



Task 6: Six knights

Name	ID
Youssef Wael Hamdy Ibrahim	2001430
Ahmad Youssef Mansour Mahfouz	2002238

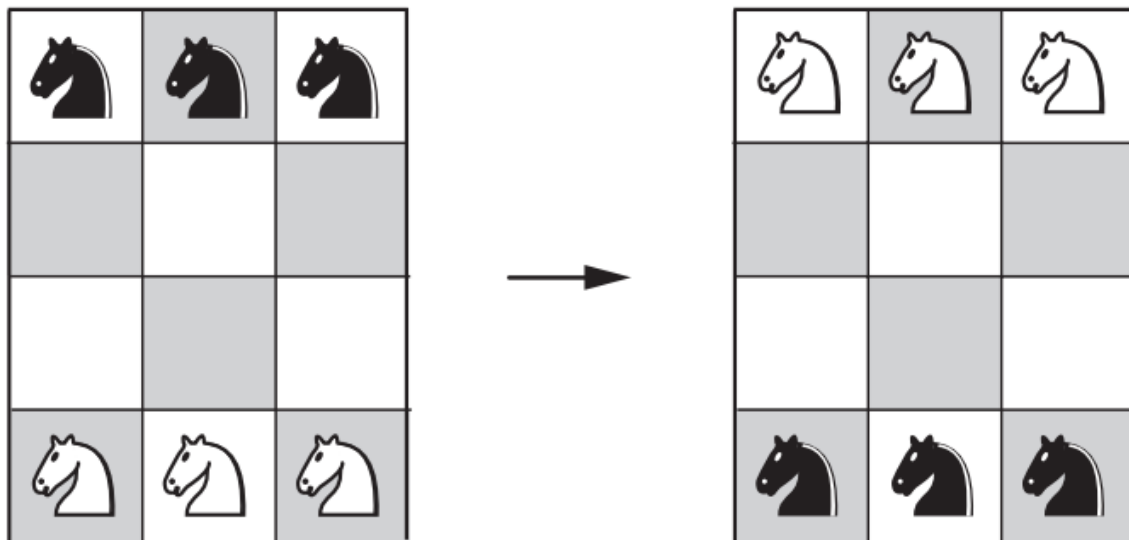
Table of Contents

1- Problem description:	2
2- Detailed assumptions:	2
3- Detailed solution including the pseudo-code and the description of your solution:	2
Detailed solution:	2
Pseudo-code: Link.....	3
4- complexity analysis for algorithm:	4
5- Code:	5
Link.....	5
6- Sample output:	6
7- comparison with another algorithm:	7
8- conclusion:.....	7
9- References:	7

1- Problem description:

There are six knights on a 3×4 chessboard: the three white knights are at the bottom row, and the three black knights are at the top row.

Design an iterative improvement algorithm to exchange the knights to get the position shown on the right of the figure in the minimum number of knights moves, not allowing more than one knight on a square at any time.



2- Detailed assumptions:

- 1) A chessboard of 4×3 is assumed.
- 2) Knights must follow an L-shaped pattern.
- 3) A knight can't go outside the board boundaries.
- 4) Each Knight placement in a board is a State.
- 5) Best first search is implemented using heuristics.
- 6) The heuristic is implemented using Manhattan distance.
- 7) Once in a path you can't visit the same state twice.

3- Detailed solution including the pseudo-code and the description of your solution:

Detailed solution:

- 1) An initial state is passed in the code.
- 2) The code starts by Generating moves for knights starting with the white knights.
- 3) Each state is stored inside a path and each path is stored inside priority queue that is sorted based on heuristic value.
- 4) The heuristic value of the path is the heuristic value of the last state of this path.
- 5) The top of the queue (the one with the lowest heuristic value) is hypothetically the closest board arrangement to our solution.
- 6) If our board arrangement is found its Heuristic will equal -1

Pseudo-code: [Link](#)

```
FUNCTION calculateHeuristic(board)
// Inputs: Board is the 2D vector representing the game board
// Output: Returns the heuristic value calculated based on the board state
heuristic <- 0

// Define goal positions for white and black knights
whiteGoalPositions <- {{0, 0}, {0, 1}, {0, 2}}
blackGoalPositions <- {{3, 0}, {3, 1}, {3, 2}}

FOR i FROM 0 TO ROWS - 1 DO
  FOR j FROM 0 TO COLS - 1 DO
    IF board[i][j] == WHITE THEN
      minDist <- INT_MAX
      FOR EACH goalPos IN whiteGoalPositions DO
        dist <- ABS(i - goalPos.first) + ABS(j - goalPos.second)
        minDist <- MIN(minDist, dist)
      END FOR
      heuristic += minDist
    ELSE IF board[i][j] == BLACK THEN
      minDist <- INT_MAX
      FOR EACH goalPos IN blackGoalPositions DO
        dist <- ABS(i - goalPos.first) + ABS(j - goalPos.second)
        minDist <- MIN(minDist, dist)
      END FOR
      heuristic += minDist
    END IF
  END FOR
END FOR

IF isGoalState(board) THEN
  RETURN -1
END IF

RETURN heuristic / 3.0 + 3
END FUNCTION

FUNCTION isGoalState(board)
// Inputs: board is the 2D vector representing the game board
// Output: Returns boolean true if the board represents a goal state
RETURN (board[0][0] == WHITE AND board[0][1] == WHITE AND board[0][2] == WHITE) AND (board[3][0] == BLACK AND board[3][1] == BLACK AND board[3][2] == BLACK)
END FUNCTION

FUNCTION getKnightMoves(x, y, board)
// Inputs: x is the X coordinate of the knight, y is the Y coordinate of the knight, board is the 2D vector representing the game board
// Output: Returns a list of valid knight moves from the given position
moves <- {}

directions <- {{1, 2}, {-2, 1}, {-2, -1}, {-1, -2}, {1, -2}, {2, -1}, {2, 1}, {-1, 2}}

FOR EACH dir IN directions DO
  newX <- x + dir.first
  newY <- y + dir.second
  IF isEmptyAndWithinLimits(board, newX, newY) THEN
    moves.push({newX, newY})
  END IF
END FOR

RETURN moves
END FUNCTION

FUNCTION generateNextState(currentState, pq, currentPath)
// Inputs: currentState is the current state of the game, pq is the priority queue of paths, currentPath is the current path
// Output: Generates and pushes next path to the priority queue
FOR i FROM 0 TO ROWS - 1 DO
  FOR j FROM 0 TO COLS - 1 DO
    IF currentState.board[i][j] != 0 THEN
      moves <- getKnightMoves(i, j, currentState.board)
      FOR EACH move IN moves DO
        nextBoard <- COPY currentState.board
        SWAP nextBoard[i][j] AND nextBoard[move.first][move.second]
        nextHeuristic <- calculateHeuristic(nextBoard)
        nextState <- State(nextBoard, nextHeuristic)
        IF NOT currentPath.hasDupeStates(nextState) THEN
          currentPath.push(nextState)
          pq.push(currentPath)
        END IF
      END FOR
    END IF
  END FOR
END FOR
END FUNCTION

// Performs Best-First Search to find the solution
FUNCTION bestFirstSearch(initialState)
// Inputs: initialState is the initial state of the game
// Output: Prints Path leading to solution if found
pq <- Priority Queue of Paths
initialHeuristic <- calculateHeuristic(initialState)
beginningState <- State(initialState, initialHeuristic)
initialPath <- Path(beginningState)
pq.push(initialPath)

WHILE NOT pq.empty() DO
  currentPath <- pq.top()
  currentState <- currentPath.path[currentPath.path.size() - 1]
  pq.pop()

  IF isGoalState(currentState.board) OR currentState.heuristic == -1 THEN
    PRINT "\n\nSolution found:"
    printBoard(currentState.board)
    PRINT "Path: "
    currentPath.printPath()
    PRINT "\nTotal moves: " + currentPath.length() - 1
    PRINT "\nTotal paths: " + pq.size()
    RETURN
  END IF

  IF currentPath.length() <= MAX_MOVES THEN
    generateNextState(currentState, pq, currentPath)
  END IF
END WHILE

PRINT "No solution found."
END FUNCTION

FUNCTION withinLimits(x, y)
// Inputs: x is the X coordinate of the knight, y is the Y coordinate of the knight
// Output: Returns boolean true if the coordinates are within the board limits
RETURN (x >= 0 AND y >= 0) AND (x < ROWS AND y < COLS)
END FUNCTION

FUNCTION isEmptyAndWithinLimits(board, x, y)
// Inputs: board is the 2D vector representing the game board, x is the X coordinate of the knight, y is the Y coordinate of the knight
// Output: Returns boolean true if the knight movement is within the board and the target square is empty
RETURN withinLimits(x, y) AND board[x][y] == 0
END FUNCTION
```

4- complexity analysis for algorithm:

Time Complexity: $O(N * \log(N))$.

Space complexity: $O(N)$.

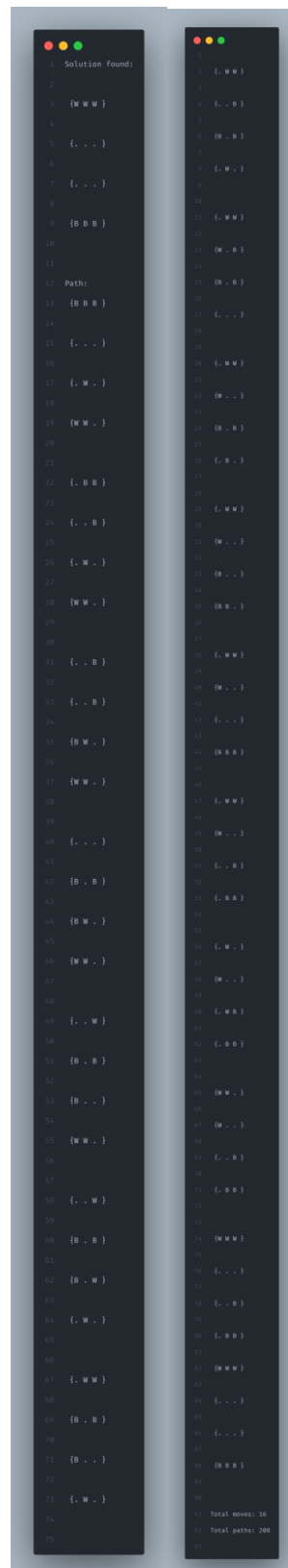
Where N is the number of paths in priority queue.

Link

```

1  #include <chrono>
2  #include <vector>
3  #include <unordered_map>
4  #include <algorithm>
5  #include <deque>
6  #include <queue>
7  #include <string>
8  #include <string_view>
9  #include <chrono>
10 using namespace std;
11
12 // Board size
13 constexpr int COLS = 3;
14 constexpr int ROWS = 3;
15 constexpr int BLACK = 1;
16 constexpr int WHITE = 2;
17 constexpr int MAX_MOVES = 16;
18
19 void printBoard(const vector<vector<int>>& board) {
20     // Print the board
21     for (int i = 0; i < ROWS; i++) {
22         for (int j = 0; j < COLS; j++) {
23             cout << board[i][j] << " ";
24             if (j < COLS - 1) cout << " ";
25         }
26         cout << endl;
27     }
28 }
29
30 int calculateManhattanDistance(const vector<vector<int>>& board) {
31     // Calculate the Manhattan distance between the current state and the goal state
32     int distance = 0;
33     for (int i = 0; i < ROWS; i++) {
34         for (int j = 0; j < COLS; j++) {
35             if (board[i][j] != 0) {
36                 // Find the goal position for this piece
37                 int goalRow, goalCol;
38                 for (int k = 0; k < ROWS; k++) {
39                     for (int l = 0; l < COLS; l++) {
40                         if (board[k][l] == board[i][j]) {
41                             goalRow = k;
42                             goalCol = l;
43                         }
44                     }
45                 }
46                 // Calculate the Manhattan distance
47                 distance += abs(i - goalRow) + abs(j - goalCol);
48             }
49         }
50     }
51     return distance;
52 }
53
54 struct VectorVectorHash {
55     // Hash function for vector<vector<int>>
56     size_t operator()(const vector<vector<int>>& vec) const {
57         size_t hash = 0;
58         for (const auto &vec : vec) {
59             for (int i = 0; i < vec.size(); i++) {
60                 // Combine hash with the hash of each element
61                 hash ^= std::hash<int>()(vec[i]) + 0x9e3779b9 + (hash << 6) + (hash >> 2);
62             }
63         }
64         return hash;
65     }
66 };
67
68 class State {
69 public:
70     vector<vector<int>> board;
71     int move = 0; // Count of moves made
72     int heuristic; // Heuristic value for best-first search
73     // Constructor
74     State(vector<vector<int>>& _board, int _heuristic, int _boardHash, int _heuristicHash) {
75         board = _board;
76         move = _move;
77         heuristic = _heuristic;
78         boardHash = _boardHash;
79         heuristicHash = _heuristicHash;
80     }
81     // Overload comparison operator for priority queue
82     bool operator<(const State &other) const {
83         return heuristic < other.heuristic;
84     }
85     // Board hash function
86     bool isSameBoard(const State &other) const {
87         for (int i = 0; i < ROWS; i++) {
88             for (int j = 0; j < COLS; j++) {
89                 if (board[i][j] != other.board[i][j]) {
90                     return false;
91                 }
92             }
93         }
94         return true;
95     }
96 };
97
98 // Path class
99 class Path {
100 public:
101     vector<State> path;
102     // Constructor
103     Path(State s) {
104         path.push_back(s);
105     }
106     // Add a state to the path
107     void push(State s) {
108         path.push_back(s);
109     }
110     // Get the path
111     const vector<State> &path() const {
112         return path;
113     }
114     // Get the size of the path
115     int size() const {
116         return path.size();
117     }
118     // Get the last state of the path
119     State &last() {
120         return path.back();
121     }
122     // Get the first state of the path
123     State &first() {
124         return path.front();
125     }
126     // Get the state at index i
127     State &at(int i) {
128         return path[i];
129     }
130     // Get the state at index i-1
131     State &at(int i-1) {
132         return path[i-1];
133     }
134     // Get the state at index i+1
135     State &at(int i+1) {
136         return path[i+1];
137     }
138     // Get the state at index i-2
139     State &at(int i-2) {
140         return path[i-2];
141     }
142     // Get the state at index i+2
143     State &at(int i+2) {
144         return path[i+2];
145     }
146     // Get the state at index i-3
147     State &at(int i-3) {
148         return path[i-3];
149     }
150     // Get the state at index i+3
151     State &at(int i+3) {
152         return path[i+3];
153     }
154     // Get the state at index i-4
155     State &at(int i-4) {
156         return path[i-4];
157     }
158     // Get the state at index i+4
159     State &at(int i+4) {
160         return path[i+4];
161     }
162     // Get the state at index i-5
163     State &at(int i-5) {
164         return path[i-5];
165     }
166     // Get the state at index i+5
167     State &at(int i+5) {
168         return path[i+5];
169     }
170     // Get the state at index i-6
171     State &at(int i-6) {
172         return path[i-6];
173     }
174     // Get the state at index i+6
175     State &at(int i+6) {
176         return path[i+6];
177     }
178     // Get the state at index i-7
179     State &at(int i-7) {
180         return path[i-7];
181     }
182     // Get the state at index i+7
183     State &at(int i+7) {
184         return path[i+7];
185     }
186     // Get the state at index i-8
187     State &at(int i-8) {
188         return path[i-8];
189     }
190     // Get the state at index i+8
191     State &at(int i+8) {
192         return path[i+8];
193     }
194     // Get the state at index i-9
195     State &at(int i-9) {
196         return path[i-9];
197     }
198     // Get the state at index i+9
199     State &at(int i+9) {
200         return path[i+9];
201     }
202     // Get the state at index i-10
203     State &at(int i-10) {
204         return path[i-10];
205     }
206     // Get the state at index i+10
207     State &at(int i+10) {
208         return path[i+10];
209     }
210     // Get the state at index i-11
211     State &at(int i-11) {
212         return path[i-11];
213     }
214     // Get the state at index i+11
215     State &at(int i+11) {
216         return path[i+11];
217     }
218     // Get the state at index i-12
219     State &at(int i-12) {
220         return path[i-12];
221     }
222     // Get the state at index i+12
223     State &at(int i+12) {
224         return path[i+12];
225     }
226     // Get the state at index i-13
227     State &at(int i-13) {
228         return path[i-13];
229     }
230     // Get the state at index i+13
231     State &at(int i+13) {
232         return path[i+13];
233     }
234     // Get the state at index i-14
235     State &at(int i-14) {
236         return path[i-14];
237     }
238     // Get the state at index i+14
239     State &at(int i+14) {
240         return path[i+14];
241     }
242     // Get the state at index i-15
243     State &at(int i-15) {
244         return path[i-15];
245     }
246     // Get the state at index i+15
247     State &at(int i+15) {
248         return path[i+15];
249     }
250     // Get the state at index i-16
251     State &at(int i-16) {
252         return path[i-16];
253     }
254     // Get the state at index i+16
255     State &at(int i+16) {
256         return path[i+16];
257     }
258     // Get the state at index i-17
259     State &at(int i-17) {
260         return path[i-17];
261     }
262     // Get the state at index i+17
263     State &at(int i+17) {
264         return path[i+17];
265     }
266     // Get the state at index i-18
267     State &at(int i-18) {
268         return path[i-18];
269     }
270     // Get the state at index i+18
271     State &at(int i+18) {
272         return path[i+18];
273     }
274     // Get the state at index i-19
275     State &at(int i-19) {
276         return path[i-19];
277     }
278     // Get the state at index i+19
279     State &at(int i+19) {
280         return path[i+19];
281     }
282     // Get the state at index i-20
283     State &at(int i-20) {
284         return path[i-20];
285     }
286     // Get the state at index i+20
287     State &at(int i+20) {
288         return path[i+20];
289     }
290     // Get the state at index i-21
291     State &at(int i-21) {
292         return path[i-21];
293     }
294     // Get the state at index i+21
295     State &at(int i+21) {
296         return path[i+21];
297     }
298     // Get the state at index i-22
299     State &at(int i-22) {
300         return path[i-22];
301     }
302     // Get the state at index i+22
303     State &at(int i+22) {
304         return path[i+22];
305     }
306     // Get the state at index i-23
307     State &at(int i-23) {
308         return path[i-23];
309     }
310     // Get the state at index i+23
311     State &at(int i+23) {
312         return path[i+23];
313     }
314     // Get the state at index i-24
315     State &at(int i-24) {
316         return path[i-24];
317     }
318     // Get the state at index i+24
319     State &at(int i+24) {
320         return path[i+24];
321     }
322     // Get the state at index i-25
323     State &at(int i-25) {
324         return path[i-25];
325     }
326     // Get the state at index i+25
327     State &at(int i+25) {
328         return path[i+25];
329     }
330     // Get the state at index i-26
331     State &at(int i-26) {
332         return path[i-26];
333     }
334     // Get the state at index i+26
335     State &at(int i+26) {
336         return path[i+26];
337     }
338     // Get the state at index i-27
339     State &at(int i-27) {
340         return path[i-27];
341     }
342     // Get the state at index i+27
343     State &at(int i+27) {
344         return path[i+27];
345     }
34
```

6- Sample output:



```
1 Solution found:
2
3 {W W W }
4
5 { . . . }
6
7 { . . . }
8
9 {B B B }
10
11
12 Path:
13 {B B B }
14
15 { . . . }
16
17 { . W . }
18
19 {W W . }
20
21
22 { . B B }
23
24 { . . B }
25
26 { . W . }
27
28 {W W . }
29
30
31 { . . B }
32
33 { . . B }
34
35 {B W . }
36
37 {W W . }
38
39 { . . . }
40
41 {B . B }
42
43 {B W . }
44
45 {W W . }
46
47
48 { . . W }
49
50 {B . B }
51
52 {B . . }
53
54 {W W . }
55
56 { . . W }
57
58 {B . B }
59
60 {B . W }
61
62 { . W . }
63
64 { . W W }
65
66 {B . B }
67
68 {B . . }
69
70 { . W . }
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
```

Each state Generated in turn generates more states each chain of states is a path so this code starts with initial state generates a new state and adds it to this state and it's a path so this path is put into Priority queue where it's sorted based on Heuristic values of each state each new iteration it takes top path of queue and then it generates new states for it and so on till we reach goal.

7- comparison with another algorithm:

	Brute force Approach	Best first search
Advantages	Guarantees to find a solution if one exists	A faster approach than brute force and can work with any size of chessboard
Disadvantages	A huge time complexity	Not guaranteed to find a solution if heuristic is poorly chosen
Time Complexity	8^N where N is the number of knights.	$O(N \cdot \log(N))$ where N is the number of paths in Priority queue

8- conclusion:

Solving the Six knights problem requires one of 2 approaches Brute force, Searching algorithms (best first, Hill climbing, A*).

The brute force method Generates all possible moves combinations till it reaches the goal, but can yield very high time complexity, and isn't feasible for bigger chessboards.

Best first algorithm is better than brute force if heuristic calculations are good and can reach solution a lot faster than brute force approach, this approach is an approximation so it might not get the best path each time and it might encounter convergence problems.

9- References:

- [Best first search.](#)
- [Wiki.](#)
- [Puzzle stack exchange.](#)
- [Princeton university](#) pages 3,4 and 5.
- [Carnegie Mellon University.](#)