Algorithm 1 Brute Force

Input: $grid \in \mathbb{R}^{m \times n}$

Output: $dist \in \mathbb{R}^{m \times n}$

for each $\boldsymbol{p}_1 \in grid$ **do**

  for each $\boldsymbol{p}_2 \in grid$ **do**

    $dist[\boldsymbol{p}_1] \leftarrow \min(dist[\boldsymbol{p}_1], \|\boldsymbol{p}_1 - \boldsymbol{p}_2\|_2)$

  end for

end for

Algorithm 2 Forward BFS

```
Input: grid ∈ ℝ^{m×n}
Output: dist ∈ ℝ^{m×n}
for each p ∈ grid do
    open.push(p), step ← 0, found ← false
    while not open.empty() and not found do
        for each c ∈ open do
            open.pop()
            closed.insert(c)
            if grid[c] == 0 then
                dist[c] ← step, found ← true
                break
            end if
            for each n ∈ c.neighbours do
                if n not in closed then
                    open.push(n)
                end if
            end for
        end for
        step ← step + 1
    end while
end for
```

Algorithm 3 Backward BFS

```
Input: grid ∈ ℝ^{m×n}
Output: dist ∈ ℝ^{m×n}
for each p ∈ grid do
    if grid[p] == 0 then
        open.push(p)
    end if
end for
step ← 0
while not open.empty() do
    for each c ∈ open do
        open.pop()
        closed.insert(c)
        dist[c] ← step
        for each n ∈ c.neighbours do
            if n not in closed then
                open.push(n)
            end if
        end for
    end for
    step ← step + 1
end while
```

Algorithm 4 L1 DP

Input: $grid \in \mathbb{R}^{m \times n}$
Output: $dist \in \mathbb{R}^{m \times n}$
for each $\boldsymbol{p} \in grid$ **do**
  if $grid[\boldsymbol{p}] == 0$ **then**
    $dist[\boldsymbol{p}] \leftarrow 0$
  end if
end for
for $x = 0$ **to** $m - 1$ **do**
  for $y = 0$ **to** $n - 1$ **do**
    $dist[x][y] \leftarrow \min(dist[x][y], dist[x-1][y] + 1)$
    $dist[x][y] \leftarrow \min(dist[x][y], dist[x][y-1] + 1)$
  end for
end for
for $x = m - 1$ **to** 0 **do**
  for $y = n - 1$ **to** 0 **do**
    $dist[x][y] \leftarrow \min(dist[x][y], dist[x+1][y] + 1)$
    $dist[x][y] \leftarrow \min(dist[x][y], dist[x][y+1] + 1)$
  end for
end for

Algorithm 5 Distance Transform 1D

Input: $f : \mathbb{R} \to \mathbb{R}$
Output: $dist \in \mathbb{R}^n$
$k \leftarrow 0$ ▷ Index of rightmost parabola in lower envelope
$v[0] \leftarrow 0$ ▷ Locations of parabolas in lower envelope
$z[0] \leftarrow -\infty,\ z[1] \leftarrow +\infty$ ▷ Locations of boundaries between parabolas
for $q = 1$ **to** $n - 1$ **do**
    $s \leftarrow ((f(q) + q^2) - (f(v[k]) + v[k]^2))/(2q - 2v[k])$
    while $s \leq z[k]$ **do**
        $k \leftarrow k - 1$
        $s \leftarrow ((f(q) + q^2) - (f(v[k]) + v[k]^2))/(2q - 2v[k])$
    end while
    $k \leftarrow k + 1$
    $v[k] \leftarrow q$
    $z[k] \leftarrow s,\ z[k+1] \leftarrow +\infty$
end for
$k \leftarrow 0$
for $q = 0$ **to** $n - 1$ **do**
    while $z[k+1] < q$ **do**
        $k \leftarrow k + 1$
    end while
    $dist[q] \leftarrow (q - v[k]^2) + f(v[k])$
end for

Algorithm 6 Incremental Backward BFS

```
Input: grid ∈ ℝ^{m×n}, insertQueue, deleteQueue
Output: dist ∈ ℝ^{m×n}
while not insertQueue.empty() do
  cur ← insertQueue.front()
  insertQueue.pop()
  deleteFrom(cur.coc, cur.pos)
  cur.coc ← cur.pos
  cur.dis ← 0
  insertTo(cur.coc, cur.pos)
  updateQueue.push(cur)
end while
while not deleteQueue.empty() do
  cur ← deleteQueue.front()
  deleteQueue.pop()
  for each vox ∈ cur.dll do
    deleteFrom(vox.coc, vox.pos)
    vox.coc ← IP
    vox.dis ← ∞
    for each nbr ∈ vox.nbrs do
      if nbr.coc exists and getDist(nbr.coc, vox.pos) < vox.dis then
        vox.dis ← getDist(nbr.coc, vox.pos)
        vox.coc ← nbr.coc
      end if
    end for
    insertTo(vox.coc, vox.pos)
    if vox.coc is not IP then
      updateQueue.push(vox)
    end if
  end for
end while
while not updateQueue.empty() do
  cur ← updateQueue.front()
  updateQueue.pop()
  for each nbr ∈ cur.nbrs do
    if getDist(cur.coc, nbr.pos) < nbr.dis then
      nbr.dis ← getDist(cur.coc, nbr.pos)
      deleteFrom(nbr.coc, nbr.pos)
      nbr.coc ← cur.coc
      insertTo(nbr.coc, nbr.pos)
      updateQueue.push(nbr)
    end if
  end for
end while
```