Section 1 Implementation

All of these three implementations (kt-depth-first, kt-best-first-hquad, kt-best-first-h) use a 2D vector to mark and store the visited squares, so that a O(1) time complexity can be attained on each node created (determine which square to move to, recursion call etc.). Otherwise, if we use a list to store the Knight tour, without using a global variable, each time the recursion call will generate a new list with up to n^2 elements, and the time-complexity (the real running time) will be a mess given a large n and the nodes-limit sufficiently large.

With this implementation, we can safely set the nodes-limit to 1000000 and n = 200 without worrying the actual running time (even for kt-depth-first implementation, the actual running time is around 10s with n = 200 and nodes-limit 1000000).

Also worth mentioning here is that the third implementation, kt-best-first-h, uses the **Warnsdorff's Rule** as the heuristic. Since it is simple and clear yet very efficient compared to other heuristic. I, myself, as a chess player for many years, I would also conduct a knight tour in this way intuitively, just as what the Warnsdorff's Rule states: “We move the knight so that we always proceed to the square from which the knight will have the *fewest* onward moves.”

Below are the codes:

```
1 #lang racket
2
3   (define (kt-depth-first init-position n nodes-limit)                ;naive depth-first version
4   (define init-pos (cons (first init-position) (last init-position)))
5   (define board (make-array n))                                       ;this board serves as reference (check if a square is visited) and printing
6   (define moves '((1 . 2) (-1 . 2) (1 . -2) (-1 . -2) (2 . 1) (-2 . 1) (2 . -1) (-2 . -1)))
7   (define nodes-created (make-parameter 0))                           ;a counter for nodes-created, used to compare with nodes-limit
8   (define num-of-steps 1)                                             ;indicates the step number, to mark on board
9   (write-chessboard (kt-depth-first-rec init-pos n nodes-limit board nodes-created num-of-steps moves)
10                  n board nodes-created))
11
12 (define (kt-depth-first-rec current-pos n nodes-limit board nodes-created num-of-steps moves)
13   (define message #f)                                                ;used to pass the return value
14   (vector-set! (vector-ref board (car current-pos)) (cdr current-pos) num-of-steps)
15   (nodes-created (add1 (nodes-created)))                             ;increase the counter by one when a new node created
16   (cond [(equal? num-of-steps (* n n)) (set! message #t)]             ;if goal reached
17         [(equal? (nodes-created) nodes-limit) (set! message "NaN")] ;if maxumum reached, and not succeed
18         [else (for/or ([i moves])                                    ;for/or terminates when a first success occur
19             (if (validate-pos (next-pos i current-pos) n board)      ;obtain the list of attainable positions
20               (set! message
21               (kt-depth-first-rec (next-pos i current-pos)
22                                   n nodes-limit board nodes-created (+ 1 num-of-steps) moves)) #f) message)
23                                   (when (equal? message #f)          ;if message indicates a failure, leave this node
24                                   (vector-set! (vector-ref board (car current-pos)) (cdr current-pos) 0))]
25                                   ) message)                         ;use message as the return value
26
27 (define (next-pos move current-pos)
28   (cons (+ (car move) (car current-pos)) (+ (cdr move) (cdr current-pos))))
29
30 (define (validate-pos pos n board)
31   (if (or  (or (< (car pos) 0)                                       ;use 'or' to reduce comparisions
32            (>= (cdr pos) n) (>= (car pos) n)
33            (< (cdr pos) 0))
34            (not (equal? 0 (vector-ref (vector-ref board (car pos)) (cdr pos)))))
35            #f #t))
36
37 (define (write-chessboard message n board nodes-created)
```

```
38  (define count 1)
39  (cond [(equal? message #f) (display "No knight's tour exists!")]
40        [(equal? message "NaN") (display "Maximum trials reached!")]
41        [else (begin
42              (for ([i board])
43               (if (equal? count 1) (display "((") (display " ("))
44               (display (string-join (for/list ([j (vector->list i)])
45                                       (~a j #:width (count-digits (* n n)) #:align 'right))))
46               (if (equal? count n) (displayln "))") (displayln ")"))
47               (set! count (add1 count)))
48              (display "Nodes created: <")
49              (display (nodes-created))
50              (display ">"))]))
51
52 (define (count-digits a)                                    ;used to obain the maximum width of move number, for printing purpose
53  (if (< a 10) 1 (+ 1 (count-digits (/ a 10)))))
54
55 (define (make-array n)                                      ;initialize an n*n matrix/2D vector with 0
56  (let ((a (make-vector n)))
57    (let loop ((i 0))
58      (when (< i n)
59        (vector-set! a i (make-vector n))
60        (loop (add1 i))))
61        a))
62
63 ;-------------------------kt-best-first-hquad--------------------------;
64
65 (define (kt-best-first-hquad init-position n nodes-limit)            ;this is another implementation with a given heuristic
66  (define init-pos (cons (first init-position) (last init-position)))
67  (define board (make-array n))
68  (define nodes-created (make-parameter 0))
69  (define num-of-steps 1)
70  (write-chessboard (kt-best-first-hquad-rec init-pos n nodes-limit board nodes-created num-of-steps)
71                    n board nodes-created))
72
73 (define (kt-best-first-hquad-rec current-pos n nodes-limit board nodes-created num-of-steps)
74  (define message #f)
75  (vector-set! (vector-ref board (car current-pos)) (cdr current-pos) num-of-steps)
76  (nodes-created (add1 (nodes-created)))                         ;increase the counter by one when a new node created
77  (cond [(equal? num-of-steps (* n n)) (set! message #t)]                ;if goal reached
78        [(equal? (nodes-created) nodes-limit) (set! message "NaN")]      ;if maxumum reached, and not succeed
79        [else (for/or ([i (best-moves-hquad current-pos n board)])        ;different from the previous implementation, since
80                (set! message                                  ;we need to try the next position in heuristic order
81                (kt-best-first-hquad-rec i n nodes-limit board nodes-created (+ 1 num-of-steps))) message)
82                          (when (equal? message #f)
83                          (vector-set! (vector-ref board (car current-pos)) (cdr current-pos) 0))]
84                          ) message)
85
86 (define (best-moves-hquad current-pos n board)                 ;return all the possible-moves at current position
87  (define r (car current-pos))
88  (define c (cdr current-pos))
89  (best-sorted-list-hquad
90        (check-board (get-valid-pos (list
91        (cons (- r 1) (- c 2)) (cons (- r 2) (- c 1))
92        (cons (- r 1) (+ c 2)) (cons (- r 2) (+ c 1))                ;construct the list of 8 candidates
93        (cons (+ r 1) (- c 2)) (cons (+ r 2) (- c 1))                ;and use the check-board to get rid of
```

```
94        (cons (+ r 1) (+ c 2)) (cons (+ r 2) (+ c 1))                    ;visited positions
95        ) n) board) n '()))
96
97 (define (best-sorted-list-hquad positions n result)                     ;obtain a sorted list (according to hquad values) of positions
98   (define maxValue -inf.f)
99   (define maxPos '())
100   (define tmpValue -inf.f)
101   (if (null? positions) result
102       (begin
103         (for ([i positions])                                           ;a naive selection sort for constant number of elements
104         (set! tmpValue (hquad-value i n))
105         (when (> tmpValue maxValue) (begin (set! maxPos i)
106                                            (set! maxValue tmpValue))))
107         (set! result (append (list maxPos) result))
108         (best-sorted-list-hquad (remove maxPos positions) n result))))
109
110 (define (hquad-value position n)
111   (define r (car position))
112   (define c (cdr position))
113   (+ (* (- (- n 1) r) r) (* (- (- n 1) c) c)))                         ;hquad = r((n-1)-r)+c((n-1)-c)
114
115 (define (check-board candilist board)                                  ;return a new list without visited squares
116    (cond [(null? candilist) '()]
117          [(equal? (vector-ref (vector-ref board (car (car candilist))) (cdr (car candilist))) 0)
118           (cons (car candilist) (check-board (cdr candilist) board))]
119          [else (check-board (cdr candilist) board)]
120       ))
121
122 (define (get-valid-pos candi n)
123   (if (null? candi) '()
124       (if (not (or (< (car (car candi)) 0)
125            (>= (cdr (car candi)) n) (>= (car (car candi)) n)
126            (< (cdr (car candi)) 0)))
127            (cons (car candi) (get-valid-pos (cdr candi) n))
128            (get-valid-pos (cdr candi) n))))
129
130 ;---------------------------kt-best-first-h------------------------------;
131
132 (define (kt-best-first-h init-position n nodes-limit)                  ;This implemntation uses Warnsdorff's Rule as heuristic
133   (define init-pos (cons (first init-position) (last init-position)))
134   (define board (make-array n))
135   (define nodes-created (make-parameter 0))
136   (define num-of-steps 1)
137   (write-chessboard (kt-best-first-h-rec init-pos n nodes-limit board nodes-created num-of-steps)
138                 n board nodes-created))
139
140 (define (kt-best-first-h-rec current-pos n nodes-limit board nodes-created num-of-steps)
141   (define message #f)
142   (vector-set! (vector-ref board (car current-pos)) (cdr current-pos) num-of-steps)
143   (nodes-created (add1 (nodes-created)))                               ;increase the counter by one when a new node created
144   (cond [(equal? num-of-steps (* n n)) (set! message #t)]              ;if goal reached
145         [(equal? (nodes-created) nodes-limit) (set! message "NaN")]    ;if maxumum reached, and not succeed
146         [else (for/or ([i (best-moves-h current-pos n board)])         ;try the next position in heuristic order
147               (set! message
148               (kt-best-first-h-rec i n nodes-limit board nodes-created (+ 1 num-of-steps))) message)
149                           (when (equal? message #f)
```

```
150                     (vector-set! (vector-ref board (car current-pos)) (cdr current-pos) 0))]
151                     ) message)
152
153 (define (best-moves-h current-pos n board)                ;return all the possible-moves at current position
154   (define r (car current-pos))                            ;and here "moves" means the next POSITION
155   (define c (cdr current-pos))
156   (best-sorted-list-h
157       (check-board (get-valid-pos (list
158       (cons (- r 1) (- c 2)) (cons (- r 2) (- c 1))
159       (cons (- r 1) (+ c 2)) (cons (- r 2) (+ c 1))       ;construct the list of 8 candidates
160       (cons (+ r 1) (- c 2)) (cons (+ r 2) (- c 1))       ;and use the check-board to get rid of
161       (cons (+ r 1) (+ c 2)) (cons (+ r 2) (+ c 1))       ;visited positions
162       ) n) board) n '() current-pos board))
163
164 (define (best-sorted-list-h positions n result current-pos board)
165   (define maxValue -inf.f)                                 ;obtain a sorted list (according to their onward possible squares)
166   (define maxPos '())                                      ;and this is known as the “Warnsdorff's Rule”
167   (define tmpValue -inf.f)
168   (if (null? positions) result
169     (begin
170       (for ([i positions])
171       (set! tmpValue (h-value i n board))
172       (when (> tmpValue maxValue) (begin (set! maxPos i)
173                                 (set! maxValue tmpValue))))
174       (set! result (append (list maxPos) result))
175       (best-sorted-list-h (remove maxPos positions) n result current-pos board))))
176
177 (define (h-value position n board)                         ;obtain the onward degree of this position
178   (define r (car position))                                ;*notice, this degree is greater than the
179   (define c (cdr position))                                ;actual value by 1, yet does not matter
180   (length (check-board (get-valid-pos (list                ;since every candidate gets same treatment
181       (cons (- r 1) (- c 2)) (cons (- r 2) (- c 1))
182       (cons (- r 1) (+ c 2)) (cons (- r 2) (+ c 1))       ;construct the list of 8 candidates
183       (cons (+ r 1) (- c 2)) (cons (+ r 2) (- c 1))       ;and use the check-board to get rid of
184       (cons (+ r 1) (+ c 2)) (cons (+ r 2) (+ c 1))       ;visited positions
185       ) n) board)))
```

Section 2 Testing

Start from **n = 1** (since 0<=r<n => n>0): (the code will be ommited for next cases)

```
(displayln "kt-depth-first:")
(kt-depth-first '(0 0) 1 1000000)
(display "\n\n")
(displayln "kt-best-first-hquad:")
(kt-best-first-hquad '(0 0) 1 1000000)
(display "\n\n")
(displayln "kt-best-first-h:")
(kt-best-first-h '(0 0) 1 1000000)
```

Result:

kt-depth-first:
((1))
Nodes created: <1>

kt-best-first-hquad:
((1))
Nodes created: <1>

kt-best-first-h:
((1))
Nodes created: <1>

For **n = 2, 3, 4** the results are the same:

kt-depth-first:
No knight's tour exists!

kt-best-first-hquad:
No knight's tour exists!

kt-best-first-h:
No knight's tour exists!

For n>4, we choose n = 5, n = 6 and 8 as examples:

n = 5:(start from (0 0))

kt-depth-first:
((1 14 19 8 25)
 (6 9 2 13 18)
 (15 20 7 24 3)
 (10 5 22 17 12)
 (21 16 11 4 23))
Nodes created: <72509>

kt-best-first-hquad:
((1 18 13 24 7)
 (12 23 8 19 14)
 (17 2 25 6 9)
 (22 11 4 15 20)
 (3 16 21 10 5))
Nodes created: <25>

kt-best-first-h:
((1 12 25 18 7)
 (22 17 8 13 24)
 (11 2 23 6 19)
 (16 21 4 9 14)
 (3 10 15 20 5))
Nodes created: <25>

Notice both of the implementations with heuristic perfomed much more efficiently than the first one, and the results are correct.

N = 6: (start from (2 4))

kt-depth-first:
Maximum trials reached!

kt-best-first-hquad:
((35 4 13 24 33 2)
 (14 27 34 3 12 25)
 (5 36 23 26 1 32)
 (18 15 28 9 22 11)
 (29 6 17 20 31 8)
 (16 19 30 7 10 21))
Nodes created: <313>

kt-best-first-h:
((35 4 13 16 29 2)
 (14 23 34 3 12 17)
 (5 36 15 30 1 28)
 (22 33 24 9 18 11)
 (25 6 31 20 27 8)
 (32 21 26 7 10 19))
Nodes created: <36>

Notice the kt-depth-first fails to find the tour within the nodes-limit (1000000); And this time, the Warnsdorff's rule performed much better than hquad heuristic, though they both found a valid tour.

n = 8:

kt-depth-first:
Maximum trials reached!

kt-best-first-hquad:
((1 38 3 18 35 40 13 16)
 (4 19 36 39 14 17 34 41)
 (37 2 49 60 43 58 15 12)
 (20 5 64 57 62 51 42 33)
 (55 48 61 50 59 44 11 26)
 (6 21 56 63 52 27 32 29)
 (47 54 23 8 45 30 25 10)
 (22 7 46 53 24 9 28 31))
Nodes created: <64>

kt-best-first-h:
((1 64 15 32 61 28 13 30)
 (16 33 62 59 14 31 56 27)
 (63 2 49 44 57 60 29 12)
 (34 17 58 53 50 47 26 55)
 (3 52 45 48 43 54 11 40)
 (18 35 20 51 46 41 8 25)
 (21 4 37 42 23 6 39 10)
 (36 19 22 5 38 9 24 7))
Nodes created: <64>

Some boundary/interesting tests:

(kt-depth-first '(2 2) 7 10000000)
(display "\n\n")

```
(displayln "kt-best-first-hquad:")
(kt-best-first-hquad '(6 3) 7 1000000)
(display "\n\n")
(displayln "kt-best-first-h:")
(kt-best-first-h '(6 6) 7 1000000)
```

Result:

```
kt-depth-first:
Maximum trials reached!

kt-best-first-hquad:
Maximum trials reached!

kt-best-first-h:
((17 14 19 38  7 12  9)
 (20 43 16 13 10 39  6)
 (15 18 45 42 37  8 11)
 (44 21 30 49 40  5 34)
 (29 46 41 36 33  2 25)
 (22 31 48 27 24 35  4)
 (47 28 23 32  3 26  1))
Nodes created: <293576>
```

It seems like for some combinations (n, position), all three implementations will meet some trouble.

Yet it is very easy for them (2nd and 3rd) in this case:

```
(displayln "kt-depth-first:")
(kt-depth-first '(2 2) 8 10000000)
(display "\n\n")
(displayln "kt-best-first-hquad:")
(kt-best-first-hquad '(6 3) 8 1000000)
(display "\n\n")
(displayln "kt-best-first-h:")
(kt-best-first-h '(6 6) 8 1000000)
```

Result:

```
kt-depth-first:
Maximum trials reached!

kt-best-first-hquad:
((26 11 48 55 28 13 32 35)
 (47 54 27 12 49 34 29 14)
 (10 25 46 51 56 31 36 33)
 (53 58 63 60 45 50 15 30)
 (24  9 52 57 64 61 44 37)
 ( 3  6 59 62 43 40 19 16)
 ( 8 23  4  1 18 21 38 41)
 ( 5  2  7 22 39 42 17 20))
Nodes created: <306>

kt-best-first-h:
((43 10 61 64 21 12 23 58)
 (62 55 42 11 60 57 20 13)
```

```
( 9 44 63 56 41 22 59 24)
(54 39 48 45 52 33 14 19)
(49  8 53 40 47 18 25 34)
(38  5 46 51 32 35 28 15)
( 7 50  3 36 17 30  1 26)
( 4 37  6 31  2 27 16 29))
Nodes created: <64>
```

When set nodes-limit to 0, the program fails to provide a result, since knight needs at least visit the square it starts tour from.

For much larger n, we will cover those cases in section 3.

In all, all of the three implementations generate the expected result as specified in the original problem.

Section 3 Analysis

To conduct the analysis efficiently, we implemented a test program which can conduct all the tests automatically; meanwhile, the write-board procedure was modified to only return the nodes-created value for test purpose. The test procedure is as follow:

```
1 (define (kt-test n nodes-limit)
2   (define kt-depth-first-success-rate 0)
3   (define kt-depth-first-average-nodes 0)
4   (define kt-best-first-hquad-success-rate 0)
5   (define kt-best-first-hquad-average-nodes 0)
6   (define kt-best-first-h-success-rate 0)
7   (define kt-best-first-h-average-nodes 0)
8   (define r 0)
9   (define c 0)
10   (define nodes-created 0)
11   (displayln "################################################")
12   (display "Test cases when n = ")
13   (displayln n)
14   (for ([i (in-range 10)])                                    ;ten tests
15     (set! r (random n))                                       ;randomly pick up r
16     (set! c (random n))                                       ;randomly pick up c
17     (set! nodes-created (kt-depth-first (list r c) n nodes-limit))
18     (when (< nodes-created nodes-limit) (set! kt-depth-first-success-rate (add1 kt-depth-first-success-rate)))
19     (set! kt-depth-first-average-nodes (+ kt-depth-first-average-nodes nodes-created))
20
21     (set! nodes-created (kt-best-first-hquad (list r c) n nodes-limit))
22     (when (< nodes-created nodes-limit) (set! kt-best-first-hquad-success-rate (add1 kt-best-first-hquad-success-rate)))
23     (set! kt-best-first-hquad-average-nodes (+ kt-best-first-hquad-average-nodes nodes-created))
24
25     (set! nodes-created (kt-best-first-h (list r c) n nodes-limit))
26     (when (< nodes-created nodes-limit) (set! kt-best-first-h-success-rate (add1 kt-best-first-h-success-rate)))
27     (set! kt-best-first-h-average-nodes (+ kt-best-first-h-average-nodes nodes-created)))
28
29   (set! kt-depth-first-average-nodes (/ kt-depth-first-average-nodes 10))
30   (set! kt-best-first-hquad-average-nodes (/ kt-best-first-hquad-average-nodes 10))
31   (set! kt-best-first-h-average-nodes (/ kt-best-first-h-average-nodes 10))
32
33   (set! kt-depth-first-success-rate (/ kt-depth-first-success-rate 10))
34   (set! kt-best-first-hquad-success-rate (/ kt-best-first-hquad-success-rate 10))
```

```
35  (set! kt-best-first-h-success-rate (/ kt-best-first-h-success-rate 10))
36
37  (displayln "\nFor kt-depth-first: ")
38  (display "kt-depth-first-success-rate: ")
39  (displayln kt-depth-first-success-rate)
40  (display "kt-depth-first-average-nodes/time complexity: ")
41  (displayln kt-depth-first-average-nodes)
42
43  (displayln "\nFor kt-best-first-hquad: ")
44  (display "kt-best-first-hquad-success-rate: ")
45  (displayln kt-best-first-hquad-success-rate)
46  (display "kt-best-first-hquad-average-nodes/time complexity: ")
47  (displayln kt-best-first-hquad-average-nodes)
48
49  (displayln "\nFor kt-best-first-h: ")
50  (display "kt-best-first-h-success-rate: ")
51  (displayln kt-best-first-h-success-rate)
52  (display "kt-best-first-h-average-nodes/time complexity: ")
53  (displayln kt-best-first-h-average-nodes))
```

Run:

```
(kt-test 8 1000000)
(kt-test 30 1000000)
(kt-test 60 1000000)
(kt-test 100 1000000)
(kt-test 200 1000000)
```

Result:

```
Welcome to DrRacket, version 5.93--2014-01-29(-/f) [3m].

Language: racket.

###############################################

###############################################

Test cases when n = 8



For kt-depth-first:

kt-depth-first-success-rate: 1/10

kt-depth-first-average-nodes/time complexity: 4590738/5



For kt-best-first-hquad:

kt-best-first-hquad-success-rate: 1

kt-best-first-hquad-average-nodes/time complexity: 1457/10
```

10

For kt-best-first-h:

kt-best-first-h-success-rate: 1

kt-best-first-h-average-nodes/time complexity: 64

##

Test cases when n = 30

For kt-depth-first:

kt-depth-first-success-rate: 0

kt-depth-first-average-nodes/time complexity: 1000000

For kt-best-first-hquad:

kt-best-first-hquad-success-rate: 9/10

kt-best-first-hquad-average-nodes/time complexity: 219365/2

For kt-best-first-h:

kt-best-first-h-success-rate: 1

kt-best-first-h-average-nodes/time complexity: 900

##

Test cases when n = 60

For kt-depth-first:

kt-depth-first-success-rate: 0

kt-depth-first-average-nodes/time complexity: 1000000

For kt-best-first-hquad:

kt-best-first-hquad-success-rate: 4/5

kt-best-first-hquad-average-nodes/time complexity: 2031059/10

For kt-best-first-h:

kt-best-first-h-success-rate: 9/10

kt-best-first-h-average-nodes/time complexity: 103240

##

Test cases when n = 100

For kt-depth-first:

kt-depth-first-success-rate: 0

kt-depth-first-average-nodes/time complexity: 1000000

For kt-best-first-hquad:

kt-best-first-hquad-success-rate: 3/5

kt-best-first-hquad-average-nodes/time complexity: 4407713/10

For kt-best-first-h:

kt-best-first-h-success-rate: 3/5

kt-best-first-h-average-nodes/time complexity: 406000

##

Test cases when n = 200

For kt-depth-first:

kt-depth-first-success-rate: 0

kt-depth-first-average-nodes/time complexity: 1000000

For kt-best-first-hquad:

kt-best-first-hquad-success-rate: 2/5

kt-best-first-hquad-average-nodes/time complexity: 6164877/10

For kt-best-first-h:

kt-best-first-h-success-rate: 3/5

kt-best-first-h-average-nodes/time complexity: 424000

Formulate it as a tabular form:

	kt-depth-first	kt-best-first-hquad	Kt-best-first-h (Warnsdorff's Rule)
Success rate (n = 8)	1/10	1	1
Average-number of nodes	918147	146	64
Success rate (n = 30)	0	9/10	1
Average-number of nodes	1000000	109682	900
Success rate (n = 60)	0	8/10	9/10
Average-number of nodes	1000000	203106	103240
Success rate (n = 100)	0	6/10	6/10
Average-number of nodes	1000000	440771	406000
Success rate (n = 200)	0	4/10	6/10
Average-number of nodes	1000000	616488	424000

Some basic facts we can see directly from this table is that both of the implementations with heuristic have much higher success rate than the naive depth first method, and the nodes created (denoting the time complexity) are much smaller.

Time Complexity Analysis:

To better illustrate the analysis, let b denotes the maximum branching factor of the search tree;
let d denotes the depth of the least-cost solution;
let m denotes maximum depth of the state space.

In this Knight Tour problem, we can recognize b as 8 (there are at most 8 possible moves to go); d as n*2 (need to traverse the board); and m is not infinite, but is exponentially large (grows with n).

Given that, in the depth-first search, as the result have shown, it has a very poor performance with a time complexity **Big-O(b^m)**, namely **O(8^m)**, where m grows exponentially with n. That is why it fails to find a solution within a node-limit (even this limit is much larger than d).

However, with good heuristics, we can have much better performance:

For kt-best-first-hquad, the time-complexity is reduced to **O(cd) = O(cn^2)** with **a large constant factor c**;

For kt-best-first-h, the time-complexity is reduced to **O(cd) = O(cn^2)** with **a large constant factor c** (but smaller than the one in kt-best-first-hquad);

Meanwhile, we also notice from the test cases results that the performance with heuristic implementations were not very stable—sometimes they would also fail resulting in a bad time-complexity. This is also understandable though: given a very large chess board and starts from aroung the middle of the chessboard, even the heuristic may walk away from the “correct” direction (getting closer to the border in our case)

Thus, we can conclude: when the solution is dense, naive depth-first method is tolerable; but in more general cases, we need good heuristic functions to dramatically reduce the search cost. And as it has shown in this research, a good heuristic makes a great difference in time complexities.