend downloading the Anaconda distribution (`https://www.continuum.io/downloads`) and looking through introductory resources like Google's Python Class (`https://developers.google.com/edu/python/`) and the Python Beginner's Guide (`https://wiki.python.org/moin/BeginnersGuide`). If you have not used scientific Python before, we recommend following introductions to NumPy (`http://www.numpy.org/`) and matplotlib (http://matplotlib.org/).

**Please make sure your code submission works with the grading environment, which uses Python 3.7 and numpy 1.15.0**

## 1 A* search (25 points) [Ivan]

### 1.1 Heuristic warm-up (10 points)

1. **[2 points]** Let $h_1$ and $h_2$ be consistent heuristics. Will heuristic $h = h_1 + h_2$ be admissible? Consistent?

2. **[3 points]** Let $h_1, h_2, \ldots, h_n$ be consistent heuristics. Define heuristic $h = \max\{h_1, ..., h_n\}$. Is heuristic $h$ admissible? Consistent?

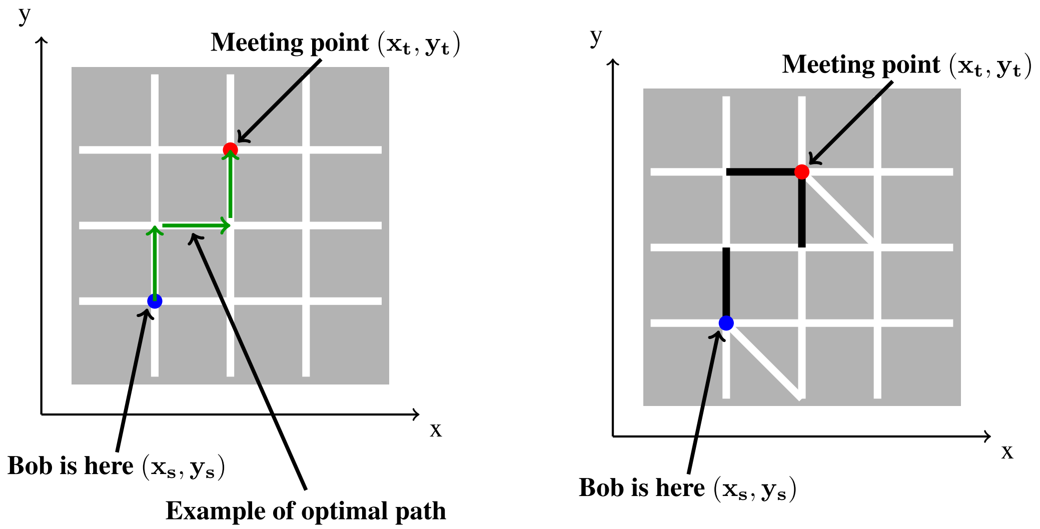
3. **[5 points]** Let $h$ be admissible heuristic. Let $g(n)$ be the cost till the node $n$. Define the expansion function $f$ as follows:

$$f(n) = (2 - t)g(n) + th(n),$$

where $t \in [0, 2]$ is a parameter. Find all values of $t$ for which the A* tree search algorithm is guaranteed to find the optimal path.

**Important: Edge costs are non-negative!**[1]

[1] Remark added


y
Meeting point $(\mathbf{x_t}, \mathbf{y_t})$
x
Bob is here $(\mathbf{x_s}, \mathbf{y_s})$
Example of optimal path
y
Meeting point $(\mathbf{x_t}, \mathbf{y_t})$
x
Bob is here $(\mathbf{x_s}, \mathbf{y_s})$

(a) Manhattan map (usually)

(b) Manhattan map (today)

Figure 1: Example for problem 1.2

## 1.2 Route planning in Manhattan (15 points)

Alice and Bob scheduled a meeting in the restaurant on top of the Empire State Building. Bob has just arrived in Manhattan and is already behind of time so he needs to find the **shortest distance** walking path between his current location and the meeting point. Usually, streets in Manhattan have regular structure (see Figure 1a) and one can find a shortest path with no problem. However, today the situation is different (see Figure 1b):

- Many of the streets are being repaired so Bob can not use them to reach the goal (black segments in Figure 1b)
- In exchange, several buildings opened the technical corridors that allow to cross the block diagonally (diagonal segments in Figure 1b)

These changes make path finding problem much harder. Fortunately, for the sake of this problem we are guaranteed that there exists a path from Bob's position to the meeting point. Bob decides to use A* search to find the optimal path. Bob has an actual map of open and closed streets. *For this problem we assume that each block is a* $1 \times 1$ *square.*

1. **[2 points]** Draw a graph that represents the road situation in Figure 1b and can be used as an input for A* search. Make sure to specify the cost of each edge in the graph.

   **Hint:** think of what edge cost is suitable for the shortest distance path finding. [2] Mind that the cost of diagonal segment is not 1.

2. **[13 points]** Now consider a general problem instance (not necessarily the one represented in Figure 1b).

[2] Hint added

The key component of the A* search is a heuristic function. Bob came up with the following candidates[3]:

- $h_1(n) = 0$ for every node $n$
- $h_2(n) = \sqrt{2}\min(|x_t - x_n|, |y_t - y_n|)$
- $h_3(n) = \sqrt{(x_t - x_n)^2 + (y_t - y_n)^2}$
- $h_4(n) = (x_t - x_n)^2 + (y_t - y_n)^2$
- $h_5(n) = |x_t - x_n| + |y_t - y_n|$
- $h_6(n) = \frac{1}{\sqrt{2}}(|x_t - x_n| + |y_t - y_n|)$

For each of the listed heuristics answer the following questions: is this heuristic admissible? consistent? If heuristic is inadmissible, construct an instance of the problem where A* tree search finds sub-optimal solution.

**Some clarification:**[4] General problem instance is constructed from 1x1 square blocks with some street segments closed and some diagonals open. You can not assign arbitrary costs for different segments – the cost must be determined by the distance.

# 2 Unconstrained Convex Optimization (25 points) [Ivan]

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, a somewhat weak guarantee:

- First, it does not specify what does "sufficiently small" mean
- Second, even if the step sizes are "sufficiently small", iteratively taking gradient descent steps will not necessarily lead us to the global minimum of the function

In this task we will explore the limitations of the result obtained in class and show that under some assumptions, gradient descent indeed leads to the global optimum.

## 2.1 Negative examples (5 points)

### Example 1 [2 points]

It would be great if there exists some "sufficiently small" constant $\alpha^* > 0$ such that for any differentiable function $f : \mathbb{R}^n \to \mathbb{R}$, the gradient descent algorithms with step size $\alpha^*$ will decrease the value of objective, provided that the starting point is not a local minimum. Unfortunately, it is not true in a general case.

For every value of $\alpha > 0$, consider a function $f(x) = \frac{2}{\alpha}x^2$ and show that the gradient descent algorithm with starting point $x_0 = 1$ and step size $\alpha$ will not converge to the global minimum.

This example demonstrates that there is no universally good step size and underscores the importance of careful step size selection.

[3] $(x_n, y_n)$ is a coordinates of node $n$

[4] Clarification added

**Example 2 [3 points]**

At this point we know that it is not trivial to select the right step size. Unfortunately, it is not the only tricky part.

Show that there exist a one dimensional differentiable non-convex function $f : \mathbb{R} \to \mathbb{R}$ with unique global minimum $x^* \in \mathbb{R}$, a value $\alpha > 0$ and a starting point $x_0 \in \mathbb{R}$, such that gradient descent algorithm with step size $\alpha$ and starting point $x_0$ at every iteration decreases the value of function $f$, but does not converge to the global minimum.

**Hint:** Think about the function with multiple local minima and choose the starting point and step size such that the gradient descent algorithm is trapped in one of them.

## 2.2 Positive results (20 points)

Examples you constructed above show that one needs to be careful with the step sizes of the gradient descent algorithm as well as be aware that even if the gradient descent algorithm converges to some point, it does not necessary finds the global minimum of the function. In this section we will make some assumptions that will ensure that gradient descent algorithm finds the solution of the optimization problem.

**Assumption 1.** We will assume that function $f$ is convex. For convenience, we will use a different definition of a convex function that is equivalent to one you've seen in class in case of differentiable function $f$ (you do not need to prove equivalence):

$$\text{for all } x, y \in \mathbb{R}^n : f(y) \geq f(x) + \nabla f(x)^T (y - x).$$

One way of interpreting this definition is that, for a convex function, the first order Taylor expansion of the function underestimates the function everywhere. As you have seen in class, convexity guarantees that all locally optimal points are globally optimal.

**Assumption 2.** The gradient of $f$ is *Lipschitz continuous with constant* $L > 0$, i.e.,

$$\text{for all } x, y \in \mathbb{R}^n : \|\nabla f(x) - \nabla f(y)\|_2 \leq L\|x - y\|_2.$$

Informally, this means that the gradient of $f$ does not vary too much as we move from one point to another in $\mathbb{R}^n$. As we will show below, this assumption gives us a simple way to select a step size of gradient descent algorithm.

**Implications**
Assumption 2 together with Taylors remainder theorem guarantees that for any $x, y \in \mathbb{R}^n$, we have:

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

Joining this bound with the bound from Assumption 1, for every $x, y \in \mathbb{R}^n$ we obtain:

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

which gives us a useful way to control the behaviour of function $f$ around $x$.

**1. [5 points]** Do each of the convex functions $f(x) = x^2$, $f(x) = x^4$ and $f(x) = e^x$ satisfy Assumption 2, for some $L > 0$? If yes, give the minimum $L$ such that the inequality is satisfied. If not, why?

**Step Size**
For each step of the gradient descent algorithm we have $x' = x - \alpha \nabla f(x)$, where $x$ is an old point and $x'$ is a new point. Substituting this equation into upper bound in (1) with $y = x'$, we obtain

$$f(x) - f(x') \geq \alpha \|\nabla f(x)\|_2^2 - \frac{L}{2}\alpha^2 \|\nabla f(x)\|_2^2.$$

The left hand side of this equation is the difference between objective value in previous point ($x$) and the new point ($x'$). To ensure that the algorithm decreases the function, this difference must be positive. Ideally, we want to keep this difference as large as possible. The right hand side is the lower bound on the quantity of interest, so the natural option is to take the step size that maximizes that lower bound.

**2. [5 points]** Prove that the step size $\alpha = \frac{1}{L}$ maximizes the lower bound on the decrease of the function. Notice that this step size does not depend on the value of the function or its gradient in $x$. Show that for this choice of step size we have:

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

For the rest of the problem we fix $\alpha = \frac{1}{L}$. To show that the gradient descent algorithm with selected step size converges to the global minimum (and not diverges as in your **Example 1**), we need to connect the value $f(x')$ of the objective function after the update, with the value of the objective function in the global optimum $f(x^*)$.

**3. [5 points]** Show that under Assumptions 1 and 2, the following holds

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

You may use all the equation introduced above without (re)deriving them.
**Hints:**

- Use Assumption 1 to upper bound $f(x)$ in terms of $f(x^*)$ in (2)
- You may want to use the following fact without proof:

$$2a^T b + \|b\|_2^2 = \|a\|_2^2 + 2a^T b + \|b\|^2 - \|a\|_2^2 = \|a + b\|_2^2 - \|a\|_2^2$$

Finally, we can formulate the main result of this section.

**4. [5 points]** 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 (2) and (3), prove the following two inequalities:

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

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

## 3 Resizing Rectangles (25 points) [Chun Kai]

Like any respectable software developer, Alice has an environment comprising dozens of windows, widgets and panels. Help her reorganize her workspace using the least amount of effort!

Each state is given by a finite number of axis aligned non-overlapping rectangles with integer coordinates. At every step, you are able to select a *single* rectangle and increase or decease the position of *one* of its edges by 1. At all times, all rectangles must have **strictly positive area** and **may not intersect** any other rectangles. All rectangles are uniquely identifiable (think of each rectangle as being colored differently). Your task is to find the shortest sequence of actions required to transform a state to another given state. You may assume that the 'playing field' extends infinitely in all directions.

**Example 1.** The initial and goal states are given in Figure 2 and Figure 3 respectively. The shortest sequence of actions to get from $s_{\text{init}}$ to $s_{\text{goal}}$ is of length 6. A possible sequence of intermediate states is shown in Figure 4.

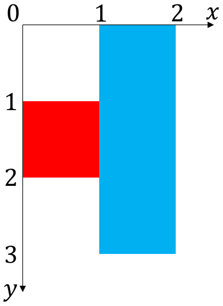


Figure 2: Initial state $s_{\text{init}}$ for Example 1

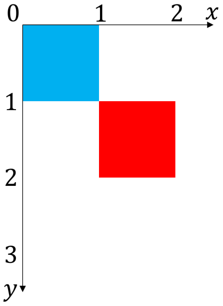


Figure 3: Goal state $s_{\text{goal}}$ for Example 1

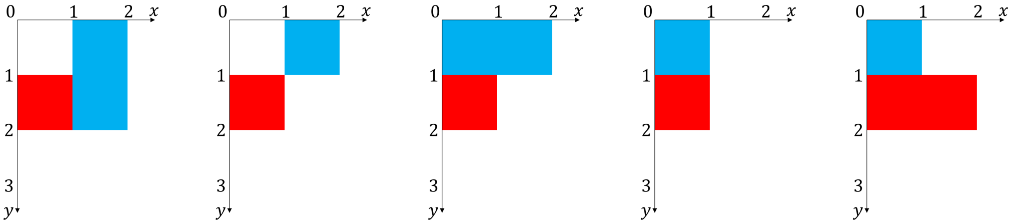


Figure 4: Sequence of intermediate moves for Example 1

**State representation** A state is a tuple of $k$ rectangles $s = (r_1, \dots, r_k)$, where each $r_i$ is a 4-tuple $(x_{\text{left}}, x_{\text{right}}, y_{\text{top}}, y_{\text{bottom}})$ of coordinates which describe the location and shape of the rectangle. For the examples in Figure 2 and 3, we have $s_{\text{init}} = ((0,1,1,2),(1,2,0,3))$ and $s_{\text{goal}} = ((1,2,1,2),(0,1,0,1))$. You can internally change the state to a format that's easier to work with if you prefer.

**Example 2.** The initial and goal states in Figure 5 and 6 are represented by $s_{\text{init}} = ((0,2,2,3),(0,1,3,4),(0,1,4,5))$ and $s_{\text{goal}} = ((0,2,2,3),(0,1,0,1),(0,1,1,2))$. A minimum of 14 steps are required for the transformation.

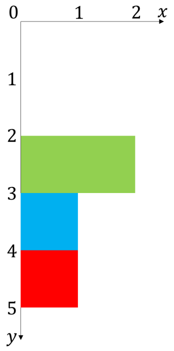


Figure 5: Initial state $s_{\text{init}}$ for Example 2

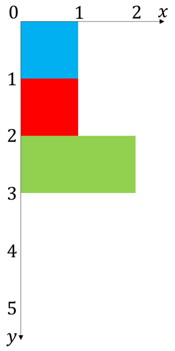


Figure 6: Goal state $s_{\text{goal}}$ for Example 2

We have provided a `rectangles.py` Python module for you to get started with. When you are done, you will submit your completed module to Diderot for automatic grading. Do not rename the file or change the function names because our grader will import the module and functions by name. You are welcome to introduce new auxiliary functions and call them from the main functions we provide.

You should be able to execute `rectangles.py` without any modifications as a starting point. It contains a `main` function that shows expected outputs of every function we will grade.

### 3.1 Implementing Heuristics (10 points)

You should first implement the following two heuristics in the stubbed `heuristic_misplaced` and `heuristic_distance` functions we have provided.

1. **The number of misplaced edges.** In Example 1, there are 5 misplaced edges. 2 of them are from left and right edges of the red rectangle, while all but the top edge of the blue rectangle are misplaced. In Example 2, there are 4 misplaced edges, 2 each from the red and blue rectangles. The mathematical expression for the heuristic is

$$h_{\text{misplaced}}(s_1, s_2) = \sum_{i=1}^{k} \sum_{j=1}^{4} 1\left(s_1[i][j] \neq s_2[i][j]\right),$$

where $1(\cdot)$ is the indicator function which returns $1$ when the condition within is true and $0$ otherwise. The heuristic $h_{\text{misplaced}}$ is obtained by relaxing the original problem. Each edge may be 'dragged' to any other location in a single step and overlap constraints are ignored.

2. **The sum of the edge distances from the target.** In Example 1, the distance is $6$ (equivalent to the optimal cost). $4$ of this is from the blue rectangle and $2$ from the red. In Example 2, the distance is $12$, $6$ for each rectangle. The mathematical expression for the heuristic is

$$h_{\text{distance}}(s_1, s_2) = \sum_{i=1}^{k} \sum_{j=1}^{4} \Big| s_1[i][j] - s_2[i][j] \Big|.$$

This heuristic is obtained by ignoring the overlap constraints.

You should convince yourself that both heuristics are consistent and that $h_{\text{distance}}$ dominates $h_{\text{misplaced}}$.

### 3.2 Implementing $A^*$ (15 points)

Next you will implement the $A^*$ *graph search* algorithm in the stubbed `astar` function we have provided to find the shortest path using the heuristics above. Your function should return a tuple containing (1) a list of states in the shortest path (include both the initial and goal states) and (2) a list of states in the order they were expanded. Since the playing field is infinitely large, there is always a solution.

$A^*$ is non-deterministic when selecting what node to explore next when they all have the same $f$ value. To make $A^*$ deterministic for grading, **we require that you break ties by selecting the lexicographically first node** based on the flattened state. For example, the state ((1, 2, 3, 4) (6, 7, 5, 8)) comes lexicographically before ((1, 2, 3, 4), (6, 8, 3, 4)). Furthermore, because there can be two paths of the same length leading to the same state, it is required that you further break ties by selecting the **smallest lexcicographic parent node** (this may make a difference when extracting the shortest path).

We recommend that you use the `heapdict` package as a priority queue to find elements with minimum cost. The `heapdict` package is included by default in Anaconda distributions and the project page is available at `https://github.com/DanielStutzbach/heapdict`. To break ties, your should use a tuple with ($f$ value, state, parent state, other information you want to store) as the value in your heapdict.

**Hints**

1. 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.
2. Since the number of explored states may be large, consider using the `set` data structure rather than iterate over lists.
3. The test cases are fairly small. You should *not* have to implement any special data structure to test for intersection, a naive implementation should work well enough.
4. We have included some helper functions which you may find useful. Feel free to use them (or not at all).

## 4 Simplex Programming (25 pts) [Junjue]

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.

### 4.1 Basic simplex (9 pts)

Consider a linear programming problem in its standard form, in which $c \in R^n, A \in R^{m\text{x}n}, b \in R^m, x \in R^n$.

$$\min_x c^T x \quad \text{subject to} \quad Ax = b, x \geq 0$$

Implement the simplex algorithm in function **simplex(I, c, A, b)** as described in the class. We will use the `numpy` package for matrix manipulations.

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

- $I$: a numpy array representing a feasible basis index set.
- $c$: a numpy array representing the cost vector.
- $A$: a 2D numpy array representing the constraints.
- $b$: a numpy array. Your solution $x$ should satisfy $Ax = b$.

Your output should be a tuple consisting of:

- $v$: a scalar value of the optimal cost.
- $x$: a numpy array that is 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 $\geq 0$ you should check if it is[5] $\geq -10^{-12}$.

### 4.2 Finding an Initial Feasible Point (8 pts)

In the simplex algorithm implementation above, an initial feasible point is given. In this problem, we will explore how to find such an feasible point.

Recall that the feasible point needs to satisfy the constraint $x_I = {A_I}^{-1} b \geq 0$. To find such a point, let's construct the following auxiliary problem from the original problem. Let $b' = |b|$. We want to maintain the equality of all original constraints, so we transform A to $A'$ correspondingly. Let $z \in R^m$ and the auxiliary linear programming problem to be the following:

$$\min_{x,z} 1^T z \quad \text{subject to} \quad A'x + z = b', x \geq 0, z \geq 0$$

If the optimal value to this auxiliary problem is 0 (i.e. $1^T z = 0$), we know $A'x = b'$ and so $Ax = b$. Hence, we have found a feasible solution to the original problem.

[5] The minus sign was missing

### 4.2.1 What is an initial feasible point for this auxiliary problem? (Optional)

(Note: this question will not be graded. It is here to guide your thoughts.)

By observation, find and write down an initial feasible point for the above auxiliary linear programming problem.

### 4.2.2 Implement a Full Simplex Algorithm (8 pts)

Implement a function that solves the above auxiliary linear program to find an initial feasible point for the original problem. Then use this function together with **simplex(I, c, A, b)** to implement function **full_simplex(c, A, b)** to solve the linear programming problem. There might be degenerate solutions, but you will not have to deal with any such cases for this problem.

The inputs and outputs to function **full_simplex** are:

- $c$: a numpy array representing the cost vector.
- $A$: a 2D numpy array representing the constraints.
- $b$: a numpy array. Your solution $x$ should satisfy $Ax = b$.

(Note: You'll need to get $A', b'$ of the auxiliary problem from $A, b$).

Your function should return:

- $v$: a scalar value of the optimal cost.
- $x$: a numpy array that is the optimal solution.

## 4.3 A More Efficient Simplex Implementation (8 pts)

Simplex algorithm requires inverting $A_I$ at each iteration. A naive implementation would re-invert this matrix every time, resulting in $O(m^3)$ computation at each iteration. A more efficient solution is to use Sherman-Morrison Formula that directly maintains and updates ${A_I}^{-1}$ at each iteration, using $O(m^2)$ computation.

The Sherman Morrison Formula states that for $C \in R^{mxm}, u \in R^m, v \in R^m$, the following equation holds. Note that the $u, v$ are m by 1 column vectors. $\otimes$ is the outer product.

$$(C + u \otimes v)^{-1} = C^{-1} - \frac{(C^{-1}u) \otimes (v^T C^{-1})}{1 + v^T C^{-1} u}.$$

### 4.3.1 Apply Sherman-Morrison Formula to Simplex Algorithm (Optional)

(Note: this question will not be graded. It is here to guide your thoughts.)

Write the update rule to $A_I$ in the last step of the simplex algorithm ($I \leftarrow I - \{i^\star\} \cup \{j\}$) as the left-hand side form of the Sherman-Morrison Formula ($C + u \otimes v$). $C$ here is $A_I$. Identify what the corresponding column vectors $u, v$ would be.

### 4.3.2 Implement a More Efficient Simplex Algorithm using Sherman-Morrison Formula (8 pts)

Implement the simplex algorithm using the Sherman-Morrison formula by keeping track of and updating the ${A_I}^{-1}$ in function **fast_simplex(I, c, A, b)**. This section will be graded based on whether it is adequately fast. If you implemented standard simplex for 4.1 then you should expect to see a factor 2-20 speedup depending on the problem instance. The input and output of your function are the same as 4.1.

## 5 Submitting to Diderot

Create a tar file containing your writeup for the first two problems and the completed `simplex.py` and `rectangles.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  rectangles.py  writeup.pdf
$ tar cvzf handin.tgz writeup.pdf simplex.py rectangles.py
a writeup.pdf
a simplex.py
a rectangles.py
$ ls
handin.tgz  simplex.py  rectangles.py  writeup.pdf
```