Aprendizaje por Refuerzo

Pablo Barranco, Thomas Bladt, Alejandro De Anda

Simulación: Proyecto Final
Instituto Tecnológico Autónomo de México
Dr. Jorge de la Vega Góngora

14 de Diciembre del 2020

Resumen

Tenemos como objetivo mostrar la aplicación de algoritmos de aprendizaje por refuerzo, basados en los métodos de Monte Carlo y Q-Learning. Nos centramos en el problema de la resolución de laberintos, creados mediante una variación del algoritmo de PRIM. Comparamos los métodos, sus soluciones y la eficiencia computacional de cada uno. Finalmente presentamos una breve discusión de las ventajas de cada método.

1. Introducción

Este trabajo representa el proyecto final de la materia de Simulación (así como nuestro último trabajo de la licenciatura). Quisimos implementar el uso de inteligencia artificial desde cero (sin utilizar librerías existentes). La mayor parte de los códigos existentes se encuentran programados en `Python`, por lo que decidimos programar nuestros propios códigos en `R`.

Dado que gran parte del material corresponde a temas relacionados con la materia de simulación, pero que no fueron vistos en clase, tuvimos que buscar buenas referencias. Principalmente nos basamos en las clases en línea del Dr.Daniel Soper, "Foundations of Reinforcement Learning." [3], "Foundations of Q-Learning."[4], así como en el libro principal de aprendizaje de máquina *"Reinforcement learning: An introduction"* de Richard S. Sutton [1].

2. Aprendizaje por Refuerzo

Antes de hablar del concepto de aprendizaje por refuerzo como técnica computacional, podemos entenderlo como un proceso de aprendizaje general. En su contexto más básico, se entiende como un proceso de aprendizaje mediante recompensas y/o castigos. Por ejemplo, pensemos en cómo un niño pequeño aprende a hacer algo nuevo. El niño observa su entorno, toma una decisión y actúa de determinada forma. Posteriormente observa cómo repercutió esa acción en su entorno y aprende de ello. Siguiendo con el ejemplo, si el niño consigue comer solo, sus padres lo felicitarán (recompensa), mientras que si acerca las manos al fuego, será regañado (castigo). Así, el niño aprende a tomar decisiones de una mejor manera. En ocasiones, se puede tomar una decisión que no maximiza la recompensa inmediata, sino que piensa a futuro para maximizar la recompensa total; tal sería el caso de faltar a una reunión de amigos (recompensa inmediata) para poder estudiar y obtener una buena calificación final (recompensa total).

Para poder traducir problemas de la vida real al lenguaje de una computadora, es necesario hacer un planteamiento teórico. Lo primero que se necesita es un agente explorador que tenga un propósito o meta que quiera lograr, que interactúe con un ambiente y obtenga información acerca de sus acciones en cada estado.

En el campo de aprendizaje de máquina existen tres paradigmas principales de aprendizaje: *supervisado*, *no supervisado* y *por refuerzo*[1]. En aprendizaje supervisado se utilizan datos de entrenamiento, que son ejemplos proporcionados por un supervisor externo, en donde se le indica a la computadora cómo reaccionar ante determinadas situaciones. A través de estos ejemplos, el algoritmo busca encontrar una estructura subyacente para poder reaccionar de manera correcta ante futuros eventos (que no sean de prueba). En aprendizaje no supervisado, de igual forma se intenta encontrar una estructura subyacente en un entorno. Sin embargo, no se proporcionan datos de entrenamiento. La diferencia principal que tiene aprendizaje por refuerzo ante estos dos métodos es que no necesariamente busca encontrar una estructura subyacente en el entorno, sino que se basa en un sistema que busca maximizar las recompensas obtenidas.

2.1. Entrada-Salida

Como en cualquier algoritmo, aprendizaje por refuerzo requiere de dos cosas: una entrada (input) y una salida (output). Como entrada se le tienen que proporcionar al algoritmo los *estados* del entorno, así como las *recompensas* de estar en cada estado. Como salida, el algoritmo nos dará acciones. Lo que se busca con el algoritmo es determinar una estrategia (policy) óptima para maximizar las recompensas acumuladas. Es decir, busca poder elegir una acción - en cada estado - de manera óptima.

2.2. Recompensas

Aprendizaje por refuerzo se basa en un sistema de recompensas. Una recompensa se puede interpretar como una métrica que nos dice qué tan buena resulta ser una acción. Hay que tener en cuenta que lo que busca el algoritmo es maximizar la recompensa acumuldada. Es posible que se elija una acción que no maximice la recompensa inmediata, sino una que maximice la recompensa en un futuro. Es por esto que no se le puede considerar a este algoritmo como uno codicioso. Matemáticamente, se define una función recompensa, que dependerá del problema y del modelo (puede querer maximizar ganancias o minimizar pérdidas).

2.3. Entorno

El entorno es el "lugar" en donde el agente se encuentra aprendiendo. Es aquello que le provee al algoritmo la información acerca de los estados en los que se encuentra y las recompensas correspondientes. Notamos que no necesariamente tiene que ser interpretado de forma espacial, sino sólo como un ente que provee de información al agente. Por ejemplo, un laberinto puede ser particionado en una cuadrícula, en donde cada cuadro puede ser visto como un estado; en el juego de blackjack un estado dependería de las cartas que se muestran en la mesa y las que ya han salido.

Una vez que se tienen estados, es natural que para cada uno de ellos se tenga una lista finita de posibles acciones. Podría ser una acción espacial (moverse hacia alguna dirección) u otro tipo de acción (en el caso de blackjack podría ser tomar una nueva carta o no). El entorno también define cómo se transiciona de un estado a otro dependiendo de la acción elegida por el agente. El entorno también tiene la información de la recompensa que se tiene por elegir cualquier acción en cualquier estado.

Conforme el algoritmo va recopilando información de las recompensas en cada estado, va aprendiendo. Algo importante que debemos notar es que el agente a veces decide elegir lo que conoce como la mejor acción (que aprendió anteriormente) o una acción completamente aleatoria. Esto permite al agente"explorar"distintas alternativas y, posiblemente, encontrar una mejor solución al problema. La probabilidad de explorar o elegir la mejor acción se controla con un parámetro que define qué proporción del tiempo el agente explora y qué proporción del tiempo elige la mejor acción.

2.4. Proceso de Decisión de Markov

Los MDP (procesos de decisión de Markov, por sus siglas en inglés) son el lenguaje matemático mediante el cual el agente decidirá qué acción elegir en el proceso de aprendizaje. Este tipo de sistema estocástico se utiliza cuando una decisión depende en parte de un factor aleatorio y en parte de un ente que toma decisiones (en este caso el agente).

Una parte importante de estos sistemas es que trabajan con tiempos discretos (es decir, cada unidad de tiempo se puede definir como $t_0, t_1, ...$). El proceso de decisión, al tiempo t, es como sigue:

1. Entrar al estado S_t.
2. Elegir una acción a_t.
3. Recibir una recompensa r_t por la acción.

con lo que se entra al estado S_{t+1} y se repiten estos pasos hasta que termine el proceso.

2.5. Entrenamiento

Al igual que en aprendizaje supervisado, la primera etapa de aprendizaje por refuerzo consiste en entrenar al algoritmo. Esto se hace mediante muchas iteraciones de transición de estados, tomando en cuenta las recompensas observadas. Se corre el "juego"(entrecomillamos juego, pues lo que se está entrenando no necesariamente es con fines de diversión) repetidas ocasiones, de tal forma que se intente lograr la meta final. Después de cada iteración el algoritmo tiene más información, por lo que actualiza su política de decisión. Una vez que el método haya sido entrenado satisfactoriamente, el sistema está listo para implementarse en un ejemplo real. Es decir, la fase de entrenamiento le sirvió al algoritmo para saber cómo reaccionar ante ciertas situaciones, y ahora es momento de afrontarse por si solo a su tarea. A esta etapa se le conoce como modo de inferencia, y aquí ya no se actualiza la política de decisión, sino que simplemente ejecuta la estrategia que generó en el entrenamiento.

3. Creación de los laberintos

Para poder implementar los algoritmos de aprendizaje por refuerzo, lo primero que requerimos fue un ambiente o entorno. Aunque existen bibliotecas de juegos en algunos lenguajes de programación (por ejemplo `pygame` en `Python`), optamos por crear un juego sencillo - la resolución de un laberinto - para tener más control sobre el código.

Crear laberintos a mano es posible, pero quisimos poder generarlos de forma arbitraria y del tamaño de nuestra elección. Para esto, programamos una variante del algoritmo de PRIM; el código está basado en el pseudocódigo encontrado en Stack Overflow [2]. Para entenderlo, se necesita saber lo siguiente: el laberinto es una matriz llena de caminos (1) y paredes (0). Una celda frontera es cualquier celda con distancia 2 a la actual, tal que sea pared. Una celda vecina es una celda con distancia 2 a la actual tal que sea camino. A continuación se muestra el pseudocódigo.

Algorithm 1 PRIM modificado

Datos: tamaño del laberinto m, n.

Resultado: matriz de $m \times n$ correspondiente a un laberinto.

crear una matriz de $m \times n$ llena de ceros (paredes)

elegir una celda aleatoria, cambiarla a 1 (camino) y calcular las celdas frontera

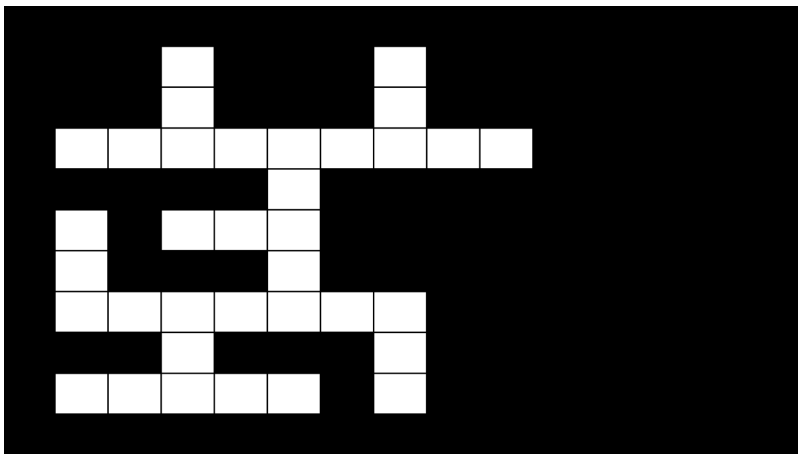
mientras *lista de celdas frontera no vacía* **hacer**

- elegir celda frontera aleatoria de la lista y hacerla celda actual
- elegir celda vecina aleatoriamente y conectarla con la celda actual
- calcular celdas frontera de la actual y agregarlas a la lista
- quitar celda actual de la lista

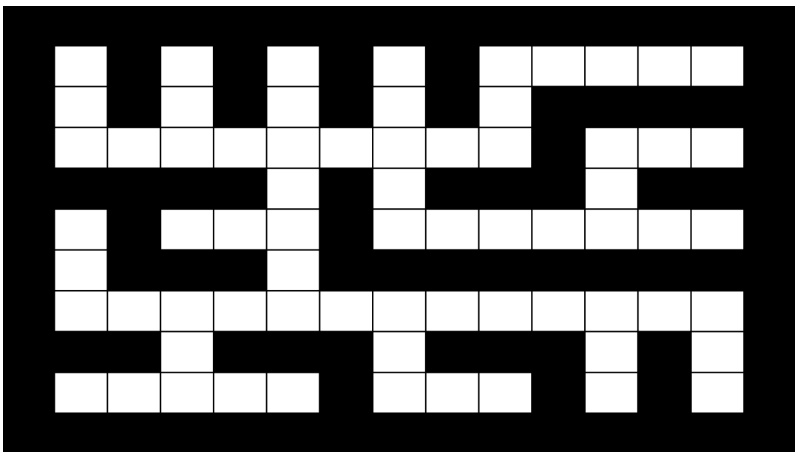
fin

En la Figura 1 mostramos distintas etapas de la creación de un laberinto de tamaño 11×15, utilizando nuestro algoritmo.

(a) Primera iteración.



(b) Iteración número 17.



(c) Iteración final.

Figura 1: Distintas etapas del algoritmo de PRIM modificado.

Finalmente, elegimos como entrada a la celda $(m-1, 1)$ y salida a $(2, n)$. Teniendo ya un laberinto, se crea la matriz de recompensas. Una vez desarrollado el laberinto, debemos construir su matriz de recompensas. Esta matriz consta de valores negativos únicamente, excepto en la celda correspondiente a la meta (cuya recompensa es positiva y muy grande).

Además, los valores son más grandes en las celdas en las que existen paredes, mientras que el valor es más pequeño en las celdas que contienen un camino. Esto es porque queremos disuadir al algoritmo de quedarse en un bucle infinito en el laberinto, maximizando cada vez más la recompensa; al asignarle un castigo a cada unidad de camino, el algoritmo buscará llegar a la recompensa de la forma más eficiente posible. En el cuadro 1 podemos ver la matriz de recompensas resultante de nuestro ejemplo y en la figura 2 mostramos la matriz de una forma más visual.

-100	-100	-100	-100	-100	-100	-100	-100	-100	-100	-100	-100	-100	-100	-100
-100	-1	-100	-1	-100	-1	-100	-1	-100	-1	-1	-1	-1	-1	1000
-100	-1	-100	-1	-100	-1	-100	-1	-100	-1	-100	-100	-100	-100	-100
-100	-1	-1	-1	-1	-1	-1	-1	-1	-1	-100	-1	-1	-1	-100
-100	-100	-100	-100	-100	-1	-100	-1	-100	-100	-100	-1	-100	-100	-100
-100	-1	-100	-1	-1	-1	-100	-1	-1	-1	-1	-1	-1	-1	-100
-100	-1	-100	-100	-100	-1	-100	-100	-100	-100	-100	-100	-100	-100	-100
-100	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-100
-100	-100	-100	-1	-100	-100	-100	-1	-100	-100	-100	-1	-100	-1	-100
-1	-1	-1	-1	-1	-1	-100	-1	-1	-1	-100	-1	-100	-1	-100
-100	-100	-100	-100	-100	-100	-100	-100	-100	-100	-100	-100	-100	-100	-100

Cuadro 1: Matriz de recompensas para el laberinto.

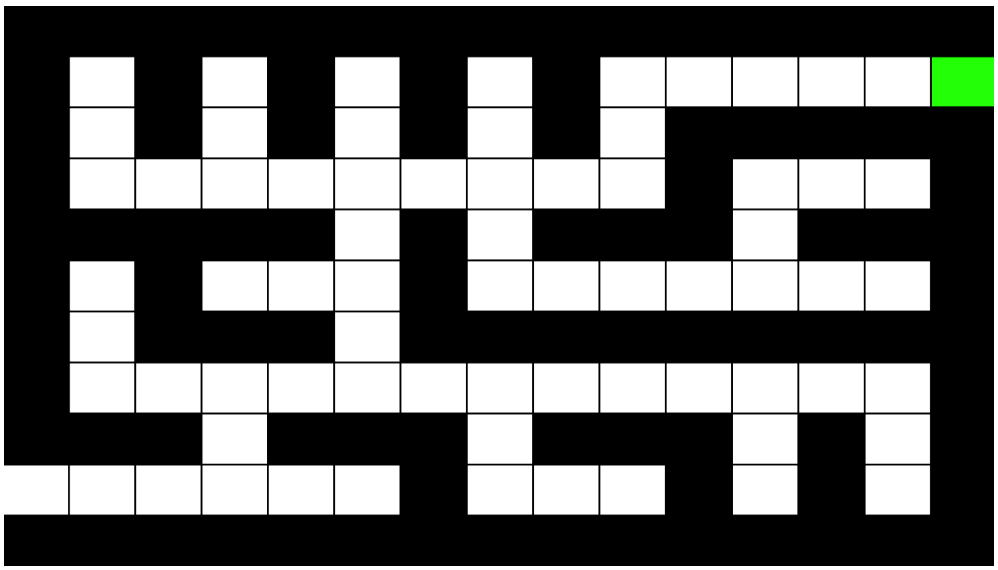


Figura 2: Laberinto final. La celda verde corresponde a la meta.

4. Q-Learning

Q-Learning es un tipo particular de aprendizaje por refuerzo, por lo que contamos con un agente explorador y con un entorno en el que éste se encuentra. A diferencia de otros métodos, en Q-Learning se busca construir una conducta mediante la interacción directa con el entorno. Así, el agente explorador interactúa una y otra vez con el entorno

y aprende mediante un mecanismo de prueba/error. Al final de cada intento, el agente actualiza sus conocimientos y es así como aprende a desarrollarse en su entorno.

Q-Learning cuenta con todas las características de un modelo de aprendizaje por refuerzo. Sin embargo, Q-Learning también requiere que tanto el conjunto de estados como de acciones sean finitas. Es decir, existe un conjunto finito de estados $\mathcal{S} = \{s_0, s_1, \ldots, s_n\}$ en los que el agente puede encontrarse y en cada uno de ellos puede elegir entre un conjunto finito de acciones $\mathcal{A} = \{a_0, a_1, \ldots, a_m\}$. Considerando nuestro ejemplo, supongamos que tenemos un laberinto de tamaño 10×10; entonces, cada celda corresponde a un estado, con lo que $\mathcal{S} = \{1, 2, \ldots, 100\}$ y las acciones que puede tomar el agente son moverse arriba, derecha, izquierda o abajo, con lo que podemos definir $\mathcal{A} = \{1, 2, 3, 4\}$.

Este método debe su nombre a que funciona mediante *Q-values*. Estos valores representan la calidad (quality en inglés, de ahí la Q) de tomar la decisión a en el estado s, que podemos escribir como una función $Q(s, a)$. Intuitivamente, un Q-value nos dice qué tanto esperamos obtener de recompensa acumulada al tomar una acción en determinado estado.

Regresando a nuestro ejemplo, necesitamos una forma de guardar los Q-values. Para cada estado, necesitamos un Q-value para cada acción. Por lo tanto, requerimos de un array de dimensión $m \times n \times 4$, donde m representa el número de filas del laberinto y n el número de columnas. La idea es que cuando el agente se encuentre en algún estado, elija la acción que tenga un mayor Q-value. De esta forma, se puede interpretar al array de los Q-values como la política del agente.

4.1. Diferencias Temporales

Ahora debemos pensar en cómo se actualizarán los Q-values conforme el agente va recopilando información de su entorno. En concreto: ¿Cómo actualizar los Q-values de determinada acción, en determinado estado, después de haberla realizado? Para esto existe un método conocido como diferencias temporales, que nos proporciona la siguiente ecuación

$$DT\left(s_t, a_t\right) = r_t + \gamma \cdot \max_a Q\left(s_{t+1}, a\right) - Q\left(s_t, a_t\right). \tag{1}$$

En la ecuación 1 tenemos que

- r_t es la recompensa inmediata (dada por la matriz de recompensas) por haber tomado la acción a_t.
- γ es un parámetro de descuento, entre 0 y 1, que controla qué tan importantes son las recompensas en un futuro. Es intuitivo que una recompensa inmediata es preferible que una futura, y γ nos dice qué tanto.

- $\text{máx}_a\, Q\,(s_{t+1}, a)$ corresponde al Q-value más grande en el estado actual s_{t+1}. Nos dice cuánto esperamos acumular en recompensas futuras, en el mejor de los casos.
- $Q\,(s_t, a_t)$ es el Q-value en el estado pasado, con la acción elegida. Es el valor que buscamos actualizar.

En resumen, la ecuación 1 nos permite considerar recompensas inmediatas y futuras a la vez.

4.2. Ecuación de Bellman

Una vez que tenemos la diferencia temporal calculada, es hora de actualizar el Q-value. Para esto introducimos la ecuación de Bellman, dada por

$$Q_{\text{nueva}}\,(s_t, a_t) = Q_{\text{vieja}}\,(s_t, a_t) + \alpha \cdot DT\,(s_t, a_t)\,. \tag{2}$$

En la ecuación 2 tenemos que

- $Q_{\text{nueva}}\,(s_t, a_t)$ corresponde al valor actualizado del Q-value en el estado s_t realizando la acción a_t
- $Q_{\text{vieja}}\,(s_t, a_t)$ corresponde al valor pasado del Q-value en el mismo estado y bajo la misma acción.
- α es un parámetro de aprendizaje, entre 0 y 1, que controla qué tan rápido se actualizan los Q-values, basado en la diferencia temporal.
- $DT\,(s_t, a_t)$ es la diferencia temporal, calculada en la ecuación 1.

Esta ecuación es sumamente importante dado que actualiza de manera adecuada los Q-values, dependiendo de la acción que se tomó en el estado anterior. Como su nombre lo indica, está ecuación fue inventada por Richard Bellman.

Entonces, recapitulando: el array de Q-values se inicializa como un array de ceros (puede ser lo que sea, en realidad), y en cada iteración se realiza el proceso de pasar de un estado a otro, a través de las acciones posibles. En cada paso se calcula la diferencia temporal, la ecuación de Bellman y se actualiza el Q-value. En la fase de entrenamiento se repite el proceso muchas veces, con lo que el array de Q-values se va refinando cada vez más. Una vez que concluye la fase de entrenamiento, obtenemos un array de Q-values que refleja la política que se debe seguir para maximizar la recompensa. Qué tan precisa es esa política depende de varios factores, uno de los más grandes siendo el número de iteraciones para el entrenamiento (problemas más complejos requieren más entrenamiento).

4.3. Implementación del método

Para programar el método de Q-Learning nos basamos en un código de Python para un problema similar [5], por parte del Dr. Daniel Soper. Presentamos el pseudocódigo de nuestra implementación:

Algorithm 2 Q-Learning.

Datos: número de entrenamientos, matriz de recompensas, ε, α y γ.

Resultado: array de Q-values.

para $i = 1$ *hasta número de entrenamientos* **hacer**
    elegir celda de inicio aleatoria.
    mientras *no sea una celda terminal* **hacer**
        elegir siguiente acción
        calcular siguiente estado (nueva celda)
        calcular recompensa inmediata
        calcular diferencia temporal
        actualizar Q-value
    fin
fin

5. Monte Carlo

Este algoritmo es más primitivo que Q-learning. La diferencia principal radica en que no utiliza la ecuación de Bellman para mapear las recompensas, sino que es un simple promedio. El término "Monte Carlo" por lo general se emplea para modelos de estimación que están compuestos por un término aleatorio. En este caso, sí existe el uso de un componente aleatorio para la exploración del agente llamado "ε - greedy". Pero no por eso este algoritmo lleva el nombre de Monte Carlo, sino porque realiza un promedio de los retornos del entorno.

Monte Carlo, al igual que Q-Learning, requiere que tanto el conjunto de estados como el conjunto de acciones sean finitos. Tras múltiples simulaciones, el agente interactua con el entorno de manera finita y estas interacciones reciben el nombre de "episodios". De forma iterativa, los episodios consiguen que el agente aprenda a interactuar de manera óptima dentro del entorno y de esta forma maximizan la recompensa previmente presentada.

Para entender Monte Carlo, primero hay que entender la forma de exploración del algoritmo. De manera similar a Q-Learning, este método explora el entorno si un número aleatorio es menor a ε. De lo contrario, el agente selecciona la siguiente acción que maximice el retorno esperado. De esta forma, a mayor ε, mayor exploración. En pocas pa-

labras, lo que hace el algoritmo de Monte Carlo es seleccionar un estado inicial aleatorio. Mediante el uso de ε - greedy, selecciona una nueva acción y, consecuentemente, un nuevo estado. Finalmente, repite esto hasta llegar a un estado terminal. Una vez finalizado el episodio, se promedia la recompensa otenida tras la primer visita de cada estado. Así, se registra memoria de los estados visitados y sus recompensas asociadas multiplicadas por un factor de descuento. Esto favorece a que el aprendizaje del agente sea robusto, priorizando los retornos inmediatos sin olvidar las últimas acciones del episodio.

La asignación del retorno ponderado al estado del episodio en el tiempo t se asigna mediante la siguiente ecuación:

$$G_t = R_{t+1} + \gamma R_{t+1} + \gamma^2 R_{t+3} + ...,$$

donde $0 \leq \gamma \leq 1$ es la tasa de retorno y R_t es el retorno en el tiempo t.

De esta forma, se aproxima el retorno esperado en cada estado. Para posteriormente optar por la politica cuyas acciones maximicen el retorno.

A continuacón se presentan los pseudocódigos de nuestra implementación de Monte Carlo:

Algorithm 3 Monte Carlo: estructura general.

Datos: número de entrenamientos, matriz de recompensas, ε y γ.
Resultado: array de Q-values.

```
Inicializar Q, v = visitas y r = recompensas como arrays de ceros.
para i = 1 hasta número de entrenamientos hacer
    ε ← máximo entre 0 y ε − i/número de entrenamientos
    episodio ← episodio(ε, γ)
    inicializar vector de estado-acción encontrados
    para k = 1 hasta duración de episodio hacer
        si estado-acción de episodio no se ha encontrado entonces
            v(estado, acción) ← v(estado, acción)+1
            r(estado, acción) ← r(estado, acción)+r
            Q(estado, acción) ← r(estado, acción)/v(estado, acción)
            cambiar estado-acción a encontrado
        fin
    fin
fin
```

Algorithm 4 Monte Carlo: episodio.

Datos: ε y γ.

Resultado: tabla estado-acción-recompensa.

inicializar tabla estado-acción-recompensa como tabla vacía de 3 columnas
elegir estado inicial aleatorio
mientras *estado no terminal* **hacer**
  elegir siguiente acción a
  calcular siguiente estado (nueva celda)
  calcular recompensa inmediata r
  agregar a estado-acción-recompensa la fila (estado anterior, acción, recompensa)
fin
$g \leftarrow 0$
$R \leftarrow$ estado-acción-recompensa al revés
para $i = 1$ *hasta dimensión* R **hacer**
  $g \leftarrow r + \gamma g$
  guardar valores de g en vector G
fin
reemplazar vector G en última columna de estado-acción-recompensa

6. Resultados

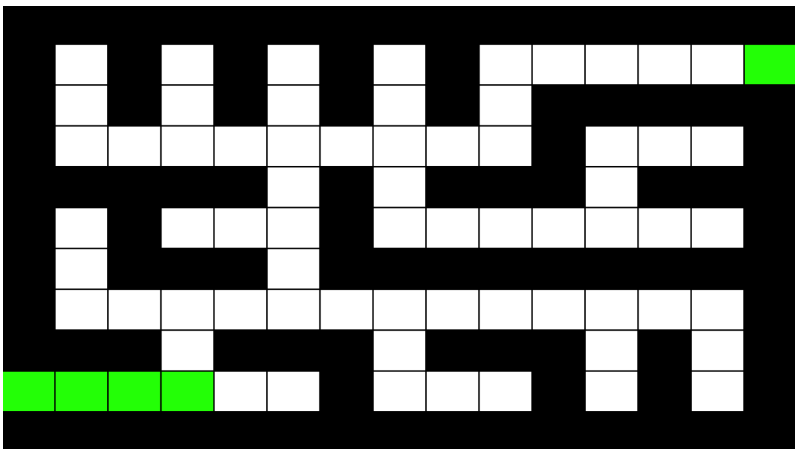
En esta sección presentamos los resultados que obtuvimos para el problema de la resolución de laberintos de distintas dimensiones. Corremos el algoritmo (sea Q-learning original o aquel basado en Monte Carlo) para obtener una matriz Q y procedemos a la etapa de inferencia, es decir, vemos si el algoritmo puede resolver el laberinto.

6.1. Ejemplo práctico

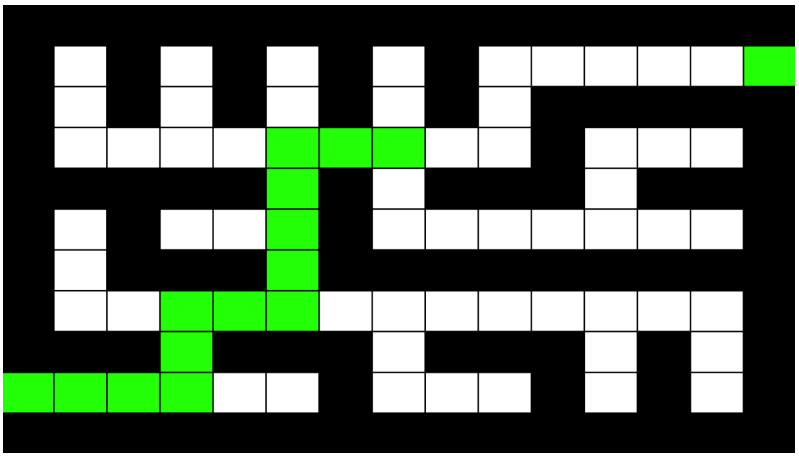
Aquí mostramos todo el proceso para resolver el problema de un laberinto en particular. Lo primero que debemos hacer es crear un laberinto, que elegimos que sea de tamaño 11×15. En particular, vamos a trabajar con el laberinto creado en la sección 3. Corrimos el algoritmo de Q-Learning, entrenando un total de $50,000$ iteraciones (se eligió este número a base de prueba y error). El algoritmo tardó un total de 2.832 segundos. El camino que resulta es el siguiente

$$[10,1] \to [10,2] \to [10,3] \to [10,4] \to [9,4] \to [8,4] \to [8,5] \to$$
$$[8,6] \to [7,6] \to [6,6] \to [5,6] \to [4,6] \to [4,7] \to [4,8] \to [4,9] \to$$
$$[4,10] \to [3,10] \to [2,10] \to [2,11] \to [2,12] \to [2,13] \to [2,14] \to [2,15]$$

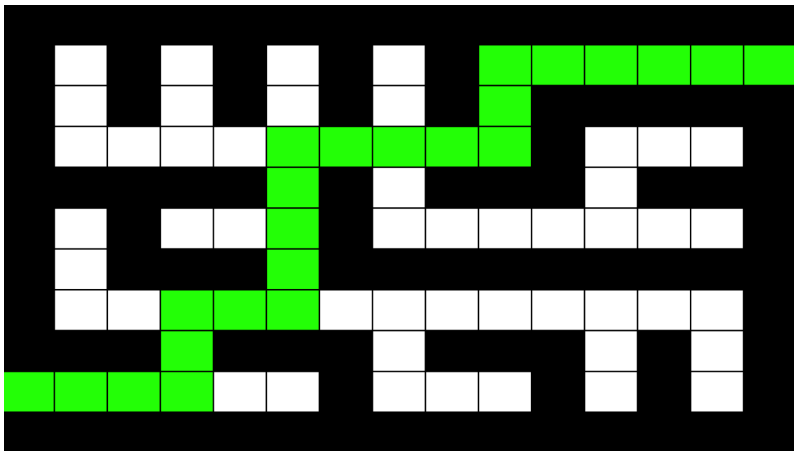
De igual forma lo podemos visualizar gráficamente. En la figura 3 se muestra el camino solución en distintas etapas.



(a) Primera iteración.



(b) Iteración número 17.



(c) Iteración final.

Figura 3: Distintas etapas del algoritmo de PRIM modificado.

Ahora que hemos visto cómo funciona el algoritmo, pasaremos a evaluar la calidad de los métodos y los compararemos.

6.2. Evaluación de los métodos

Para evaluar los métodos, corrimos el algoritmo para laberintos cuadrados, de tamaños:

- Laberinto 1: 5×5 (25 casillas)
- Laberinto 2: 7×7 (49 casillas)
- Laberinto 3: 9×9 (81 casillas)
- Laberinto 4: 11×11 (121 casillas)
- Laberinto 5: 13×13 (169 casillas)
- Laberinto 6: 15×15 (225 casillas)

Notamos que los tamaños siempre consisten de números impares. Esto es porque el algoritmo de PRIM modificado genera laberintos simétricos (márgenes simétricos) para estos casos.

Queremos aclarar que el número de iteraciones lo elegimos de manera "artesanal", basado en nuestras experiencias de prueba y error. Intentamos elegir el menor número posible de iteraciones para el entrenamiento, de tal forma que nos proveyera con una matriz de Q-values con una solución correcta al problema.

6.3. Comparativa

Los valores obtenidos con ambos métodos se reportan a continuación en los cuadros 2 y 3.

Cuadro 2: Resultados utilizando Q-Learning.

Laberinto	Tamaño	Casillas	Iteraciones	Tiempo
1	5×5	25	1,000	0.151
2	7×7	49	2,500	0.240
3	9×9	81	12,500	0.647
4	11×11	121	12,500	0.820
5	13×13	169	25,000	1.232
6	15×15	225	50,000	2.819

Cuadro 3: Resultados utilizando Monte Carlo.

Laberinto	Tamaño	Casillas	Iteraciones	Tiempo
1	5×5	25	1,000	0.54
2	7×7	49	10,000	8.717
3	9×9	81	-	-
4	11×11	121	-	-
5	13×13	169	-	-
6	15×15	225	-	-

Nota. El método de Monte Carlo no encontró la solución en un tiempo razonable de entrenamiento para los laberintos 3, 4, 5 y 6. Hablamos más al respecto en la sección 7.

Finalmente, quisimos investigar la complejidad del algoritmo. Es claro que un laberinto más grande requiere más iteraciones de entrenamiento, por lo que investigamos cómo crece el tiempo con respecto a las iteraciones. Corrimos el código para laberintos de tamaño $n = \{50000, 75000, \ldots, 500000\}$ para el método de Q-Learning. En la figura ?? se muestran los resultados obtenidos.

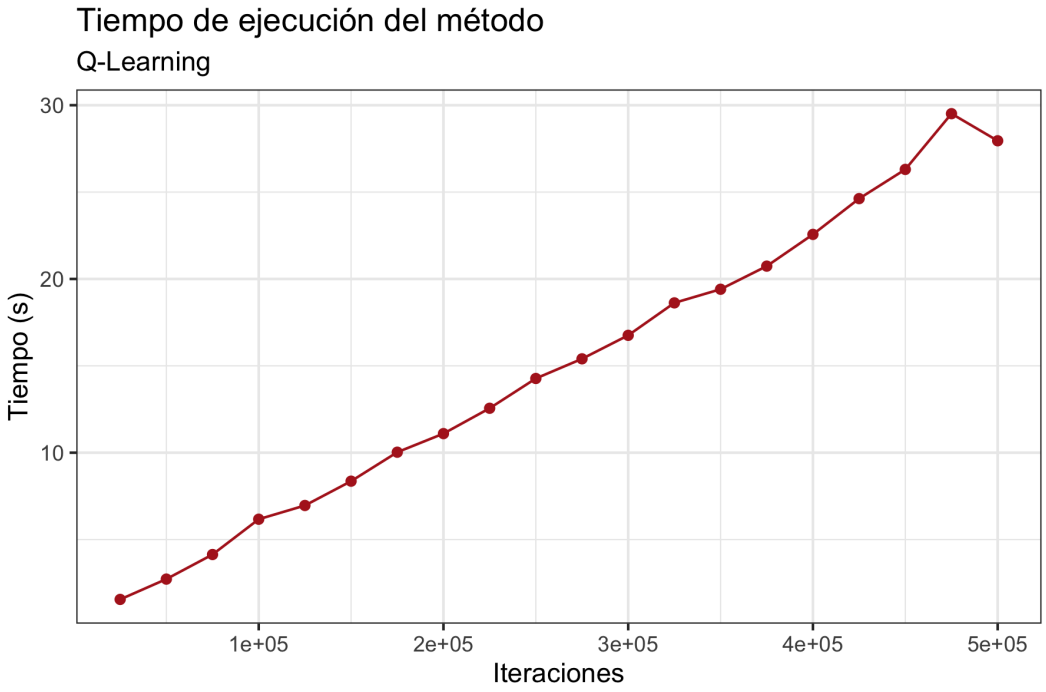


Figura 4: Tiempos de ejecución del método Q-Learning.

Como podemos ver, parecer ser que el algoritmo crece de forma lineal, es decir, concluimos empíricamente que el método tiene orden $\mathcal{O}(n)$.

7. Conclusiones

Logramos implementar el método de aprendizaje por refuerzo de Q-learning en `R`, así como el algoritmo de PRIM modificado para la creación de laberintos.

El método de Q-Learning presenta una manera aceptable de tratar el problema de la resolución de los laberintos. En tan solo 2.8 segundos, y después de $50,000$ iteraciones, logra aprender satisfactoriamente el ambiente de un laberinto de 15×15 y llega a la solución correcta. Pudimos observar que el orden de complejidad de este algoritmo es lineal.

En cuanto a la implementación de un método de Monte Carlo, no resultó ser muy exitosa. Es muy probable que exista un error en el código, pues aún cuando la teoría nos dice que el método es lento (y peor que Q-Learning), el desempeño del método es demasiado malo. A partir del laberinto de tamaño 9×9, el algoritmo no logró llegar a la solución, incluso después de $5,000,000$ de iteraciones (el laberinto pasado requirió de tan sólo $10,000$). Además, el tensor Q no parece ser correcto, al inspeccionarlo cuidadosamente. Sin embargo, presentamos la teoría de Monte Carlo, junto con los pseudocódigos, esperando poder corregir el código en un futuro y así lograr una comparación adecuada entre los métodos.

Referencias

[1] Sutton, R. S., and Barto, A. G.: *Reinforcement learning: An introduction*. The MIT Press, Cambridge, MA, 2018.

[2] BitTickler (https://stackoverflow.com/users/2225104/bittickler), Implementing a randomly generated maze using Prim's Algorithm, URL (consultado: 14-12-2020): https://stackoverflow.com/questions/29739751/implementing-a-randomly-generated-maze-using-prims-algorithm

[3] Soper, Daniel. "Foundations of Reinforcement Learning.", YouTube, publicado 7 de abril del 2020, URL (consultado: 14-12-2020)) www.youtube.com/watch?v=wVXXLLT6srY.

[4] Soper, Daniel. "Foundations of Q-Learning.", YouTube, publicado 22 de abril del 2020, URL (consultado: 14-12-2020) https://www.youtube.com/watch?v=__t2XRxXGxI.

[5] Soper, Daniel. "Notebook for Topic 08 Video - Q-Learning - A Complete Example in Python.ipynb." Google Drive, URL (consultado: 14-12-2020) colab.research.google.com/drive/1E2RViy7xmor0mhqskZV14_NUj2jMpJz3.

Apéndice: código principal

```
---
title: "Aprendizaje por Refuerzo"
author: "Pablo Barranco,Thomas Bladt, Alejandro De Anda"
date: "14/12/2020"
output: html_document
---

```{r setup, include=FALSE}
knitr::opts_chunk$set(echo = TRUE)
rm(list=ls())
source("prim_final.R")
```

C digo para encontrar la ruta m s corta entre dos puntos
Aplicado para encontrar la soluci n a un laberinto
Basado en explicaci n y c digo en Python de Dr. Daniel Soper:
https://www.youtube.com/watch?v=iKdlKYG78j4
14 de diciembre del 2020
Pablo Barranco, Thomas Bladt, Alejandro De Anda

Primero generamos un laberinto con el algoritmo de PRIM y creamos el
    ambiente del "juego".

Definimos funci n para crear matriz de recompensas y graficarla.
```{r}
creaRecompensas <- function(lab,rmax){
  recompensas <- lab
  recompensas[recompensas == 0] <- -100
  recompensas[recompensas == 1] <- -1
  recompensas[2,ncol(lab)] <- rmax
  recompensas[nrow(lab)-1,1] <- -1
  return(recompensas)
}

plotRecompensas <- function(rec){
  rec[2,ncol(rec)] <- 100
  plot(rec, col = c("black","white","red","green"), key = NULL,breaks =
     NULL,axis.col=NULL, axis.row=NULL,xlab = "", ylab="", main = paste0
    ("Laberinto de ", nrow(rec), " x ",ncol(rec)))
}
```

Generamos un laberinto de tama o $m \times n$.
```{r}
# Tama o del laberinto (m x n)
```
```

```
m <-9
n <- 9

set.seed(2022)
laberinto <- genera_laberinto(m,n,0)

recompensas <- creaRecompensas(laberinto,10000)

# Inicializamos matriz Q
q_val <- array(0, dim = c(m,n,4))

# Ploteo laberinto
#png("lab_15.png",width = 1800, height = 1300,res=250)
plotRecompensas(recompensas)
#dev.off()
```

Definimos las funciones ocupadas

```{r}
es_terminal <- function(fila,columna){
  ifelse(recompensas[fila,columna] == -1,FALSE,TRUE)}

inicio_aleatorio <- function(){
  fila <- sample(1:m,size = 1)
  columna <- sample(1:n,size = 1)
  while(es_terminal(fila,columna)){
    fila <- sample(1:m,size = 1)
    columna <- sample(1:n,size = 1)
  }
  return(c(fila,columna))
}

sig_accion <- function(fila,columna,epsilon){
  if(runif(1) < epsilon )
    return(which.max(q_val[fila,columna,]))
  else
    return(sample(1:4,size = 1))
}

sig_lugar <- function(fila,columna,accion){
  if(accion == 1 && fila > 1){fila = fila - 1}
  else
    if(accion == 2 && columna < n){columna = columna + 1}
    else
      if(accion == 3 && fila < m){fila = fila + 1}
      else
```

```
        if(accion == 4 && columna > 1){columna = columna - 1}
  return(c(fila,columna))
}

dist_mas_corta <- function(fila_inicio,columna_inicio){
  if(es_terminal(fila_inicio,columna_inicio)){return("[,]")}
  else
    fila <- fila_inicio
    columna <- columna_inicio
    camino <- paste0("[",fila,",",columna,"]")
    k <- 0

    solucion_df <- data.frame(x= fila_inicio,y = columna_inicio)

    while(!(es_terminal(fila,columna)) && (k < 500)){
      k <-  k+1
      accion <- sig_accion(fila, columna,1)
      aux <- sig_lugar(fila,columna,accion)
      fila <- aux[1]
      columna <- aux[2]
      camino <- append(camino,paste0("[",fila,",",columna,"]"))
      solucion_df[k+1,1] <- fila
      solucion_df[k+1,2] <- columna
    }
    return(list(camino = camino,sol = solucion_df))
}

look <- function(vec,mat){
  resp <- apply(mat, 1, function(x, want) isTRUE(all.equal(x, want)),
   vec)
  if(TRUE %in% resp)
    return(TRUE)
  else
    return(FALSE)
}
```

El algoritmo de Monte Carlo:

```{r}
episodio <- function(epsilon,gamma = 0.9) # corre un episodio de
   iteracion pero con tasa de descuento
{
  bandera = TRUE
  # Obtenemos S_0
  inicio <- inicio_aleatorio()
  fila <- inicio[1]
  columna <- inicio[2]
```

```
135
136   state_action_reward = c(NULL,NULL,NULL)
137
138   while(!(es_terminal(fila,columna))){
139     accion <- sig_accion(fila,columna,epsilon)
140     fila_vieja <- fila
141     columna_vieja <- columna
142     lugar <- sig_lugar(fila,columna,accion)
143     fila <- lugar[1]
144     columna <- lugar[2]
145     recompensa <- recompensas[fila,columna]
146     state_action_reward <- rbind(state_action_reward,c(fila_vieja,
      columna_vieja,accion,recompensa))
147   }
148
149   g = 0
150   G <- NULL
151   R <- apply(t(state_action_reward[,4]),1,rev)
152   for(r in R){
153     g = r + gamma*g
154     G <- append(G,g)
155   }
156   G <- apply(t(G),1,rev)
157   state_action_reward[,4] <- G
158   return(state_action_reward)
159 }
160
161 monte_carlo <- function(env,epsilon,gamma,N_episodes=1000)
162 {
163   # Inicializamos Q,V,PI
164   dim <- dim(env)
165   m <- dim[1]
166   n <- dim[2]
167   q_val <- array(0, dim = c(m,n,4))
168   visit <- array(0, dim = c(m,n,4))
169   rewards <- array(0, dim = c(m,n,4))
170
171
172   for(i in 1:N_episodes){
173
174     #cambiar epsilon
175
176     eps = max(0,epsilon - i/N_episodes)
177     #generamos un episodio
178     episode <- episodio(eps,gamma) #aqui
179     seen_state_action = matrix(nrow=1,ncol=3)
180     k <- dim(episode)[1]
```

```
181
182     for(i in 1:k){
183       if(!look(episode[i,1:3],seen_state_action)){ #Si no esta en los
    estados vistos
184         # Estados
185         fila <- episode[i,1]
186         columna <- episode[i,2]
187         # Accion
188         accion <- episode[i,3]
189         #Recompensa
190         recompensa <- episode[i,4]
191
192         #Actualizacion
193         visit[fila, columna, accion] <- visit[fila, columna, accion] +
    1
194         rewards[fila,columna,accion] <- rewards[fila,columna,accion] +
    recompensa
195         q_val[fila,columna,accion] <- rewards[fila,columna,accion]/
    visit[fila, columna, accion]
196         seen_state_action <- rbind(seen_state_action,episode[i,1:3])
197       }
198     }
199   }
200   return(q_val)
201 }
202
203 ```
204 ```{r}
205
206 tiempos_x <- seq(from = 25000, to = 500000, by = 25000)
207 tiempos_y <- rep(0,20)
208
209 for(i in 1:20){
210   q_val <- array(0, dim = c(m,n,4))
211   tiempos_y[i] <- system.time(q_val <- entrena_qlearn(tiempos_x[i]))[3]
212 }
213 ```
214
215 El algoritmo de Q-Learning:
216 ```{r}
217 entrena_qlearn <- function(num_entren,q = q_val){
218
219   epsilon <- 0.9
220   descuento <- 0.9      #gamma
221   tasa_aprendizaje <- 0.9   #alpha
222
223   for(i in 1:num_entren){
```

```
    inicio <- inicio_aleatorio()
    fila <- inicio[1]
    columna <- inicio[2]

    while(!(es_terminal(fila,columna))){
      accion <- sig_accion(fila,columna,epsilon)
      fila_vieja <- fila
      columna_vieja <- columna
      lugar <- sig_lugar(fila,columna,accion)
      fila <- lugar[1]
      columna <- lugar[2]
      recompensa <- recompensas[fila,columna]
      viejo_q_val <- q_val[fila_vieja,columna_vieja,accion]
      dif <- recompensa + descuento*max(q_val[fila,columna,])-viejo_q_
    val

      nuevo_q_val = viejo_q_val + (tasa_aprendizaje*dif)
      q_val[fila_vieja, columna_vieja, accion] = nuevo_q_val
    }
  }

  return(q_val)
}
```

```{r}
#entrena
q_val <- array(0, dim = c(m,n,4))
system.time(q_val <- entrena_qlearn(100000))

#Resuelve para la ruta
(solucion <- dist_mas_corta(m-1,1))

#coloreaa
colorea <- function(rec,sol){
  aux <- list()
  for(i in 1:nrow(sol)){
    aux[[i]] <- rec
    rec[sol[i,1],sol[i,2]] <- 100
  }

  saveGIF({
    for (i in 1:nrow(sol)) plot( aux[[i]],col = c("black","white","
    green"),key = NULL,breaks = NULL,axis.col=NULL, axis.row=NULL,xlab =
    "", ylab="", main = "Soluci n del laberinto")
  },interval = 0.25,movie.name = "solucion1.gif")
}
```

```
#guardas gif coloreado
colorea(recompensas,solucion$sol)
recompensas[2,49] <- 100
```

```{r}
#gr fica de complejidad
tiempos <- data.frame(x= tiempos_x, y = tiempos_y)

ggsave("tiempo_q.png",width = 6, height = 4)
ggplot(data = tiempos, aes(x= x, y = y))+
  geom_line(color = "firebrick")+
  geom_point(color = "firebrick")+
  labs( x= "Iteraciones", y = "Tiempo (s)", title= "Tiempo de
    ejecuci n del m todo", subtitle = "Q-Learning")+
  theme_bw()
dev.off()
```

Apéndice: código PRIM

```
#
    ##############################################################################

## C digo para generar laberintos utilizando el algoritmo de PRIM
    aleatorio
## Basado en el pseudoc digo encontrado en:
## https://stackoverflow.com/questions/29739751/implementing-a-randomly
    -generated-maze-using-prims-algorithm
## 19 de noviembre del 2020
## Pablo Barranco, Thomas Bladt, Alejandro De Anda
#
    ##############################################################################


# Librerias necesarias
    ----------------------------------------------------
library(animation)
library(plot.matrix)

# Funciones auxiliares
    ------------------------------------------------------------------

# Crea la base del laberinto de mxn
creaGrid <- function(num_fil,num_col){
  grid_lab <- matrix(0, nrow = num_fil,ncol = num_col)
  return(grid_lab)
}

# Selecciona una cela aleatoriamente
celdaAleatoria <- function(num_fil,num_col){
  fila    <- sample(2:num_fil-1,size = 1, replace = F)
  columna <- sample(2:num_col-1,size = 1, replace = F)
  return(c(fila,columna))
}

# Checa si determinada celda cae dentro del laberinto
esLegal <- function(fila,columna, lab){
  num_fil <- nrow(lab)
  num_col <- ncol(lab)
  return( (fila > 1) &&(fila < num_fil) && (columna > 1)&& (columna <
   num_col) )
}

# Calcula la frontera de una celda espec fica
# Frontera: celdas a distancia 2 de tipo "cerradas"
```

```
frontera <- function(fila,columna,lab){
  #Inicializamos un data frame
  frontera_df <-  data.frame(x = integer(), y = integer())

  #Consideramos los 4 puntos posibles
  if(esLegal(fila+2,columna,lab)){
    if(lab[fila+2,columna] == 0)
      frontera_df <- rbind(frontera_df,data.frame(x= fila+2, y =
   columna))
  }
  if(esLegal(fila-2,columna,lab)){
    if(lab[fila-2,columna] == 0)
      frontera_df <- rbind(frontera_df,data.frame(x= fila-2, y =
   columna))
  }
  if(esLegal(fila,columna+2,lab)){
    if(lab[fila,columna+2] == 0)
      frontera_df <-rbind(frontera_df,data.frame(x= fila, y = columna
   +2))
  }
  if(esLegal(fila,columna-2,lab)){
    if(lab[fila,columna-2] == 0)
      frontera_df <-rbind(frontera_df,data.frame(x= fila , y = columna
   -2))
  }

  return(frontera_df)
}

# Calcula las celdas vecinas a una espec fica
# Vecina: celdas a distancia 2 de tipo "abierta"
vecinos <- function(fila,columna,lab){
  vecinos_df <-  data.frame(x = integer(), y = integer())

  if(esLegal(fila+2,columna,lab)){
    if(lab[fila+2,columna] == 1)
      vecinos_df <- rbind(vecinos_df,data.frame(x= fila +2, y = columna
   ))
  }
  if(esLegal(fila-2,columna,lab)){
    if(lab[fila-2,columna] == 1)
      vecinos_df <-rbind(vecinos_df,data.frame(x= fila-2, y = columna))
  }
  if(esLegal(fila,columna+2,lab)){
    if(lab[fila,columna+2] == 1)
      vecinos_df <-rbind(vecinos_df,data.frame(x= fila, y = columna+2))
  }
```

```
  if(esLegal(fila,columna-2,lab)){
    if(lab[fila,columna-2] == 1)
      vecinos_df <-rbind(vecinos_df,data.frame(x= fila , y = columna-2)
   )
  }
  return(vecinos_df)
}

# Funci n principal
   ------------------------------------------------------------

genera_laberinto <- function(m,n,opcion){
#opcion 1 = return plotss, 0 = return laberinto
#set.seed(2025)

# Creamos un grid de m x n
num_fil <- m
num_col <- n
laberinto <- creaGrid(num_fil,num_col)

# Seleccionamos la celda inicial (puede ser aleatoria tambi n)
#inicio <- celdaAleatoria(num_fil,num_col)
inicio <- c(num_fil-1,2)

# Definimos celda inicial como abierta
laberinto[inicio[1],inicio[2]] <- 1

# Inicializamos lista de la frontera en un dataframe
listaFrontera <- data.frame(x = integer(), y = integer())
listaFrontera <- rbind(listaFrontera,frontera(inicio[1],inicio[2],
   laberinto))

# Empezamos el contador de iteraciones
k=0

# Lista para guardar las matrices en cada iteraci n (para
   visualizaciones  nicamente )
plotss <- list()

#Mientras no est  vac a la lista
while( nrow(listaFrontera) != 0 ){
  k=k+1
  # Inicializamos lista de vecinos vac a
  vecinos_df <- data.frame(x = integer(), y = integer())

  # Seleccionamos aleatoriamente un elemento de la frontera
  tama oLista <- nrow(listaFrontera)
```

```
122   celda <- sample(1:tama oLista , size = 1, replace = F)
123
124   # Calculamos a los vecinos del elemento frontera
125   vecinos_df <- rbind(vecinos_df, vecinos(listaFrontera[celda,1],
       listaFrontera[celda,2],laberinto))
126
127   # Seleccionamos un vecino aleatoriamente y lo conectamos con la
       frontera
128   if(nrow(vecinos_df) != 0)
129   {
130      vecino <- sample(1:nrow(vecinos_df),size = 1)
131      laberinto[(vecinos_df[vecino,1]+listaFrontera[celda,1])/2,  (
       vecinos_df[vecino,2]+listaFrontera[celda,2])/2] = 1
132      laberinto[listaFrontera[celda,1],listaFrontera[celda,2]] <- 1
133      #laberinto[vecinos_df[vecino,1],vecinos_df[vecino,2]] <- 1
134
135   # Calculamos la frontera de la celda frontera actual y agregamos a la
       lista
136   listaFrontera <- rbind(listaFrontera,frontera(listaFrontera[celda,1],
       listaFrontera[celda,2], laberinto))
137
138   # Quitamos la frontera anterior de la lista
139   listaFrontera <-  listaFrontera[-celda,]
140
141   # Cuidamos que no se repitan elementos en la lista
142   listaFrontera <- unique(listaFrontera)
143   }
144
145   # Guardamos el plot (para animaci n   nicamente  )
146   plotss[[k]] <- laberinto
147 }
148
149 if(opcion == 1){
150
151   # png("plot1_prim.png",width = 1800, height = 1300,res=250)
152   # plot( plotss[[1]],col = c("black","white"),key = NULL,breaks = NULL
       ,axis.col=NULL, axis.row=NULL,xlab = "", ylab="", main = "")
153   # dev.off()
154   #
155   # png("plot2_prim.png",width = 1800, height = 1300,res=250)
156   # plot( plotss[[floor(k/2)]],col = c("black","white"),key = NULL,
       breaks = NULL,axis.col=NULL, axis.row=NULL,xlab = "", ylab="", main
       = "" )
157   # dev.off()
158   #
159   # png("plot3_prim.png",width = 1800, height = 1300,res=250)
160   # plot( plotss[[k]],col = c("black","white"),key = NULL,breaks = NULL
```

```
    ,axis.col=NULL, axis.row=NULL,xlab = "", ylab="", main = "")
  # dev.off()
  # En caso de querer guardar un gif de la creaci n
  saveGIF({
    for (i in 1:k) plot( plotss[[i]],col = c("black","white"),key =
   NULL,breaks = NULL,axis.col=NULL, axis.row=NULL,xlab = "", ylab="")
  },interval = 0.01)
  return(plotss)
}
else
  return(laberinto)
}
```