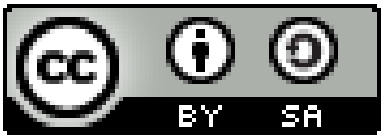
Análisis y Diseño de Algoritmos II

Trabajo Práctico Especial

Mummy Maze

Cursada 2019

Consultas y sugerencias: laboratorio.ayda@alumnos.exa.unicen.edu.ar



Introducción

El juego Mummy Maze está compuesto de una serie de escenarios o laberintos, en los cuales hay que guiar al explorador hasta encontrar la salida de los mismos. Cada escenario contiene paredes que constituyen obstáculos para el movimiento, así como enemigos (momias y escorpiones) que intentarán evitar que el explorador llegue a la salida del laberinto.

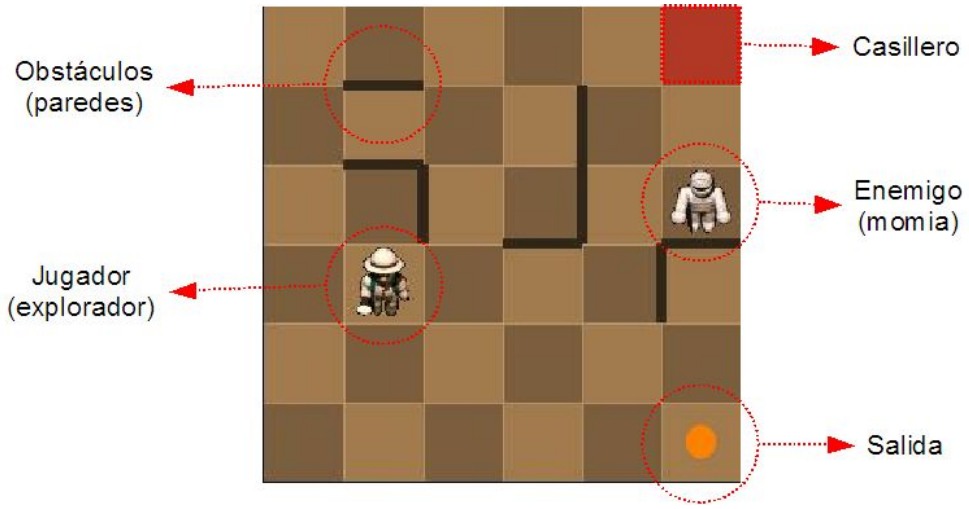


Diagrama mostrando los elementos que forman parte de los distintos laberintos.

La versión del juego que utilizaremos tiene las siguientes características:

El tamaño del laberinto de 6 casilleros de ancho por 6 casilleros de alto.

Los posibles movimientos de todos los personajes son hacia casilleros contiguos; en las direcciones hacia arriba, abajo, derecha e izquierda. Adicionalmente los personajes pueden quedarse en el lugar. En el caso del explorador esto es una acción que se puede elegir ("hacer nada"); para los enemigos esto sólo es posible cuando se encuentran bloqueados por alguna pared.

La dinámica del juego se da en turnos alternados entre los personajes:

El orden es primero el explorador, luego los escorpiones y finalmente las momias, uno a la vez.

Durante su turno cada personaje debe optar por realizar un movimiento válido o bien no moverse y pasar el turno.

Tanto la momia como el escorpión tienen una lógica determinística para realizar sus movimientos. Los enemigos siguen las mismas reglas para moverse:

- Cuando pueden moverse horizontalmente para acercarse al explorador, realizarán ese movimiento primero.
- En caso de no poder moverse horizontalmente, y contar con la posibilidad de hacerlo en forma vertical, realizarán ese movimiento para acercarse al explorador.

En cada turno tanto el explorador como el escorpión pueden moverse como máximo un casillero; mientras que la momia puede realizar como máximo dos movimientos por turno.

Tanto el explorador como los enemigos ocupan un casillero a la vez; no puede haber más de un personaje por casillero al mismo tiempo. Las reglas para resolver la situación que se da cuando un personaje se mueve a un casillero ocupado son:

- Cuando el explorador se mueve a un casillero con cualquier enemigo se pierde el juego.
- Cuando cualquier enemigo se mueve al casillero donde se encuentra el explorador también se pierde el juego.
- Cuando se genera una colisión de la momia con otro enemigo (momia o escorpión) sobrevive la momia.

El juego se gana sólo cuando el explorador ocupa el casillero marcado como la salida.

Fundamentos teóricos

La propuesta para resolver el problema es utilizar el concepto de grafo para modelar la dinámica del juego. De esta forma cada configuración o estado del laberinto se considerará un vértice. Luego, teniendo en cuenta que los movimientos de los enemigos son determinísticos, consideramos que sólo los movimientos válidos que puede realizar el explorador, a partir de un estado, constituirán los arcos hacia nuevos estados. La configuración de los nuevos estados será la obtenida a partir de procesar el movimiento del explorador y cada uno de los movimientos que realizan los enemigos durante sus turnos, hasta que el explorador pueda moverse otra vez.

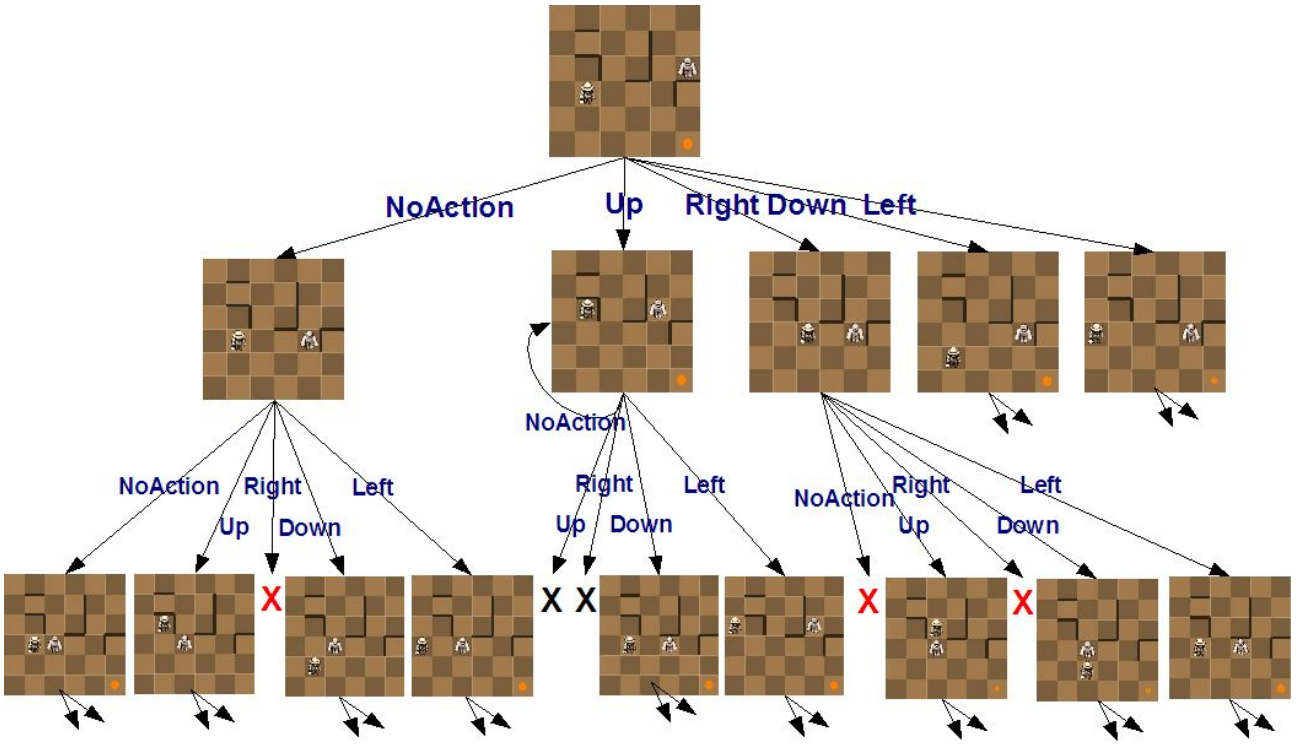


Diagrama de ejemplo mostrando el grafo de búsqueda, donde las configuraciones del laberinto luego de los movimientos posibles del explorador constituyen los nodos y las acciones que realiza el mismo son los arcos.

Una vez que conseguimos modelar con un grafo el desarrollo del juego, vemos que es posible aplicar los algoritmos clásicos de recorrido (DFS, BFS) adaptados a la búsqueda del estado objetivo (es decir una configuración del laberinto donde el explorador se encuentra en la salida). Esta adaptación lleva a lo que se conoce como algoritmos de búsqueda por fuerza bruta, ya que exploran sistemáticamente los

estados, generando los mismos en el orden que define cada tipo de recorrido. Vemos entonces que la base de estas estrategias de búsqueda son los recorridos simples sobre grafos a los cuales se les agrega una evaluación para detenerse en el momento en que se alcanza el estado objetivo.

```
DFS(EstadoJuego E, Solución S):
      Si E es el estado objetivo
            Devolver la solución con los pasos ejecutados hasta el momento
      Sino
            Si E no fue visitado anteriormente
                  Marcar a E como visitado
                  Para cada movimiento válido, M, del explorador en E
                        Generar el nuevo estado E' ejecutando M y procesando los
                        turnos de los enemigos
                        Agregar M a S
                        DFS(E', S)
                        Si se encontró solución
                              Terminar y devolver la solución
                        Sino
                              Volver el estado actual a E, y sacar a M de S
```

Pseudocódigo de ejemplo de la búsqueda en profundidad aplicada a la resolución del problema.

Como parte de la implementación no sólo se requieren los algoritmos más básicos de fuerza bruta, sino que aprovechando el contexto del problema, se deberá proveer algoritmos de búsqueda más avanzados que utilicen información adicional, específica a dicho contexto, para orientar la búsqueda de la solución. Este tipo de algoritmos se denomina de búsqueda heurística, ya que su objetivo es reducir el tiempo de proceso al orientar la exploración de los estados esperando evaluar una cantidad menor que las estrategias por fuerza bruta. El hecho de que se los llame heurísticos se debe a que en general no es posible garantizar una mejora para todos los posibles escenarios. En particular se pide que se desarrolle el algoritmo de búsqueda heurística más popular llamado A* (A estrella) probando varias funciones heurísticas de evaluación de estados.

Funciones de evaluación heurística

Las funciones de evaluación heurística (o simplemente heurísticas) permiten estimar el costo del camino óptimo entre dos estados. Esta definición es la utilizada en el contexto de problemas de búsqueda desde la perspectiva de un único jugador (o agente). Para juegos de dos jugadores también se utilizan heurísticas, pero las mismas son estimaciones de la utilidad de los estados teniendo en cuenta la perspectiva de ambos jugadores.

En general, los estados evaluados serán un estado no terminal y el estado objetivo. En este tipo de juegos, donde las condiciones que determinan los estados objetivos permanecen invariables, el parámetro principal de la evaluación es el estado no terminal. La distancia estimada es la correspondiente a una configuración donde se alcanza el objetivo a través del cumplimiento de dichas condiciones.

Se llama estática a una función heurística que tiene como parámetro sólo la configuración de un estado. Dichas funciones tienen en cuenta ciertas variables para calcular la estimación de la distancia. Frecuentemente la función heurística es una combinación lineal de las variables consideradas, que se pesan de algún modo para obtener un valor en término de las unidades elegidas.

A la hora de diseñar una función heurística se debe hacer un balance entre:

- el costo computacional de cada evaluación;
- la calidad de la estimación retornada.

Algoritmo A*

El algoritmo A* combina dos estrategias de búsqueda, la expansión de estados de igual costo (como el BFS) y la búsqueda heurística pura, con el objetivo de encontrar una solución óptima.

A* expande los nodos de acuerdo al mejor costo heurístico asociado a un estado. A este costo se lo denomina f(n). El mismo se define como f(n) = g(n) + h(n), donde g(n) es el costo del camino desde el estado inicial al estado n, y h(n) es la estimación heurística para alcanzar el estado objetivo a partir del estado n. Entonces, f(n) estima el costo total menor de cualquier solución que pase por el estado n. Si dos estados tienen igual valor de f se debe desempatar a favor del nodo con menor valor de h. El algoritmo termina cuando se elige un estado objetivo para expandir.

El algoritmo A* garantiza que las soluciones encontradas son óptimas si la función heurística h(n) es admisible, es decir, no sobrestima el valor real de la distancia óptima. La principal desventaja del algoritmo A*, al igual que algoritmos como el BFS, es la utilización de la memoria, ya que, por lo menos, la lista de estados por expandir debe ser almacenada durante la búsqueda de la solución.

Objetivo

El objetivo del trabajo es desarrollar distintos algoritmos que obtengan la sucesión de movimientos que debe realizar el explorador para llegar al casillero de salida de cada uno de los laberintos. Luego, se deberán analizar y comparar estos algoritmos en base a distintas métricas que reflejen el funcionamiento de los mismos.

Soporte provisto para el desarrollo y análisis de los algoritmos

Para la implementación de los algoritmos se provee un proyecto, que consiste en la implementación completa del juego Mummy Maze. El mismo incluye la lógica del juego (reglas, configuración y comportamiento de cada uno de los personajes), así como una interfaz de usuario que permite probar los distintos escenarios, controlando al explorador manualmente y también, a través de las soluciones obtenidas por los distintos algoritmos que se van a implementar.

El control del modo manual es mediante las flechas para los movimientos arriba, abajo, derecha e izquierda y la barra espaciadora para quedarse en el lugar. También se puede poner en pausa el juego o la reproducción de una solución mediante la tecla P.

Los algoritmos a implementar

Cada uno de los algoritmos de búsqueda deberá ser implementado cumpliendo la siguiente interfaz:

```
bool (*PlanAlgorithm)(GameSimulator & sim, unsigned int & depth,
                      list<Action::Type> & solution)
```

Esta interfaz presentada por **`PlanAlgorithm`** determina que se debe retornar un valor booleano que indica si se pudo alcanzar el estado objetivo o no. Además, establece que se recibe como parámetros:

> un objeto ***`sim`*** de tipo **`GameSimulator`**, que permite simular el juego y gestionar los estados generados;
>
> un valor entero ***`depth`*** que nos permite limitar la profundidad de la búsqueda;
>
> y una lista ***`solution`*** donde se almacenarán las acciones que nos permiten alcanzar el estado objetivo, si fue encontrado.

En los archivos **`MazeGameAlgorithms.h`** y **`MazeGameAlgorithms.cpp`**, se encuentran las declaraciones y las definiciones del algoritmo DFS de ejemplo, y el resto de los algoritmos cuya implementación se deberá completar (todos ellos de acuerdo a la interfaz general dada por **`PlanAlgorithm`**). Es posible agregar más algoritmos en estos archivos.

En el archivo **`main.cpp`** se invoca a la función **`runMazeGame.`** Esta función recibe en un vector, punteros a las funciones que implementan los algoritmos y una cadena de caracteres necesaria para crear la opción correspondiente en el menú. La opción para el modo manual es una función especial y el resto (**`dfs`**, **`bfs`**, **`astar1`**, etc) corresponden a los algoritmos de ejemplo y los stubs del resto de los algoritmos a implementar. Si se implementan algoritmos adicionales, se deberá agregar la opción correspondiente al vector.

```
vector<pair<PlanAlgorith, string> > algorithms;
algorithms.push_back(make_pair(&manualMode, "Modo Manual"));
algorithms.push_back(make_pair(&dfs, "DFS Recursivo"));
algorithms.push_back(make_pair(&bfs, "BFS"));
algorithms.push_back(make_pair(&astar1, "A* Heuristica 1"));
algorithms.push_back(make_pair(&astar2, "A* Heuristica 2"));
algorithms.push_back(make_pair(&astar3, "A* Heuristica 3"));
```

Interfaz de programación de la aplicación

Como se vió en la especificación de la interfaz de los algoritmos uno de los parámetros es del tipo **`GameSimulator`**. Este objeto permite manipular los estados del juego proveyendo las facilidades necesarias para implementar los algoritmos de búsqueda para resolver el problema planteado.

Este objeto simulador posee varios métodos que pueden dividirse en dos clases principales:

Consulta del estado del juego: a través del método **`getGame`** se obtienen el objeto que representa el juego propiamente dicho (de tipo **`Game`**).

Manipulación de estados: el tipo **`GameState`** representa un estado de juego que puede guardarse y recuperarse más tarde. A través del método **`update`** el simulador va avanzando de estado, guardando automáticamente los estados anteriores. El método **`undoUpdate`** permite volver al estado inmediatamente anterior.
El método **`getState`** retorna la representación del estado del juego actual como una referencia a una instancia de **`GameState`** que puede guardarse en una estructura externa al simulador. El método **`setState`** lleva el estado de la simulación al estado que se indica como parámetro.
Se distinguen dos tipos de estados que guarda el simulador.

Los estados anteriores: es una lista de los estados por los que pasó la simulación, ya sea por una invocación al método **`update`** o **`setState`**; una invocación al método **`undoUpdate`** eliminará el último estado alcanzado. El método **`isInPath`** consulta si un estado dado se encuentra en la lista de los estados anteriores.

Los estados generados: es un conjunto de todos los estados por los que pasó la simulación. Para estos estados se puede guardar información; la más importante un valor de visitado (booleano) que se activa con el método **`addVisited`** y se consulta con el método **`getVisited`** (de esta forma el simulador implementa directamente la lista de estados visitados o cerrados de muchos algoritmos).

El juego en sí está representado por el tipo **`Game`**. El mismo permite acceder a los elementos del juego (explorador, enemigos, paredes, etc) por su tipo, a través del método **`getObject(s)`**. También se puede acceder a los objetos a través de su posición en el espacio de juego. Para esto se utiliza el método **`getGameSpace`** que retorna un objeto del tipo **`GameSpace`** que permite hacer ese tipo de consultas.

Los elementos del juego son instancias de la clase **`LogicObject`**. Esta clase posee muchos métodos de consulta siendo los más importantes los que proveen la información sobre el tipo (**`getType`**) y la posición (**`getShape`**, que tiene las coordenadas **`getX/getY`**). Al mismo tiempo, esta clase provee métodos mediante los cuales los elementos del juego pueden modificar su estado a través de la ejecución de acciones, representadas por la clase **`Action`**. El método **`getActions`** retorna una lista de las acciones que puede realizar un elemento del juego; el método **`canPerform`** permite determinar si se puede realizar una acción y el método **`performAction`** ejecutarla.

Para una información completa de cada uno de los métodos que conforman estos tipos de datos principales consultar la documentación HTML (subdirectorio **doc** del proyecto).

Algoritmo de ejemplo provisto

Como parte del proyecto se incluye la implementación del algoritmo de búsqueda exhaustiva por prufundidad (DFS).

En la primera parte, se ve la utilización de los métodos del simulador para la consulta del estado; a través del método **`isGameOver`** se determina si el juego finalizó. En el caso de que el juego terminó, se consulta el resultado de finalización, con el método **`getGameOverResult`**.

A continuación se utilizan los métodos del simulador para la manipulación de los estados de la simulación. Como primer paso se obtiene una referencia al estado actual con el método **`getCurrentState`**. Luego, se consulta si el mismo se encuentra visitado con el método **`isVisited`**, para lo cual se procede marcándolo como visitado. Así se evitan los ciclos infinitos que pueden ocurrir al visitar un estado en el camino actual y el recorrido de una rama ya visitada anteriormente (notar la diferencia con el método **`isInPath`** que sólo consulta el camino actual).

Luego se sigue con la expansión de los estados hijos. Para esto se debe ejecutar una acción del explorador y procesar los turnos hasta que nos toque jugar otra vez. Es así que se obtiene una referencia al jugador utilizando el método **`getObject(EXPLORER)`**. Un detalle de implementación muy importante es que dicha referencia se debe manipular a través de un tipo de puntero especial dado por la clase **`ComponentPtr`**. **No se debe utilizar un puntero del tipo `LogicObject *`** (porque el mismo se invalidará durante la ejecución) .

Una vez obtenido el elemento del juego que representa al jugador, se procede a obtener los movimientos o acciones (**`getActions`**) y realizar (**`performAction`**) aquellas que es posible en el estado actual (**`canPerform`**). Luego de ejecutar la acción se pasa al estado siguiente del juego utilizando el método **`update`** del simulador.

Es en este punto que se hace el llamado recursivo. Al retornar de la recursión se puede haber encontrado la solución o no. En caso negativo se retorna al estado previo a la ejecución de la acción a través del método **`undoUpdate`**.

Durante la ejecución de la acciones se va manteniendo la lista de acciones realizadas que formarán parte de la solución, teniendo cuidado de quitar la última acción al retornar a un estado previo del juego.

```
bool dfs_recursivo(GameSimulator & sim, unsigned int & depth,
                   list<Action::Type> & solution)
{
      if (sim.isGameOver()) {
            if (sim.getGameOverResult() == GameSimulator::WIN) {
                  return true;
            } else {
                  return false;
            }
      } else if (solution.size() == depth) {
            return false;
      } else {
            const GameState * currentState = sim.getCurrentState();
            if (!sim.isInPath()) {
                  ComponentPtr<LogicObject>
                    explorer(sim.getGame()->getObject(EXPLORER));
                  list<Action::Type> explorerActions;
                  explorer->getActions(explorerActions);
                  bool found = false;
                  list<Action::Type>::iterator action = explorerActions.begin();
                  while (!found && (action != explorerActions.end())) {
                        if (explorer->canPerform(*action)) {
                              solution.push_back(*action);
                              explorer->performAction(*action);
                              sim.update();
                              found = dfs_recursivo(sim, depth, solution);
                              if (!found) {
                                    solution.pop_back();
                                    sim.undoUpdate();
                              }
                        }
                        action++;
                  }
                  return found;
            } else {
                  return false;
            }
      }
}
```

Desarrollo

Implementación

Se deberá entregar un proyecto que permita compilar correctamente el código de los algoritmos que se solicitan a continuación. Así mismo, se solicita la entrega de un ejecutable de ese proyecto.

Los algoritmos que se deberán implementar son:

- Búsqueda en amplitud (BFS).
- Búsqueda heurística A*, junto con tres heurísticas. Ninguna de las funciones heurísticas podrá utilizar como único parámetro de evaluación la distancia entre el explorador y la salida (cualquier tipo de distancia, ya sea la euclídea, Manhattan, etc).

Informe

Se deberá entregar un informe impreso conteniendo al menos la siguiente información:

- Identificación del grupo (número de grupo, nombres de los integrantes, email y ayudante asignado).
- Introducción al problema.
- Código de los algoritmos implementados.
- Explicación breve del funcionamiento general de cada algoritmo.
- Explicación de los detalles de la implementación realizada de cada uno, incluyendo comentarios sobre las estructuras de datos utilizadas, y cualquier decisión de implementación que considere relevante.
- Explicación detallada de las heurísticas
  - Explicar las hipótesis bajo las cuales se basaron las heurísticas implementadas.
  - Explicar los pasos y pruebas que determinaron la forma específica de cada función de evaluación (variables, pesos o factores, combinación de las mismas, etc).
- Análisis teórico y empírico de los algoritmos
  - Realizar una comparación de los resultados obtenidos teniendo en cuenta aspectos como la cantidad de estados visitados, calidad de la solución, y otros parámetros que considere importantes. Se deberán comparar, como mínimo, los resultados para los escenarios 4, 6, 8, 10, 13 y 15.
  - Presentar los resultados en un tabla de datos y mediante los gráficos que consideren apropiados.
  - Interpretar los resultados obtenidos y realizar una explicación de los mismos.
  - Incluir una explicación del comportamiento del Explorador de acuerdo a cada función de evaluación utilizada. Mencionar si este comportamiento justifica o no la hipótesis de la evaluación heurística.
- Conclusiones a las que llegaron luego del trabajo de análisis anterior.