SEAM CARVING

DONE BY:
DEMA AL-THUNAYAN
NOUF AL-QAHTANI
HANEEN AL-AQEEL

SUPERVISED BY :
MS.MALAK ALMOJALY

Our Algorithmic Approach

BRUTE FORCE

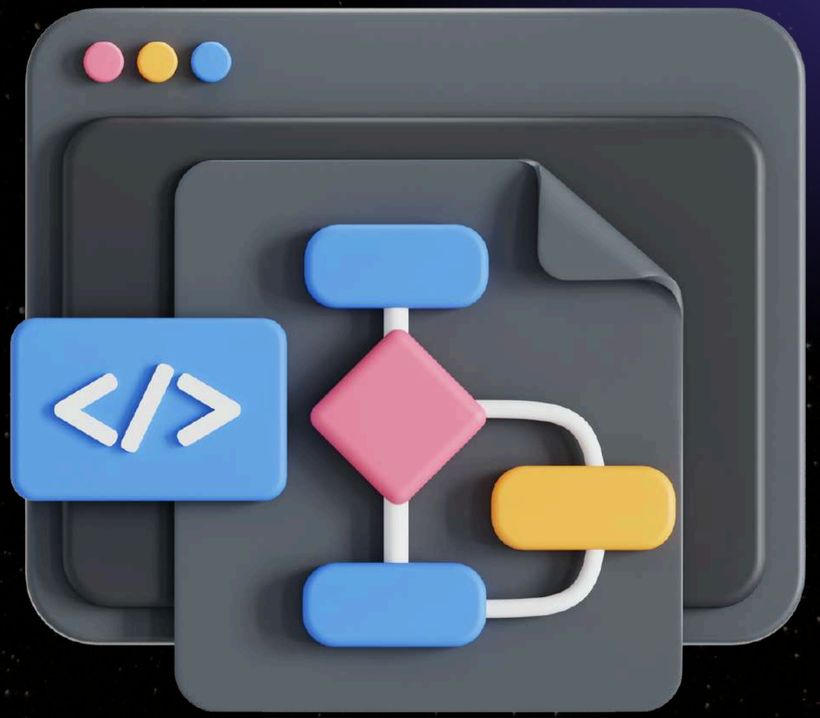
Tries all possible solutions to find the best seam, but can be slow for large images.

DYNAMIC PROGRAMMING

Efficiently finds the optimal seam by breaking the problem into smaller, overlapping subproblems.

GREEDY

Chooses the best local option at each step, faster but may not always find the global best seam.



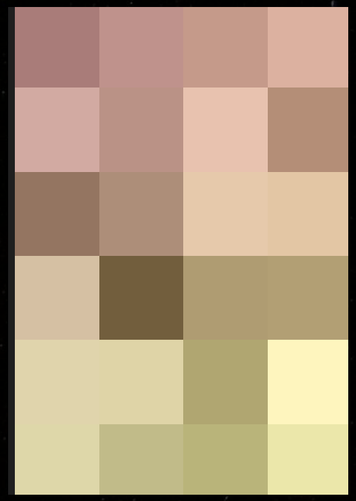
Brute force

In the brute force method, we generate and evaluate all possible paths from the first to the last column. For each path, we calculate the total energy by summing the energies along the path. After evaluating all paths, we select the one with the minimum total energy.

Although this guarantees finding the optimal solution, it is computationally expensive because the number of paths grows exponentially with the grid size.

OREGENAL IMAGE:

IMAGE AFTER BRUTE FORCE CARVING:



Brute force

COST = 0

MINCOST = +∞

ENERGY MATRIX :

0.1	0.8	0.8	0.3	0.5	0.4
0.7	0.8	0.1	0.0	0.8	0.4
0.8	0.0	0.4	0.7	0.2	0.9
0.9	0.0	0.0	0.5	0.9	0.4
0.2	0.4	0.0	0.2	0.4	0.5
0.2	0.4	0.2	0.5	0.3	0.0

MINSEAM ARRAY:

Brute force

COST = 1

MINCOST = 1

ENERGY MATRIX :

0.1	0.8	0.8	0.3	0.5	0.4
0.7	0.8	0.1	0.0	0.8	0.4
0.8	0.0	0.4	0.7	0.2	0.9
0.9	0.0	0.0	0.5	0.9	0.4
0.2	0.4	0.0	0.2	0.4	0.5
0.2	0.4	0.2	0.5	0.3	0.0

MINSEAM ARRAY:

0
0
1
2
2
2

Brute force

MINSEAM ARRAY

MINSEAMPERCOL MATRIX :

0	...	...	...	...	...
0	...	...	...	...	...
1	...	...	...	...	...
2	...	...	...	...	...
2	...	...	...	...	...
2	...	...	...	...	...

MINSEAM ARRAY:

0
0
1
2
2
2

MINCOST

MINCOSTPERCOL ARRAY:

1	...	...	...	...	...

MINCOST = 1

PSEUDO CODE :

```
findSeamByBruteForce( ) {

minCostPerCol ← new double[width]
minSeamPerCol ← new int[width][height]

For col ← 0 to width - 1 do
minCost ← ∞
minSeam ← new int[height]
int[] StartSeam ← new int[height]
StartSeam[0] ← col
findSeamByBruteForce(0, col, Enargy[0][col], StartSeam)

minCostPerCol[col] ← minCost
minSeamPerCol[col] ← minSeam.clone()
}
```

```
findSeamByBruteForce(row, col, cost, seam)  {

If row == height - 1 then
    If cost < minCost then
        minCost ← cost
          minSeam ← seam.clone()
Return

For c = col - 1 to col + 1 do
If c >= 0 AND c < width then
int[] newSeam ← seam.clone()
newSeam[row + 1] ← c

findSeamByBruteForce(row + 1, c, cost + Enargy[row + 1]
[c], newSeam)
}
```

Dynamic Programming

Dynamic programming efficiently finds the optimal seam by breaking the problem into smaller overlapping subproblems.

At each pixel, it computes the minimum total energy needed to reach that point based on its neighboring pixels. This guarantees finding the optimal solution while being much faster than brute force.

OREGENAL IMAGE:

IMAGE AFTER DYNAMIC PROGRAMMING CARVING:

Dynamic Programming

OVERLAPPING SUBPROBLEMS

ENERGY MATRIX :

0.1	0.8	0.8	0.3	0.5	0.4
0.7	0.8	0.1	0.0	0.8	0.4
0.8	0.0	0.4	0.7	0.2	0.9
0.9	0.0	0.0	0.5	0.9	0.4
0.2	0.4	0.0	0.2	0.4	0.5
0.2	0.4	0.2	0.5	0.3	0.0

BRUTE FORCE RECALCULATES SIMILAR PATHS MANY TIMES

Dynamic Programming

ENERGY MATRIX :

0.1	0.8	0.8	0.3	0.5	0.4
0.7	0.8	0.1	0.0	0.8	0.4
0.8	0.0	0.4	0.7	0.2	0.9
0.9	0.0	0.0	0.5	0.9	0.4
0.2	0.4	0.0	0.2	0.4	0.5
0.2	0.4	0.2	0.5	0.3	0.0

MINIMAL ENERGY TO BOTTOM :

Dynamic Programming

ENERGY MATRIX :

0.1	0.8	0.8	0.3	0.5	0.4
0.7	0.8	0.1	0.0	0.8	0.4
0.8	0.0	0.4	0.7	0.2	0.9
0.9	0.0	0.0	0.5	0.9	0.4
0.2	0.4	0.0	0.2	0.4	0.5
0.2	0.4	0.2	0.5	0.3	0.0

MINIMAL ENERGY TO BOTTOM :

0.2	0.4	0.2	0.5	0.3	0.0

Dynamic Programming

ENERGY MATRIX :

0.1	0.8	0.8	0.3	0.5	0.4
0.7	0.8	0.1	0.0	0.8	0.4
0.8	0.0	0.4	0.7	0.2	0.9
0.9	0.0	0.0	0.5	0.9	0.4
0.2	0.4	0.0	0.2	0.4	0.5
0.2	0.4	0.2	0.5	0.3	0.0

MINIMAL ENERGY TO BOTTOM :

0.2	0.4	0.2	0.5	0.3	0.0

Dynamic Programming

ENERGY MATRIX :

0.1	0.8	0.8	0.3	0.5	0.4
0.7	0.8	0.1	0.0	0.8	0.4
0.8	0.0	0.4	0.7	0.2	0.9
0.9	0.0	0.0	0.5	0.9	0.4
0.2	0.4	0.0	0.2	0.4	0.5
0.2	0.4	0.2	0.5	0.3	0.0

MINIMAL ENERGY TO BOTTOM :

0.4					
0.2	0.4	0.2	0.5	0.3	0.0

Dynamic Programming

ENERGY MATRIX :

0.1	0.8	0.8	0.3	0.5	0.4
0.7	0.8	0.1	0.0	0.8	0.4
0.8	0.0	0.4	0.7	0.2	0.9
0.9	0.0	0.0	0.5	0.9	0.4
0.2	0.4	0.0	0.2	0.4	0.5
0.2	0.4	0.2	0.5	0.3	0.0

MINIMAL ENERGY TO BOTTOM :

1.0	1.1	1.1	0.6	1.1	1.7
0.9	1.0	0.3	0.6	1.7	1.3
1.0	0.2	0.6	0.9	0.9	1.7
1.3	0.2	0.2	0.7	1.3	0.8
0.4	0.6	0.2	0.4	0.4	0.5
0.2	0.4	0.2	0.5	0.3	0.0

Dynamic Programming

ENERGY MATRIX :

0.1	0.8	0.8	0.3	0.5	0.4
0.7	0.8	0.1	0.0	0.8	0.4
0.8	0.0	0.4	0.7	0.2	0.9
0.9	0.0	0.0	0.5	0.9	0.4
0.2	0.4	0.0	0.2	0.4	0.5
0.2	0.4	0.2	0.5	0.3	0.0

SHORTEST PATH

MINIMAL ENERGY TO BOTTOM :

1.0	1.1	1.1	0.6	1.1	1.7
0.9	1.0	0.3	0.6	1.7	1.3
1.0	0.2	0.6	0.9	0.9	1.7
1.3	0.2	0.2	0.7	1.3	0.8
0.4	0.6	0.2	0.4	0.4	0.5
0.2	0.4	0.2	0.5	0.3	0.0

PSEUDO CODE :

```
computeMinimalEnergyToBottom( ) {

For x ← 0 to width - 1 do
MinimalEnergyToBottom[height - 1][x] ← Energy[height - 1][x]

For y ← height - 2 down to 0 do
  For x ← 0 to width - 1 do
    If x == 0 then
MinimalEnergyToBottom[y][x] ← Energy[y][x] + min(MinimalEnergyToBottom[y+1][x], MinimalEnergyToBottom[y+1][x+1])
    Else if x == width - 1 then
MinimalEnergyToBottom[y][x] ← Energy[y][x] + min(MinimalEnergyToBottom[y+1][x-1], MinimalEnergyToBottom[y+1][x])
    Else
MinimalEnergyToBottom[y][x] ← Energy[y][x] + min(MinimalEnergyToBottom[y+1][x-1], MinimalEnergyToBottom[y+1][x],
MinimalEnergyToBottom[y+1][x+1])

}
```

PSEUDO CODE :

findSeamByDP( ) {

```
min ← ∞
For i ← 0 to width - 1 do
  If MinimalEnergyToBottom[0][i] < min then
     min ← MinimalEnergyToBottom[0][i]
     index ← i
minSeam[0] ← index

For y ← 0 to height - 2 do
  If index == 0 then
     If MinimalEnergyToBottom[y+1][index+1] < MinimalEnergyToBottom[y+1][index]
        index ← index + 1
  Else if index == width - 1 then
     If MinimalEnergyToBottom[y+1][index-1] < MinimalEnergyToBottom[y+1][index]
        index ← index - 1
  Else
     min ← ∞
     For i ← index - 1 to index + 1 do
        If MinimalEnergyToBottom[y+1][i] < min
           min ← MinimalEnergyToBottom[y+1][i]
           index ← i
  minSeam[y+1] ← index
}
```

Greedy Algorthim

This approach is based on the greedy strategy, where the algorithm selects the lowest-energy pixel from the top row and continues downward, choosing the least-energy option among the three adjacent pixels (left, middle, right) at each step.

It makes decisions **LOCALLY**, row by row, without considering the full path.

OREGENAL IMAGE:

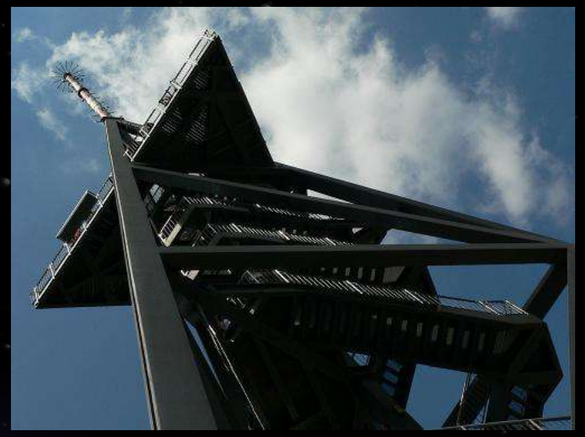
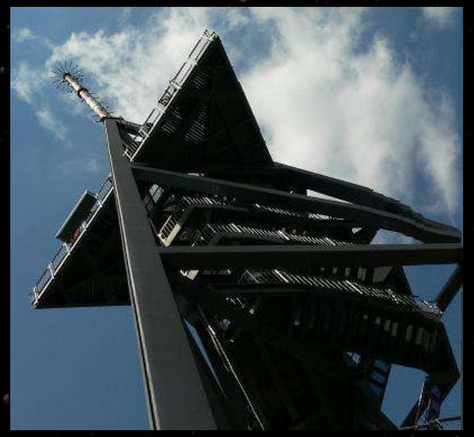


IMAGE AFTER GREEDY CARVING:



Dynamic Programming

ENERGY MATRIX :

0.1	0.8	0.8	0.3	0.5	0.4
0.7	0.8	0.1	0.0	0.8	0.4
0.8	0.0	0.4	0.7	0.2	0.9
0.9	0.0	0.0	0.5	0.9	0.4
0.2	0.4	0.0	0.2	0.4	0.5
0.2	0.4	0.2	0.5	0.3	0.0

Greedy Algorithm

ENERGY MATRIX :

0.1	0.8	0.8	0.3	0.5	0.4
0.7	0.8	0.1	0.0	0.8	0.4
0.8	0.0	0.4	0.7	0.2	0.9
0.9	0.0	0.0	0.5	0.9	0.4
0.2	0.4	0.0	0.2	0.4	0.5
0.2	0.4	0.2	0.5	0.3	0.0

PSEUDO CODE :

```
findGreedySeam() {
seam ← new int[height]
// Find the column in the first row with minimum energy
currentCol ← 0
minEnergy ← ∞
For col ← 0 to width - 1 do
  If Energy[0][col] < minEnergy then
     minFirstRowEnergy← Energy[0][col]
     currentCol ← col
Seam[0] ← currentCol
// Traverse from the second row to the bottom
For y ← 1 to height - 1 do
   If currentCol - 1 ≥ 0:
     left ← currentCol - 1
  Else: left ← currentCol
  If currentCol + 1 < width:
     right ← currentCol + 1
  Else: right ← currentCol   leftEnergy ← Energy[y][left]
  middleEnergy ← Energy[y][currentCol]
  rightEnergy ← Energy[y][right]
  // Choose the direction with the lowest energy
  If leftEnergy ≤ middleEnergy and leftEnergy ≤ rightEnergy then
     currentCol ← left
  Else if rightEnergy ≤ middleEnergy and rightEnergy < leftEnergy then
     currentCol ← right
  // Otherwise, currentCol stays the same (middle)
  Seam[y] ← currentCol
Return Seam
}
```

FINDING:

- **Brute Force** is best for **small** images or when absolute accuracy is required.
- **Dynamic Programming** offers a balance between **optimal results** and **efficiency**.
- **Greedy** is the **fastest** and **most efficient** but compromises on accuracy. It's ideal when you need quick results and can tolerate some loss in quality

CHALLENGES:

- How to process and apply the algorithms to the images through the code

 We searched and discovered that the image is treated like any other file using the File class and its object instances.

- The slowness of the brute-force algorithm.

 We used very small images, with dimensions of 6x6 pixels or less.

Thank You for Listening