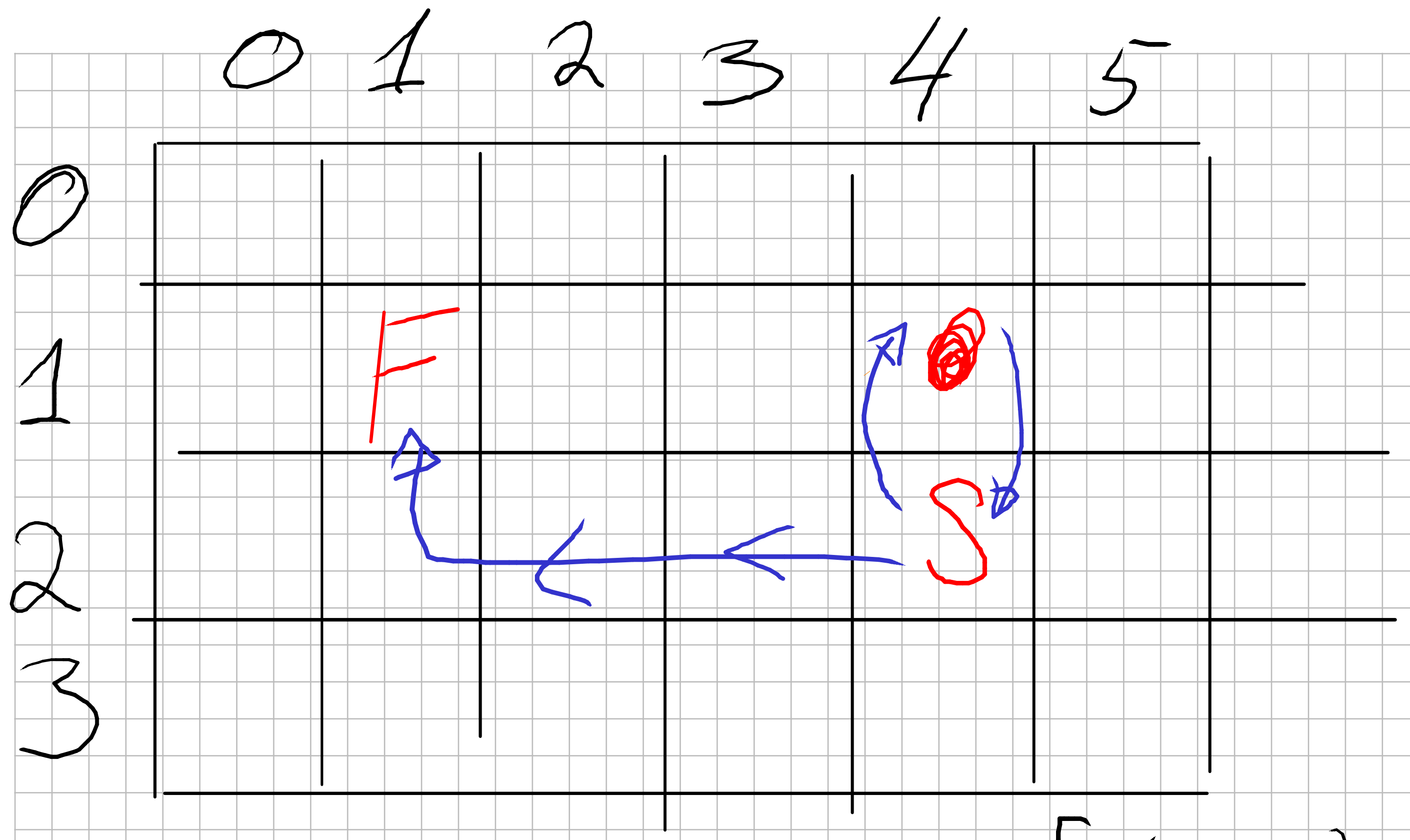


S = (4, 2)
F = (1, 1)

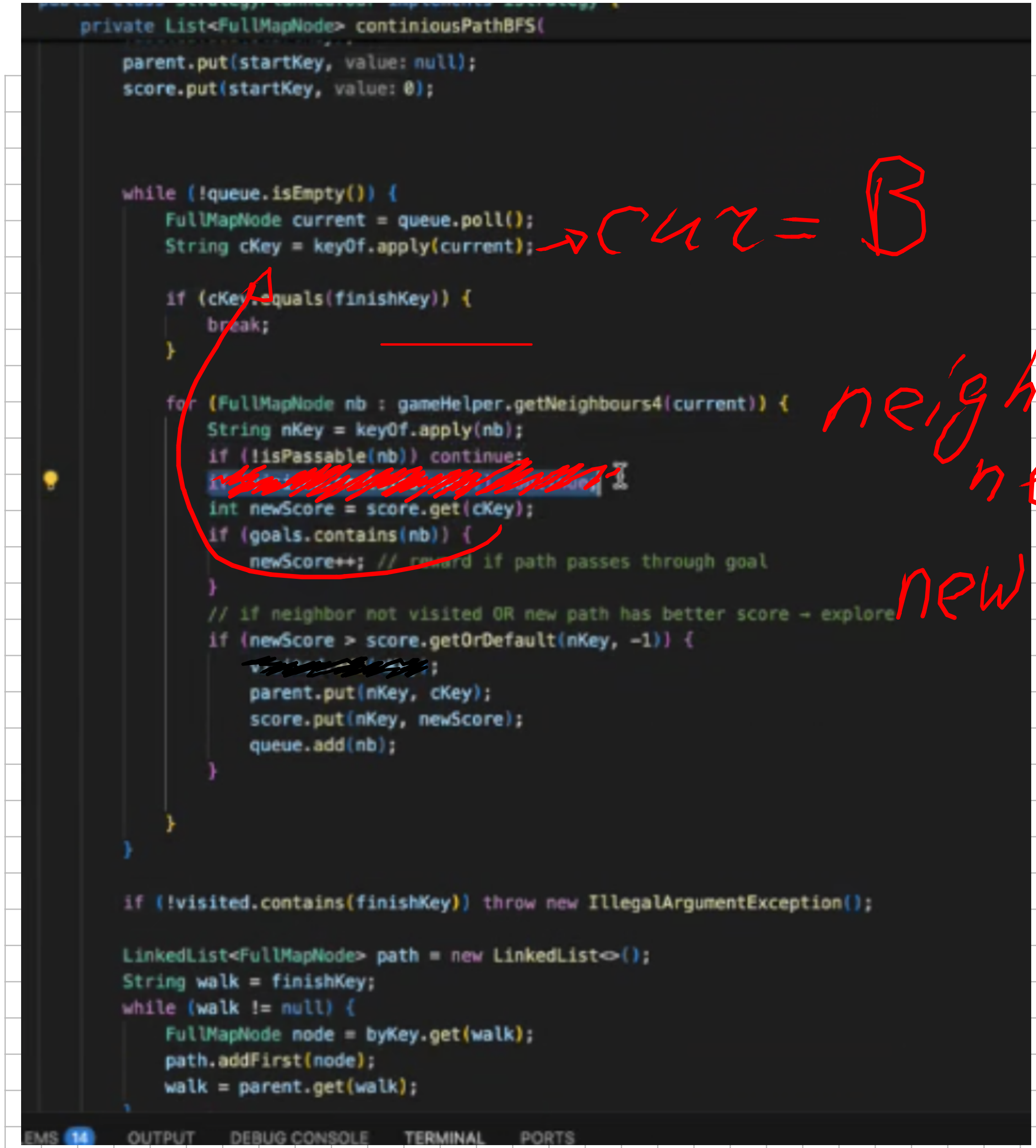
Visited

queue = [(5,2) (4,3) (3,2)]

cur = (4,1)

(5,1) (3,1) (4,0) (4,2)

$(4,2) \to (4,1) \longrightarrow (4,2)$



```
    private List<FullMapNode> continiousPathBFS(
        parent.put(startKey, value: null);
        score.put(startKey, value: 0);


        while (!queue.isEmpty()) {
            FullMapNode current = queue.poll();
            String cKey = keyOf.apply(current);

            if (cKey.equals(finishKey)) {
                break;
            }

            for (FullMapNode nb : gameHelper.getNeighbours4(current)) {
                String nKey = keyOf.apply(nb);
                if (!isPassable(nb)) continue;
                [redacted]
                int newScore = score.get(cKey);
                if (goals.contains(nb)) {
                    newScore++; // reward if path passes through goal
                }
                // if neighbor not visited OR new path has better score → explore
                if (newScore > score.getOrDefault(nKey, -1)) {
                    [redacted];
                    parent.put(nKey, cKey);
                    score.put(nKey, newScore);
                    queue.add(nb);
                }

            }
        }

        if (!visited.contains(finishKey)) throw new IllegalArgumentException();

        LinkedList<FullMapNode> path = new LinkedList<>();
        String walk = finishKey;
        while (walk != null) {
            FullMapNode node = byKey.get(walk);
            path.addFirst(node);
            walk = parent.get(walk);
```

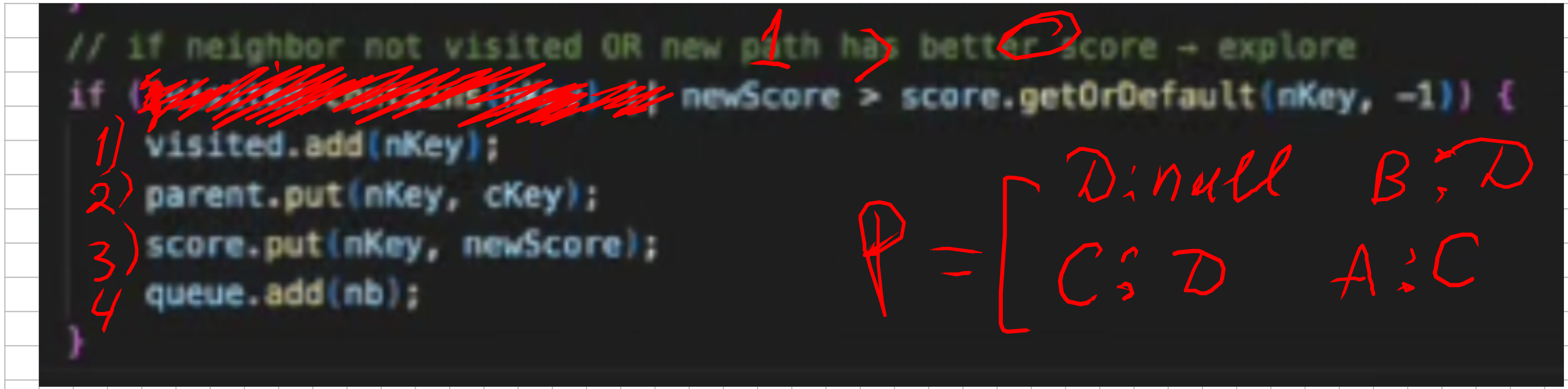
→ cur = B

neighbours = [D A]

nB = D

new Score = 1 + 0 = 1

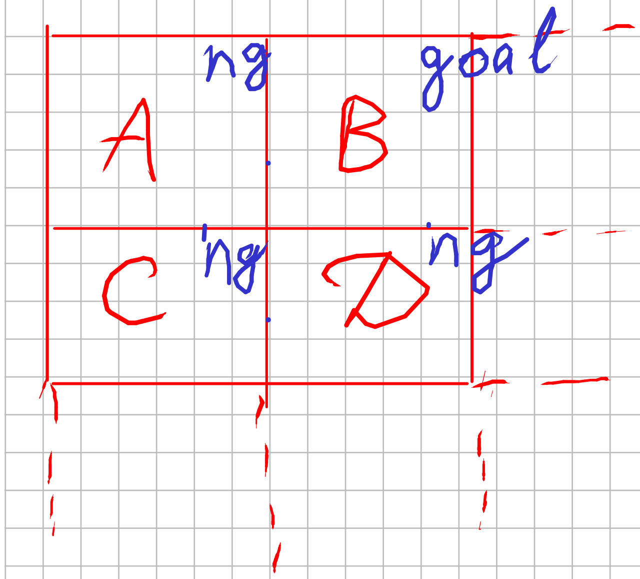
Start = D Finish = A



```
// if neighbor not visited OR new path has better score → explore
if (~~!visited.contains(nKey) ||~~ newScore > score.getOrDefault(nKey, -1)) {
1)  visited.add(nKey);
2)  parent.put(nKey, cKey);
3)  score.put(nKey, newScore);
4   queue.add(nb);
}
```

1 > 0

P = [D: null B: D
C: D A: C



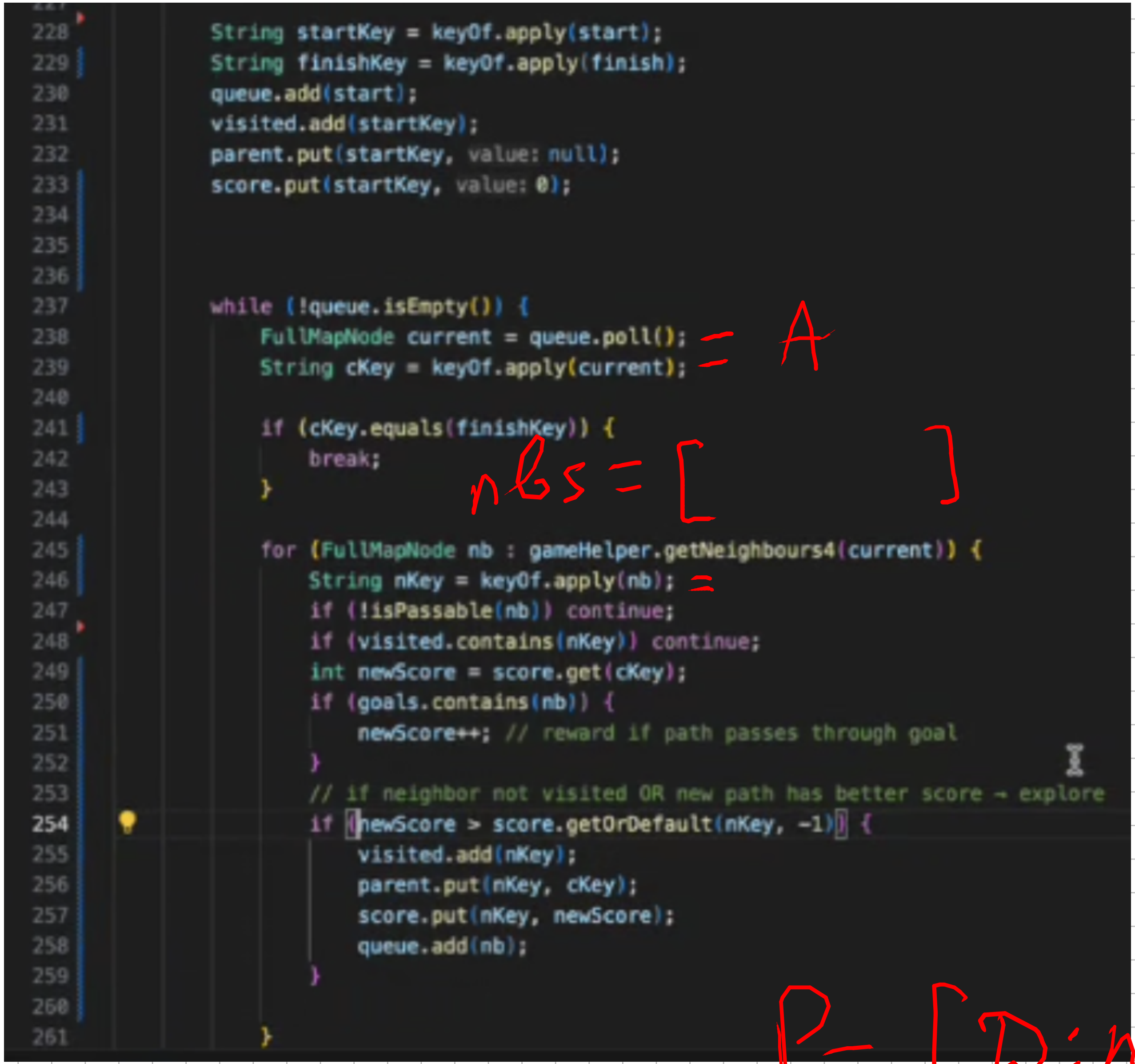
Q = [A]

V = [D, C, B, A]

Score = [(D, 0) (C, 0) (B, 1) (A, 0)

cur = B nb = D

Start = D   Finish = A

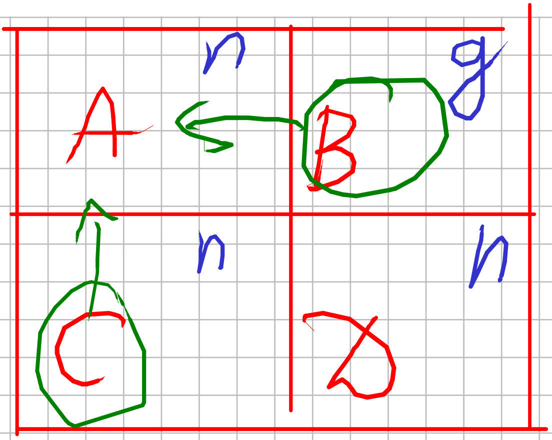


```
String startKey = keyOf.apply(start);
String finishKey = keyOf.apply(finish);
queue.add(start);
visited.add(startKey);
parent.put(startKey, value: null);
score.put(startKey, value: 0);



while (!queue.isEmpty()) {
    FullMapNode current = queue.poll();
    String cKey = keyOf.apply(current);

    if (cKey.equals(finishKey)) {
        break;
    }

    for (FullMapNode nb : gameHelper.getNeighbours4(current)) {
        String nKey = keyOf.apply(nb);
        if (!isPassable(nb)) continue;
        if (visited.contains(nKey)) continue;
        int newScore = score.get(cKey);
        if (goals.contains(nb)) {
            newScore++; // reward if path passes through goal
        }
        // if neighbor not visited OR new path has better score → explore
        if (newScore > score.getOrDefault(nKey, -1)) {
            visited.add(nKey);
            parent.put(nKey, cKey);
            score.put(nKey, newScore);
            queue.add(nb);
        }

    }
```

= A

nbs = [ ]

=



Q = [ ]

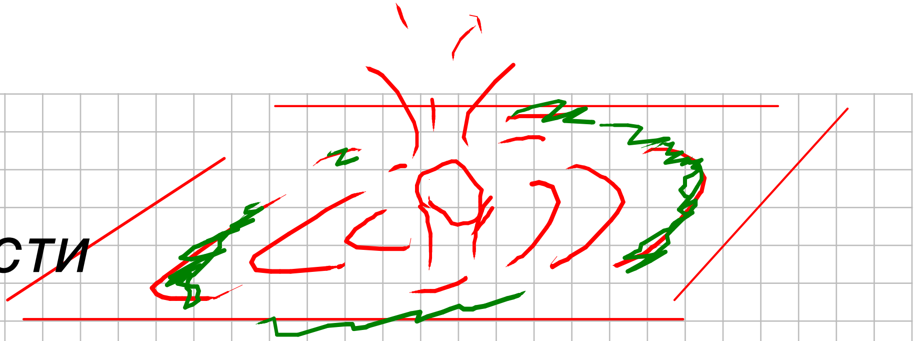
V = [D, C, B, A]

Score = D:0
C:0 B:1 A:0

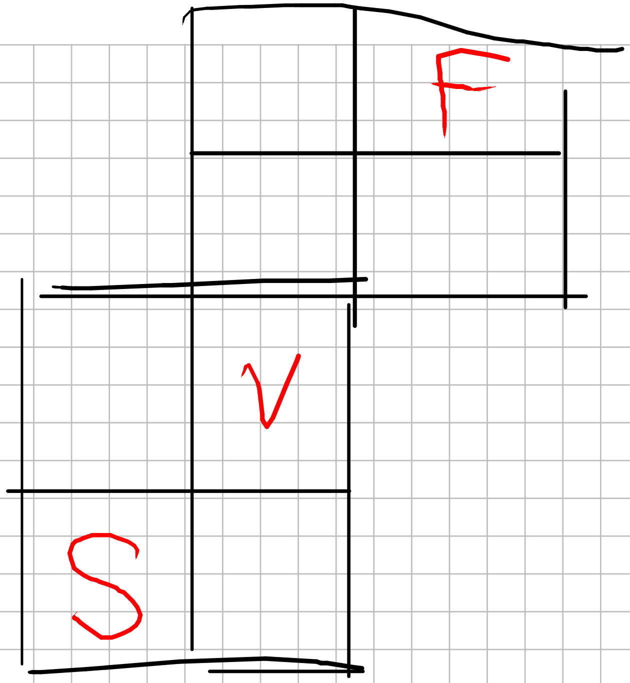
P = [D:null, C:D, B:D
A:C

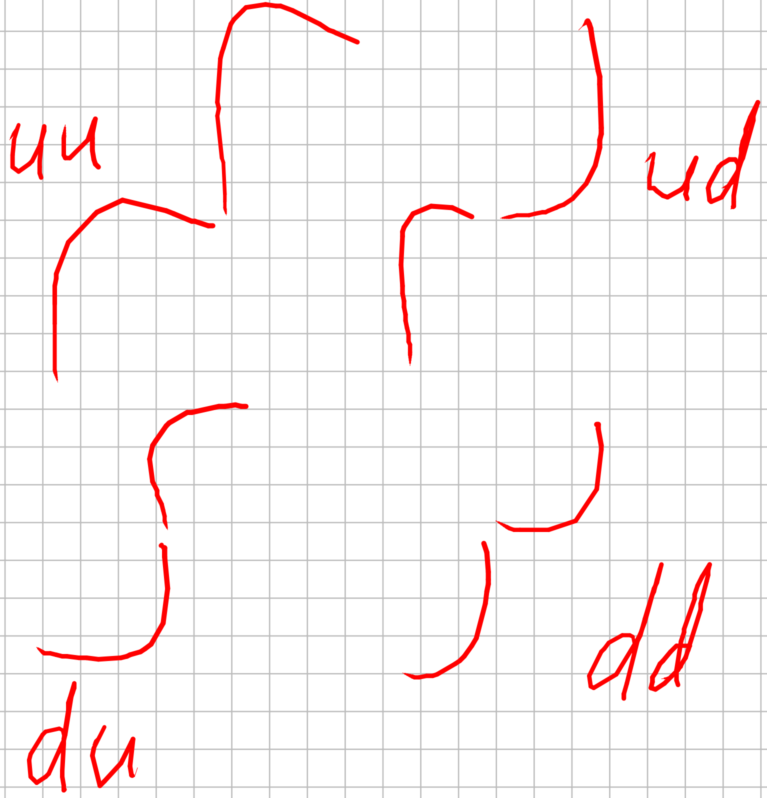
D → C → A

Волновой фронт (frontier)



камень падает в озеро, на поверхности воды поянляются круги, концентрические (вложенные) окружности.
Та окружность, которая появилась самой первой, и находится в любой момент времени дальше всего от центра, называется волновой фронт.


F
V
S


uu
ud
du
dd

Path Priorities:

1) Path length (Terrain cost)

2) Intermed. goals

3) Random choice

↑ getNeighb4(node) →

→ [V_N, V_E, V_S, V_W] → shuffle

Simple queue (w/o priority part) doesn't solve "shortest path" task when facing map with different terrain costs!

Q: is it true?
Provide an example that shows that simple queue fails the task.

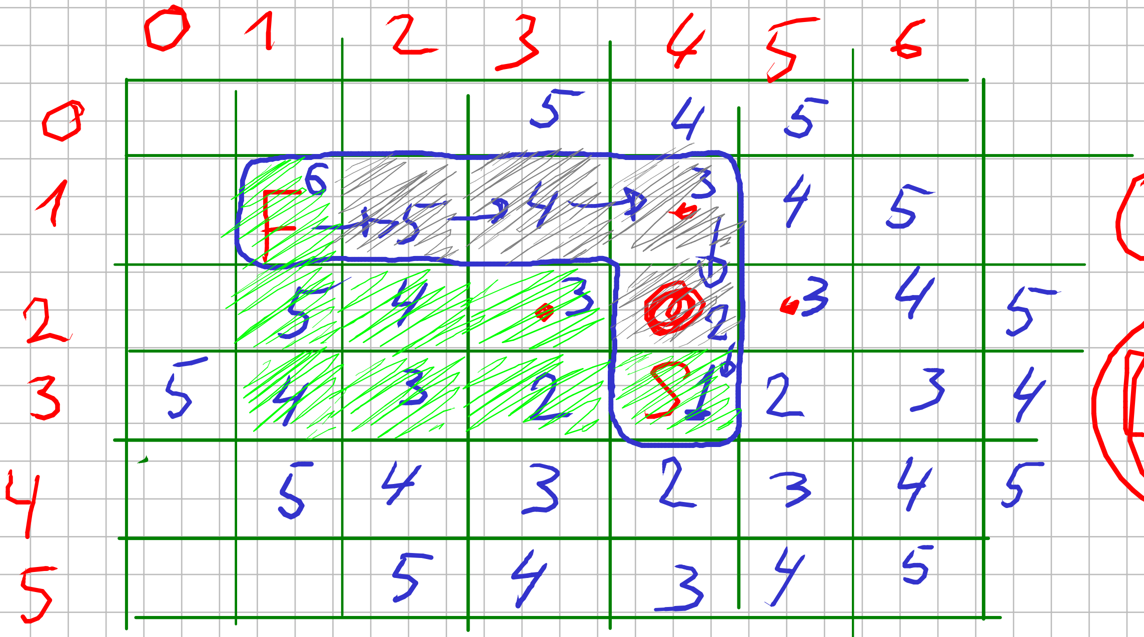
Q: How do we need to modify the algorithm so that "shortest path" problem gets solved correctly?

Ter. cost = ?

1

4 n 2

3



	0	1	2	3	4	5	6	
0				5	4	5		
1		F 6	5	4	3	4	5	
2		5	4	3	◎ 2	3	4	5
3	5	4	3	2	S 1	2	3	4
4		5	4	3	2	3	4	5
5			5	4	3	4	5	

Q = [4,3]

[4,2 5,3 4,4 3,3]

[5,3 4,4 3,3] + [nb(4,2)]

Priority Queue vs Simple Queue

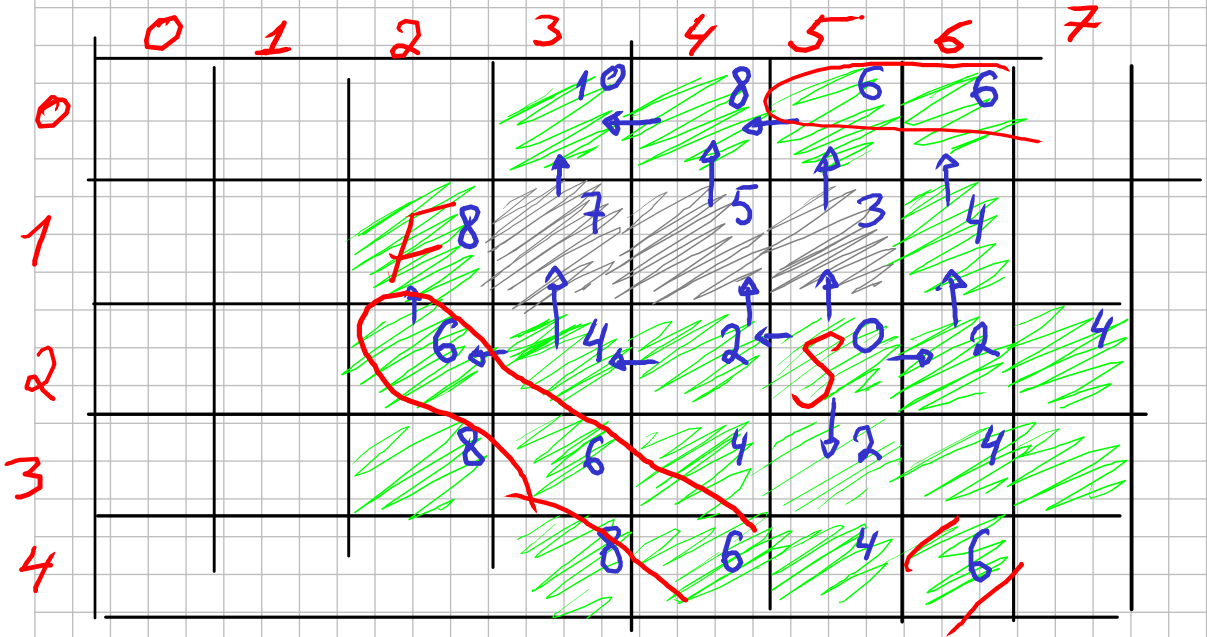
*Simple Queue: elements are retrieved in order they were added.
in other words FIFO, First In First Out*

** LIFO, Last In Forst Out - stack protocol*

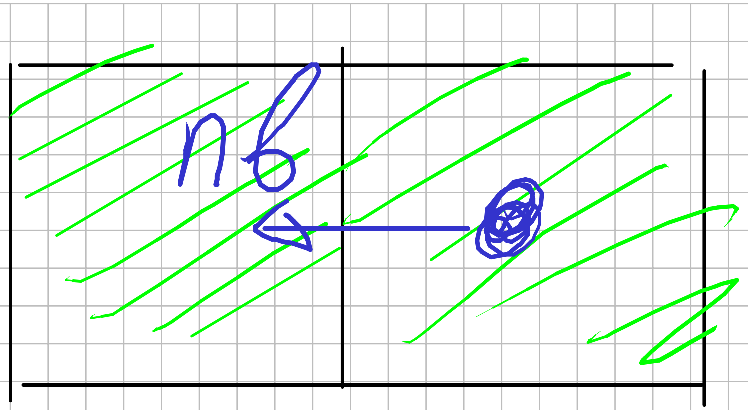
*Priority Queue: upon placement elements are being assigned
a priority value. Elements with lowerst pririty are being
retrieved fist.*

M_in = 2
M_out = 2

G_in = 1
G_out = 1



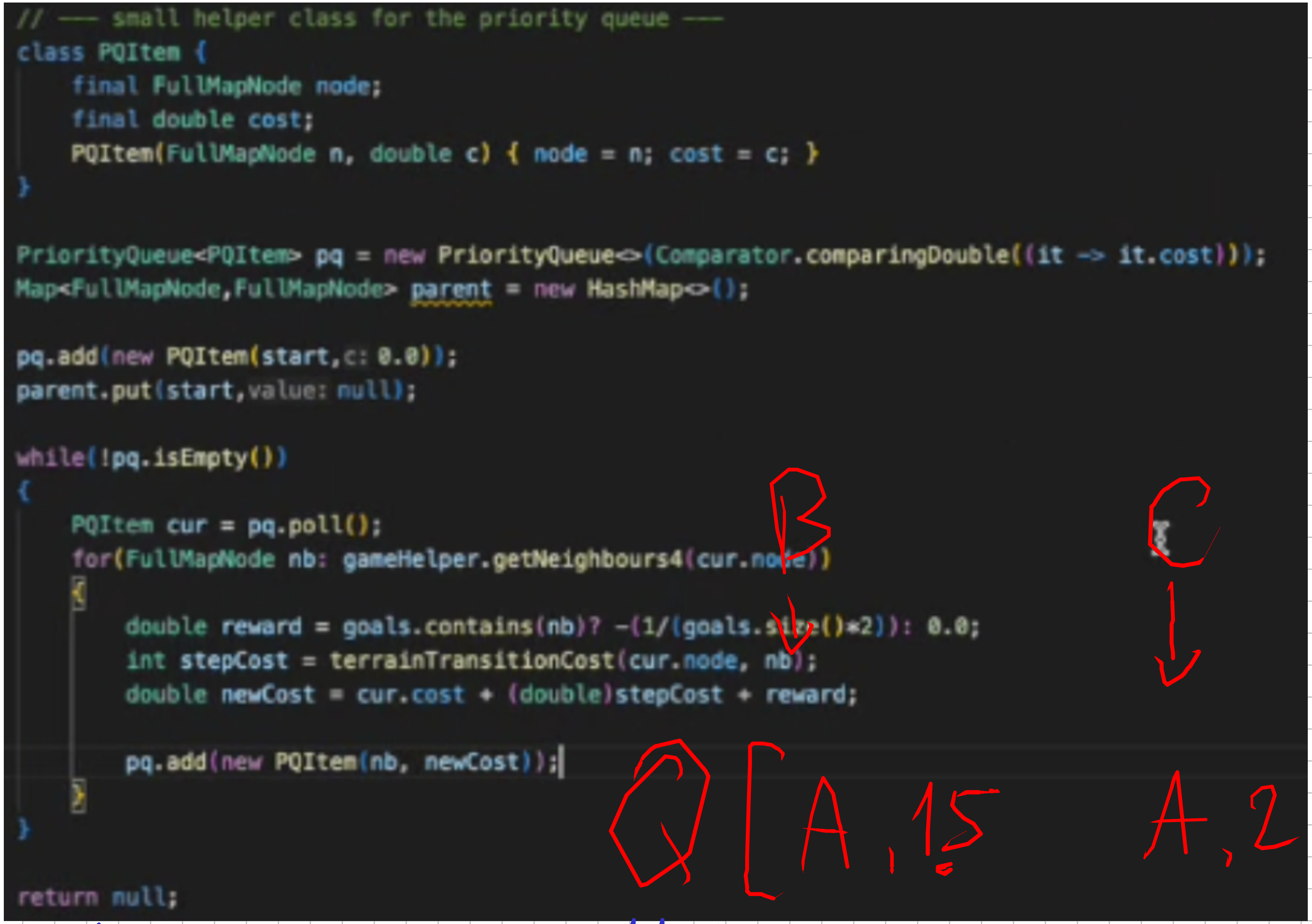
turn:	frontier
0:	(5,2)
1:	null
2:	(4,2) (6,2) (5,3)
3:	(5,1)
4:	(3,2), (6,1), (7,2) (4,3) (6,3), (5,4)
5:	(4,1)



if (nb is a goal) ↳

terain_cost = TC − 1/1000

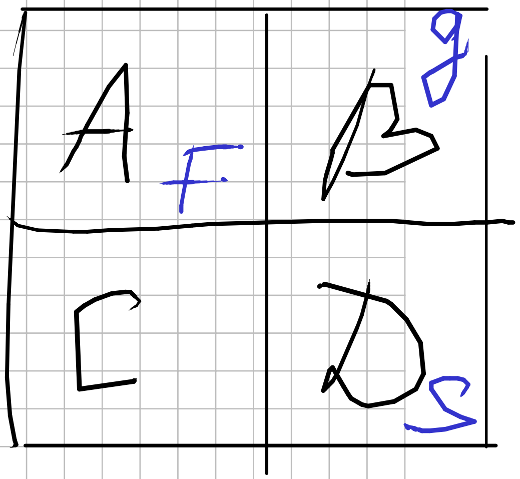
3 → 2.999



```
// --- small helper class for the priority queue ---
class PQItem {
    final FullMapNode node;
    final double cost;
    PQItem(FullMapNode n, double c) { node = n; cost = c; }
}

PriorityQueue<PQItem> pq = new PriorityQueue<>(Comparator.comparingDouble((it -> it.cost)));
Map<FullMapNode,FullMapNode> parent = new HashMap<>();

pq.add(new PQItem(start,c: 0.0));
parent.put(start,value: null);

while(!pq.isEmpty())
{
    PQItem cur = pq.poll();
    for(FullMapNode nb: gameHelper.getNeighbours4(cur.node))
    {
        double reward = goals.contains(nb)? -(1/(goals.size()*2)): 0.0;
        int stepCost = terrainTransitionCost(cur.node, nb);
        double newCost = cur.cost + (double)stepCost + reward;

        pq.add(new PQItem(nb, newCost));
    }
}

return null;
```

B ↓   C ↓

Q [A, 1.5   A, 2]



Q = {(D, 0)}

nB = C ;   Q.add(C, 1)

nB = B ;   Q.add(B, 0.5)   ⟼ Q = [B 0.5   C 1]

cur = B 0.5   nB = A   Q.add(A, 1.5)

E^{g}	F	G	H_{F}
A_{S}	B	C^{g}	D^{g}

$$Q = [(\overset{B}{\downarrow}V, 5) \ldots]$$

Parent : V:B

$C \rightarrow V$ cost = 4.7

$Q = [V,(4.7),..]$

Parent
V:C

update both Q and Parent

no update

$D \rightarrow V$ cost = 5.2

$Q = [V,5 \ldots]$

Parent : V:B