

$$(x\ y) = (y\ (x\ y))^1$$

Proyecto: “N-Queens Problem”

JAIME RANGEL-OJEDA¹, LEONARDO FLORES-TORRES²



^{1,2}Instituto de Investigaciones en Inteligencia Artificial, Universidad Veracruzana, Xalapa 91000, México

Autores de correspondencia: Jaime Rangel-Ojeda (zs22000513@estudiantes.uv.mx) y Leonardo Flores-Torres (zs22000519@estudiantes.uv.mx).

Abstract: En el presente trabajo presenta una revisión al problema de las N-reinas el cual suele ser usado como problema introductorio para enseñar algoritmos, en este caso el algoritmo de depth-first-search. Por motivos históricos que rodean el área de inteligencia artificial se decidió implementar el algoritmo en un lenguaje de programación dentro de la familia de `lisp`s, descendiente del `lisp` creado por John McCarthy, del que después se derivó `scheme` y del que posteriormente nació `plt scheme`, más conocido como `racket` hoy en día.

Palabras clave: Inteligencia artificial, algoritmo, backtracking, paradigma funcional.

I. Introducción

El problema de las n-reinas es un problema clásico que ilustra la búsqueda en profundidad, el problema consiste en colocar n-reinas en un tablero de ajedrez de una forma que ninguna  se ataque entre sí las reglas son simples, cada  puede moverse en los cuadros de las filas, columnas o diagonales del tablero.

Este problema es muy popular debido a que la búsqueda en profundidad es un árbol lo que significa que no hay posibilidad de que un estado previo sea obtenido de nuevo, esto ayuda a no tener que dar seguimiento a todos los estados y hacer una exploración exhaustiva.

II. Justificación

Hay factores por los cuales `lisp` es un buen lenguaje para aplicaciones de inteligencia artificial.

- Herramientas incorporadas para uso de listas
- Manejo automático de almacenamiento
- Tipado dinámico

- Funciones de primera clase
- Sintaxis uniforme
- Ambiente interactivo
- Extensibilidad

Herramientas incorporadas para uso de listas

Las listas son una estructura de datos muy versátil, `lisp` hace fácil su uso. Muchas aplicaciones de la inteligencia artificial (IA) involucran constantemente el cambio de tamaño de ellas.

Manejo automático de almacenamiento

Un programador en `lisp` no necesita tener un seguimiento de asignación de memoria, esto se hace automáticamente. Esto libera al programador y hace sencillo el uso de programación funcional.

Tipado dinámico

Programadores en `lisp` no necesitan utilizar tipos debido a que el lenguaje hace un seguimiento del tipo de cada objeto en tiempo de ejecución, en lugar de conocer todo el tipificado de objetos en tiempo de compilación.

Funciones de primera clase

¹Referencia al logo de `mit scheme`.

Un objeto de primera clase puede ser usado en cualquier parte y puede ser manipulado de la misma forma como cualquier otro objeto, en otros lenguajes no es posible crear nuevas funciones mientras el programa se encuentra en ejecución, no es posible crear funciones anónimas. En `lisp` podemos hacer las dos cosas.

Sintaxis uniforme

La sintaxis de `lisp` es simple. Esto hace al lenguaje fácil de aprender y muy poco tiempo es usado para corregir typos o errores de sintaxis. También hace fácil a los editores de texto parsearlos en `lisp`.

Ambiente interactivo

Tradicionalmente, un programador escribiría un programa completo, compilarlo y corregir errores detectados por el compilador, después ejecutarlo y debugearlo. Esto es conocido como el modo batch. Para programas largos, esperar por el compilador ocupa una gran porción en tiempo de ejecución. En `lisp` normalmente se escriben pequeñas funciones cada vez, obteniendo un feedback después de cada evaluación. Esto es conocido como ambiente interactivo.

Extensibilidad

Desde su invención en 1958 `lisp` ha podido sobrevivir debido a su adaptabilidad, debido a que es extensible, ha cambiado para incorporar nuevas características, por lo que se ha vuelto popular.

III. Enfoque

Los agentes son muy usados en inteligencia artificial cuando una secuencia de decisiones necesitan realizarse en orden de alcanzar una meta, como por ejemplo jugar ajedrez, los agentes generalmente son llamados agentes inteligentes.

La decisión de un agente se realiza en el contexto del ambiente que provee una plataforma para las interacciones del ambiente. El agente interactúa con el ambiente gracias a la percepción (o toma de información) y acciones (haciendo cambios en el ambiente). La forma principal en la que los agentes interactúan con el ambiente es con la noción de los estados. En el contexto de aplicaciones de inteligencia artificial, un estado

corresponde a la configuración actual de las variables, en nuestro ejemplo la posición de las piezas de ajedrez en un tablero.

Los dos conceptos clave asociados con los agentes interactuando con el ambiente son la percepción (convirtiendo la información del ambiente en representaciones internas) y la acción (cambiando el estado del ambiente).

Un agente interactúa con el ambiente a través de sensores (para percepción) y un conjunto de actuadores (para acciones). Las entradas de datos por el agente desde el ambiente se les llama percepciones.

Agente	Agente de ajedrez
Sensor	Interfaz de entrada del juego de mesa
Actuador	Interfaz de salida de movimientos
Objetivo	Evaluación de posición
Estado	Posición de piezas
Ambiente	Tablero de ajedrez

IV. Tipos de ambientes

Los ambientes que no son completamente observables tienen incertidumbre en ellos, un agente de ajedrez puede observar completamente los efectos de sus acciones en el tablero, por lo tanto es determinista y completamente observable.

Ambientes multi-agentes pueden parecer probabilísticos desde la perspectiva de cada agente individual (debido a que no pueden predecir los movimientos individuales de cada agente), pero aun así los ambientes son tratados como deterministas debido a que cada agente está en control de sus acciones

V. Razonamiento deductivo

El razonamiento deductivo inicia con la base de conocimientos de hechos y el uso de lógica u otros métodos sistemáticos para realizar inferencias. Hay varias aproximaciones al razonamiento deductivo, incluida la búsqueda basada en lógica.

En general, problemas con tareas bien definidas como la inferencia lógica basada en hechos, los cuales requieren grandes cantidades del conocimiento del domi-

nio o aquellos que requieren grandes cantidades de poder computacional (para el aprendizaje inductivo) son a menudo resueltos usando métodos de razonamiento deductivo.

VI. Problema de satisfacción de restricciones

El problema consiste en instanciar un conjunto de variables a unos valores en particular, las variables y las restricciones pueden ser de varios tipos, lo cual conduce a diferentes versiones del problema de satisfacción de restricciones. Estas versiones diferentes podrían ser más útiles en diferentes aplicaciones.

En general la definición es la siguiente: Dado un conjunto de variables, la cual es asignada a un dominio particular y una restricción, encuentra una asignación de valores a las variables, donde la restricción se satisface.

VII. Problemas NP-Duros

Problemas NP encuentran una solución óptima a problemas bajo restricciones previamente especificadas y el tamaño de la búsqueda es de tamaño exponencial. Estos problemas parecen no tener una solución en tiempo polinomial a pesar de una prueba formal de la falta de una solución en tiempo polinomial, una solución no ha sido propuesta. Un problema NP-Completo es una versión de decisión de un problema NP-duro en donde se tiene que determinar si existe una solución válida a un problema de restricción.

Una generalización del problema de las n-reinas es un problema NP-Completo, en esta generalización un subconjunto de n-reinas se colocan en el tablero en una configuración válida, y se desea completar esta configuración agregando ♔s adicionales en posiciones válidas.

VIII. El juego

Muchos juegos de mesa como ajedrez o Go tienen un alto nivel de complejidad y el problema puede obser-

varse como un árbol de posibilidades correspondientes a las posiciones del tablero obtenidas de los movimientos exitosos, el hijo de cada nodo corresponde a las posiciones del tablero alcanzadas usando movimientos individuales.

Las posiciones del tablero son los estados encontrados por el agente, este tipo de juegos se manejan por un árbol de posibles movimientos y seleccionando las mejores jugadas desde la perspectiva de los jugadores, idealmente sería conveniente usar un árbol completo de movimientos pero esto no es posible en la práctica (debido al tamaño de las combinaciones posibles del árbol), en vez de esto una aproximación es construir el árbol con una profundidad restringida y evaluar las posiciones desde el más bajo nivel usando una función heurística diseñada por una persona experta.

IX. Algoritmo

El algoritmo [1] que computa todas las permutaciones, como ya se mencionó anteriormente, esta basado en la búsqueda en profundidad.

Primero uno debe ser capaz de obtener una representación para el tablero, esto se logra a través de una lista de n entradas donde cada entrada es a su vez otra lista de n elementos. Esto genera un tablero de $n \times n$, y se indica si hay o no una ♔ ocupando un lugar en el tablero con 1's y 0's, respectivamente. El tablero es generado mediante la función `make-board`, y al principio del archivo se definen dos valores `cellfig` y `queenfig` que corresponden a las figuras por usar para cambiar la representación de "." y "Q" a imágenes de una celda vacía y una reina, respectivamente, al momento de mostrar el tablero usando `DrRacket`.

```

1 ;; Define figures for empty cells and for the
  ↪ queens
2 (define-values (cellfig queenfig) (values
  ↪ "cellfigure" "queenfigure"))
3
4 ;; Function that creates a chessboard as a
  ↪ n-squared array
5 (define (make-board n)
6   (let loop ([v n]
7             [l '()])
8     (if (zero? v)
9         (list→vector l)
10        (loop (sub1 v) (cons (make-vector n 0)
                              ↪ l)))))

```

```

11 ;; Function to print the state of the board
12 (define (board-print board [showfigs #false])
13   (let*-values ([n (vector-length board)]
14                 [(cell) (if (not showfigs) "."
15                               ↪ cellfig)]
16                 [(queen) (if (not showfigs) "Q"
17                               ↪ queenfig)])
17     (for* ([r n]
18            [c n])
19       (when (zero? c) (newline))
20       (let ([v (board-ref board r c)])
21         (if (zero? v)
22             (display cell)
23             (display queen))
24         )))
25   (newline))

```

De esta manera se pueden tener tableros de la dimensión deseada:

```

1 # REPL
2 > (make-board 4)
3 '#(0 0 0 0) #(0 0 0 0) #(0 0 0 0) #(0 0 0 0))
4
5 > (board-print (make-board 4))
6 . . . .
7 . . . .
8 . . . .

```

```

25 ;; Function to allow the access to the chessboard
  ↪ by row and column and replace the value in
  ↪ that position
26 (define (board-set! board row col val)
27   (vector-set! (vector-ref board col) row val))
28
29 ;; Function to allow the access to the chessboard
  ↪ by row and column
30 (define (board-ref board row col)
31   (vector-ref (vector-ref board col) row))


```

De esta manera se accede al tablero y se puede cambiar su estado, como se muestra a continuación:

```

1 # REPL
2 > (let ([board (make-board 4)])
3   (board-set! board 2 2 1)
4   (board-print board))
5
6 . . . .
7 . . Q .
8 . . . .



```

Además, la función que se encarga de mostrar el estado del tablero es `board-print`. Los 0's y 1's por los que está representado el tablero se sustituyen por `.` o `Q`, respectivamente, para visualizar de una mejor manera donde hay s. Si se quisiera mostrar el tablero con las imágenes en vez de caracteres para los espacios vacíos y para las reinas solamente se llama a la función `board-print` cambiando el segundo argumento opcional por `#true`.

Al hacer esto se está cambiando el valor de las entradas del tablero, y es necesario poder guardar el estado


del tablero cuando una solución es encontrada. Esto se logra iterando sobre todas las listas dentro dentro del arreglo del tablero y copiándolas en un arreglo nuevo:



```
32 ;; Function to copy/save board state
33 (define (board-copy board)
34   (for/vector ([v board])
35     (vector-copy v)))
```


En la siguiente función `depth-first-search` es donde se lleva a cabo la búsqueda en profundidad, las soluciones que se encuentran son copiadas con `board-copy` y guardadas en una lista `sols`. Primero se entra a un `loop` que itera sobre las celdas del tablero, y verifica con `(not (attacked board row col))` que la posición de esa celda se encuentra libre, esto quiere decir que si hay s dentro del tablero ninguna de ellas puede atacar directamente a la celda en curso. En dado caso que sea válido el movimiento, `valid` toma el valor de `#true` y solamente en este caso una nueva  es puesta en el tablero.

```
36 ; Depth first search procedure
37 (define (depth-first-search n)
38   (let ([sols '()])
39     [board (make-board n)])
40     (let loop ([row 0]
41               [col 0])
42       (when (< col n)
43         (let ([valid (not (attacked board row
44                               → col))])
45           (when valid
46             (board-set! board row col 1)
47             (if (= col (sub1 n))
48                 (let ([copy (board-copy board)])
49                   (set! sols (cons copy sols)))
50                 (loop 0 (add1 col)))
49             (board-set! board row col 0))
50         (when (< (add1 row) n) (loop (add1 row
51                               → col)))))
52     sols))
```




La búsqueda en profundidad se lleva a cabo al probar si la celda actual, en el caso de ser válida, se encuentra en la última columna. Si no se encuentra en la última columna entonces el algoritmo se mueve a la siguiente columna lo que lleva al siguiente nivel de profundi-

dad del árbol de búsqueda. En el caso contrario en que la celda sí se encuentre en la última columna se ha encontrado una solución, se guarda en `sols` como se mencionó anteriormente, y se quita la .

Cuando ya se han puesto s y la celda se ha seguido moviendo, si se llega a una celda en que la posición no es válida porque puede ser atacada por otra  ya en el tablero, entonces se ha llegado al punto en el que el proceso de *backtracking* ocurre, la ejecución falla, se abandona el estado del tablero, se *retrocede* en el árbol de búsqueda y se toma la siguiente alternativa.

Ahora es necesario incluir la función `attacked` cuyo propósito es determinar si la posición actual del tablero puede ser atacada por alguna  ya en el tablero.

```
53 ;; Function that tests if the current position in
54   → the board can be attacked by a queen already
55   → in the board
56 (define (attacked board row col)
57   (let ([n (vector-length board)])
58     (let loop ([ac (sub1 col)])
59       (if (< ac 0) #f
60           (let ([r1 (+ row (- col ac))]
61                 [r2 (+ row (- ac col))])
62             (if (or (= 1 (board-ref board row
63                               → ac))
64                     (and (< r1 n) (= 1 (board-ref
65                               → board r1 ac)))
66                     (and (>= r2 0) (= 1
67                               → (board-ref board r2
68                               → ac)))))
69             #t
70             (loop (sub1 ac)))))))
```

Este algoritmo permite computar las permutaciones válidas en que las s se encuentran posicionadas de acuerdo a las restricciones preestablecidas, todas deben estar libres, esto quiere decir que no deben encontrarse en una posición en que otra  ya en el tablero pueda atacarlas. Este método resulta en uno más rápido que el computar todas las posibles permutaciones de las posiciones de las s en el tablero y después filtrar cuáles permutaciones sí cumplen con las restricciones.

Finalmente, las soluciones encontradas por la función `depth-first-search` se muestran una por una mediante la llamada a la función `solve` donde a su vez se llama

internamente al algoritmo de búsqueda en profundidad:

```

65 ;; Solve procedure to compute and print all
    ↳ solutions found by depth first search
    ↳ algorithm
66 (define (solve n [showfigs #false])
67   (let* ([sols (depth-first-search n)]
68         [nsols (length sols)])
69     (display (format "Number of solutions found:
    ↳ ~a. ~%~%" nsols))
70     (for ([board sols]
71           [i (in-range 0 nsols)])
72       (display (format "Solution (~a):" (add1
    ↳ i)))
73       (board-print board showfigs)
74       (newline))))

```

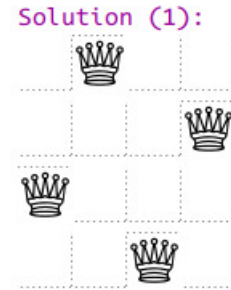


Figura 1: Primer tablero solución para 4 ♔'s.

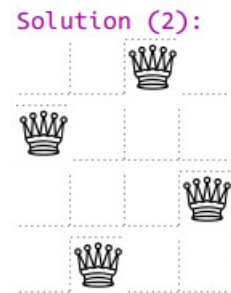


Figura 2: Segundo tablero solución para 4 ♔'s.

Con el algoritmo terminado de implementar es posible probarlo con un número N de ♔'s mayores a 8. De hecho, este algoritmo se puede usar desde valores de $N \geq 4$. En el apéndice se incluyen algunas pruebas pequeñas, por ejemplo se muestra en caso cuando $N = 4$ y cuando $N = 5$. Primero se imprime el número total de soluciones, y posteriormente se imprimen uno a uno las soluciones válidas. Aunque es importante observar que de la manera en como se implementó el algoritmo no es posible reconocer soluciones únicas, esto es, soluciones las cuáles sean diferentes a otras si el tablero es rotado un número deliberado de veces. Por ejemplo, se observó que en las soluciones para el caso cuando $N = 5$ la solución marcada por Solution (6) es la misma que la marcada como Solution (9). De manera similar, los casos marcados por Solution (8) y Solution (10) son equivalentes.

Anteriormente se mencionó el cambio en la representación del problema, de caracteres a imágenes. Si se llama a la función `solve` con su argumento opcional igual a `#true`, para el caso de $N = 4$ se obtienen las figuras 1 y 2 que coinciden con los mostrados en el Apéndice para este caso.

Referencias

- [1] James W Stelly. *Racket Programming the Fun Way: From Strings to Turing Machines*. No Starch Press, 2021.
- [2] C.C. Aggarwal. *Artificial Intelligence: A Textbook*. Springer International Publishing, 2021.
- [3] Peter Norvig. *Paradigms of artificial intelligence programming: case studies in Common LISP*. Morgan Kaufmann, 1992.

Apéndice

```
1  # REPL
2  > (solve 4)
3  Number of solutions found: 2.

4  Solution (1):
5  . Q . .
6  . . . Q
7  Q . . .
8  . . Q .

9  Solution (2):
10 . . Q .
11 Q . . .
12 . . . Q
13 . Q . .

14 > (solve 5)
15 Number of solutions found: 10.

16 Solution (1):
17 . . Q . .
18 . . . . Q
19 . Q . . .
20 . . . Q .
21 Q . . . .

22 Solution (2):
23 . . . Q .
24 . Q . . .
25 . . . . Q
26 . . Q . .
27 Q . . . .

28 Solution (3):
29 . . . . Q
30 . Q . . .
31 . . . Q .
32 Q . . . .
33 . . Q . .

34 Solution (4):
35 . Q . . .
36 . . . . Q
37 . . Q . .
38 Q . . . .
39 . . . Q .

40 Solution (5):
41 . . . . Q
42 . . Q . .
43 Q . . . .
```



```
44   . . . Q .
45   . Q . . .

46 Solution (6):
47   . Q . . .
48   . . . Q .
49   Q . . . .
50   . . Q . .
51   . . . . Q

52 Solution (7):
53   . . . Q .
54   Q . . . .
55   . . Q . .
56   . . . . Q
57   . Q . . .

58 Solution (8):
59   . . Q . .
60   Q . . . .
61   . . . Q .
62   . Q . . .
63   . . . . Q

64 Solution (9):
65   Q . . . .
66   . . Q . .
67   . . . . Q
68   . Q . . .
69   . . . Q .

70 Solution (10):
71   Q . . . .
72   . . . Q .
73   . Q . . .
74   . . . . Q
75   . . Q . .

76 > (solve 8)
77 Number of solutions found: 92.

78 Solution (1):
79   . . Q . . . .
80   . . . . . Q .
81   . . . Q . . .
82   . Q . . . . .
83   . . . . . . Q
84   . . . . Q . .
85   . . . . . Q .
86   Q . . . . . .

87 Solution (2):
88   . . Q . . . .
89   . . . . Q . .
```



```

90  . Q . . . . .
91  . . . . . Q
92  . . . . . Q .
93  . . . Q . . .
94  . . . . . Q .
95  Q . . . . .

96  ; Removed output

97  Solution (92):
98  Q . . . . .
99  . . . . . Q .
100 . . . . Q . .
101 . . . . . Q
102 . Q . . . . .
103 . . . Q . . .
104 . . . . . Q .
105 . . Q . . . .

106 > (solve 9)
107 Number of solutions found: 352.

108 Solution (1):
109 . . . . . Q .
110 . . . Q . . .
111 . . . . . Q .
112 . . Q . . . .
113 . . . . . Q
114 . . . . . Q .
115 . Q . . . . .
116 . . . . Q . .
117 Q . . . . .

118 Solution (2):
119 . . . . . Q .
120 . . . . Q . .
121 . . Q . . . .
122 . . . . . Q
123 . . . . . Q .
124 . . . . . Q .
125 . Q . . . . .
126 . . . Q . . .
127 Q . . . . .

128 ; Removed output

129 Solution (352):
130 Q . . . . .
131 . . . . Q . .
132 . Q . . . . .
133 . . . . . Q .
134 . . . . . Q
135 . . Q . . . .
136 . . . . . Q .

```

```

137   . . . Q . . . . .
138   . . . . . Q . .

139 > (solve 10)
140 Number of solutions found: 724.

141 Solution (1):
142   . . . . Q . . . . .
143   . . . . . Q . . . .
144   . . . Q . . . . . .
145   . . . . . . . . Q
146   . . Q . . . . . . .
147   . . . . . Q . . . .
148   . . . . . . . Q .
149   . Q . . . . . . . .
150   . . . . . . Q . .
151   Q . . . . . . . .

152 Solution (2):
153   . . . . . Q . . . .
154   . . . Q . . . . . .
155   . . . . . . . . Q
156   . . . . Q . . . . .
157   . . Q . . . . . . .
158   . . . . . . . Q .
159   . . . . . Q . . . .
160   . Q . . . . . . . .
161   . . . . . . Q . .
162   Q . . . . . . . .

163 ; Removed output

164 Solution (724):
165   Q . . . . . . . . .
166   . . . . . . Q . .
167   . Q . . . . . . . .
168   . . . . . . . Q .
169   . . . . . Q . . . .
170   . . Q . . . . . . .
171   . . . . . . . . Q
172   . . . Q . . . . . .
173   . . . . . Q . . . .
174   . . . . Q . . . . .

175 > (solve 11)
176 Number of solutions found: 2680.

177 Solution (1):
178   . . . . . Q . . . . .
179   . . . . . . . . . Q
180   . . . . Q . . . . . .
181   . . . . . . . . Q .
182   . . . Q . . . . . . .
183   . . . . . . . . Q .

```

```

184   . . Q . . . . . . .
185   . . . . . . Q . . .
186   . Q . . . . . . . .
187   . . . . . Q . . . .
188   Q . . . . . . . . .

189 Solution (2):
190   . . . . . Q . . . .
191   . . . . . . . . Q
192   . . . Q . . . . . .
193   . . . . . Q . . . .
194   . . . . . . . Q . .
195   . . Q . . . . . . .
196   . . . . . . . . Q .
197   . . . . . . Q . . .
198   . Q . . . . . . . .
199   . . . . Q . . . . .
200   Q . . . . . . . . .

201 ; Removed output

202 Solution (2680):
203   Q . . . . . . . . .
204   . . . . . Q . . . .
205   . Q . . . . . . . .
206   . . . . . . Q . . .
207   . . Q . . . . . . .
208   . . . . . . . Q . .
209   . . . Q . . . . . .
210   . . . . . . . . Q .
211   . . . . Q . . . . .
212   . . . . . . . . Q
213   . . . . . Q . . . .

214 > (solve 12)
215 Number of solutions found: 14200.

216 Solution (1):
217   . . . . . Q . . . .
218   . . . . . . Q . . .
219   . . . . Q . . . . .
220   . . . . . . . . Q .
221   . . . Q . . . . . .
222   . . . . . . . . Q .
223   . . . . . . Q . . .
224   . . Q . . . . . . .
225   . . . . . . . . Q
226   . Q . . . . . . . .
227   . . . . . . . Q . .
228   Q . . . . . . . . .

229 Solution (2):
230   . . . . . . Q . . .
231   . . . . . Q . . . .

```

```

232   . . . Q . . . . . . .
233   . . . . . . . Q . .
234   . . . . Q . . . . .
235   . . . . . . . . Q .
236   . . . . . . . Q . .
237   . . Q . . . . . . .
238   . . . . . . . . . Q
239   . Q . . . . . . . .
240   . . . . . Q . . . .
241   Q . . . . . . . . .

```

```

242   ; Removed output

```

```

243   Solution (14200):

```

```

244   Q . . . . . . . . .
245   . . . . . . . Q . .
246   . Q . . . . . . . .
247   . . . . . . . . . Q
248   . . Q . . . . . . .
249   . . . . . Q . . . .
250   . . . . . . . Q . .
251   . . . Q . . . . . .
252   . . . . . . . . Q .
253   . . . . Q . . . . .
254   . . . . . . . Q . .
255   . . . . . Q . . . .

```