CE1106 - Minesweeper
Paradigmas de Programación
Escuela de Ingeniería en Computadores
Estudiantes:
Eduardo José Canessa Quesada
Luis Felipe Chaves Mena
Deiler Morera Valverde
Fecha: Septiembre 2025

A continuación se documentan los aspectos técnicos del proyecto `BusCEMinas`, una implementación del clásico juego de lógica Buscaminas desarrollado en **Racket** bajo el paradigma de programación funcional. El sistema genera un tablero de dimensiones configurables con un número aleatorio de minas, permitiendo al usuario descubrir celdas y marcar banderas. La aplicación incluye una interfaz gráfica para una interacción amigable, y su lógica se basa en el uso de recursividad y manipulación de listas para revelar celdas vacías y calcular minas adyacentes, cumpliendo así con los principios del paradigma funcional.

1. Algoritmos

A continuación se presentan los algoritmos usados para el desarrollo de la tarea. Cada diagrama de flujo se encuentra en su sección respectiva dentro del documento.

1.1. Generación del Tablero con Minas Aleatorias

Propósito: Genera la matriz completa del tablero de juego, asignando minas en posiciones aleatorias y calculando los números de minas adyacentes para cada celda.

```
(define (generar-tablero filas columnas num-minas)
  ; 1. Generar todas las coordenadas posibles del tablero
  (define todas-coordenadas (generar-coordenadas filas columnas))

  ; 2. Seleccionar aleatoriamente las posiciones de las minas
  (define posiciones-minas
    (tomar-primeros (mezclar-lista todas-coordenadas) num-minas))

  ; 3. Funcion recursiva para construir el tablero fila por fila
  (define (generar-filas fila-actual acumulador)
    (cond
      ; Caso base: cuando se procesaron todas las filas
      [(>= fila-actual filas) (invertir-lista acumulador)]

      ; Caso recursivo: generar una fila y continuar con la siguiente
      [else
       (generar-filas
        (+ fila-actual 1)
        (cons
         (generar-fila-campo fila-actual columnas posiciones-minas filas)
         acumulador))]))

  ; 4. Iniciar la recursion con fila 0 y acumulador vacio
  (generar-filas 0 '()))
```

La función `generar-tablero` opera en cuatro etapas claramente definidas: primero genera todas las coordenadas posibles del tablero, luego selecciona aleatoriamente las posiciones donde se ubicarán las minas mediante una mezcla y selección de las primeras N coordenadas. Posteriormente, utiliza una función recursiva interna que construye el tablero fila por fila, donde cada fila es generada individualmente determinando para cada celda si contiene una mina o el número correspondiente de minas adyacentes. Finalmente, al completar todas las filas, invierte el acumulador para devolver la matriz en el orden correcto, resultando en una estructura de datos que representa completamente el tablero de juego listo para su uso.

1.2. Conteo de Minas Adyacentes

Propósito: Obtiene las coordenadas de todas las celdas adyacentes a una posición específica en el tablero, filtrando aquellas que están dentro de los límites del mismo.

```
(define (get-adyacentes fila columna filas columnas)
  ; 1. Definir las 8 posiciones adyacentes posibles
  (define todos-adyacentes
    (list (cons (- fila 1) (- columna 1))  ; Arriba-izquierda
          (cons (- fila 1) columna)        ; Arriba
          (cons (- fila 1) (+ columna 1))  ; Arriba-derecha
          (cons fila (- columna 1))        ; Izquierda
          (cons fila (+ columna 1))        ; Derecha
          (cons (+ fila 1) (- columna 1))  ; Abajo-izquierda
          (cons (+ fila 1) columna)        ; Abajo
          (cons (+ fila 1) (+ columna 1)))) ; Abajo-derecha

  ; 2. Filtrar solo las coordenadas que estan dentro del tablero
  (filter (lambda (coordenada)
            (coordenada-dentro-limites? (car coordenada) (cdr coordenada) filas columnas))
          todos-adyacentes))
```

La función `get-adyacentes` primero define las ocho posiciones vecinas posibles alrededor de una celda específica , creando una lista completa de todas las direcciones adyacentes. Posteriormente, aplica un filtro para eliminar aquellas coordenadas que caen fuera de los límites del tablero, utilizando la función auxiliar `coordenada-dentro-limites?` que verifica si cada par (fila, columna) está dentro del rango válido [0, filas-1] y [0, columnas-1]. El resultado es una lista que contiene únicamente las posiciones adyacentes válidas.

1.3. Revelado de celdas

Propósito: Gestiona la lógica de descubrimiento de celdas en el tablero, además de las condiciones de victoria/derrota.

```
(define (descubrir-campo fila columna valor)
  (when (eq? (obtener-estado-celda fila columna) 'cubierta)
    ; Marcar celda como descubierta y actualizar interfaz
    (establecer-estado-celda! fila columna 'descubierta)
    (send (obtener-lienzo-campo-minas fila columna) refresh)

    ; Incrementar contador y verificar victoria
    (set! estado-juego-actual (incrementar-contador-campos-descubiertos estado-juego-actual))
    (when (= (get-campos-desc estado-juego-actual)
             (- (* (unbox alto) (unbox ancho)) (unbox minas)))
      (ganar))))

; Decide que accion tomar segun el contenido de la celda
(define (intentar-descubrir fila columna)
  (define valor (obtener-valor-celda fila columna (get-campo estado-actual)))
  (cond
    [(equal? valor "X") (perder)]
    [(number? valor) (if (= valor 0)
                         (descubrir-campos-cero fila columna)
                         (descubrir-campo fila columna valor))]
    [else (perder)]))  ; Error

; Expansion recursiva de areas vacias
(define (descubrir-campos-cero fila col [visitados (set)])
  (define actual (cons fila col))
  (when (not (visitada? actual visitados))
    (define nuevos-visit (agregar-visitada visitados actual))
    (define valor (obtener-valor-celda fila col (get-campo estado-actual)))

    (when (and (cubierta? fila col) (number? valor))
      (descubrir-campo fila col valor)  ; Descubrir celda actual
      (when (= valor 0)  ; Si es cero, expandir a vecinos
        (for ([vecino (get-vecinos fila col (unbox alto) (unbox ancho))])
          (descubrir-campos-cero (car vecino) (cdr vecino) nuevos-visit))))))
```

Este conjunto de funciones son una parte fundamental de la interacción del jugador con el tablero. `intentar-descubrir-campo` verificando si la celda seleccionada contiene una mina (derrota inmediata) o un número. Si el valor es cero, se activa `descubrir-campos-cero` que realiza una expansión recursiva todas las celdas vacías conectadas. La función `descubrir-campo` se encarga de la lógica individual de descubrimiento, actualizando el estado visual del tablero y verificando si se ha alcanzado la condición de victoria (todas las celdas sin minas descubiertas).

1.4. Estado de juego

Propósito: Gestiona el estado interno del juego usando una estructura de datos funcional.

```
; Configuracion del juego (se establecen desde el menu)
(define ancho-tablero 0)     ; Columnas del tablero
(define alto-tablero 0)      ; Filas del tablero
(define num-minas 0)         ; Total de minas

; Estado del juego: par (campo-minas . campos-descubiertos)
(define init-estado (cons null 0))

; Accesores del estado
(define (get-estado-minas estado) (car estado))         ; Obtener matriz del tablero
(define (get-estado-minas-desc estado) (cdr estado))    ; Obtener contador de descubiertas

; Modificadores del estado (inmutables)
(define (set-estado-val-minas estado valores) (cons valores (cdr estado)))
(define (set-estado-minas-desc estado contador) (cons (car estado) contador))

; Operaciones especificas
(define (reiniciar-contador estado) (cons (car estado) 0))           ; Reset a cero
(define (incrementar-contador estado) (cons (car estado) (+ (cdr estado) 1))) ; +1
```

Sistema que mantiene el estado del juego usando un par `cons` donde el `car` almacena la matriz del tablero y el `cdr` lleva la cuenta de celdas descubiertas.

1.5. Generación de Coordenadas

Propósito: Creación de la matriz del tablero con minas distribuidas aleatoriamente.

```
(define (generar-tablero filas columnas num-minas)
  ; 1. Crear todas las coordenadas y seleccionar minas aleatoriamente
  (define todas-coordenadas (generar-coordenadas filas columnas))
  (define posiciones-minas (tomar-primeros (mezclar-lista todas-coordenadas) num-minas))

  ; 2. Construir tablero fila por fila usando recursion
  (define (generar-filas fila-actual acumulador)
    (cond
      [(>= fila-actual filas) (invertir-lista acumulador)]  ; Caso base: terminar
      [else  ; Caso recursivo: generar fila y continuar
       (generar-filas (+ fila-actual 1)
                      (cons (generar-fila-campo fila-actual columnas posiciones-minas filas)
                            acumulador))]))

  ; 3. Iniciar recursion desde fila 0
  (generar-filas 0 '()))
```

Función que crea el tablero completo al generar coordenadas, seleccionar minas aleatoriamente y construir la matriz procesando cada fila.

1.6. Mezcla de aleatoria de lista

Propósito: De acuerdo con la recomendación de `AI` se implementa el algoritmo Fisher-Yates para mezclar una lista aleatoriamente.

```
(define (mezclar-lista elementos)
  (define (iterar elementos-restantes resultado)
    (cond
      [(null? elementos-restantes) resultado]  ; Caso base: lista vacia
      [else
       ; 1. Seleccionar elemento aleatorio
       (define indice (random (length elementos-restantes)))
       (define elemento (list-ref elementos-restantes indice))

       ; 2. Remover elemento seleccionado de la lista
       (define nuevos-restantes (append
         (tomar-primeros elementos-restantes indice)
         (eliminar-primeros elementos-restantes (+ indice 1))))

       ; 3. Llamada recursiva con elemento agregado al resultado
       (iterar nuevos-restantes (cons elemento resultado))]))

  ; Llamada inicial con lista completa y resultado vacio
  (iterar elementos '()))
```

En cada paso selecciona un elemento aleatorio, lo remueve de la lista y lo agrega al resultado, repitiendo hasta vaciar la lista original.

2. Estructuras de datos desarrolladas

Estructura	Módulo	Función Principal	Tipo
Estado del Juego	Lógico	`init-estado`	Par cons
Estados Visuales	GUI	`inicializar-estados-celdas!`	Vector
Tablero de Minas	Lógico	`generar-tablero`	Lista de listas
Lienzos Celdas	GUI	`lienzo-celda-mina%`	Clase objeto
Coordenadas	Lógico	`generar-coordenadas`	Pares cons
Adyacentes	Lógico	`get-adyacentes`	Lista de pares
Sprites	GUI	`inicializar-sprites!`	Bitmaps

2.1. Ejemplo de uso

2.2. Estado Completo del Juego

```
; Estado lógico:
(cons '((0 1 "X" 0 2 1 0 0 1)
        (1 2 "X" 1 2 "X" 1 1 1)
        ("X" 2 1 1 2 2 2 1 "X")
        ...)
      15) ; 15 celdas descubiertas
; Donde 0 representa cubierta
; Numeros del 1-8 representa minas adjacentes
; X representa una mina en el tablero
; Estados visuales correspondientes (GUI):
#('cubierta 'descubierta 'cubierta 'bandera 'cubierta ...)
```

2.3. Flujo de Datos entre Módulos

1. **Lógico**: `generar-tablero` → crea estructura de minas y números
2. **GUI**: `inicializar-estados-celdas!` → crea vector de estados visuales
3. **Interacción**: Usuario clickea → `establecer-estado-celda!` actualiza estado
4. **Sincronización**: GUI usa `obtener-valor-campo` para renderizar valores
5. **Contador**: `incrementar-contador-campos-descubiertos` lleva cuenta
6. **Victoria**: `ganar-juego` cuando se descubren todas las celdas seguras

2.4. Correspondencia de Nombres entre Módulos

Módulo Lógico	Módulo GUI
`init-estado`	`estado-juego-actual`
`generar-tablero`	`nuevo-juego` (lo llama)
`obtener-valor-campo`	`intentar-descubrir-campo` (lo consulta)
`get-adyacentes`	`descubrir-campos-cero` (lo usa)
`incrementar-contador...`	`descubrir-campo` (lo actualiza)

3. Estructuras de Datos Clave

3.1. Vector de Estados Visuales

- **Tipo**: Vector unidimensional
- **Índice**: `(+ (* fila ancho-tablero) columna)`
- **Valores**: `'cubierta`, `'bandera`, `'descubierta`
- **Función**: Controlar qué ve el usuario en cada celda

3.2. Lista de Listas del Tablero

- **Tipo**: Lista anidada (matriz)
- **Valores**: Números (0-8) o "X"para minas
- **Función**: Representar la configuración real del juego

3.3. Par Cons del Estado

- **Tipo**: Par (cons) inmutable
- **Partes**: `(tablero . contador-descubiertas)`
- **Función**: Estado global del juego para lógica pura

3.4. Sistema de Sprites

- **Tipo**: Bitmaps escalados
- **Elementos**: Números 0-8, mina, bandera, cubierta
- **Función**: Representación visual responsive

4. Funciones implementadas

4.1. Configuración de Tamaño de tablero

Se implementó la funcionalidad que permite al usuario ingresar el tamaño deseado para la cuadrícula del juego a través de campos de texto en el menú principal.

```
(define (iniciar-juego-desde-menu entrada-filas entrada-columnas seleccion-dificultad)
  (define filas-usuario (string->number (send entrada-filas get-value)))
  (define columnas-usuario (string->number (send entrada-columnas get-value)))

  ; Validacion de entrada
  (cond
    [(or (not (number? filas-usuario)) (not (number? columnas-usuario)))
     (send (new message% [parent marco-menu-principal]
                [label "Por favor ingrese numeros validos"]) show #t)]
    [(or (<= filas-usuario 0) (<= columnas-usuario 0))
     (send (new message% [parent marco-menu-principal]
                [label "Las filas y columnas deben ser > 0"]) show #t)]
    [(> (* filas-usuario columnas-usuario) 400)
     (send (new message% [parent marco-menu-principal]
                [label "Cuadricula demasiado grande (max 400 celdas)"]) show #t)]
    [else
     ; Calcular porcentaje de minas
     (define porcentaje-usuario
       (cond [(= (send seleccion-dificultad get-selection) 0) 0.10]
             [(= (send seleccion-dificultad get-selection) 1) 0.15]
             [else 0.20]))
     (define total-celdas (* filas-usuario columnas-usuario))
     (set-box! num-minas (max 1 (inexact->exact (floor (* total-celdas porcentaje-usuario)))))
     (set-box! ancho-tablero columnas-usuario)
     (set-box! alto-tablero filas-usuario)
     (send marco-menu-principal show #f)
     (crear-interfaz-grafica)]))
```

4.2. Selección de dificultad

Se implementó un sistema de tres niveles de dificultad (Fácil, Medio, Difícil) que determina el porcentaje de minas en el tablero. Cada nivel corresponde a un porcentaje específico: 10 %, 15 % y 20 % respectivamente.

```
(define (iniciar-juego-desde-menu entrada-filas entrada-columnas seleccion-dificultad)
...
(define porcentaje-usuario
(cond [(= (send seleccion-dificultad get-selection) 0) 0.10]
      [(= (send seleccion-dificultad get-selection) 1) 0.15]
      [else 0.20]))
(define total-celdas (* filas-usuario columnas-usuario))
(set-box! num-minas (max 1 (inexact->exact (floor (* total-celdas porcentaje-usuario)))))
```

4.3. Reinicio y volver al menú principal

Después de cumplirse una condición ya sea de victoria o derrota, se muestra un menú que presenta al usuario con la opción de reiniciar el juego o volver al menú principal.

```
(define (nuevo-juego)
  (set! juego-terminado? #f)
  (inicializar-estados-celdas!)

  ; Inicializar estado del juego con nuevo campo de minas
  (set! estado-juego-actual
        (set-estado-val-minas
         (reiniciar-contador-campos-descubiertos init-estado)
         (generar-tablero (unbox alto-tablero)
                          (unbox ancho-tablero)
                          (unbox num-minas))))

  (for ([fila (in-range (unbox alto-tablero))])
    (for ([columna (in-range (unbox ancho-tablero))])
      (send (obtener-lienzo-campo-minas fila columna) refresh))))
```

4.4. Lienzo especializado

La clase `lienzo-celda-mina%` es componente gráfico interactivo para cada celda del tablero de buscaminas que responde a clics del mouse y se dibuja según el estado de la celda. Esta clase hereda

de `canvas%` y se especializa en gestionar la representación visual e interacción de una celda individual dentro del tablero de juego. Cada instancia de esta clase corresponde exactamente a una celda en la matriz bidimensional que conforma el campo minado.

5. Problemas Encontrados

1. **Separación de lógica e interfaz:** Al integrar la parte gráfica con la lógica funcional surgieron errores de comunicación entre funciones. Fue necesario depurar y modularizar mejor el código para mantener ambas capas independientes.
2. **Gestión de la recursividad:** La implementación del descubrimiento automático de celdas adyacentes sin minas (valor 0) generó problemas de recursividad infinita en etapas iniciales. Esto obligó a establecer condiciones base más estrictas para garantizar la correcta terminación del algoritmo.
3. **Validación de índices fuera de rango:** Durante la exploración de celdas vecinas, en múltiples ocasiones el programa intentaba acceder a posiciones inexistentes en el tablero, provocando errores de ejecución. Fue necesario implementar verificaciones de límites antes de procesar cada coordenada.
4. **Manejo de estados de las celdas:** Al principio, resultó complejo distinguir entre celdas no descubiertas, descubiertas, marcadas con bandera y aquellas que contenían minas. Esto requirió diseñar una representación de datos más clara para evitar ambigüedades.
5. **Probabilidad y número de minas:** Ajustar la generación aleatoria de minas conforme a los porcentajes de dificultad establecidos (10 %, 15 %, 20 %) presentó inconsistencias, ya que en ocasiones el número de minas no correspondía exactamente al esperado. Se implementaron correcciones para redondear y validar el total generado.
6. **Limitaciones del paradigma funcional:** La restricción de no usar variables imperativas ni estructuras típicas como `let`, `map` o `apply` obligó a replantear soluciones aparentemente simples, requiriendo un mayor nivel de abstracción y planeamiento en el diseño de funciones.

6. Problemas sin solución

1. **Optimización en tableros grandes:** No se encontró una solución ya implementada en repositorios públicos ni en controladores de versiones que resolviera de manera eficiente el problema de la recursividad en tableros grandes. Por esta razón, se desarrolló una solución propia, aunque con limitaciones de rendimiento en configuraciones máximas (15x15).
2. **Interfaz gráfica avanzada:** No existía un recurso en GitHub u otros repositorios que ofreciera directamente una integración de Racket con funcionalidades avanzadas de interfaz gráfica como temporizador o contador dinámico de banderas. Por tanto, estas características fueron descartadas y se mantuvo una versión básica implementada manualmente.
3. **Persistencia de partidas:** No se encontró una implementación previa que permitiera guardar y cargar partidas en Racket bajo un paradigma puramente funcional. La solución fue intentada desde cero, pero no se logró integrar de manera estable al proyecto final.
4. **Gestión de errores del usuario:** Tampoco se hallaron ejemplos completos en repositorios que resolvieran de forma robusta los casos de clics múltiples o entradas inválidas en la interfaz. Fue necesario crear rutinas propias, aunque algunas validaciones avanzadas quedaron pendientes.

7. Plan de Actividades

Cuadro 1: Plan de Actividades del Proyecto Minesweeper (Planificación Inicial)

No.	Actividad	Responsable(s)	Tiempo	Fecha
1	Investigación inicial: Estudio de Racket y análisis de implementaciones existentes	Todos	2d	08-09-2024
2	Configuración del entorno: Instalación de herramientas y creación de repositorio Git	Luis Chaves	1d	07-09-2024
3	Diseño de lógica básica: Generación de coordenadas y verificación de límites	Deiler Morera	2d	10-09-2024
4	Algoritmo de minas: Generación aleatoria y conteo de minas adyacentes	Eduardo Canessa	3d	12-09-2024
5	Expansión de zonas: Implementación recursiva de áreas seguras	Eduardo Canessa	2d	14-09-2024
6	Sistema de sprites: Carga y escalado de imágenes para celdas	Luis Chaves	3d	16-09-2024
7	Menú principal: Interfaz de configuración y validación	Deiler Morera	2d	17-09-2024
8	Canvas interactivo: Manejo de eventos de ratón y actualización visual	Todos	3d	19-09-2024
9	Detección fin de juego: Lógica de victoria/derrota y pantallas	Eduardo Canessa	2d	20-09-2024
10	Integración completa: Conexión entre lógica e interfaz gráfica	Luis Chaves	2d	22-09-2024
11	Pruebas funcionales: Testing con diferentes configuraciones	Todos	2d	23-09-2024
12	Corrección de errores: Depuración y optimización	Todos	2d	24-09-2024
13	Pruebas cross-platform: Verificación en diferentes sistemas	Deiler Morera	1d	25-09-2024
14	Documentación técnica: Comentarios de código y manual técnico	Todos	2d	26-09-2024
15	Informe final: Redacción de bitácora y resultados	Todos	2d	27-09-2024
16	Entrega final: Preparación de archivos y envío	Todos	1d	28-09-2024

8. Conclusiones

1. **Reafirmación del paradigma funcional:** El desarrollo de la tarea permitió aplicar de manera práctica los principios de la programación funcional, reforzando el uso de recursividad, funciones puras y estructuras de datos inmutables.

2. **Aplicación de estructuras de datos:** La representación del tablero como listas demostró la utilidad de las estructuras de datos en la resolución de problemas complejos, facilitando la manipulación y almacenamiento de información en el juego.

3. **Desarrollo de lógica algorítmica:** La implementación de algoritmos para descubrir celdas, contar minas adyacentes y determinar condiciones de victoria o derrota permitió fortalecer el razonamiento lógico y la capacidad de diseñar soluciones eficientes.

4. **Integración de interfaz gráfica y lógica:** La separación entre la lógica del juego y la interfaz gráfica promovió una mejor organización del código, facilitando el mantenimiento, la depuración y la comprensión del proyecto.

5. **Trabajo en equipo y gestión de tareas:** La planificación de actividades y la documentación técnica fomentaron la colaboración entre integrantes del grupo, demostrando la importancia de la comunicación y la organización en proyectos de programación.

9. Recomendaciones

1. **Optimizar algoritmos recursivos:** En futuras implementaciones se recomienda analizar la eficiencia de las funciones recursivas para evitar redundancias y mejorar el rendimiento en tableros de gran tamaño.

2. **Ampliar funcionalidades:** Se sugiere extender el juego con características adicionales, como niveles personalizados, cronómetro o registro de puntajes, con el fin de enriquecer la experiencia del usuario.

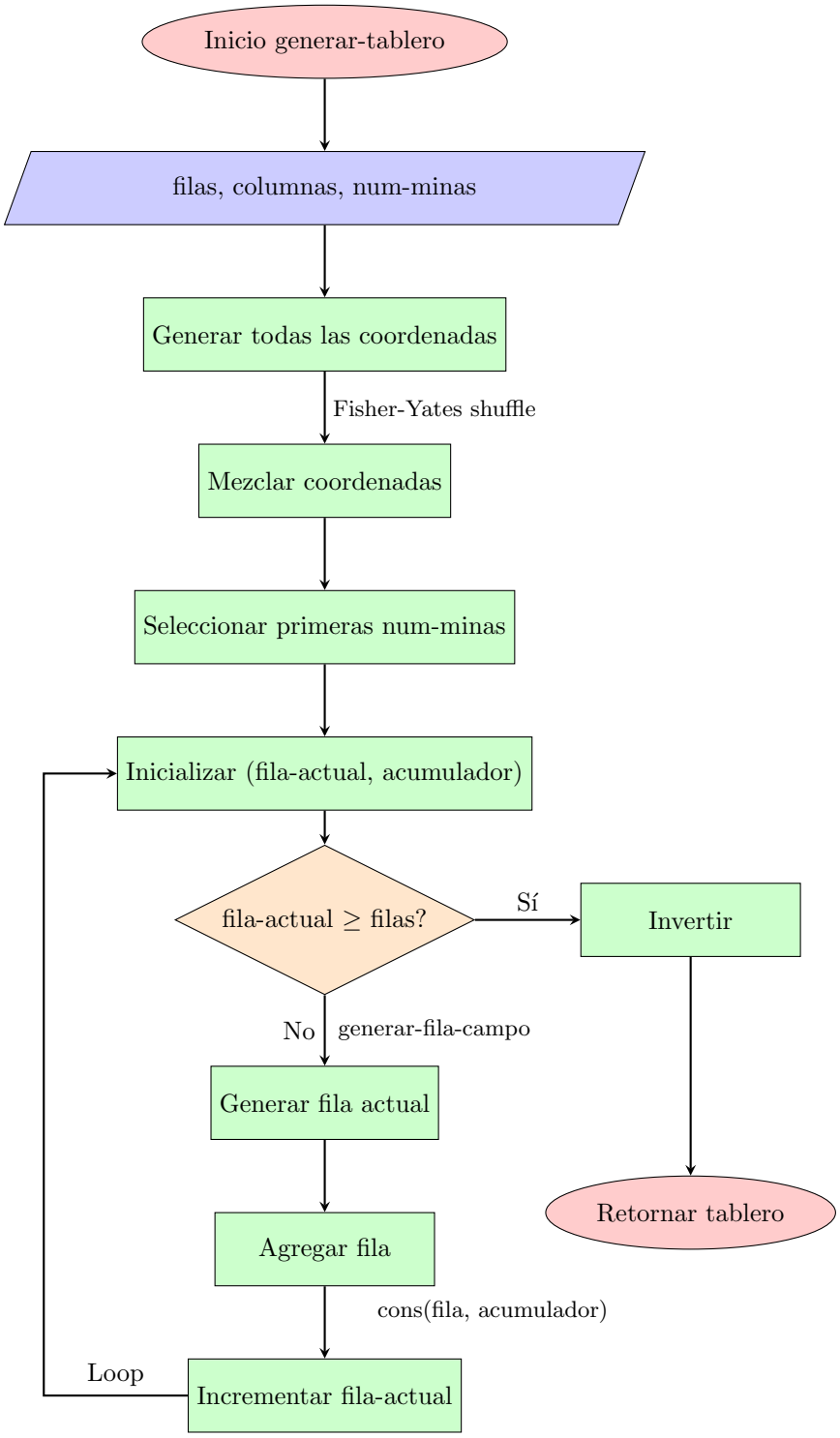
3. **Mejorar la interfaz gráfica:** Una interfaz más intuitiva, con mejores indicadores visuales y mensajes claros, facilitaría la interacción y aumentaría la usabilidad del sistema.

4. **Documentación continua:** Mantener una documentación clara y actualizada durante todo el desarrollo contribuirá a la comprensión del código y permitirá una más fácil colaboración entre integrantes.

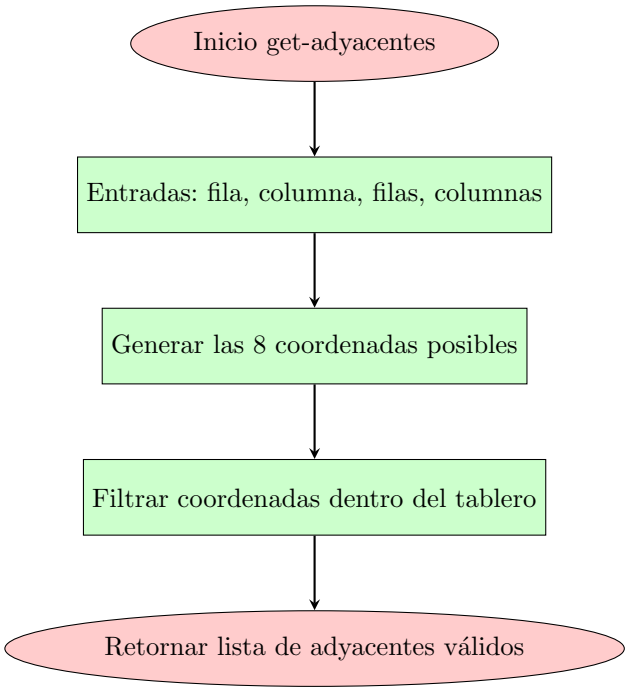
5. **Explorar otros paradigmas:** Aunque el objetivo fue reforzar la programación funcional, sería enriquecedor comparar esta solución con implementaciones en otros paradigmas (imperativo u orientado a objetos) para valorar ventajas y desventajas.

10. Diagramas

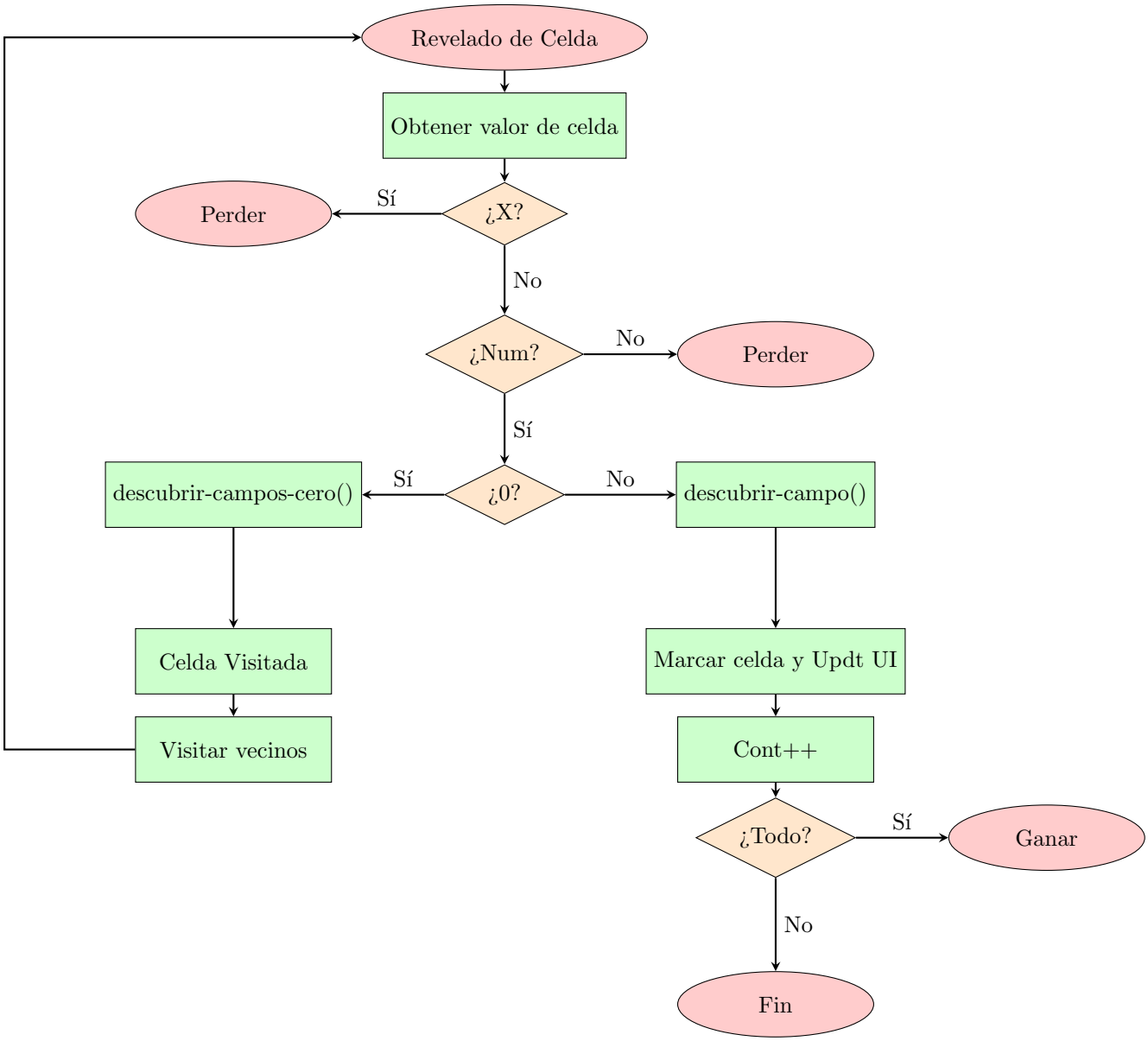
10.1. Gestión de Estado del Juego



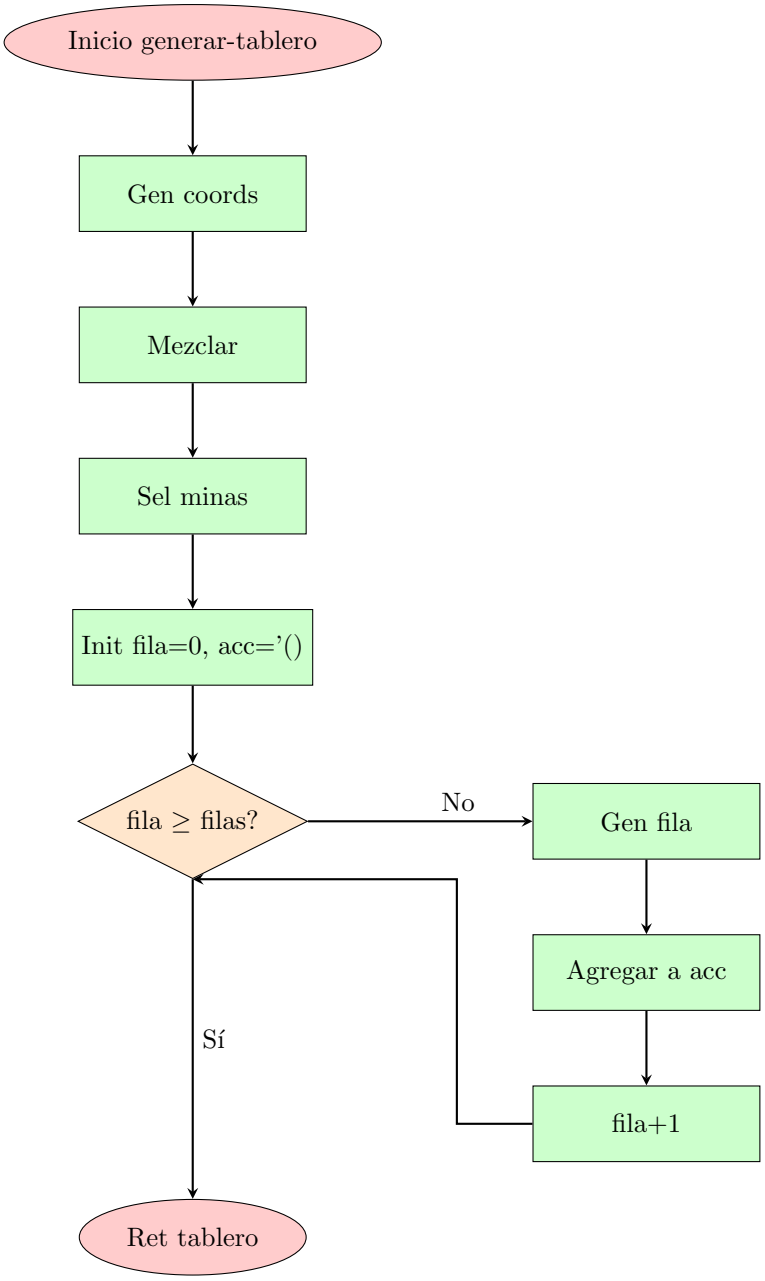
10.2. Conteo de celdas adyacentes



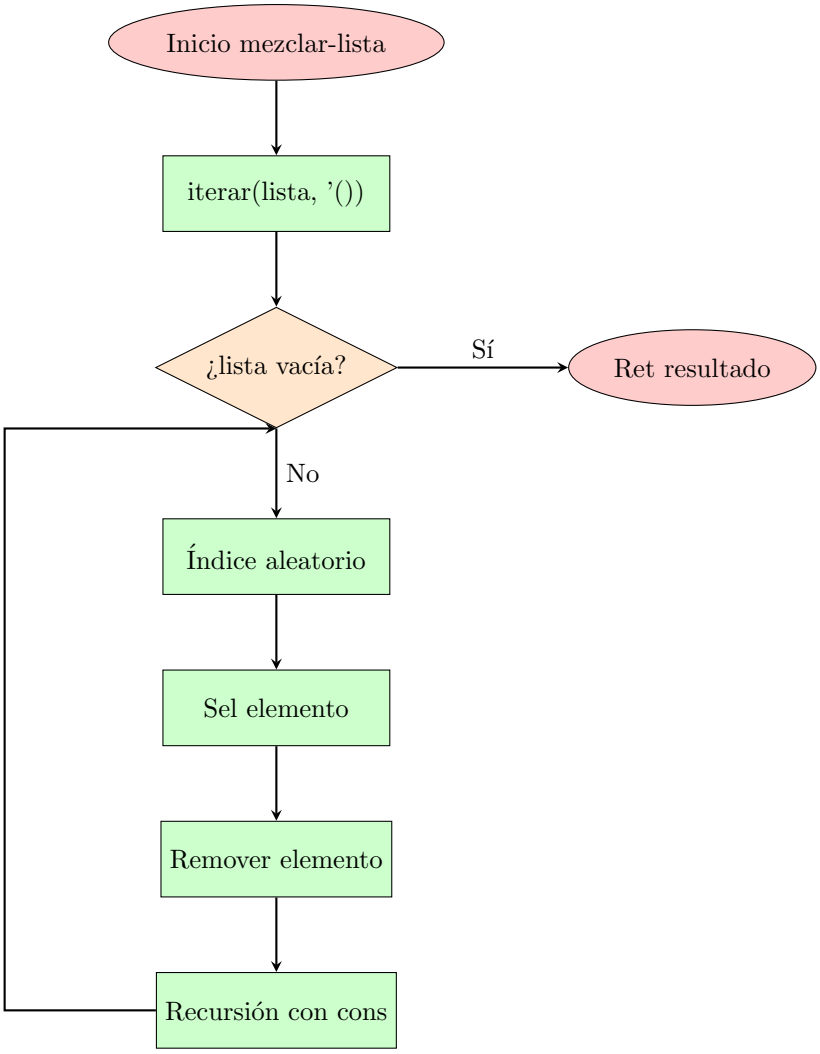
10.3. Revelado de Celdas



10.4. Generación de coordenadas



10.5. Mezcla aleatoria de lista



11. Bibliografía

Referencias

[1] Racket Documentation. *The Racket Guide*. Disponible en: `https://docs.racket-lang.org/`

[2] Abelson, H., & Sussman, G. J. (1996). *Structure and Interpretation of Computer Programs*. MIT Press.

[3] Hutton, G. (2016). *Programming in Haskell*. Cambridge University Press.

[4] Bird, R. (2010). *Thinking Functionally with Haskell*. Cambridge University Press.

[5] Knuth, D. E. (1997). *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*. Addison-Wesley.