Pentominós

Aido Teruel, Suriel

Balsera Martinez, Paula

Contenido

1. Decisiones de diseño:

El problema que se nos presenta es el de colocar doce pentominós en un tablero rectangular de mxn, considerando que pueden quedar 4 huecos vacíos en dicho tablero, además de que debemos considerar todas las rotaciones de las piezas y contamos con la posibilidad de repetir todas las piezas las veces que sean necesarias.

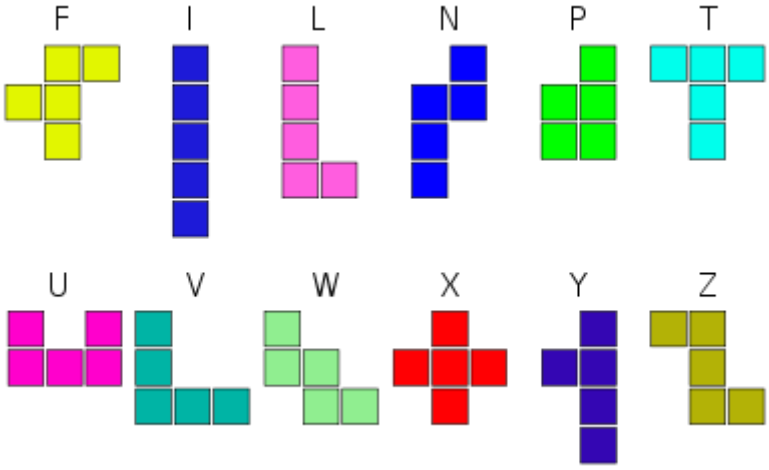

Ilustración 1. Tipos de pentominó.

Siendo así, y contando con las clases dadas durante la realización de la práctica dos durante la evaluación continua sobre *búsqueda en espacios de estado,* diseñamos el problema creando una clase de *Acción* para cada pieza, es decir, 63 acciones teniendo en cuenta que consideramos todos los posibles giros y simetrías como piezas independientes de las originales. Esta decisión se toma en base a la dificultad de realizar un solo algoritmo que se recorra todas las piezas, además las gire y las coloque correctamente.

Además de esto, creamos un método que nos resuelva la aplicabilidad en el cual comprobamos que las piezas no se salgan del tablero y que no se superpongan unas con otras.

Por último, diseñamos el algoritmo que servirá para identificar un estado como estado final en base a uno de los requisitos, citado ya, que tenemos para resolver el problema: todo tablero que tenga 4 huecos o menos después de colocar n número de piezas se considerará estado final.

Por último, no contamos con un estado final prefijado, sino que implementaremos un método que nos permitirá insertar un número de filas y un número de columnas determinado por el usuario, pudiéndose elegir las dimensiones del tablero (un matriz llena de 0)

Así pues, teniendo ya definido el diseño de estos elementos (acciones, estado final y estado inicial) y con las librerías facilitadas en la práctica dos, tenemos todos los elementos para poder realizar el proyecto.

2. Desarrollo de la implementación

2.1. Primera versión (desechada):

En primera instancia, nos enfrentamos a un planteamiento cuyo diseño llevó a un nivel de complejidad demasiado alto. Intentamos introducir todas las fichas a través de un fichero el cual sería leído a través de un método de una clase *ColocarPieza* que heredaría de *Acción.* Dicho método leería cada línea del fichero (cada una representaría una ficha en forma matricial, es decir, como una lista de listas, al igual que el tablero), e interpretaría cada una de estas líneas como una lista de listas. Después, el método introduciría en el tablero las fichas y el algoritmo de la práctica aplicaría las búsquedas pertinentes.

De igual manera, la aplicabilidad se comprobaría a través de la lectura del fichero mencionado, y comprobaría que a la hora de colocar la ficha, ésta no se saliera del tablero y solo se colocara sobre una casilla vacía (es decir, un punto de la matriz cuyo valor fuera 0).

Por último, el estado final se calcularía de la misma forma que en la versión final: contaría el número de 0 del tablero final. Cuando este fuese menor que cuatro, se consideraría estado final.

Finalmente desechamos esta idea. Los motivos que acompañan al abandono de este planteamiento de diseño son los siguientes:

- **Complejidad:** después de varios meses intentando resolver el problema de esta forma, vimos finalmente que sería excesivamente complejo realizar un algoritmo que actuase de esta forma.
- **Tiempo:** cuando la tercera convocatoria estuvo notablemente cerca, decidimos plantearnos un cambio de diseño ya que, debido a la complejidad ya mencionada, no podríamos terminar a tiempo el proyecto.
- **Práctica:** la práctica que utilizamos en todo momento de referencia, fue estudiada exhaustivamente por nosotros mismo cuando vimos que habíamos llegado a un punto muerto en el diseño del problema. Después de un estudio bastante importante, llegamos a la conclusión de que podríamos resolver el problema si lo plateábamos como un conjunto de problemas pequeños, en lugar de un problema grande que resolviese todos los pequeños. Dicho de otra manera, si en lugar de intentar hacer una acción que resolviese todas las piezas, hacíamos una acción para cada una de las mismas, podríamos alcanzar una serie de problemas de baja complejidad, mucho más fácilmente resolubles. El volumen de trabajo se multiplicaría varias veces, pero la dificultad en suma de la resolución de todos los nuevos problemas sería menor que nuestro planteamiento actual.
- **Ayuda de la tutora del proyecto:** finalmente, la ayuda que nos prestó la profesora en sucesivas tutorías que tuvimos con ella, nos dio también un empujón para cambiar el planteamiento del problema a uno que nos presentara una dificultad menor, ya que en las sucesivas tutorías pudimos ver como las ideas para resolver el algoritmo, que a priori pensábamos bien encaminadas, no estabas enfocadas de la forma correcta.

Por todas estas razones, pasamos a llevar el problema a un diseño muy distinto, tal y como es el diseño final.

Para poder comprender los problemas de desarrollo, adjuntaremos junto con esta memoria un zip que contenga todo el trabajo anterior en el punto en el que fue abandonado.

2.2. Versión definitiva:

La siguiente versión del trabajo se basa en, en lugar de hacer unos métodos genéricos, hacer una clase para cada una de las piezas que forman el grupo de pentominós y después haremos la heurística pertinente. Vamos a pasar a explicar a continuación las piezas (como están hechas y por qué), las clases que conforman las piezas, la heurística y por último el menú creado.

2.2.1. Piezas:

Creamos las piezas de una forma que resulta bastante sencilla de implementar: utilizamos una de las casillas de la pieza como referencia. Después comenzamos a restar o sumar a las piezas que forman esta matriz que forma la pieza. Para explicarnos mejor, mostremos un ejemplo con la pieza i:

- Estado[fila][columna]: Pieza que utilizaremos de origen. Será la última de ellas (abajo del todo).
- Estado[fila-1][columna]: Colocamos, a partir de la anterior, una pieza en la fila anterior.
- Estado[fila-2][columna]: Colocamos otra ficha en dos casillas anteriores a la de origen.
- Estado[fila-3][columna]: Colocamos otra ficha en tres casillas anteriores a la de origen.
- Estado[fila-4][columna]: Colocamos otra ficha en cuatro casillas anteriores a la de origen.

Aquí tenemos como se formaría la pieza I. Además de esto, a cada una de las fichas le asignaremos un número que la identifique. De esta forma, podremos distinguir entre varias fichas. Por ejemplo, la I horizontal tendrá asignado el número 1, la Z4 (cuarto giro) tendrá el número 63 (el último) o la Y3 (tercer giro), que tendrá el número 54.

2.2.2. Clase ponerPiezaX:

La case ponerPiezaX estará formada por tres métodos: el método hay_hueco_x(estado), el es_aplicable(estado) y el aplicar(estado). El método hay_hueco_x es un método auxiliar que utilizamos para resolver el método es_aplicable. Tanto el método es_aplicable como el método aplicar son métodos necesarios para que las clases que se nos da para resolver el problema (en la práctica 2) funcionen correctamente.

- **hay_hueco_x(estado):** Este método comprueba si el punto que se va a colocar una ficha x es un espacio en blanco. En ese caso se podrá colocar. Como parámetro e entrada le pasamos un estado, que será una matriz. La salida será un True o un False, en función de si alguno de los métodos encuentra una ficha en una de las posiciones o si, por el contrario, todas las fichas se colocan en huecos vacíos.
- **es_aplicable(estado):** En este método, al que le pasamos un tablero (mismas condiciones que en el método anterior), comprobamos, en primer lugar, que la pieza que vamos a introducir tiene unas dimensiones que están dentro de los límites de nuestro tablero. Después, le pasamos el método anterior. Si ambas condiciones nos dan True, la aplicabilidad será válida y el método nos devolverá un True. En caso contrario nos devolverá un False y la pieza no será colocada.
- **aplicar(estado):** Por último, en este método le asignamos a cada una de las casillas, una vez validada la aplicabilidad, colocamos la pieza en las posiciones que le corresponda. Esto es, colocaremos el número que corresponda a cada pieza en el lugar indicado.

2.2.3. Heurística

Pensar la heurística nos presentó varios problemas. No encontrábamos una forma efectiva de realizarla, de forma que decidimos acudir a tutoría con la profesora para ver si podía guiarnos. Con su ayuda, tomamos la decisión de afrontarlo de esta forma: calcularíamos las casillas vacías y como están agrupadas. De esta forma, si un estado tenía las casillas vacías dispersas recibiría una penalización, ya que sería peor que otro tablero que tuviese juntas al menos cinco de sus casillas vacías.

Esto lo hicimos con un procedimiento relativamente simple: en primer lugar, hicimos un método que nos dijese, para una casilla (que ya fuese 0), si sus vecinos eran también cero. Evidentemente, tuvimos que poner unas condiciones a las esquinas, otra a las casillas del borde del tablero, y otras a las fichas centrales del tablero. Hacemos las comprobaciones de forma similar que en el método hay_hueco_x explicado anteriormente, es decir, nos recorremos el tablero y cuando encontramos una casilla que contenga un 0, comprobamos sus vecinos (3 en caso de una esquina, 5 en caso de un lateral, 8 en el caso de una central). Por cada uno de sus vecinos que son 0 incrementamos una variable contador, que nos indicará el número de casillas vacías entorno a nuestra casilla.

Tras esto, lo que haremos será el método h (que será el que nos calculará la heurística). Lo que haremos en dicho método será comprobar el número de vecinos que tiene una casilla vecina. Si este método es múltiplo de 5, es decir, es módulo 5, quiere decir que podremos meter alguna pieza. En caso contrario, ese estado sufrirá una penalización. El método calculará la heurística como la suma del número de 0 total del estado más la penalización pertinente menos la profundidad del nodo.

2.2.4. Menú:

El método menú es solo un método que le pedirá al usuario que introduzca un número determinado de filas y columnas que formarán el tablero. Después de esto nos pedirá que introduzcamos un número del 1 al 5 para elegir con el algoritmo que queramos que se desarrolle (se nos indicará, claro, los números que hay que introducir para usar un algoritmo u otro). Una vez introducido un número mayor que 0 de columnas y de filas y un algoritmo válido, se mostrará el funcionamiento del algoritmo (con la variable detallado en activo), hasta que llegue finalmente a un estado final. Mostrará también el tiempo de ejecución segundos (con detalles).

2.2.5. Otros:

Por último, comentar la clase Colocar(fila, columna) y el método es_estado_final.

- Clase Colocar(fila, columna): Simplemente señalar que es la clase que le meteremos como parámetro de entrada a los algoritmos de búsqueda, ya que dicha clase tendrá la lista de acciones (una lista con todas las clases poner_ficha_x) y el estado inicial.
- Método es_estado_final(estado): simplemente comprueba que un tablero tenga un número menor o igual que casillas vacías de 4, ya que solo así será un estado final.

4. Conclusiones de la fase de experimentación:

Algoritmo/Tiempo(s)	**Tablero 4x4**	**Tablero 8x4**	**Tablero 6x7**	**Tablero 2x10**	**Tablero 9x4**
B. en profundidad	0.0175330	0.3382396	*inestimable	0.52185177	31.3039584
B. en anchura	5.7405948	*inestimable	*inestimable	43.7770476	*inestimable
B. Óptima	0.0760552	0.9316797	4.84043622	0.10259056	1.71571469
B. A Estrella	0.0745515	0.9171545	4.0248720	0.10356950	1.66816806
B. Primero mejor	0.0765538	0.9331624	4.0723931	0.10155463	1.69169545

**inestimable = tiempo de resolución > 2 minutos.*

Todo testeado sobre el mismo PC: Sony VAIO con Intel Core i7 4500u 1.8GHz y 8Gb de ram

Como podemos observar, en los tableros más pequeños domina el algoritmo de Búsqueda en profundidad. Luego, mientras más grande se va haciendo el tablero, los algoritmos con heurística van siendo más eficientes con respecto al resto, hasta llegar al punto de una diferencia de menos de 5 segundos frente a los más de dos minutos de los otros. Por otra parte, podemos observar que el algoritmo más ineficiente es siempre el de Búsqueda en Anchura

5. Bibliografía

Como parte de nuestra bibliografía podemos encontrar:

- La práctica 2,(https://www.cs.us.es/cursos/iais-2016/?contenido=practicas.php) de la asignatura Inteligencia Artificial de la titulación de Ingeniería Informática del Software impartida en la US, del curso académico 2016/2017.
- https://stackoverflow.com/ , para consultas de determinados problemas con el lenguaje de programación.
- http://lineadecodigo.com/, para consultas sobre el lenguaje de programación.
- https://es.slideshare.net/martinarosasvelazquez/heuristica-38526359, para aclarar el concepto de heurística.

Para el resto de cuestiones que no hemos podido resolver vía web, hemos acudido a tutorías con la profesora para resolver los problemas que quedaban.

6. Comentarios

Por último, hacer una aclaración para entenderé mejor el código: el nombre de los métodos se llama como poner[letra]2, poner[letra]3, etc. Para aclarar que pieza está en cada posición se adjuntará junto a esta carpeta una carpeta con las representaciones gráficas de las piezas giradas, así como el número asignado.