Métodos de la clase Game

Se dan principalmente dos modificaciones respecto al juego original de MsPacMan, tan sólo se realiza una acción al alcanzar una intersección y no se puede realizar el movimiento contrario con el que se ha alcanzado dicha intersección (no se puede volver atrás).

Para no perderse se puede mirar inicialmente sólo el apartado de *Métodos* y el de *Constants*, dejando el apartado de *Métodos para visibilidad parcial*, para más adelante.

Métodos

Estos métodos pueden ser útiles en todas las entregas. Los métodos que se van a ver son de la clase Game, por lo que para ejecutar la funcionalidad habría que escribir `game.`*metodo(params)*.

- Para obtener las variables de MsPacMan, nodo y movimiento, tenemos: `int getPacmanCurrentNodeIndex()` y `MOVE getPacmanLastMoveMade()`
- De forma análoga, para las variables de cada uno de los fantasmas están: `int getGhostCurrentNodeIndex(GHOST)` y `MOVE getGhostLastMoveMade(GHOST)`
  - Adicionalmente, los fantasmas tienen la variable de *tiempo restante que son comibles*, a la cual podemos acceder de la forma: `int getGhostEdibleTime(GHOST)`
    - (Obs. podemos usar el tiempo para formular la pregunta de boolean, boolean isGhostEdible(), con la sentencia (getGhostEdibleTime(GHOST) > 0) $\sim$ isGhostEdible()).
  - Por último, puede ser útil saber cuánto tiempo le queda a un fantasma para salir de la cárcel: `int getGhostLairTime(GHOST)`
- Respecto a obtener los posibles movimientos dados dos nodos, el último movimiento con el que se alcanza el nodo origen y una métrica de medida de distancia, tenemos:
  - Dado dos nodos, origen y destino, devuelve el movimiento que nos acerca al destino: `MOVE getNextMoveTowardsTarget(int, int, MOVE, Constants.DM.PATH)`
  - Dados dos nodos, origen y destino, devuelve el movimiento con el se huye del destino: `MOVE getNextMoveAwayFromTarget(int, int, MOVE, Constants.DM.PATH)`
    - (Obs. en ambos casos, existen una sobrecarga de estos métodos sin el parámetro *MOVE* y por lo tanto no lo tiene en cuenta)
    - (Obs. la métrica de medida en las distancias puede ser diversa, *PATH* nos devuelve el camino óptimo)

- Métodos adicionales en función de dos nodos o un nodo y un movimiento:
  - Dado un nodo y el último movimiento, devuelve una lista de movimientos posibles:
    `MOVE[] getPossibleMoves(int, MOVE)`
    - (Obs. existe una sobrecarga de este método que no necesita un MOVE y no lo tiene en cuenta).
  - Dado un nodo y un movimiento, nos devuelve el nodo vecino al origen tomando ese movimiento:
    `int getNeighbouringNodes(int, MOVE)`
  - Dados un nodo origen y un nodo destino vecino, nos da el movimiento para ir al destino:
    `MOVE getMoveToMakeToReachDirectNeighbour(int, int)`
- Para saber los nodos de las pills y las power pills:
  `int[] getActivePillIndices()` y `int[] getActivePowerPillIndices()`
  - (Obs. tener cuidado y utilizar este método para tomar las posiciones de pills/powerpills ACTIVAS, es decir, que MsPacMan aún no se ha comido. Los métodos, *int[] getPill/PowerPillIndices()*, devuelve las posiciones de las pills/powerpills iniciales del mapa).
- Métodos para obtener caminos (secuencias de nodos) y relacionados:
  - Dado dos nodos, origen y destino, y el último movimiento con el que se alcanzó el nodo origen, obtenemos el camino óptimo desde el origen al destino, dicho camino seguirá el movimiento que le acerca al nodo destino siguiendo la medida de la distancia DM.PATH:
    `int[] getShortestPath(int, int, MOVE)`
  - Dado dos nodos, origen y destino, y el último movimiento con el que se alcanzó el nodo origen, obtenemos la distancia más corta del origen al destino, distancia del camino óptimo:
    `int getShortestPathDistance(int, int, MOVE)`
    - (Obs. si necesitas el camino y la distancia, la distancia se puede obtener en función del camino, por su tamaño *getShortestPath(int, int, MOVE).length ~ getShortestPathDistance(int, int, MOVE)*).
- Para saber si es necesario tomar una decisión de movimiento, en función de si estás en una intersección o no, se puede comprobar de forma sencilla con:
  - Dado un nodo, preguntar si es una intersección:
    `boolean isJunction(int)`
  - Sin embargo, esta función no tiene en cuenta el caso inicial, en el que MsPacMan o los fantasmas aparecen/reaparecen poseen un movimiento *MOVE.NEUTRAL*, por lo tanto SÍ se puede decir el movimiento que puede llevar tanto los fantasmas como MsPacMan.
  - Se puede sustituir la función *boolean isJunction(int)* para contemplar este caso, apoyándose en la función de obtener los posibles

movimientos dado un nodo y el último movimiento *MOVE[] getPossibleMoves(int, MOVE)*:
`(getPossibleMoves(int, MOVE).lenght > 1) ~ isJunctio(int)`

- Puedes obtener la lista de intersecciones mediante el método:
`int[] getJunctionIndices()`

Métodos para visibilidad parcial

Para la práctica de lógica difusa, en la que se aplica una visibilidad parcial, hay que tener en cuenta información adicional y para eso tenemos otros métodos. Tener en cuenta que tanto estos métodos como los anteriores pueden devolver un dato inesperado (por ejemplo, la posición de un fantasma devuelva -1), hay que tener en cuenta antes de hacer una consulta si se "ve" el objeto sobre el que se pregunta (el fantasma, un nodo, una pill/powerpill ...).

- Preguntar si un nodo se puede ver desde un punto de vista un fantasma o MsPacMan, dado el nodo:
`boolean isNodeObservable(int)`
- Como no se puede ver a MsPacMan y puede ser interesante cuándo se ha comido una power pill, por ejemplo para localizar a MsPacMan aún sin verla:
`boolean wasPowerPillEaten()`
  - Este método tan sólo devuelve *true* en el "tick" en el que MsPacMan se come la power pill, por lo que si se valora este método posteriormente (por ejemplo, en la primera intersección, donde se toman las decisiones) no será *true* si no se ha comido la power pill exactamente en el mismo "tick" en el que se alcanzó la intersección.
- Si se quiere tener un registro del estado del mapa se pueden usar distintos métodos.
  - Por ejemplo, dado los índices de las pills y las power pills podemos preguntar si siguen "vivas":
    `boolean isPill/PowerPillStillAvailable(int)`
  - Lo más importante es que ya no podemos acceder a variables importantes, tales como la posición de las power pills, al menos a través del game.
  - Sin embargo, el game tiene un "currentMaze" y gracias a esta variable podemos acceder al objeto maze donde se está jugando:
    `Maze getCurrentMaze()`
    - Gracias a esta variable Maze, podemos acceder a los parámetros públicos de la clase, donde destacan:
      - Los índices de las pills y las power pills *int[] pillIndices, powerPillIndices*.

· Las posiciones iniciales de MsPacMan y de los fantasmas *int initialPacManNodeIndex, initialGhostNodeIndex*.

Constants

Un apartado aparte para describir la clase importante *Constants*, en la cual tenemos los distintos parámetros que rigen el juego, tales como:

- El tiempo que son comibles los fantasmas `EDIBLE_TIME`.
- El factor de reducción de movimiento cuando los fantasmas son comibles `GHOST_SPEED_REDUCTION`.
- El factor por el que se reduce el tiempo que pasan los fantasmas en la cárcel una vez son comidos en una partida `LAIR_REDUCTION`.
- La distancia a la que se "comen" unos a otros `EAT_DISTANCE`.

Siendo estos los que más he tenido en cuenta, es recomendable visualizar la lista completa de parámetros accediendo a la clase.