ECSE 507 Final Project

Mohamed-Amine Azzouz

2024-05-02

Contents

Section 1: Introduction to the Tetromino Tiling Problem

In general, assume that there are k disjoint tetronimos $\{T_1, \ldots, T_k\}$ are such that $\cup_{i=1}^{k} T_i \subseteq R \subseteq \mathbb{R}^2$. One defines a (closed) tetromino as a union of four closed cubes with disjoint interiors.
We can write those closed cubes (of side length 1) as

$$C_{[x,y]} := [x, x+1] \times [y, y+1].$$

To make sure those cubes' corners stay within the grid $\mathbb{Z}^2$, we shall let $x, y \in \mathbb{Z}$. In that context, any tetromino T used to model the tiling problem can be written as:

$$T = \bigcup_{i=1}^{4} C_{[x_i,y_i]},$$

With $[x_i, y_i] \neq [x_j, y_j]$ for $i \neq j$ to ensure the cubes are disjoint.
If $R = [0, 5] \times [0, 4]$, the pigeonhole principle states the maximal number of tetrominos that can fit in R is $k = \frac{m(R)}{m(T)} = 5 \cdot 4/4 = 5$, where m is the Lebesgue measure in $\mathbb{R}^2$.
The problem consists in comparing the total rents of different tiling configurations of tetrominos, which is trivial to do with an algorithm to compute said rents. For a given list of disjoint tetrominos $\mathcal{T} = \{T_1, \ldots, T_k\} \subseteq R$, the total rent it generates is

$$R_{\text{total}}(\mathcal{T}) = \sum_{T \in \mathcal{T}} R(T) = \sum_{i=1}^{k} R(T_i),$$

where the rent of a given tetromino T is

$$R(T) := \min\{f_0(x, y) : (x, y) \in T\},$$

and $f_0(x, y) := (x-2)^2 + 3(y-1)^2 + e^{2x-3y}$ is an arbitrary smooth convex objective function. We need smoothness to be able to compute gradients of the objective.

Section 2: Problem Decomposition into a Convex Optimization Problem

Therefore, the critical problem to solve is the following non-convex optimization problem:

$$\begin{array}{ll} \text{minimize} & f_0(x, y) \\ \text{subject to} & (x, y) \in T \end{array}$$

since T is generally non-convex. However, since $T = \cup_{i=1}^{4} C_{[x_i,y_i]}$, the problem reduces to

$$\begin{array}{ll} \text{minimize} & f_0(x, y) \\ \text{subject to} & (x, y) \in C_{[x_i,y_i]} \text{ for some } i \in \{1, 2, 3, 4\} \end{array}$$

Now, notice that the closed cubes $C_{[x_i,y_i]}$ are convex, and that in general, it is very easy to see that (with a linear ordering on $\mathbb{R}^2$ induced by the linear ordering on the image of f_0):

$$\min T = \min\{\cup_{i=1}^{4} C_{[x_i,y_i]}\} = \min_{i=1,2,3,4} \min\{C_{[x_i,y_i]}\},$$

meaning it is enough to focus on one of the convex problems

$$\begin{array}{ll} \text{minimize} & f_0(x,y) \\ \text{subject to} & (x,y) \in C_{[x_i,y_i]} \end{array}$$

for $i = 1,2,3,4$, producing four optimal points $\{[x_i^*, y_i^*]\}_{i=1}^4$. Then, the original optimization problem will be found by taking

$$[x^*, y^*] = \text{argmin}_{[x_i^*,y_i^*]}\{f_0([x_i^*, y_i^*]), i = 1,2,3,4\}.$$

Section 3: Procedure to Solve the Tiling Problem, and Results

The previous section showed us that the tiling problem's core optimization challenge is to solve the problem

$$\begin{array}{ll} \text{minimize} & f_0(x,y) \\ \text{subject to} & (x,y) \in C_{[x_i,y_i]} \end{array}$$

One can rewrite the feasible cube as the intersection of half planes (ensuring smooth constraints). Since $C_{[x_i,y_i]} = \{x \geq x_i\} \cap \{y \geq y_i\} \cap \{x \leq x_i + 1\} \cap \{y \leq y_i + 1\}$, we obtain:

$$\begin{array}{ll} \text{minimize} & f_0(x,y) \\ \text{subject to} & -x + x_i \leq 0 \\ & -y + y_i \leq 0 \\ & x - (x_i + 1) \leq 0 \\ & y - (y_i + 1) \leq 0 \end{array}$$

That idea is declared in Python below (in `Part_2.py`). The tetronimos in tiling A and B are converted into lists of tuples:

```
def compare_rent_costs(list_of_tetronimos_A, list_of_tetronimos_B):
    cost_A = total_rent_list_tetrominos(list_of_tetronimos_A)

    cost_B = total_rent_list_tetrominos(list_of_tetronimos_B)

    grid_A = create_grid(list_of_tetronimos_A)
    display_grid(grid_A)
    print("Rent cost of tiling A:")
    print(cost_A)
```

```
    grid_B = create_grid(list_of_tetronimos_B)
    display_grid(grid_B)
    print("Rent cost of tiling B:")
    print(cost_B)

    if cost_A > cost_B:
        print("The tiling with higher rent is A")

    if cost_A < cost_B:
        print("The tiling with higher rent is B")

    if cost_A == cost_B:
        print("The two tilings have the same rent")


list_of_tetronimos_B = [[(0,3),(1,3),(1,2),(2,2)], [(2,3),(3,3),(3,2),(3,1
list_of_tetronimos_A = [[(0,3),(1,3),(1,2),(2,2)], [(0,2),(0,1),(0,0),(1,0

compare_rent_costs(list_of_tetronimos_A, list_of_tetronimos_B)
```

Next, we compute the rent cost of a tiling by adding up the rent costs of the tetronimos it is made of, as shown below:

```
def is_disjoint(tetrominos):
    # Convert the list of tuples into a set of tuples
    cell_sets = [set(tetromino) for tetromino in tetrominos]

    # Check if any two tetrominos share common cells
    for i in range(len(cell_sets)):
        for j in range(i + 1, len(cell_sets)):
            if cell_sets[i].intersection(cell_sets[j]):
                return False  # Tetrominos are not disjoint if we find an
    return True  # Tetrominos are disjoint

def total_rent_list_tetrominos(list_of_tetronimos):
    # Check if tetrominos are disjoint
    if not is_disjoint(list_of_tetronimos):
       print("The tetrominos are not disjoint.")
       return

    total_cost = 0

    for tetromino in list_of_tetronimos:
        tetromino_cost = minimize_group_of_cells_rent(tetromino)
        total_cost += tetromino_cost[0]
```

```
    return total_cost
```

Notice that we made sure that the tetronimos are disjoint before computing their total cost. Otherwise, computing the total rents by adding up them up would not make sense in practice.
To continue, one must compute the rent cost of individual tetronimos, which is done by minimizing over each of its cubes and picking the minimum:

```
def is_tetromino(cells):
    if len(cells) != 4:
        return False

    for k in range(4): # check if the tetromino is inside the region
        x, y = cells[k]
        if x < 0:
            return False
        if y < 0:
            return False
        if x >= horizontal_size:
            return False
        if y >= vertical_size:
            return False

    for i in range(4):
        for j in range(i+1, 4):
            x1, y1 = cells[i]
            x2, y2 = cells[j]
            if abs(x1 - x2) + abs(y1 - y2) == 1: # two cells are adjacent
                return True
    return False

def minimize_group_of_cells_rent(cell_group):

    if is_tetromino(cell_group):
        print("The given list represents a tetromino.")
    else:
        print("The given list does not represent a tetromino.")

    total_rent_cost = np.inf
    optimal_point = jnp.array([-1, -1])
    for i, j in cell_group:

        f_ineq = f_ineq_cell(i, j) # Generate constraints and initial poin
        x_initial = initial_guess(i, j)
```

```
        result = barrier_method(f_0, x_initial, t, mu, eps, alpha, beta, A
        min_cell_rent = f_0(result) # Find the optimal value of the same c

        if min_cell_rent < total_rent_cost: # if we find a smaller rent co
            total_rent_cost = min_cell_rent
            optimal_point = result

    return total_rent_cost, optimal_point
```

The program checks if the list inputted represents a tetronimo by making sure that its size is 4 and that every cube it's made is a adjacent to another cube in the list. To see if two cubes are adjacent, one uses the fact that:

$$C_v \text{ and } C_w \text{ are adjacent iff } ||v - w||_1 = 1.$$

If the lists provided do not correspond to tetronimos, one can still compute optimal costs with them, as long as a set is an arbitrary finite union of cubes the algorithm works the exact same way.
The objective and constraints are defined as such:

```
f_ineq_cell = lambda i, j: lambda x: jnp.array([-x[0]+i, -x[1]+j, x[0]-(i+

f_0 = lambda x: (x[0] - 2)**(2) + 3*(x[1] - 1)**2 + jnp.exp(2*x[0]-3*x[1])

initial_guess = lambda i,j: jnp.array([i+0.5, j+0.5]) # start guess at the
```

The initial guess for any cube is in its middle, to minimize the odds of ending up outside the feasible set. One solves the convex optimization problems with the barrier method `barrier_method` from `Part_1.py`:

```
import jax
import jax.numpy as jnp

# Define derivative operations from JAX
def grad(f):
    return jax.grad(f)

def hessian(f):
    return jax.jacfwd(jax.grad(f))

# Set up tests for iterated algorithm termination
def feasibility_test(f_ineq, x):
    return jnp.max(f_ineq(x)) <= 0

def armijo_goldstein(f, g, alpha, x, Dx, T):
    return f(x + T*Dx) - f(x) <= (alpha * T)*(g @ Dx)
```

```
def stop_newton(lambda_squared,eps):
    return (1/2)*(lambda_squared) < eps

def stop_barrier(m,t,eps):
    return m/t < eps

def log_minus(x):
    return jnp.log(-x)

# Backtracking line search algorithm to determine the step size in Newton
def backtracking_line_search(f, g, x, Dx, alpha, beta, f_ineq):
    max_iterations = 1e2
    T = 1

    counter = 0
    # First, we check that our position after step size is feasible
    while not feasibility_test(f_ineq, x + T*Dx):
        counter += 1
        T *= beta # Shorten step to try to stay in the feasible set

        # Stopping message after max_iterations
        if counter >= max_iterations:
            error_message = f'Did␣not␣manage␣to␣step␣into␣a␣feasible␣set␣a
            print(error_message)
            #raise no_convegence_error(error_message)
            return x + T*Dx

    counter = 0 # Reset counter for the new iteration

    while not armijo_goldstein(f, g, alpha, x, Dx, T):
        counter += 1
        T *= beta

        # Stopping message after max_iterations
        if counter >= max_iterations:
            error_message = f'Did␣not␣obtain␣convergence␣of␣the␣backtracki
            print(error_message)
            #raise no_convegence_error(error_message)
            x
    return T

# Solve the KKT system in Newton's method
def solve_KKT_system(Q, g, A):
```

```
    p = A.shape[0] # size of A
    n = len(g) # length of gradient vector


    if p: # If p > 0, we compute the KKT matrix with equality constraints
        # Construct KKT matrix with the Hessian and A
        KKT_matrix = jnp.block(
            [[Q,   jnp.transpose(A)],
            [A, jnp.zeros((p,p))]]
        )

        KKT_vector = jnp.concatenate((-g,jnp.zeros(p)), axis = 0) # add p


        Dx_concat_w = jnp.linalg.solve(KKT_matrix, KKT_vector)

        Dx = Dx_concat_w[:n] # grab the first n entries
        w = Dx_concat_w[n:] # the dual optimal vector
        return Dx

    else: # If p > 0, we compute the KKT matrix without equality constrain
        Dx = jnp.linalg.solve(Q,-g) # Construct KKT matrix with the Hessia
        return Dx

# Method of unconstrained minimization for the centering problem
def newtons_method(f, x, eps, A, b, alpha, beta, f_ineq):
    counter = 1
    max_iterations = 1e2

    # First iteration off the initial guess
    Q = hessian(f)(x)
    g = grad(f)(x)

    # Compute the direction of the step needed
    Dx = solve_KKT_system(Q,g,A)

    # Compute stopping criterion
    lambda_squared = Dx @ Q @ Dx

    # Initialize position for loop
    x_t = x

    # Continue the loop as long as the convergence criterion is not satisf
    while not stop_newton(lambda_squared,eps):
```

```
        counter += 1

        # Compute size of step with backtracking line search
        T = backtracking_line_search(f, g, x_t, Dx, alpha, beta, f_ineq)

        # Update position
        x_t += T*Dx

        # Derivatives at new iteration
        Q = hessian(f)(x_t)
        g = grad(f)(x_t)

        # Compute the direction of the step needed
        Dx = solve_KKT_system(Q, g, A)

        # Compute stopping criterion
        lambda_squared = Dx @ Q @ Dx

        # Stopping message after max_iterations
        if counter >= max_iterations:
            error_message = f'Did not obtain convergence of the Newton ste
            print(error_message)
            #raise no_convegence_error(error_message)
            return x_t


    return x_t


# Main function to be called to solve an optimization problem
def barrier_method(f, x, t, mu, eps, alpha, beta, A, b, f_ineq):
    m = jnp.shape(f_ineq(jnp.zeros(len(x))))[0]

    # Define the logarithmic barrier
    def phi(x):
        return -jnp.sum(jnp.array( list( map(log_minus, f_ineq(x)) ) ) )

    # Perturb the original problem to be usable in the centering problem
    def f_perturbed(x):
        return t * f(x) + phi(x)

    counter = 0
    max_iterations = 1e2
```

```
    # Continue the loop as long as the convergence criterion is not satisf
    while not stop_barrier(m,t,eps):
        x_star = newtons_method(f_perturbed,x,eps,A,b,alpha,beta,f_ineq)

        counter += 1

        # Increase t to lower the influence of the log barrier and approac
        t *= mu

        # Stopping message after max_iterations
        if counter>=max_iterations:
            error_message = f'Did not obtain convergence of the barrier al
            print(error_message)
            return x_star

        x = x_star

    return x
```

To very briefly explain `Part_1.py`, we shall remember that `barrier_method`, which is the main method called by `Part_2.py`, calls `newtons_method`, which calls `backtracking_line_search` to solve the KKT system with `solve_KKT_system`. At its turn, `backtracking_line_search` makes sure the step taken after the line search lands in a feasible set.
All of this ensures we get proper costs for tilings A and B, meaning that the code gives us the following prompt on the terminal:

```
The given list represents a tetromino.
The given list represents a tetromino.
The given list represents a tetromino.
Did not obtain convergence of the Newton step algorithm after 100.0 steps.
Computed point at interruption is [4.0000205 1.9999762].
Did not obtain convergence of the Newton step algorithm after 100.0 steps.
Computed point at interruption is [4.0000014 0.999999 ].
The given list represents a tetromino.
The given list represents a tetromino.
The given list represents a tetromino.
The given list represents a tetromino.
The given list represents a tetromino.
Did not obtain convergence of the Newton step algorithm after 100.0 steps.
Computed point at interruption is [4.0000205 1.9999762].
Did not obtain convergence of the Newton step algorithm after 100.0 steps.
Computed point at interruption is [4.0000014 0.999999 ].
The given list represents a tetromino.
The given list represents a tetromino.
1 1 4 4 4
```

```
2 1 1 4 3
2 5 5 5 3
2 2 5 3 3
Rent cost of tiling A:
22.243195
1 1 2 2 3
4 1 1 2 3
4 4 5 2 3
4 5 5 5 3
Rent cost of tiling B:
21.974768
The tiling with higher rent is A
```

Therefore, the tiling with higher cost is tiling A. We do get some prompts about non-convergence, which could be a sign of some errors, which may be partly corrected by increasing the maximum number of iterations at the cost of computing time.

Section 4: Optional Questions

For completeness and personal interest in the provided questions, this report will give theoretical answers to the optional problems. However, for the sake of wanting to end this exhausting 18-credit semester as soon as possible, no code will be produced to numerically solve them. I hope the grader understands that situation.

Optional Question 1

The algorithm presented above clearly shows that it can be replicated in higher dimensions (where the cells can be described with higher dimension arrays), with polyminos (if we write them as unions of any number of cubes instead of 4 cubes). To change from cubes to other shapes is possible, if the constraints are changed, but it may be harder to determine whether different shapes are disjoint or contained inside a predetermined region.

Optional Question 2

Again, the objective function can be made to be an arbitrary smooth and convex function, and one can use an ASCII system to feed any group of disjoint tetrominos to compute a maximal rent for.

Optional Question 3

One can turn cubes into closed balls $B_r(x_0, y_0)$ of radius r centered at $(x_0, y_0) \in [0, 5] \times [0, 4] = R$. To see if any two balls are disjoint, we only need to use the fact that:

$$B_{r_1}(v_1) \text{ and } B_{r_2}(v_2) \text{ have disjoint interiors iff } ||v_1 - v_2||_2 \geq r_1 + r_2.$$

Furthermore, to see if a ball $B_r(x)$ is contained in R, it is enough to check that

$$|x - x_0| \in [0, 5] \text{ and } |y - y_0| \in [0, 4]$$

Given those conditions met, the code only needs to be modified to replace the cube constraints with circular constraints, which are $(x-x_0)^2+(y-y_0)^2-r^2 \leq 0$ for $B_r(x, y)$.

Optional Question 4

One way to approach the problem of optimizing over a non-convex annulus $B_r(v) \setminus B_{\frac{r}{2}}(v)$ is to approximate it with a finite number of convex sets, say balls.
First, we assume the optimal value when minimizing f_0 over $B_r(v)$ is not in the annulus, otherwise the problem can just be solved with convex optimization, since if a point is optimal for the entire ball it must also be optimal for the annulus subset.
The n'th order approximation of $B_r(x) \setminus B_{\frac{r}{2}}(x)$ could be some union of 2^n balls of the form:

$$\bigcup_{k=0}^{2^n-1} B_{\frac{r}{4}}\left(v + \frac{3r}{4}\left(\cos\left(\frac{k}{2^n}\right), \sin\left(\frac{k}{2^n}\right)\right)\right)$$

Each of those balls are convex optimization problems, so the more balls taken, the more likely the minimum is found in one of those balls such that the answer is accurate.
Another way to approach the problem is to see that if the optimal point when optimizing in the inner ball $B_r(v)$ is $v^* \in B_{\frac{r}{2}}(v)$, one must have a solution for the annulus problem at the boundary $\{||x - v||_2 = \frac{r}{2}\}$. To give a sketch of that proof, this is because otherwise if the solution is x^*, one can draw a straight line between x^* and v^* over which f_0 is convex, meaning that unless f_0 is constant over the line segment between x^* and v^*, the convexity of f_0 ensures that another point in the line segment and the annulus, called x^{**}, is such that $f_0(x^{**}) < f_0(x^*)$, resulting in a contradiction.
Then, one can use gradient descent at various points to solve the generally non-convex one-dimensional optimization problem over the inner boundary and find a possible solution.