

Group Project 2 (Knight Moves)



Mahidol University
International College

Akkhrawin Nair	6580013
Nitchayanin Thamkunanon	6580081
Panupong Sangaphunchai	6580587
Sukumarn Srimai	6581085

Mahidol University International College
EGCI 221 Data Structures and Algorithms

Assoc. Prof. Dr. Rangsipan Marukatat

7th December 2024

Contents

User Manual	2
Graph data structures	7
Other data structures	12
Graph algorithm	15
Limitations	20
References	21

User Manual

This program shows the user the path of the input Knight chess piece to the input castle piece while avoiding the bomb placed on the board given an $N \times N$ board with a Knight, a Castle, and location(s) of the bombs.

The Knight can only move in L shape, i.e.,

- 2 cells horizontally, then 1 cell vertically OR
- 2 cells vertically, then 1 cell horizontally
- But it cannot leave the board and must not land on a Bomb

How the program works

1. Get $N \times N$ input of the board size from the user

```
run:
Enter N for N*N board (N must be at least 5)

5
```

2. Get the location of the Knight from the user

```
Enter Knight ID

7
```

3. Get the location of the castle from the user

```
Enter Castle ID

0
```

4. Get the location(s) of the bombs (can not be left blank) Since inserting **invalid IDs will be ignored**, inserting a string, character or any number less than 0 or exceeds board size will result in bombIDs being null, meaning a user has not inserted a bomb. If the user inserted **some** invalid numbers or strings in the list of bombs, the user will be asked to insert data again.

```
Enter Knight ID
1
Enter Castle ID
14
Enter bomb IDs separated by commas (invalid IDs will be ignored)

you havent inserted anything
Enter bomb IDs separated by commas (invalid IDs will be ignored)
25
```

When the inserted ID is invalid, ignore and treat it as the user has not inserted any bombs

```
Enter bomb IDs separated by commas(invalid IDs will be ignored)
```

```
apple
```

```
Initial --> knight at 7
```

0:C*	1:	2:	3:	4:	5:
6:	7:K*	8:	9:	10:	11:
12:	13:	14:	15:	16:	17:
18:	19:	20:	21:	22:	23:
24:	25:	26:	27:	28:	29:
30:	31:	32:	33:	34:	35:

```
Best route to Castle = 4 moves
```

But when **some** IDs in the list are invalid, the user will be asked to insert bomb IDs again

```
Enter bomb IDs separated by commas(invalid IDs will be ignored)
```

```
0,32,1000
```

```
Some IDs are exceeding the board, please enter the full list again
```

```
Enter bomb IDs separated by commas(invalid IDs will be ignored)
```

```
|
```

- Once the user has inserted the bombs, the program will calculate the shortest path based on the data given by the user

Example : KnightID = 7, CastleID = 0, no bombs

```
Initial --> knight at 7
```

0:C*	1:	2:	3:	4:	5:
6:	7:K*	8:	9:	10:	11:
12:	13:	14:	15:	16:	17:
18:	19:	20:	21:	22:	23:
24:	25:	26:	27:	28:	29:
30:	31:	32:	33:	34:	35:

```
Best route to Castle = 4 moves
```

```
Move 1 --> jump to 15
```

0:C*	1:	2:	3:	4:	5:
6:	7:	8:	9:	10:	11:
12:	13:	14:	15:K*	16:	17:
18:	19:	20:	21:	22:	23:
24:	25:	26:	27:	28:	29:
30:	31:	32:	33:	34:	35:

Move 2 --> jump to 4					
0:C*	1:	2:	3:	4:K*	5:
6:	7:	8:	9:	10:	11:
12:	13:	14:	15:	16:	17:
18:	19:	20:	21:	22:	23:
24:	25:	26:	27:	28:	29:
30:	31:	32:	33:	34:	35:
Move 3 --> jump to 8					
0:C*	1:	2:	3:	4:	5:
6:	7:	8:K*	9:	10:	11:
12:	13:	14:	15:	16:	17:
18:	19:	20:	21:	22:	23:
24:	25:	26:	27:	28:	29:
30:	31:	32:	33:	34:	35:
Move 4 --> jump to 0					
0:C+K	1:	2:	3:	4:	5:
6:	7:	8:	9:	10:	11:
12:	13:	14:	15:	16:	17:
18:	19:	20:	21:	22:	23:
24:	25:	26:	27:	28:	29:
30:	31:	32:	33:	34:	35:

This means, the shortest path from KnightID 7 to CastleID 0 is 4 moves.

6. After finish running, get answer for restarting option

If y

```
New game (y/n)?
y
Enter N for N*N board (N must be at least 5)
|
```

If n

```
=====
New game (y/n)?
n
BUILD SUCCESSFUL (total time: 11 minutes 7 seconds)
|
```

If others, the user will be asked to insert y or n again

```
New game (y/n)?
d
Please insert y or n!
New game (y/n)?
|
```

The program's board size must be more than 4×4 for the Knight to move.

Error Handling

Board size error will be solved by the program asking for input again

```
Enter N for N*N board (N must be at least 5)
3
=====
N should be at least 5, insert again!
=====

Enter N for N*N board (N must be at least 5)
|
```

invalid locations of the knight and the castle input will be solved by the program asking for input again.

```
Enter N for N*N board (N must be at least 5)
5
Cell IDs          0   1   2   3   4
                  5   6   7   8   9
                  10  11  12  13  14
                  15  16  17  18  19
                  20  21  22  23  24

Enter Knight ID
a
=====
Invalid input! Please enter an integer.
=====

Enter Knight ID
0
Enter Castle ID
24
Enter bomb IDs separated by commas (invalid IDs will be ignored)
|
```

invalid input for bomb(s) will result in the bomb id's ignored.

```

Enter bomb IDs separated by commas (invalid IDs will be ignored)
asdf

Initial --> knight at 0
    0:K*      1:      2:      3:      4:
    5:      6:      7:      8:      9:
   10:     11:     12:     13:     14:
   15:     16:     17:     18:     19:
   20:     21:     22:     23:     24:C*

Best route to Castle = 4 moves

Move 1 --> jump to 7
    0:      1:      2:      3:      4:
    5:      6:     7:K*     8:      9:
   10:     11:     12:     13:     14:
   15:     16:     17:     18:     19:
   20:     21:     22:     23:     24:C*

```

Graph data structures

In our program, we are not using a graph object from jgrapht. We did not build the graph explicitly but instead, we use an implicit graph which consists of an ArrayDeque (Name : q), HashMap (Name : parent) and ArrayLists (Names : knightXMoves, knightYMoves). The knight's moves are dynamically calculated during the BFS traversal. For each current position of the knight, it computes its neighbors (possible knight moves) and store in ArrayDeque q

Roles of each object

ArrayDeque (q) : Used to explore current knight's possible moves to see which one is valid. Once it finds the valid move, the coordinates of the new position will be pushed into the queue. It will repeat until the queue is empty.

HashMap (parent) : Used to store the current knight position as a key and the previous knight position as a value.

ArrayList (knightXMoves) : stores default knight moves (X coordinates)

ArrayList(knightYMoves) : stores default knight moves (Y coordinates)

```
ArrayDeque<Coordinate> q = new ArrayDeque<>();
HashMap<Coordinate, Coordinate> parent = new LinkedHashMap<>();
```

Class Board, line 100

```
int[] knightXMoves = knight.getKnightXMoves();
int[] knightYMoves = knight.getKnightYMoves();
```

Class Board, line 107

Type of the graph : undirected unweighted graph

Reason : We are using Breadth First Search to solve this problem. Our edges do not have any weights and they are bidirectional because in this problem, all knight moves have the same cost. We are just finding the shortest path from the source(Knight) to the destination(Castle).

What nodes and edges in our graph represent?

Nodes : the nodes represent the positions on the board which are written as Coordinates object (x,y) where each coordinate corresponds to a possible knight position that it can move to.

Edges : each edge represents a possible knight move from the current node to the next node that a knight can visit. That node has to pass the conditions in the “IsSafe()” function in order to be considered as the valid next node.

In **IsSafe()** function, we checked each of the possible knight position’s coordinates X and Y that

1. if X or Y coordinate is below 0, that means either X or Y coordinate is out of the board
2. if X or Y coordinate exceeds the size of the board, that means either X or Y coordinate is also out of the board
3. If we have already visited that position (`if(visited[x][y]==1)`), then we cannot visit that position again.
4. Lastly, if the position is the same position as one of the bombs, then it cannot visit that position.

If that possible knight position’s coordinates satisfies one of these 4 conditions, then we cannot add the edge. Otherwise, the edge between the current knight position and the possible knight position can be added.

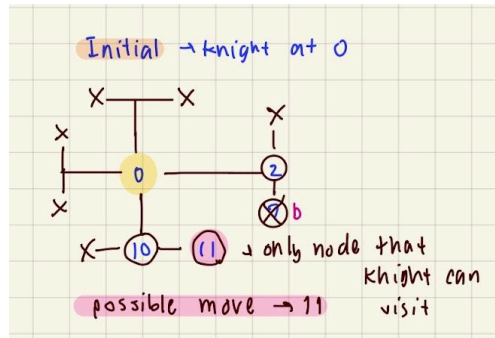
Why not use jgrapht/Explicit graph?

- We don’t store all nodes and edges of the graph, which saves memory, especially for larger boards.
- For implicit graphs, only the necessary nodes are explored, avoiding unnecessary access.
- Does not need to import an external library

Graph from Demo 2

Board size = 5, Knight ID = 0, Castle ID = 24, bomb IDs = 0,24,7,4,14,23

1. Initial -> Knight at 0

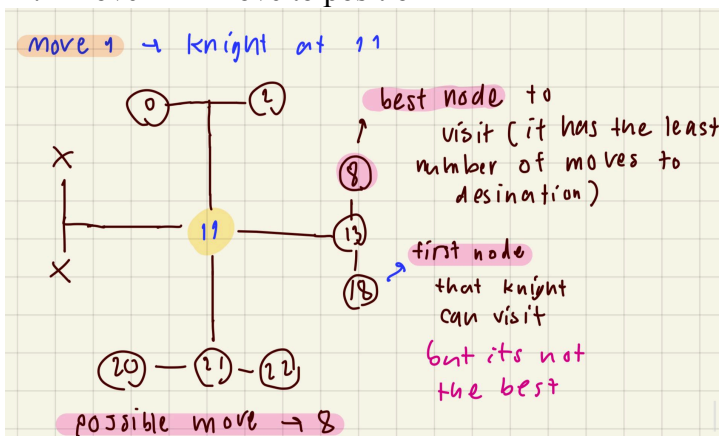


Since bomb IDs 0 and 24 have already been used, they are considered as invalid. So we cannot add ID 0 and 24 as bombs.

First, we start at the initial KnightID position, which is 0. This position is added to the queue, and it is marked as visited. (Position 0 is visited). From position 0, the program checks all possible knight moves and 6 of them are out of bounds, so it cannot visit these nodes. 1 of the possible moves (move to ID9) is blocked by a bomb, so it cannot visit that node either.

Therefore, the only position that the current knight can visit is node ID 11. The edge is now constructed from node 0 to node 11.

2. Move 1 -> Move to position 11

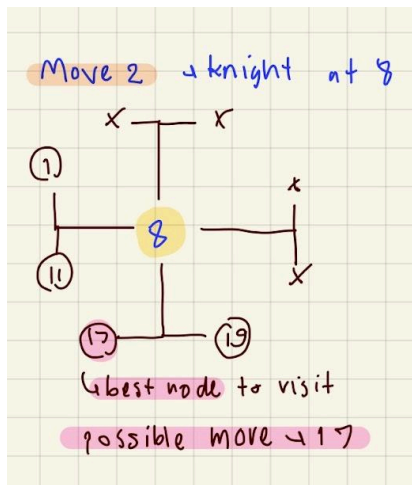


Now, node ID 11 is at the start of the queue, so it gets popped off the queue in order for the program to calculate possible knight moves from position 11. It has 3 positions (0,2,8,18,20,22) that pass the conditions in the IsSafe() function. So all of them get added into the queue. From our program, it adds 18 first and then adds in clockwise direction (18 - 22 - 20 - 0 - 2 - 8). 18 is now at the front of the queue. The BFS algorithm will process 18 first, then 22, 20, 0, 2 and then 8. When it processes each possible knight move, it checks all 8 of its possible moves (just like it did with ID 0), and if any of them are valid and safe, they will be added to the

queue as well. (It adds to the queue level by level and to process any node's possible moves, that node will get popped out of the queue and the possible moves will be added to the queue).

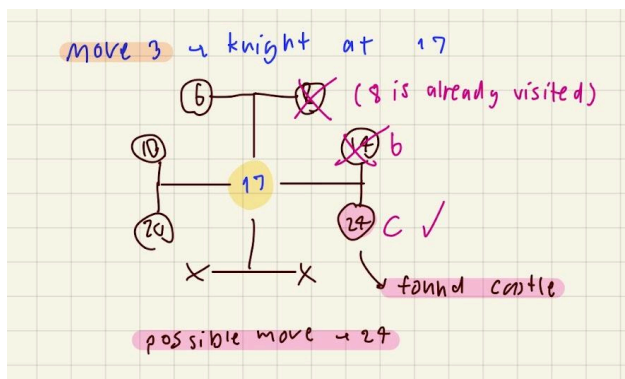
Once the program has finished processing all these nodes, it finds out that 8 has the shortest path from current knight position (11) to destination (Castle : ID24). Therefore, the edge is constructed from node 11 to node 8.

3. Move 2 -> Move to position 8



As I have explained earlier, the program will calculate possible knight moves from position 8. Since the possible knight moves that satisfy the conditions from IsSafe() are more than 1 (19, 17, 11, 1), the algorithm will process all these possible knight moves level by level until any of them reaches the castle. It finds out that if the current knight moves to position 17, the number of paths will be the least compared to when visiting other nodes. Therefore, the edge is constructed between node 9 and node 17.

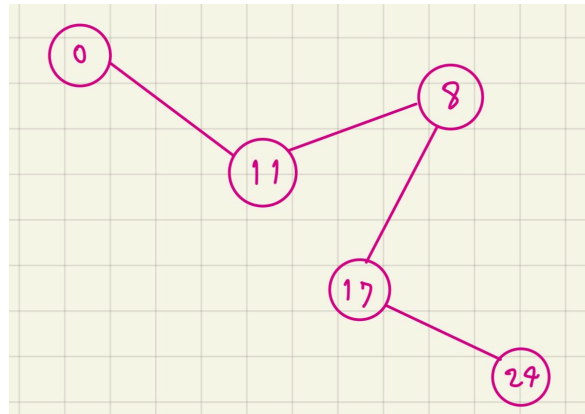
4. Move 8 -> Move to position 17



The program cannot move to position 8 since it's already visited (Current “visited” list is 0,11,8,17). Node 14 cannot be visited since the bomb is located at position 14. Nodes 20,10,6 and 24 are processed since they all passed the conditions from IsSafe(). Then, the BFS algorithm

finds out that moving to node 24 will have the shortest path to the destination. Since node 24 is the destination itself, we can construct the last edge from node 17 to node 24.

Therefore, the shortest path from position 0 to 24 with bomb IDs = 0,24,7,4,14,23 is
(0-11-8-17-24)



Other data structures

This section describes the program's key data structures, including their roles, implementation details, and the values they store.

1. ArrayList

The name of Objects that have been used in ArrayList is “BombsPosition” and “BombsID”.

Which is Located in:

- Class: Bombs

```
public class Bombs {
    private ArrayList<Coordinate> BombsPosition;
    private ArrayList<Integer> BombsID;
```

Add a bomb ID and a bomb position to the list.

```
BombsID.add(id);
BombsPosition.add(new Coordinate(x, y));
```

Using an ArrayList is advantageous because it can be dynamically resized, which is important. After all, the number of bombs is determined by user input. It also enables efficient iteration and random access, which are required to process bomb data during pathfinding. BombsPosition has been used to store the coordinates (x, y) of all bombs placed on the board and BombsID has been used to store the unique cell IDs of all bomb positions.


2. HashMap

The name of Objects that have been used in Hashmap is “parent”.

Which is Located in:

- Class: Board specifically within the bfs() method.

```
public void bfs() {
    ArrayDeque<Coordinate> q = new ArrayDeque<>();
    HashMap<Coordinate, Coordinate> parent = new LinkedHashMap<>();
```



Adds parent-child relationship for BFS traversal.

```
parent.put(neighborKnightPosition, current);
```

The HashMap provides constant time complexity for insertion and lookup, which is essential for efficiently tracking parent-child relationships during the BFS traversal. This makes it suitable for mapping each board position (key) to the position from which it was reached (value) during the traversal. The HashMap stores coordinate objects as keys, representing visited positions on the board and values, representing the parent positions from which the current position was reached.

3. ArrayDeque

The name of Objects that have been used in ArrayDeque is “q”.

Which is Located in:

- Class: Board within the bfs() method.

```
public void bfs() {
    ArrayDeque<Coordinate> q = new ArrayDeque<>();
```

Adds initial knight position to the queue.

```
q.add(knightPosition);
```

Adds neighboring positions to the queue during BFS.

```
q.add(neighborKnightPosition);
```

ArrayDeque offers efficient add and remove operations at both ends, making it ideal for implementing a queue. Its performance characteristics align well with the requirements of BFS traversal. The ArrayDeque stores coordinate objects representing the current positions of the Knight during the BFS traversal.

Graph algorithm

Explanation of the graph algorithm that we use with an example of finding at least 1 solution.

In this project, we are using BFS, a.k.a breadth first search algorithm for finding the shortest path. Because we are using a simple undirected unweighted graph. If we want to find the best route to the castle, we need to find path length between vertices.

Let's go to one of the solutions that we will show. Let's say knightID is 0. castleId is 24. And there's a bomb at 14. So we convert both positions to coordinates (0,0),(4,4) and (4,2) respectively.

```
public void bfs() {
    ArrayDeque<Coordinate> q = new ArrayDeque<>();
    HashMap<Coordinate, Coordinate> parent = new LinkedHashMap<>();
    Coordinate knightPosition = knight.getKnightPosition();
    q.add(knightPosition);
    parent.put(knightPosition, null);
    int[][] visited = new int[size][size];
    visited[knightPosition.getX()][knightPosition.getY()] = 1;
    int[] knightXMoves = knight.getKnightXMoves();
    int[] knightYMoves = knight.getKnightYMoves();
}
```

In our BFS algorithm, we created a queue using ArrayDeque. We use HashMap to collect the answer. Then, we get the knight coordinate which is 0,0, add it to the queue, and add the coordinate to the Hashmap, which keeps the current position as key, parent as value which is null in this case. then we initialize the visited 2D array for keeping track of the coordination we have visited. Then make the knight coordinate being visited by initializing its coordinate as 1 in the visited array. And get the array for iterating through knight moves.

```
private int[] KnightXMoves = new int[]{2,2,1,-1,-2,-2,-1,1};
private int[] KnightYMoves = new int[]{1,-1,-2,-2,-1,1,2,2};
```

From these 2 arrays, we have 8 members which collect 8 possible moves for the knight. We will iterate through all possible knight moves; we will move to (2,1) to (2,-1) to (1,-2) to (-1,-2) to (-2,-1) to (-2,1) to (-1,2) to (1,2). Now, it's time to use the queue for BFS traversal.


```

while (!q.isEmpty()) {
    Coordinate current = q.poll();
    if (current.equals(CastlePosition)) {
        break;
    }
    for (int i = 0; i < 8; i++) {
        int x = current.getX() + knightXMoves[i];
        int y = current.getY() + knightYMoves[i];
        Coordinate neighborKnightPosition = new Coordinate(x, y);

```

First, we pop the coordinate of 0,0 from the queue. And we will break from the queue if we are currently considering the location of the castle, which means our knight is at the castle already and there's no need to consider further locations. But this is not the case for now, so we ignore this if condition. And we iterate through 8 possible moves using a for loop. The possible locations would be (2,1),(2,-1),(1,-2),(-1,-2),(-2,-1),(-2,1),(-1,2),(1,2).but before we consider these possibilities, we need constraints in case of the move being in the same coordination as the bombs and being outside the board. Therefore, we use isSafe() function to handle these constraints.

```

public boolean isSafe(int[][]visited,Coordinate neighborKnightPosition)
{
    int x = neighborKnightPosition.getX();
    int y = neighborKnightPosition.getY();
    if(x<0||y<0)return false;
    if(x>=size||y>=size)return false;
    if(visited[x][y]==1)return false;
    ArrayList<Coordinate> bombsList = bombs.getBombsPosition();
    for(Coordinate bombs:bombsList)
    {
        if(bombs.equals(neighborKnightPosition))return false;
    }
    return true;

```

Consequently, only (2,1) and (1,2) which are 7 and 11 respectively. Then we visit all of them by putting 1 in the visited array, add them to the queue, and put their locations as keys with their parent being the current position which is (0,0).

```

    if (isSafe(visited,neighborKnightPosition)) {
        visited[x][y] = 1;
        q.add(neighborKnightPosition);
        parent.put(neighborKnightPosition, current);
    }

```

Afterward, the steps repeated themselves until we reached the castle. from (2,1) or 7, we will consider (4,2),(4,0),(3,-1),(1,-1),(0,0),(0,2),(1,3),(3,3). But only possible moves,according to isSafe(), (4,0),(0,2),(1,3) and (3,3) which are 4,10,16,18 (0 and 14 can't be possible since 0 is

already visited and 14 is the bomb's location). so we visit them all, push them to the queue, and put them in parent HashMap with 7 as their parents.

Now, we move on to (1,2) in the queue, possible moves are 8,2,20,22(0 and 18 aren't possible since we have already visited them). So we visit them, push them to the queue, and add them to HashMap with 11 as their parent.

Now, we move on to (4,0) or 4 and we consider all possible moves from 4 which is (3,2) or 13 (7 is not possible since we visited it), then we do the same procedures.

Then we move on to (0,2) which is 10 and repeat the same process. The possible moves are 17,1, and 21. and do the same procedures with these locations.

For the sake of simplicity, we will assume that we have already explored all of the neighbor positions from 16 and 18, which are possibilities from 7. Now, we will arrive at 13 since it's the first neighbor position from 4. The only possible locations are 6 and 24 (4,2,16 and 22 are visited) so we push them to the queue, and add them to HashMap with 13 being their parent. This is the moment that we arrive at the answer because 24 is the castle position.

However, we need to iterate through 17,1 and other possible locations until the queue reaches the castle position because the condition is in the while loop. The reason we don't exit the loop from the moment we have found the answer is because we want to keep the simplicity of the code without using labels or other complicated functions.

Now we will initialize the array list that will collect the best route, and use the last node or the castle position as key to access its parent or the location that the knight came from before reaching the castle.

```
ArrayList<Coordinate> path = new ArrayList<>();
Coordinate node = CastlePosition;
```

Then, we ask the answer that we have collected whether it contains the castle position or not. If not, then it means that we have done BFS traversal with all of the possible moves generated from the starting position and did not find a single route to the castle.

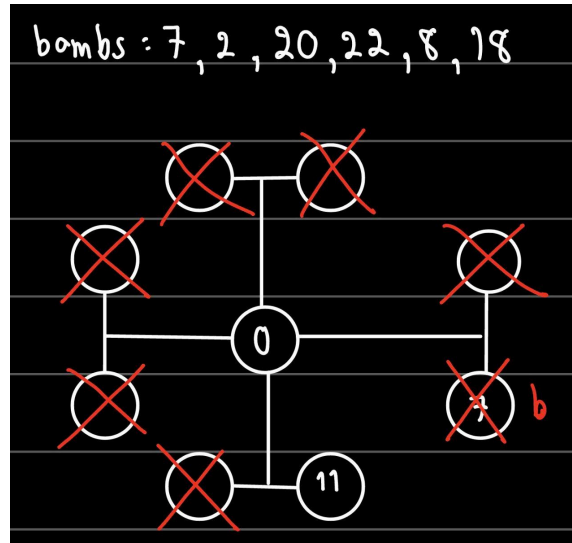
```
int countRoute = -1;
if (parent.containsKey(CastlePosition)) {
    while (node != null) {
        path.add(node);
        node = parent.get(node);
        countRoute++;
    }
    Collections.reverse(path);

    System.out.printf("\nBest route to Castle = %d %s\n", countRoute, (countRoute>1)? "moves": "move");
    int count = 0;
    for (Coordinate coord: path)
    {
        int id = coord.getY()*size+coord.getX();
        if (count==0)
        {
            count++;
            continue;
        }
        else{
            System.out.printf("Move %d --> jump to %d\n", count, id);
            int knightID = coord.getY() * size + coord.getX();
            int castleID = CastlePosition.getY() * size + CastlePosition.getX();
            printAns(knightID, castleID, bombs.getBombsID());
            count++;
        }
    }
} else {
    System.out.println("\nNo route from Knight to Castle");
}
```

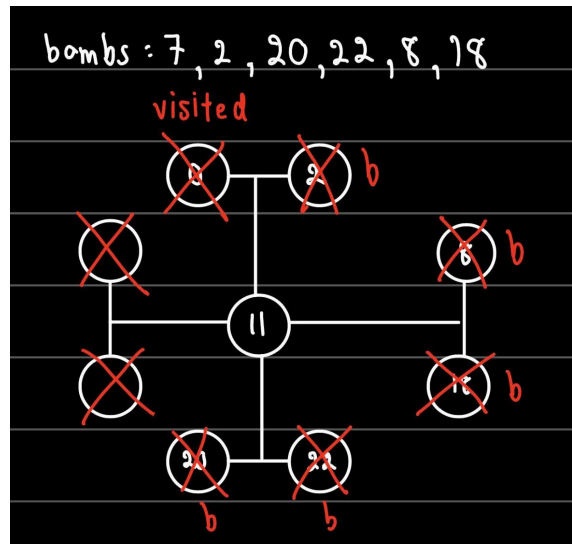
First, we add the castle position to the ArrayList that collects our path and traverses to its previous location by initializing the node as the parent. Recalling that we have put the starting position's parent as null. Therefore, we will traverse backward until we reach the starting position, so that our path will be complete, and while doing so, keep count of the numbers of moves we have made along the way. Afterward, we reverse the path using reverse() function. Once, we have best route with every coordinates, we loop through all of them, skip the first node since we have already printed its board position, and print out the answer in terms of id using a simple mathematical expressions of $y * size + x$. And we're done.

Explanation for Demo 3 and what is wrong with the graph.

Knight ID: 0, Castle ID: 24, Bombs ID: 7,2,20,22,8,18.



Firstly, we start from the knight ID position and iterate through all of the 8 possible moves, and quickly realize that only 11 passes the constraints since 7 is the one of the bombs' positions and other positions are out of the board. So we visit 11, put it in the queue, and keep it in HashMap with 0 being the parent.



Now, we will use BFS traversal to travel to 11 and iterate through all of the 8 possible moves again. However, in this case, there's not a single position that we can put into the queue because they have already visited them, are the bombs' location or out of the board. Therefore, the queue becomes empty and we break out of the loop. And since we did not keep the castle position in the HashMap, we can conclude that there's not the best route to the castle.

Limitations

1. Size of $N \times N$ grid

There should be a limit to the $N \times N$ size input.

Size shouldn't be more than 31 because it will cause numbers to overlap and when running a large $N \times N$ input question some slow computer might not be able to handle the program and crash. However the program functions properly even with large input.

size: 30

```

Enter N for N*N board (N must be at least 5)
30
Cell IDs
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29
30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59
60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89
90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119
120 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144 145 146 147 148 149
150 151 152 153 154 155 156 157 158 159 160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177 178 179
180 181 182 183 184 185 186 187 188 189 190 191 192 193 194 195 196 197 198 199 200 201 202 203 204 205 206 207 208 209
210 211 212 213 214 215 216 217 218 219 220 221 222 223 224 225 226 227 228 229 230 231 232 233 234 235 236 237 238 239
240 241 242 243 244 245 246 247 248 249 250 251 252 253 254 255 256 257 258 259 260 261 262 263 264 265 266 267 268 269
270 271 272 273 274 275 276 277 278 279 280 281 282 283 284 285 286 287 288 289 290 291 292 293 294 295 296 297 298 299
300 301 302 303 304 305 306 307 308 309 310 311 312 313 314 315 316 317 318 319 320 321 322 323 324 325 326 327 328 329
330 331 332 333 334 335 336 337 338 339 340 341 342 343 344 345 346 347 348 349 350 351 352 353 354 355 356 357 358 359
360 361 362 363 364 365 366 367 368 369 370 371 372 373 374 375 376 377 378 379 380 381 382 383 384 385 386 387 388 389
390 391 392 393 394 395 396 397 398 399 400 401 402 403 404 405 406 407 408 409 410 411 412 413 414 415 416 417 418 419
420 421 422 423 424 425 426 427 428 429 430 431 432 433 434 435 436 437 438 439 440 441 442 443 444 445 446 447 448 449
450 451 452 453 454 455 456 457 458 459 460 461 462 463 464 465 466 467 468 469 470 471 472 473 474 475 476 477 478 479
480 481 482 483 484 485 486 487 488 489 490 491 492 493 494 495 496 497 498 499 500 501 502 503 504 505 506 507 508 509
510 511 512 513 514 515 516 517 518 519 520 521 522 523 524 525 526 527 528 529 530 531 532 533 534 535 536 537 538 539
540 541 542 543 544 545 546 547 548 549 550 551 552 553 554 555 556 557 558 559 560 561 562 563 564 565 566 567 568 569
570 571 572 573 574 575 576 577 578 579 580 581 582 583 584 585 586 587 588 589 590 591 592 593 594 595 596 597 598 599
600 601 602 603 604 605 606 607 608 609 610 611 612 613 614 615 616 617 618 619 620 621 622 623 624 625 626 627 628 629
630 631 632 633 634 635 636 637 638 639 640 641 642 643 644 645 646 647 648 649 650 651 652 653 654 655 656 657 658 659
660 661 662 663 664 665 666 667 668 669 670 671 672 673 674 675 676 677 678 679 680 681 682 683 684 685 686 687 688 689
690 691 692 693 694 695 696 697 698 699 700 701 702 703 704 705 706 707 708 709 710 711 712 713 714 715 716 717 718 719
720 721 722 723 724 725 726 727 728 729 730 731 732 733 734 735 736 737 738 739 740 741 742 743 744 745 746 747 748 749
750 751 752 753 754 755 756 757 758 759 760 761 762 763 764 765 766 767 768 769 770 771 772 773 774 775 776 777 778 779
780 781 782 783 784 785 786 787 788 789 790 791 792 793 794 795 796 797 798 799 800 801 802 803 804 805 806 807 808 809
810 811 812 813 814 815 816 817 818 819 820 821 822 823 824 825 826 827 828 829 830 831 832 833 834 835 836 837 838 839
840 841 842 843 844 845 846 847 848 849 850 851 852 853 854 855 856 857 858 859 860 861 862 863 864 865 866 867 868 869
870 871 872 873 874 875 876 877 878 879 880 881 882 883 884 885 886 887 888 889 890 891 892 893 894 895 896 897 898 899
Enter Knight ID

```

size:32

```

Enter N for N*N board (N must be at least 5)
32
Cell IDs
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63
64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95
96 97 98 99 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127
128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159
160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177 178 179 180 181 182 183 184 185 186 187 188 189 190 191
192 193 194 195 196 197 198 199 200 201 202 203 204 205 206 207 208 209 210 211 212 213 214 215 216 217 218 219 220 221 222 223
224 225 226 227 228 229 230 231 232 233 234 235 236 237 238 239 240 241 242 243 244 245 246 247 248 249 250 251 252 253 254 255
256 257 258 259 260 261 262 263 264 265 266 267 268 269 270 271 272 273 274 275 276 277 278 279 280 281 282 283 284 285 286 287
288 289 290 291 292 293 294 295 296 297 298 299 300 301 302 303 304 305 306 307 308 309 310 311 312 313 314 315 316 317 318 319
320 321 322 323 324 325 326 327 328 329 330 331 332 333 334 335 336 337 338 339 340 341 342 343 344 345 346 347 348 349 350 351
352 353 354 355 356 357 358 359 360 361 362 363 364 365 366 367 368 369 370 371 372 373 374 375 376 377 378 379 380 381 382 383
384 385 386 387 388 389 390 391 392 393 394 395 396 397 398 399 400 401 402 403 404 405 406 407 408 409 410 411 412 413 414 415
416 417 418 419 420 421 422 423 424 425 426 427 428 429 430 431 432 433 434 435 436 437 438 439 440 441 442 443 444 445 446 447
448 449 450 451 452 453 454 455 456 457 458 459 460 461 462 463 464 465 466 467 468 469 470 471 472 473 474 475 476 477 478 479
480 481 482 483 484 485 486 487 488 489 490 491 492 493 494 495 496 497 498 499 500 501 502 503 504 505 506 507 508 509 510 511
512 513 514 515 516 517 518 519 520 521 522 523 524 525 526 527 528 529 530 531 532 533 534 535 536 537 538 539 540 541 542 543
544 545 546 547 548 549 550 551 552 553 554 555 556 557 558 559 560 561 562 563 564 565 566 567 568 569 570 571 572 573 574 575
576 577 578 579 580 581 582 583 584 585 586 587 588 589 590 591 592 593 594 595 596 597 598 599 600 601 602 603 604 605 606 607
608 609 610 611 612 613 614 615 616 617 618 619 620 621 622 623 624 625 626 627 628 629 630 631 632 633 634 635 636 637 638 639
640 641 642 643 644 645 646 647 648 649 650 651 652 653 654 655 656 657 658 659 660 661 662 663 664 665 666 667 668 669 670 671
672 673 674 675 676 677 678 679 680 681 682 683 684 685 686 687 688 689 690 691 692 693 694 695 696 697 698 699 700 701 702 703
704 705 706 707 708 709 710 711 712 713 714 715 716 717 718 719 720 721 722 723 724 725 726 727 728 729 730 731 732 733 734 735
736 737 738 739 740 741 742 743 744 745 746 747 748 749 750 751 752 753 754 755 756 757 758 759 760 761 762 763 764 765 766 767
768 769 770 771 772 773 774 775 776 777 778 779 780 781 782 783 784 785 786 787 788 789 790 791 792 793 794 795 796 797 798 799
800 801 802 803 804 805 806 807 808 809 810 811 812 813 814 815 816 817 818 819 820 821 822 823 824 825 826 827 828 829 830 831
832 833 834 835 836 837 838 839 840 841 842 843 844 845 846 847 848 849 850 851 852 853 854 855 856 857 858 859 860 861 862 863
864 865 866 867 868 869 870 871 872 873 874 875 876 877 878 879 880 881 882 883 884 885 886 887 888 889 890 891 892 893 894 895
896 897 898 899 900 901 902 903 904 905 906 907 908 909 910 911 912 913 914 915 916 917 918 919 920 921 922 923 924 925 926 927
928 929 930 931 932 933 934 935 936 937 938 939 940 941 942 943 944 945 946 947 948 949 950 951 952 953 954 955 956 957 958 959
960 961 962 963 964 965 966 967 968 969 970 971 972 973 974 975 976 977 978 979 980 981 982 983 984 985 986 987 988 989 990 991
992 993 994 995 996 997 998 999 1000 1001 1002 1003 1004 1005 1006 1007 1008 1009 1010 1011 1012 1013 1014 1015 1016 1017 1018 1019 1020 1021 1022 1023
Enter Knight ID

```

- Fixed chessboard size of $N \times N$. if the board size is suddenly unequal then the solution may be unfindable.
- By computing the neighbor knight positions, we can save some memory. However, it may take longer to compute large boards or many queries since we have to recompute the neighboring knight positions every time.
- There is also a limit for performance to the input $N \times N$. Once inputting more than 1000, then the output and computing time will be too large and long, making the performance suboptimal.

References

<https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/>