GAROÉ

UNIVERSIDAD NACIONAL DE RÍO CUARTO

Trabajo final de Analista en Computación

Garoé: Una Plataforma Orientada a Objetos para el Desarrollo de Soluciones Algorítmicas basadas en Búsqueda

Fiori, Carla Noelia

Gutiérrez Brida, Simón Emmanuel

Director: Dr. Aguirre, Nazareno Matías

-Diciembre 2011-

Agradecimientos

GAROÉ

Índice

1. Introducción

1.1. Introducción General

Uno de los elementos centrales en la construcción de soluciones algorítmicas a problemas es el uso de las denominadas técnicas de diseño de algoritmos. Las técnicas de diseño de algoritmos brindan formas, en cierto sentido genéricas, de pensar en soluciones a problemas algorítmicos. Algunos ejemplos de técnicas de diseño de algoritmos tradicionales son Divide & Conquer (fuertemente ligada a recursión), programación dinámica (con importantes aplicaciones en problemas de optimización), y greedy (que muchas veces se aplica en la resolución aproximada de problemas).
Una técnica de diseño de algoritmos de gran importancia es la denominada Búsqueda. Esta técnica es sumamente general, e incluye backtracking, técnicas de búsqueda en problemas con adversarios, y otras. Esta técnica tiene una característica distintiva: su ámbito de aplicación es sumamente amplio. De hecho, existen paradigmas de programación basados en búsqueda (en particular, la Programación Lógica).
Un problema que se observa con frecuencia en soluciones algorítmicas basadas en búsqueda es que las implementaciones suelen ser muy cohesivas, mezclando cuestiones vinculadas al problema específico con otras más "generales", como la implementación de motores de búsqueda. Esto en muchos casos está justificado por razones de eficiencia, pero hace muy difícil reutilizar elementos presentes en las implementaciones.
En este trabajo proponemos desarrollar una plataforma genérica y extensible, para el desarrollo de soluciones algorítmicas para problemas de búsqueda. A través del uso de esta plataforma, los motores de inferencia/búsqueda desarrollados podrán reutilizarse fácilmente por diferentes problemas, y la definición de problemas de búsqueda se podrá hacer de manera incremental sobre porciones "pre-implementadas" del concepto de problema de búsqueda. La plataforma será desarrollada en Java, y se hará uso intensivo de genericidad y herencia para facilitar extensibilidad y parametrización en la solución.

1.2. Introducción detallada del problema

Hay numerosos problemas que utilizan a búsqueda como técnica para diseñar la solución. En todos los casos siempre se pueden remarcar cuatro partes, estas son, el estado o modelo que describe una instancia del problema, el problema a resolver, métodos para cambiar un estado a otro aplicando un único paso de transformación[1] y el motor de búsqueda que a partir de un estado inicial y un problema encuentra un estado que bien puede ser la solución al problema planteado o bien ser un estado mas cercano a resolver al mismo.
En todas las aplicaciones que utilizan búsqueda para resolver un problema, las partes previamente mencionadas están usualmente "pegadas" o incorrectamente modularizadas. El problema que esto trae es que para cada aplicación que requiere la resolución de algún problema mediante búsqueda se hace necesario el diseño e implementación de todas las partes.
En el desarrollo de software es común enfrentarse a un problema que tiende a repetirse, en esos casos es muy común que el o los desarrolladores diseñen e implementen su solución, la cual

1Los pasos de transformación pueden entenderse como cambios atómicos al estado, cambios que no pueden subdividirse en pasos mas pequeños.

probablemente estará muy “atada” al problema particular que estén solucionando. Los patrones de diseño son una solución mas elegante, probada y con muchas ventajas como una mayor independencia a cual es el problema particular que se esté solucionando. Un ejemplo de esto puede ser el patrón MVC (modelo-vista-controlador) el cual representa un problema muy común en aplicaciones que requieren el uso de interfaces gráficas que actúan sobre un modelo de datos. Si bien es completamente posible resolver dicho problema de forma “casera”, el patrón MVC presenta una estructura sobre la cual es posible diseñar cualquier aplicación (parte de la aplicación) que requiera una o mas vistas, un modelo y un controlador, al mismo tiempo que permite intercambiar estas partes.
Para el problema diseñar aplicaciones que hagan uso de búsqueda proponemos crear una plataforma de desarrollo que tenga divididas estas cuatro partes en **estado**, **problema**, **reglas de cambio de estado** (funciones de cambio de estado) y **motor de búsqueda** de forma que se pueda reutilizar una gran cantidad de código. Esta meta se logra al hacer que el problema, estado, reglas y motor estén modularizados de forma totalmente independiente mediante interfaces, clases abstractas y genericidad con restricciones[2].

1.3. Descripción de la solución

Se desea construir una plataforma (conjunto de clases, clases abstractas e interfaces) que actúe de base para el desarrollo de: motores de búsqueda con y sin adversario (a partir de ahora llamados MBA y MBSA respectivamente); problemas de búsqueda con y sin adversario (a partir de ahora llamados PBA y PBSA respectivamente) mediante la descripción de sus componentes (noción de estado, descripción del problema, reglas de cambio de estado). Y dos ejemplos de uso, un juego con adversario, y otro juego sin adversario, usados para demostrar la utilización de la plataforma y mostrar sus ventajas frente a tener que desarrollarlos sin tener a la misma.
El motor de búsqueda con adversario debe ser MinMax con poda Alfa-Beta, en cambio el motor de búsqueda para problemas sin adversario es a libre elección. Ambos problemas de ejemplo deben ser juegos (uno con adversario y el otro sin) y deben proveer una interfaz gráfica de usuario.

Para que el producto final sea aceptado primero debe satisfacer al problema descripto, debe ser capaz de ejecutar dos aplicaciones (un juego con y uno sin adversario) de manera correcta donde cada juego debe cumplir con las reglas del mismo y los motores de búsqueda deben ser desafiantes en los casos de juegos con adversario o dar ayudas (o resolver la partida) para el motor sin adversario.

1.4. Estructura del Informe

En primer lugar se encontrara a la sección preliminares, la cual ofrecerá al lector la información necesaria para poder comprender las nociones utilizadas en el resto del informe. Esta no es de lectura obligatoria y cualquier usuario que tenga conocimientos sobre programación orientada a objetos, patrones de diseño y que tenga conocimientos mínimos del lenguaje Java, podrá saltear dicha sección sin perder información importante. En segundo lugar se explicara las bases de la

2La genericidad con restricciones es aquella que restringe a los parámetros de clases para que deriven de una clase en particular, ej: Clase<C implements Comparable<C>>.

búsqueda como técnica de programación, así como distintos motores de búsqueda contando cada uno con una breve discusión así también como sus ventajas y desventajas. Los distintos motores estar organizados en grupos dependiendo del tipo de problema de búsqueda que solucionan y así también las formas básicas que tiene el algoritmo. En la sección Framework se encontrara con toda la información del proyecto, como por ejemplo especificación de requerimiento, implementaciones, y testing, entre otros. La próxima sección describirá los ejemplos prácticos utilizados para probar la plataforma y demostrar tanto su uso como ventaja del mismo. Luego el manual de usuario. Y finalizando con la conclusión.

2. Preliminares

2.1. Orientación a objetos

La primera característica, orientado a objetos (“OO”), se refiere a un método de programación y al diseño del lenguaje. Aunque hay muchas interpretaciones para OO, una primera idea es diseñar el software de forma que los distintos tipos de datos que usen estén unidos a sus operaciones. Así, los datos y el código (funciones o métodos) se combinan en entidades llamadas objetos. Un objeto puede verse como un paquete que contiene el “comportamiento” (el código) y el “estado” (datos). El principio es separar aquello que cambia de las cosas que permanecen inalterables. Frecuentemente, cambiar una estructura de datos implica un cambio en el código que opera sobre los mismos, o viceversa. Esta separación en objetos coherentes e independientes ofrece una base más estable para el diseño de un sistema software. El objetivo es hacer que grandes proyectos sean fáciles de gestionar y manejar, mejorando como consecuencia su calidad y reduciendo el número de proyectos fallidos. Otra de las grandes promesas de la programación orientada a objetos es la creación de entidades más genéricas (objetos) que permitan la reutilización del software entre proyectos, una de las premisas fundamentales de la Ingeniería del Software. Un objeto genérico “cliente”, por ejemplo, debería en teoría tener el mismo conjunto de comportamiento en diferentes proyectos, sobre todo cuando estos coinciden en cierta medida, algo que suele suceder en las grandes organizaciones. En este sentido, los objetos podrían verse como piezas reutilizables que pueden emplearse en múltiples proyectos distintos, posibilitando así a la industria del software a construir proyectos de envergadura empleando componentes ya existentes y de comprobada calidad; conduciendo esto finalmente a una reducción drástica del tiempo de desarrollo.

2.2. Java

Java es un lenguaje de programación Orientada a objetos, desarrollada por Sun Microsystems, al principio de los años '90. Tiene un modelo de objeto mas simple que C ++, y una gestión automática de memoria.
La compilación de código fuente JAVA se hace a “Byte Code”, el cual es un código intermedio, el cual es interpretado por una maquina virtual, la cual es especifica de plataforma, esto es lo que permite portabilidad.
El lenguaje JAVA se creo con cinco objetivos principales:
1- Debería usar la metodología de la Programación orientada a objetos
2- Debería permitir la ejecución de un mismo programa en múltiples sistemas operativos
3- Debería incluir por defecto soporte para trabajo en red
4- Debería diseñarse para ejecutar código en sistemas remotos de forma segura
5- Debería ser fácil de usar y tomar lo mejor de otros lenguajes orientados a objetos como C++.

Cuando se habla de programación orientada a objeto, se encuentran dos definiciones importantes; la primera se refiere a clases las cuales son definiciones de objetos abstractos compuestos por una estructura que representa a un objeto y las funciones asociadas al mismo. La segunda son los objetos los cuales son instancias de una clase. En un lenguaje orientado a objeto todo lo utilizado es un objeto incluyendo a ciclos, cualquier variable utilizada, acciones y funciones, etc. Estos lenguajes permiten cambiar una gran parte del comportamiento del lenguaje, como por ejemplo la

función de creación de objetos, llamadas a métodos. Una ventaja que trae este tipo de lenguajes es que permite una introspección prácticamente sin limites, lo cual significa que las clases son redefinibles en tiempo de ejecución, esto se hace mediante el uso de metaclases. En cambio los orientados a clases, no posee metaclases y tienen parte de su comportamiento definido fuera del ámbito de clases y objetos, como por ejemplo operadores, constructores y destructores de objetos; en este tipo del lenguaje la introspección es limitada o inexistente y en el caso de existir solo permite la consulta de cierta información de la clase, pero nunca la modificación de la misma.

2.2.1. Recolector de basura

En Java el problema de las fugas de memoria se evita en gran medida gracias a la recolección de basura (o automatic garbage collector). El programador determina cuándo se crean los objetos y el entorno en tiempo de ejecución de Java (Java runtime) es el responsable de gestionar el ciclo de vida de los objetos. El programa, u otros objetos pueden tener localizado un objeto mediante una referencia a éste. Cuando no quedan referencias a un objeto, el recolector de basura de Java borra el objeto, liberando así la memoria que ocupaba previniendo posibles fugas (ejemplo: un objeto creado y únicamente usado dentro de un método sólo tiene entidad dentro de éste; al salir del método el objeto es eliminado). Aun así, es posible que se produzcan fugas de memoria si el código almacena referencias a objetos que ya no son necesarios— es decir, pueden aún ocurrir, pero en un nivel conceptual superior. En definitiva, el recolector de basura de Java permite una fácil creación y eliminación de objetos, mayor seguridad y puede que más rápida que en C++

2.2.2. Rendimiento

El rendimiento de una aplicación está determinado por multitud de factores, por lo que no es fácil hacer una comparación que resulte totalmente objetiva. En tiempo de ejecución, el rendimiento de una aplicación Java depende más de la eficiencia del compilador, o la JVM, que de las propiedades intrínsecas del lenguaje. Algunas características del propio lenguaje conllevan una penalización en tiempo, aunque no son únicas de Java. Algunas de ellas son el chequeo de los límites de arrays, chequeo en tiempo de ejecución de tipos, y la indirección de funciones virtuales.
El uso de un recolector de basura para eliminar de forma automática aquellos objetos no requeridos, añade una sobrecarga que puede afectar al rendimiento, o ser apenas apreciable, dependiendo de la tecnología del recolector y de la aplicación en concreto. Java fue diseñado para ofrecer seguridad y portabilidad, y no ofrece acceso directo al hardware de la arquitectura ni al espacio de direcciones. Java no soporta expansión de código ensamblador, aunque las aplicaciones pueden acceder a características de bajo nivel usando bibliotecas nativas (JNI, Java Native Interfaces).

2.3. Patrones de diseño

Diseñar software orientado a objetos es difícil, y diseñar software orientado a objetos que sea reutilizable es aún mas difícil.
Es necesario encontrar objetos pertinentes, factorizarlos en clases con la correcta granularidad, definir interfaces de clase y jerarquías de herencia, y establecer relaciones claves entre si.
El diseño debería ser especifico al problema que se está resolviendo pero lo suficientemente general para resolver futuros problemas y requerimientos. También es deseable evitar el rediseño o al menos

minimizarlo. Es muy difícil resolver un problema desde cero, en cambio es mucho mas sencillo reutilizar soluciones que ya han probado ser efectivas anteriormente. En general es común encontrar patrones
recurrentes de clases y comunicación entre objetos en muchos sistemas orientados a objetos. Estos patrones resuelven problemas de diseño específicos y hacen a los diseños orientados a objetos mas flexibles,
elegantes y finalmente reutilizables. Ayudan a los diseñadores a reusar diseños exitosos al basar nuevos diseños en experiencias anteriores.
En forma simple un patrón de diseño ayuda a un diseñador a diseñar "bien" mas rápidamente.

2.3.1. ¿Que es un patrón de diseño?

Christopher Alexander dijo "Cada patrón describe un problema que ocurre una y otra vez en nuestro ambiente, y luego describe el núcleo de la solución a dicho problema, en una forma que se puede utilizar
esta solución un millón de veces, nunca haciéndola de la misma forma". En general un patrón tiene cuatro elementos esenciales:
1- El nombre del patrón es una forma de describir un problema de diseño, sus soluciones y consecuencias en una o dos palabras. Tener un nombre para un patrón nos permite diseñar en un nivel mas alto
de abstracción, utilizarlo en la documentación, al hablar con colegas, etc.
2- El problema describe cuando aplicar el patrón. Explica el problema y su contexto. Puede describir problemas específicos de diseño tales como la forma de representar algoritmos como objetos. Puede
describir clases o estructuras de objetos que son síntomas de un diseño inflexible. A veces el problema contendrá una lista de condiciones que deben ser cumplidas para que tenga sentido aplicar el patrón.
3- La solución describe los elementos que hacen al diseño, sus relaciones, responsabilidades y colaboraciones. La solución no describe un diseño concreto o implementación en particular, dado que un patrón es
como un molde que puede ser aplicado en muchas situaciones diferentes. En cambio, el patrón provee una descripción abstracta de un problema de diseño y como una organización general de elementos (clases y objetos)
la resuelve.
4- Las consecuencias son los resultados y la relación “costo/beneficio” de aplicar el patrón. Si bien las consecuencias son a menudo obviadas cuando se describe un decisión de diseño, estas son criticas para evaluar las alternativas de diseño y para entender los costos y beneficios de aplicar el patrón. Las consecuencias para el software a menudo conciernen al espacio y tiempo.
Un patrón de diseño, nombra, resume e identifica los aspectos clases de una estructura de diseño común que lo hace útil para crear un diseño reutilizable orientado a objetos. El patrón de diseño identifica
las clases e instancias participantes, sus roles y colaboraciones, y la distribución de responsabilidades. Cada patrón de diseño se enfoca en un problema de diseño orientado a objetos particular. Describe
cuando se aplica, si puede aplicarse en vista de otras restricciones de diseño y las consecuencias y la relación “costo/beneficio” de utilizarlo.

3. Búsqueda como técnica de programación

3.1. Búsqueda

Para poder aplicar la técnica de búsqueda es necesario poder describir el problema como una búsqueda de configuraciones exitosas a partir de ciertas configuraciones iniciales, aplicando reglas de avances. Un problema debe contener estado o configuración, estado Inicial, Reglas de avance o de configuración, estado exitoso.

3.2. Backtracking

Backtracking (vuelta atrás) dentro de las técnicas de diseño de algoritmo, el método de vuelta atrás es uno de los de mas amplia utilización, en el sentido de que puede aplicarse en la resolución de un gran numero de problemas, especialmente en aquellos de optimización.
Para ciertos problemas la única forma de resolverlos es a través de un estudio exhaustivo de un conjunto conocido a priori de posibles soluciones, en la que tratamos de encontrar una o todas las soluciones importantes y también la óptima.
Para llevar a cabo este estudio exhaustivo, el diseño vuelta atrás propone una manera sistemática de generar todas las posibles soluciones siempre que dichas soluciones sean susceptibles de resolverse en etapas.
En su forma básica la vuelta atrás se semeja a un recorrido en profundidad dentro de un árbol cuya existencia solo es implícita, y que denominaremos árbol de expansión. En donde cada nodo nivel k representa una parte de la solución y esta formada por k-etapas que se suponen ya realizadas.
Sus hijos son las prolongaciones posibles al añadir una nueva etapa. Para examinar el conjunto de posibles soluciones es suficiente recorrer este árbol construyendo soluciones parciales a medida que se avanza en el recorrido.
En este recorrido pueden suceder dos cosas. La primera es que tenga éxito si, procediendo de esta manera, se llega a una solución. Si lo único que buscábamos era una solución al problema, el algoritmo finaliza aquí; ahora bien, si lo que buscábamos eran todas las soluciones o la mejor entre todas ellas, el algoritmo seguirá explorando el árbol en búsqueda de soluciones alternativas.
Por otra parte, el recorrido no tiene éxito si en alguna etapa la solución parcial construida hasta el momento no se puede completar; nos encontramos en lo que llamamos nodo fracaso. En tal caso, el algoritmo vuelve atrás (y de ahí su nombre) en su recorrido eliminando los elementos que se hubieran añadido en cada etapa a partir de ese nodo. En este retroceso, si existe uno o mas caminos aun no explorados que pueden conducir a soluciones, el recorrido del árbol continua por ellos.

3.3. Ramificación y poda

Este método de diseño y algoritmos es en realidad una variante del diseño vuelta atrás.
Esta técnica de diseño, cuyo nombre proviene del termino ingles “branch and bound”, se aplica normalmente a problemas de optimización.
Una característica que le hace diferencia al diseño anterior (vuelta atrás) es la posibilidad de generar nodos siguiendo distintas estrategias. Recordemos que el diseño vuela atrás realiza la generación de descendientes de una manera sistemática y de la misma forma para todos los problemas, haciendo

un recorrido en profundidad del árbol que representa el espacio de soluciones.
La técnica de ramificación y poda utiliza cotas, para poder podar aquellas ramas que no conducen a la solución óptima. Para ello calcula cada nodo una cota del posible valor de aquellas soluciones alcanzables desde este. Si la cota muestra que cualquiera de estas soluciones tiene que ser necesariamente peor que la mejor solución hallada hasta el momento no necesitamos seguir explorando pro esta rama del árbol, lo que permite realizar el procedimiento de poda.
Definimos nodo vivo del árbol a un nodo con posibilidades de ser ramificado, es decir, que no ha sido podado. Para determinar en cada momento que nodo va a ser expandido, y dependiendo de la estrategia de búsqueda seleccionada, necesitaremos almacenar todos los nodos vivos en una estructura que podamos recorrer. Emplearemos una pila para almacenar los nodos que se han generado pero no han sido examinados en una búsqueda de profundidad (LIFO). La búsqueda en amplitud usan una cola (FIFO) para almacenar los nodos vivos de tal manera que van explorando nodos en el mismo orden que son creados. La estrategia de mínimo coste utiliza una función de coste para decidir en cada momento que nodo debe explorarse, con la esperanza de alcanzar lo mas rápidamente posible una solución mas económica que la mejor encontrada hasta el momento. Utilizaremos en este caso una estructura de montículo (o cola de prioridades) para almacenar los nodos ordenados por su coste.
Básicamente, en un algoritmo de ramificación y poda, se realiza tres etapas. La primera de ellas, denominada *selección* se encarga de extraer un nodo de entre el conjunto de los nodos vivos. La forma de escogerlos va a depender directamente de la estrategia de búsqueda que decidamos para el algoritmo. En la segunda etapa, la ramificación, se construye los posibles nodos hijos del nodo seleccionado en el paso anterior. Por ultimo se realiza la tercera etapa, la poda, en la que se eliminan algunos de los nodos creados en la etapa anterior. Esto contribuye a disminuir en lo posible y así atenuar la complejidad de estos algoritmos basados en la exploración de un árbol de posibilidades. Dependiendo de lo que se busca el algoritmo podría terminar en la primer solución encontrada si es que la hay o seguir buscando soluciones hasta que no le queden nodos disponibles.
Para cada nodo del árbol dispondremos de una función de coste que nos estime el valor optimo de la solución si continuáramos por ese camino. De esta manera, si la cota que se obtiene para un nodo, es peor que una solución obtenida por otra rama, podemos podar esa rama porque no es interesante seguir por ella.
En consecuencia, y a la vista de todo esto, podemos afirmara que lo que le da valor a esta técnica es la posibilidad de disponer distintas estrategias de exploración del árbol y de acotar la búsqueda de solución, en definitiva se traduce en eficiencia.
Inicialmente, y antes de proceder a la poda de nodos, tendremos que disponer del coste de la mejor solución encontrada hasta el momento que permite excluir de futuras expansiones cualquier solución parcial con un coste mayor. Como muchas veces no se desea encontrar la primer solución para empezar a podar, un buen recurso para el problema de optimización es tomar como mejor solución inicial la obtenida por un algoritmo ávido, que como vimos no encuentra siempre la solución óptima, pero si la mas cercana a la óptima.

3.4. Motores de búsqueda

Los motores de búsqueda representan implementaciones de las distintas estrategias para recorrer y encontrar soluciones en un árbol de búsqueda. Por lo cual se explicaran las estrategias y se darán algunos detalles de implementación.
Las dos primeras estrategias y de las cuales se derivan las demás, son búsqueda a lo ancho

(breadth-first search) y búsqueda en profundidad (depth-first search) . Ambos deben tener una forma de marcar los nodos como visitados.

3.4.1 Búsqueda a lo ancho

En esta búsqueda se utiliza una cola en la cual se van analizando los nodos almacenados de forma que si un nodo no es solución entonces se encolan todos sus hijos. Este análisis se va realizando hasta que se encuentra una solución o hasta que la cola quede vacía. El algoritmo comienza colocando el nodo correspondiente al estado inicial en la cola.
Esta estrategia tiene la ventaja de que si existe una solución, esta sera encontrada, si hay varios nodos exitosos en el árbol, la solución corresponderá al nodo de profundidad mínima. Y la desventaja de que la cantidad de nodos almacenados crece exponencialmente.

3.4.1.1. Primero el Mejor

Esta técnica esta basada en búsqueda informada y se basa en ordenar los estados sucesores de mayor a menor según la función de evaluación. Dependiendo de la función de evaluación, esta técnica puede derivar en una búsqueda en profundidad o búsqueda a lo ancho dependiendo de la calidad de la misma. En términos de implementación, el algoritmo es una modificación del Breath-First Search donde se cambia la cola por una cola de prioridades.

3.4.2. Búsqueda en profundidad

En esta búsqueda se comienza analizando un nodo, si este es exitoso se retorna, y sino primero se obtienen todos sus hijos y se realiza una búsqueda recursiva comenzando con el primero de estos hijos. Este análisis comienza con la búsqueda del nodo correspondiente al estado inicial. Otra implementación podría ser mediante el uso de una pila, pero en este caso el tamaño de almacenamiento sera mucho mas grande que utilizar recursión sobre los hijos.
Esta estrategia tiene la ventaja de tener un almacenamiento lineal con respecto a la altura del árbol. Pero tiene la desventaja de poder entrar en un ciclo infinito (un nodo tiene como hijo a otro que a su vez lo tiene como hijo al primero).

3.4.2.1. Iterative Deepening

Esta estrategia agrega una cota de profundidad al método de búsqueda en profundidad, haciendo que este busque hasta una profundidad n y si no encuentra una solución, este vuelve a realizar la búsqueda desde el estado inicial pero de una profundidad n + k. Repitiendo el proceso hasta encontrar una solución. Esta modificación a búsqueda en profundidad da la ventaja que tiene búsqueda a la ancho, sobre encontrar la solución en profundidad mínima y agrega la desventaja de volver a analizar nodos previamente analizados.

3.4.2.2. Hill Climbing

Utilizando una función de valoración de estados, podemos definir un nuevo algoritmo:
1. Comenzar con el estado inicial como estado actual

2(a). Si el estado actual es un estado exitoso, terminar exitosamente
2(b). Sino, obtener los hijos del estado actual, uno a la vez, hasta encontrar uno con mayor valoración que el estado actual o agotar todos los hijos
3(a). Si no existen hijos con mayor valoración que el estado actual, terminar
3(b). Sino, volver al paso 2, con el primer hijo con valoración mayor como estado actual
Esta técnica es un ejemplo de búsqueda informada y se basa en la idea de que todo hijo con un mejor puntaje que el estado actual debe ser un paso hacia la solución. Esta idea no funciona cuando hay problemas donde se pueda tener etapas con un bajo puntaje que sin embargo llegan a una solución. Casos del estilo "a veces para avanzar es necesario retroceder"

3.4.3. Búsqueda con adversarios

Los motores de búsqueda con adversario cambian con respecto a los motores de búsqueda mencionados anteriormente en que durante la búsqueda se tiene que considerar que hay dos tipos de soluciones, una para cada jugador y cada nivel corresponde a un jugador en particular.

3.4.3.1. MinMax

Es una búsqueda en profundidad en la cual se considera que el adversario siempre realiza la mejor jugada. Cada nivel del árbol corresponde a un nivel MIN o un nivel MAX, los cuales corresponden cada uno a cada jugador. Con lo cual el algoritmo básico se puede explicar como:
Si llegaste a una hoja retorna la evaluación de esa hoja, sino si estas en un estado MAX entonces hay que quedarse con el sucesor que tenga la máxima evaluación y si estas en un estado MIN nos quedaremos con el sucesor que tenga la mínima evaluación.
Una variación de este algoritmo es agregarle una condición extra en el caso base considerando una altura máxima, permitiendo ejecutar el algoritmo sin tener que llegar a estados finales, los cuales pueden estar a una profundidad muy alta en casos como por ejemplo el ajedrez. Esta variación se puede considerar como "pensar las próximas n jugadas", donde n seria la altura máxima del árbol.

3.4.3.2. MinMax con poda Alfa-Beta

Este algoritmo es una variación del MinMax en donde los sucesores de un nodo se evalúan hasta que se llega a una situación donde se puede saber con certeza que no es posible conseguir un estado mejor. [REF#1]
El valor Alfa, representa a la mejor evaluación conseguida para un nivel MAX y el valor Beta representa la mejor evaluación conseguida para un nivel MIN.
Cada nodo hijo que se evalúa actualiza el valor de Alfa-Beta según corresponda. El ciclo de evaluación tiene la condición de "para cada nodo hijo y mientras alfa < beta".

3.5. Ejemplos de búsqueda

A continuación presentaremos algunos ejemplos de problemas de búsqueda, especificando que tipo de búsqueda es (con y si adversario, informada y no informada), finalmente se mostrara que cosas en común tienen los problemas anteriores. A si mismo se mostrará como sería la solución que se le daría normalmente a cada uno, la solución está basada en que definir 2 clases específicas al problema y la o las clases necesarias para el motor de búsqueda, cabe notar que el motor de búsqueda es en general genérico pero solo para el tipo de problema (con y si adversario, informada y no informada), un motor de búsqueda no informada no serviría para un problema de búsqueda informada. Las dos clases a definir para cada problema son el estado y el problema, esto es por que es una de las formas mas comúnes al definir problemas de búsqueda.

3.5.1. Problema de las jarras de agua

Tenemos dos jarras, con capacidades de 4 y 3 litros respectivamente. Las jarras no tienen líneas de medición, y queremos que la primera quede con exactamente 2 litros de agua. Contamos con una fuente de agua ilimitada, y podemos llenar alguna de las jarras, o volcar el contenido de una de las jarras en la otra o tirar el contenido de una jarra.

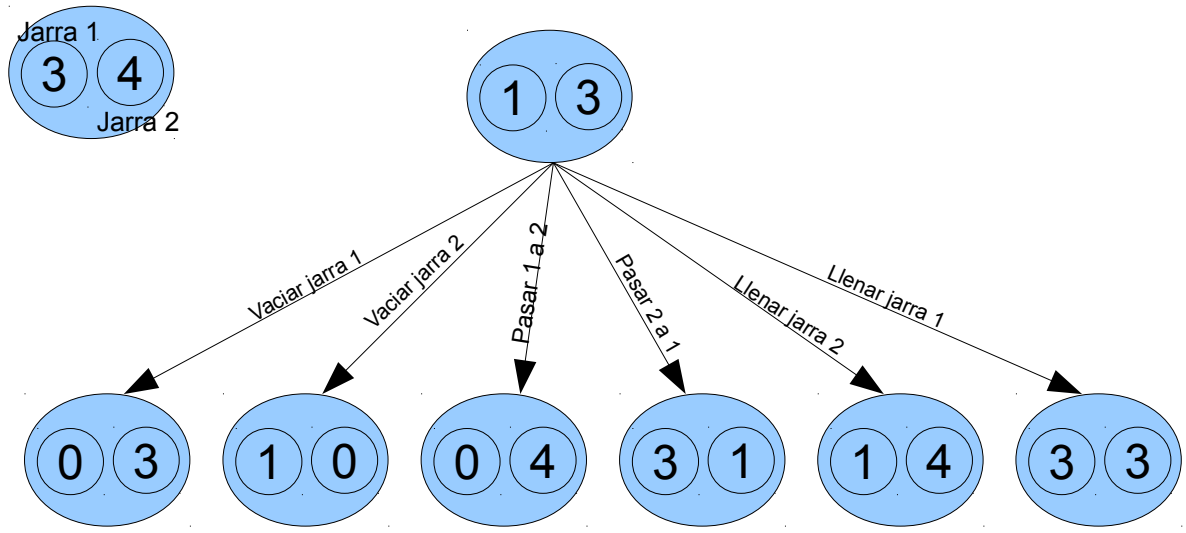

Figura 1 – ejemplo de pasos a realizar durante el problema de las jarras

Este problema se resuelve algoritmicamente utilizando un motor de búsqueda no informado (en la resolución mas simple, ya que seria posible implementarlo con un motor de búsqueda informado), hay que tener en cuenta que por la falta de restricciones (en cuanto a cantidad de pasos) en los cambios de estados seria posible generar ciclos infinitos de cambio, como por ejemplo llenar y vaciar la misma jarra como únicos dos pasos.

Solución:

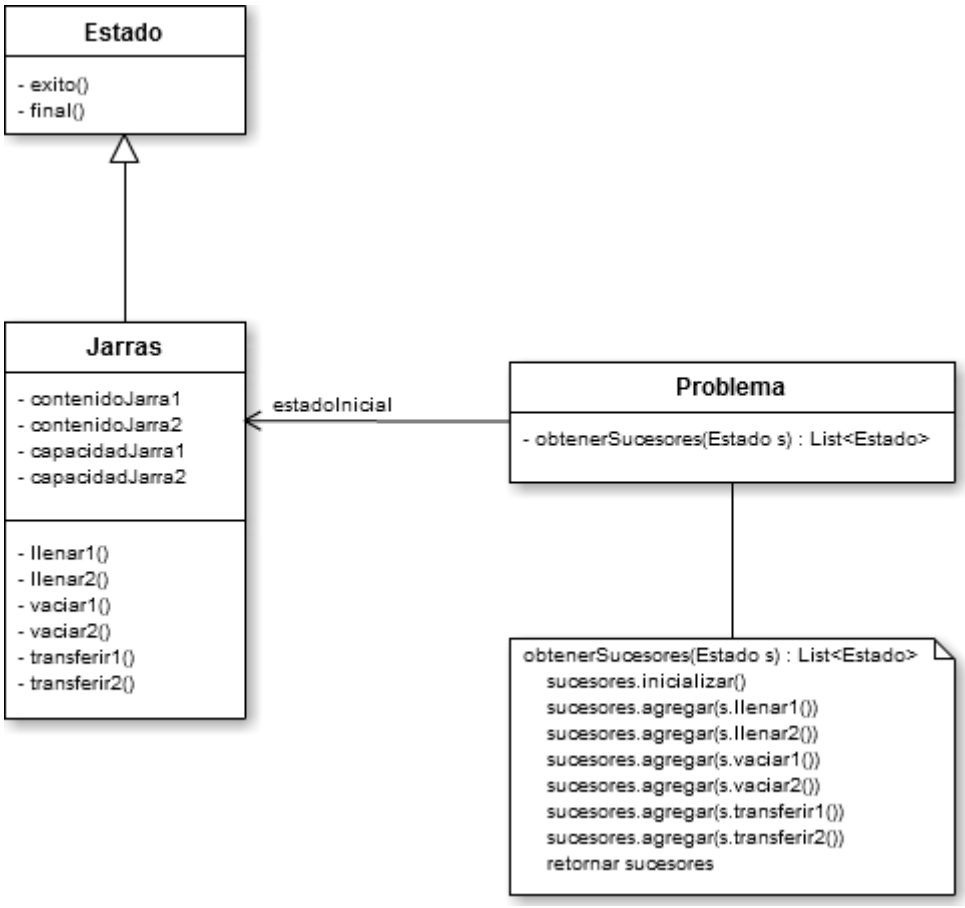

Figura ??? – solución al problema de las jarras de agua

En el diagrama anterior podemos ver la solución para el problema, con un pseudocódigo del método para obtener sucesores. Se obviaron ciertas cuestiones como la generación de clones del estado al ir agregando cada sucesor a la lista y la comprobación para cada cambio de estado (verificar que una jarra no está vacía antes de aplicar el método vaciar por ejemplo).
Los método obligatorios son los que se encuentran en la clase "Estado" y son los únicos que el motor de búsqueda necesita utilizar.

3.5.2. Problema de Indiana Jones

Indiana Jones, acompañado de su novia, su padre y su suegro, necesita cruzar un puente colgante, un tanto peligroso, de 1 kilometro de longitud. Esta tan oscuro en el lugar, que es imposible cruzar el puente sin una linterna. Además, el puente es tan débil que solo soporta como máximo a dos personas sobre el puente, y la luz de la linterna es tan débil que cuando dos personas caminan juntas sobre el puente, estas se ven forzadas a hacerlo a la velocidad del mas
lento de ellos. Indiana Jones puede cruzar el puente en i minutos, su novia en 2 × i minutos, el padre de Indiana en 4 × i minutos, y el suegro en 5 × i minutos. Para que no los atrapen los villanos, deben poder cruzar el puente en M minutos como máximo.

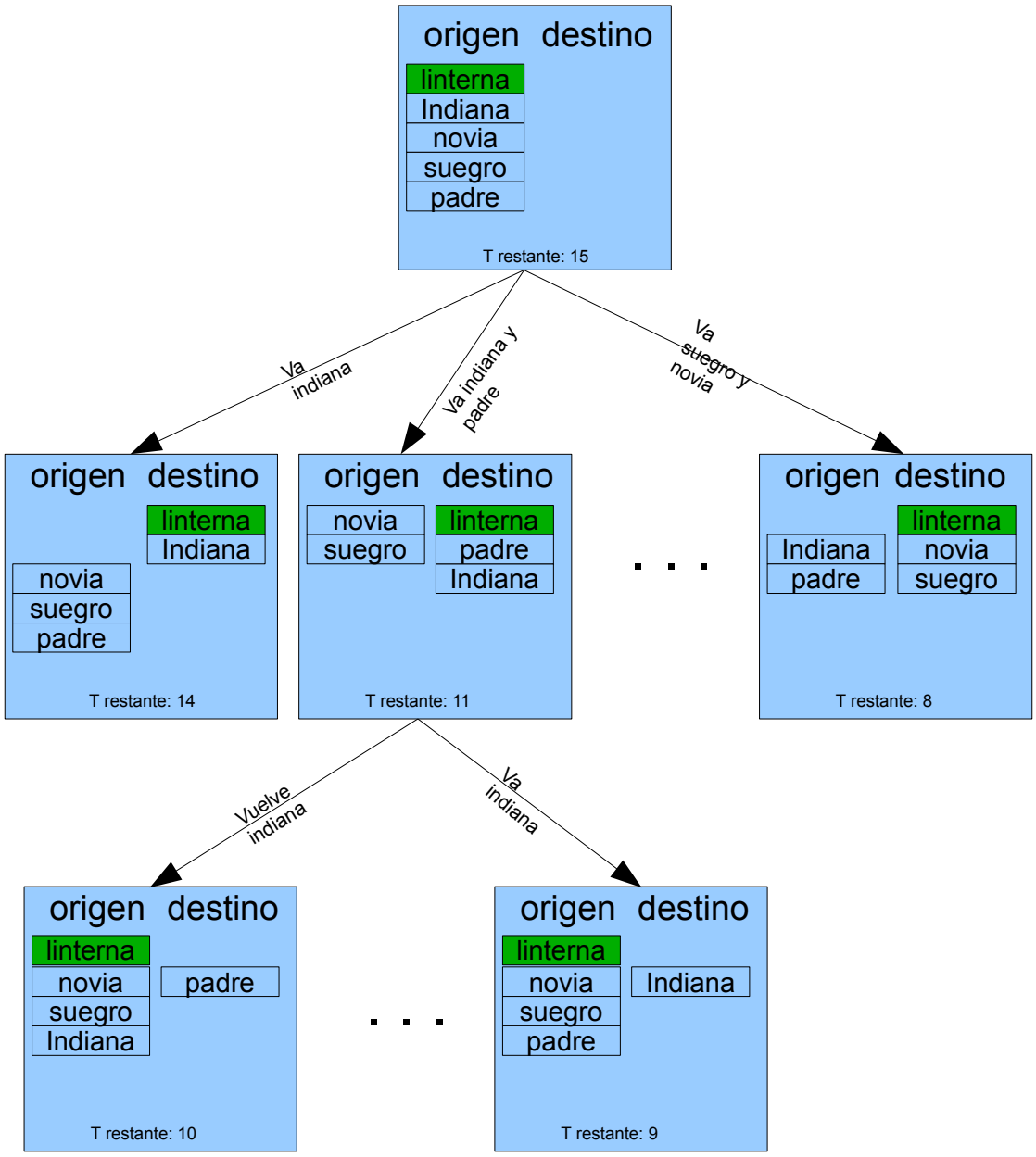

Figura 2 – ejemplo de pasos a realizar durante el problema de Indiana Jones

Este problema se resuelve de la misma forma que el ejemplo anterior, con la diferencia de que no se pueden generar ciclos infinitos en el árbol de búsqueda ya que se tiene la restricción de tiempo y cada paso disminuye el tiempo disponible, y para que se pueda generar un sucesor de un estado es necesario que almenos alguno de los personajes del problema tenga un tiempo menor igual al disponible. Cabe aclarar que es posible saber la altura máxima del árbol de búsqueda dividiendo el tiempo disponible del problema por el tiempo mínimo entre los de los distintos personajes.

Solución:

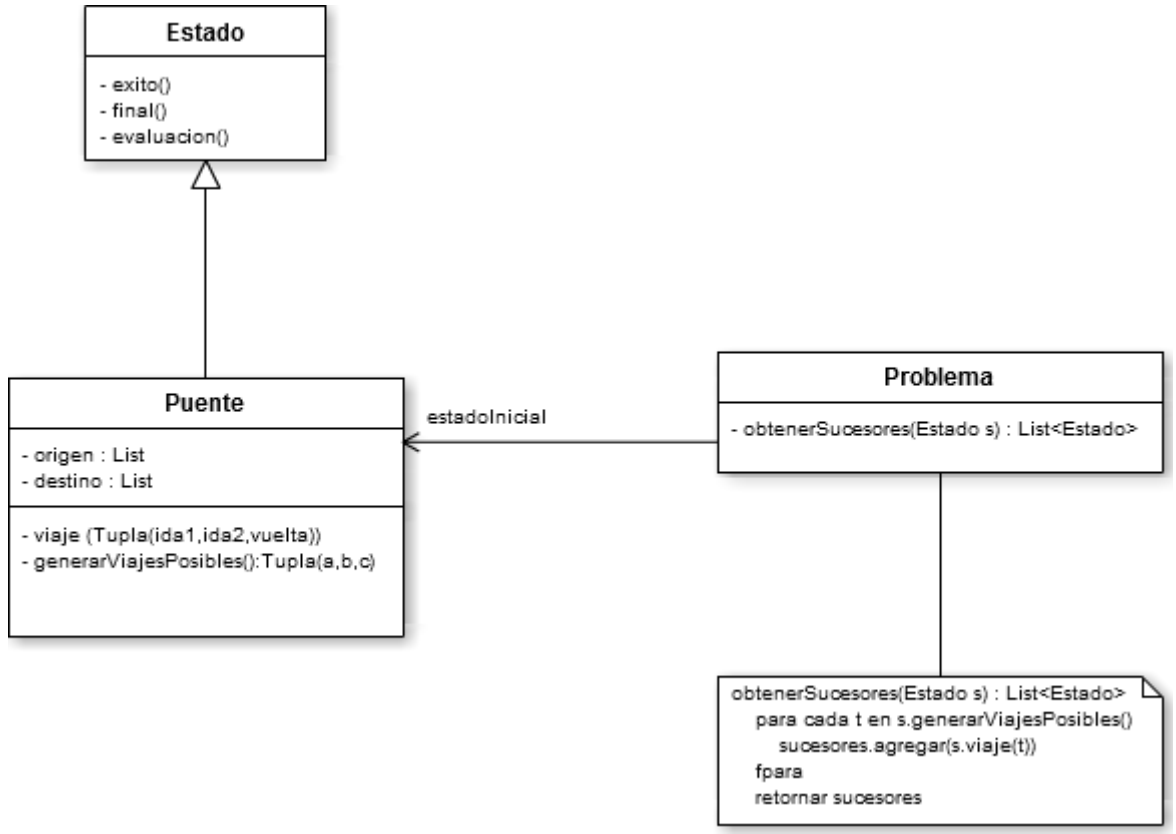

Figura ??? – solución al problema de indiana

Una solución es ignorar la linterna y tener en cuenta que un viaje es una tupla de tres elementos, dos personas que van y una que vuelve, teniendo en cuenta que es posible usar algún valor especial (NULL por ejemplo) para caso donde solo una persona viaja o viajes donde nadie vuelve (en el último viaje por ejemplo). Esto facilita el estado y es más fácil la obtención de sucesores.
[VALE LA PENA PONER UNA FUNCIÓN DE EVALUACIÓN???]

3.5.3. Problema de Ta-Te-Ti

Pueden participar dos personas y para jugar al Ta Te Ti, cada una ellas deberá escoger entre X ú O, símbolos con los cuales irán completando los distintos casilleros de un tablero de 3x3 respetando su turno. Ganará aquel que consiga ubicar tres símbolos en una misma línea, la que puede ser diagonal, horizontal o vertical

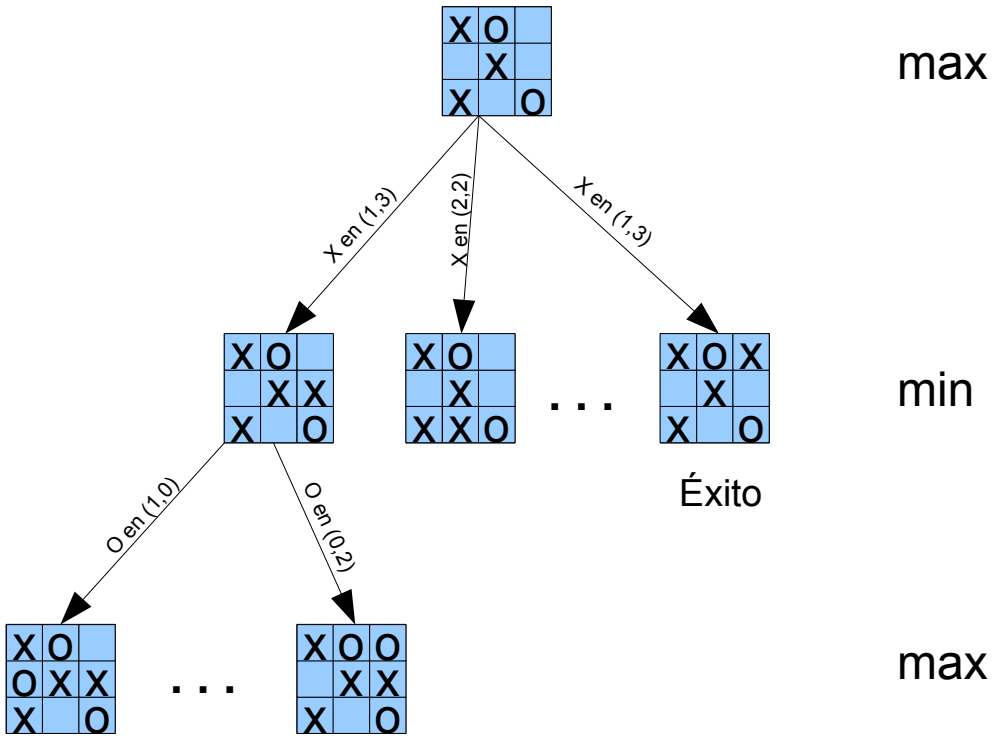

Figura 3 – ejemplo de pasos a realizar durante el problema del ta-te-ti

Este problema pertenece a la clase de problema de búsqueda con adversario y por lo tanto hay que tener en cuenta que hay dos soluciones posibles, una para cada jugador, y las cuales son opuestas, es decir, que si un estado es solución para el jugador uno entonces no puede serlo para el jugador dos y viceversa. La forma de resolver los problemas con adversarios requiere que la solución a obtener cambie según el jugador que esta jugando (el nivel del árbol), generalmente esto se logra buscando la solución asumiendo que el jugador que le toca en el turno siguiente va a realizar la mejor jugada. En caso que el árbol no se recorra hasta llegar a un estado final (se gano, se perdió, se empato) sera necesario cambiar la búsqueda de una que se basa en encontrar la solución del problema a una que se basa en encontrar el mejor movimiento posible sin saber si el mismo llevara a la solución. En este caso sera necesario una función de valoración para saber cual cerca esta un jugador de ganar dado un estado particular y como se debe tener valoración (una por cada jugador) sera necesario dos constantes que representen los dos extremos de esta valoración.

Solución:

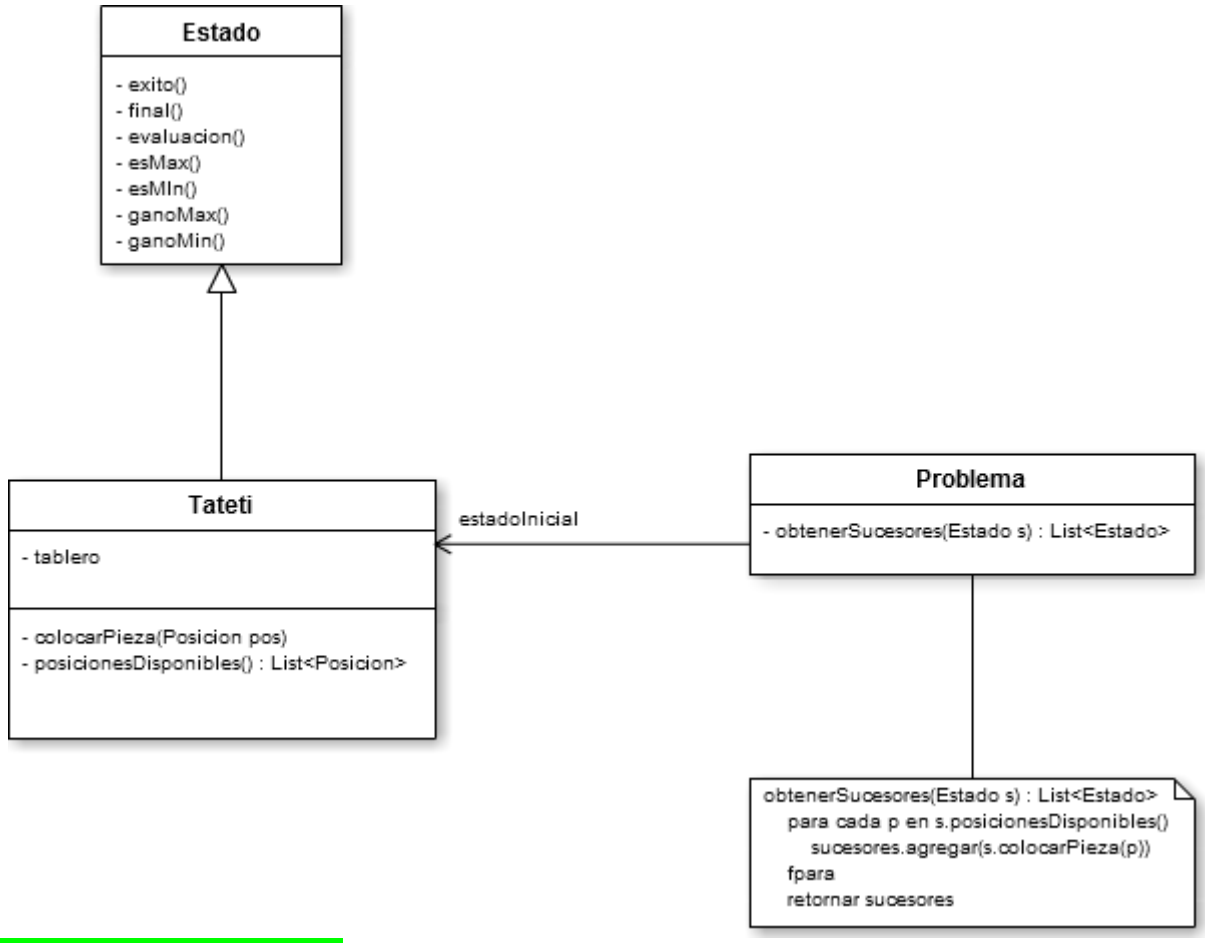

Figura ??? – solución ta-te-ti

4. Framework

4.1. Metodología de trabajo

Para el desarrollo del producto primero se va a comenzar por realizar un diseño (diagrama de clases) hasta obtener un diseño coherente y estable (un diseño con pocas probabilidades de ser modificado o que solo se le hagan pequeñas modificaciones). Una vez obtenido el diseño se va a proceder por desarrollar la plataforma, luego el motor de búsqueda con adversario y el juego con adversario para poder testear al motor, cuando el motor este funcionando correctamente se desarrollara el motor de búsqueda sin adversario, el juego sin adversario para luego seguir con la interfaz gráfica y testear el motor. En períodos de entre 7 a 21 días se realizarán reuniones con el director del proyecto para corroborar el diseño que se está siguiendo, informar estado de avance, realizar consultas y organizar reuniones, estas tendrán una duración máxima de 2 horas.

4.2. Especificaciones de requerimientos

4.2.1. Introducción

El propósito de este documento es dar toda la información importante sobre el proyecto a realizar "*Garoé: Una Plataforma Orientada a Objetos para el Desarrollo de Soluciones Algorítmicas basadas en Búsqueda*", incluidas su descripción, requerimientos y restricciones. Así también como documentación auxiliar que acompañe a la información del proyecto y documentación sobre la cual nos basamos para realizar el mismo.

El público esperado para este documento son personas con una formación técnica suficiente para que no sea necesario ampliar esta documentación a fin de explicar técnicas y conocimiento aplicados en el desarrollo del proyecto. Aunque cualquier documentación que sea considerada importante o que haya sido utilizada durante el desarrollo será incluida en la sección de referencias. Personas incluidas en el alcance incluyen a cualquiera que tenga conocimientos sobre: Programación orientada a objetos, técnicas de programación para problemas búsqueda y conocimientos básicos de diseño (utilizados para comprender los diagramas de diseño).

4.2.1.1. Alcance

Producto : ***Garoé****: Una Plataforma Orientada a Objetos para el Desarrollo de Soluciones Algorítmicas basadas en Búsqueda*
El producto proveerá una plataforma (conjunto de interfaces, clases abstractas y clases concretas) que permita el desarrollo de:

- Motores de búsqueda (con y sin adversario)
- Especificación de estados
- Especificación de reglas de cambio de estado
- Aplicaciones que utilicen un motor de búsqueda

- Extensiones de la plataforma (contemplan a estados, problemas, motores y reglas)

Así mismo el producto contendrá motores de búsqueda implementados, de los cuales al menos uno será para búsquedas sin adversario y otro con adversario.

Finalmente el producto tendrá dos ejemplos de uso, uno utilizaran búsqueda con adversario y el otro búsqueda sin adversario.

El producto esta pensado para ser utilizado en los siguientes casos:

- Desarrollar elementos para realizar búsquedas
  - Motores de búsqueda
  - Reglas de cambio de estado
  - Especificaciones de estados
- Especializar partes de la plataforma
- Utilizar instancias de la plataforma (estados, reglas, motores de búsqueda) en otras aplicaciones

Los beneficios de esta plataforma en contraste con no tenerla es liberarse de tener que diseñar toda la estructura base y estructuras auxiliares necesarias para comenzar con el verdadero problema que se quería resolver.

Mas específicamente los beneficios se pueden dividir según los casos de uso:

- Desarrollar un motor de búsqueda: Ya se tienen motores abstractos sobre los cuales se implementaría el motor y ya están definidas las operaciones de los objetos que el motor puede o debería usar. Solo es necesario implementar el método de búsqueda y algunos métodos secundarios cumpliendo con los perfiles correspondientes.

- Desarrollar un problema: En caso de querer desarrollar un problema solo es necesario definir el estado (utilizando el estado abstracto correspondiente) y las reglas de cambio de estado.

- Utilizar la plataforma con un problema y motor dado dentro de una aplicación: Si se tiene implementado un problema y uno o mas motores de búsqueda (compatibles con el mismo) uno podría usar la plataforma para conectar la aplicación con los mismos.

Para cualquiera de los casos se asegura que la integración con el resto de la plataforma es completamente funcional

En el resto del documento se detallarán las secciones de, **descripción general** donde describirá de forma superficial el producto; **características de usuario** donde detallarán que propiedades en común tendrán los mismos y se listarán los conocimientos básicos que deberán tener los mismos; **restricciones** que deberá cumplir la plataforma, cabe aclarar que estas no prevén los productos que se puedan generar a partir de la utilización de la plataforma; **dependencias y suposiciones** donde se describe qué requerimientos se deben cumplir para la correcta utilización de la plataforma;

requerimientos funcionales donde se describen los comportamientos que debe tener cada funcionalidad del sistema a las distintas entradas (válidas e invalidas), las validaciones que se aplican a los valores de entrada, la secuencia de operaciones que realiza en los distintos escenarios; **restricciones de diseño** donde se encontrarán restricciones como (en nuestro caso) seguir un determinado patrón, aspectos de visibilidad y responsabilidades que deben mantener ciertos componentes; **atributos del sistema** en donde sólo se considerará la portabilidad de la plataforma y todo sistema que se implemente sobre la misma; **información de soporte** incluirá índice, apéndice y referencias.

El resto del documento se organiza de la siguiente forma:

Una descripción general del producto, seguido de las características de cada tipo de usuario esperado para la utilización del mismo. Las restricciones, dependencias y suposiciones que se tuvieron en cuenta para el producto. Una sección con los requerimientos funcionales de cada submodulo, es decir, el motor, problema, estado y reglas de cambio de estado.
Se detallará si existen y cuales son las restricciones del diseño y los atributos del sistema que se deben/desean lograr.
Finalmente se podrá encontrar todo lo relacionado a información de soporte de esta sección del informe, índice, apéndice y referencias utilizadas.

4.2.1.2. Descripción general

El producto a desarrollar es una plataforma que permita el desarrollo de motores de búsqueda (con y sin adversario), definición de problemas de búsqueda (estado y reglas de cambio de estado) y la utilización de motores y problemas ya desarrollados para una aplicación en particular (ej: un juego de tablero). La razón para desarrollar esta plataforma es facilitar las tareas de diseño e implementación en los casos previamente nombrados.
Los usuarios para los cuales esta plataforma está pensada se pueden dividir en tres categorías dependiendo de las metas de los mismos. Cada categoría de usuario tiene distintas características, pero es posible dar una serie de características generales las cuales son compartidas por todos los usuarios por igual.
Esta plataforma está destinada a programadores, con conocimientos sobre programación orientada a objetos, con lo que eso implica (herencia, encapsulamiento, etc), cierta experiencia o conocimiento sobre especificaciones de programas mediante pre y post condiciones, etc. Dado que el sistema está desarrollado en Java, experiencia en este lenguaje es sugerida pero no requerida. Particularmente cada categoría de usuario tendrá características particulares:

Desarrollador de motores de búsqueda:
Utiliza la plataforma para el desarrollo de motores de búsqueda tanto sin y con adversario, no tiene en cuenta los problemas específicos para los cuales se utilizará el o los motores. Además de los conocimientos generales, el desarrollador de motores de búsqueda tiene conocimientos sobre algoritmos de búsqueda y sus implementaciones.

Desarrollador de soluciones algorítmicas a problemas de búsqueda:
Utiliza la plataforma para desarrollar soluciones algorítmicas para problemas que pueden ser resueltos mediante algoritmos de búsqueda. Además de los conocimientos generales, el desarrollador de soluciones algorítmicas a problemas de búsqueda debe tener experiencia

reconociendo características significativas de un problema, cómo representarlo como un problema de búsqueda, y encontrando buenas representaciones para el o los objetos sobre los cuales actúa el problema.

Desarrollador de aplicaciones que utilizan motores de búsqueda:
Utiliza la plataforma como un nexo entre un motor de búsqueda y un problema particular con una aplicación que requiere resolver un problema (de búsqueda) como función principal o como parte de un sistema mas grande. Este tipo de usuarios debe tener conocimientos sobre utilización de APIs (interfaces) además de los conocimientos generales antes descriptos.

4.2.1.3. Restricciones del producto

El producto está desarrollado enteramente en Java, con lo cual el mismo es portable a todas las plataformas para las cuales se cuente con entornos de ejecución Java. Cualquier versión de Java igual o superior a la versión 1.6 es suficiente para utilizar el producto (el mismo fue testeado utilizando versiones Sun JDK 1.6 y 1.7). Salvo estas restricciones, el producto no posee otras identificables.

4.2.1.4. Dependencias y suposiciones

Este producto requiere la disponibilidad de una maquina virtual de Java y un SDK de Java que permita las siguientes funciones:

- Genericidad
- Restricciones en parámetros de clase
- Utilización de estructura “for” con iteradores, ej: for (elem : collection) {código}

Se asume que el usuario de esta plataforma va a desarrollar utilizando como lenguaje a Java (con las restricciones de versión anteriores)

4.2.1.5. Restricciones de diseño y requerimientos funcionales

La plataforma esta restringida a la siguiente estructura:

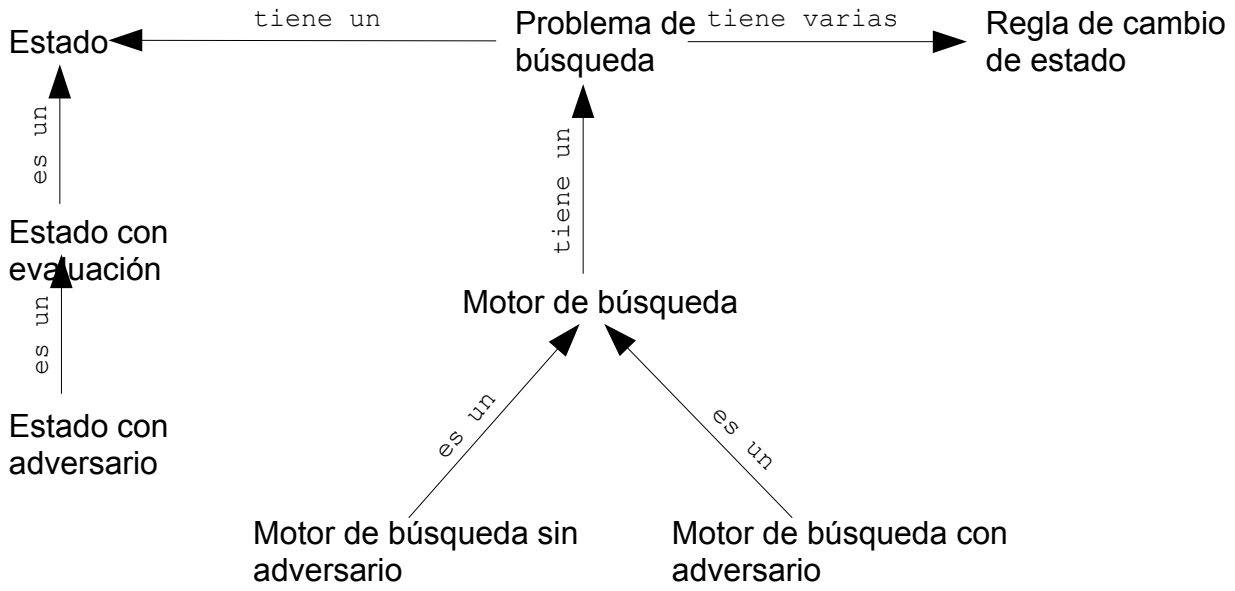

Figura 4 – estructura básica de la plataforma

Ya que el producto es una plataforma y no un sistema funcional en si mismo, los requerimientos funcionales se darán en cuanto a lo que "debería" hacer cada parte de la plataforma, definiendo responsabilidades y comportamientos esperados. Las partes de la plataforma son las que se ven en el diagrama anterior y las mismas deben respetar la estructura dada en sus responsabilidades y la manera en la que las cumplen.

4.2.1.5.1. Requerimientos funcionales del motor de búsqueda

- Definir el problema de búsqueda sobre el cual se va a ejecutar el algoritmo del motor de búsqueda:
entrada: un problema de búsqueda con un estado inicial definido y el conjunto de reglas de cambio de estado asociado a dicho problema.
restricciones del parámetro (problema): El problema, el motor, el estado y las reglas operan sobre un determinado tipo; estos tipos tienen que ser iguales o compatibles para los elementos.
comportamiento: establecer al parámetro de entrada como problema del motor.

- Obtener el estado solución del problema de búsqueda que encuentra el algoritmo de luego de ejecutarse:
restricciones de la función: el algoritmo de búsqueda debe haber sido ejecutado y debe haber encontrado una solución.
Comportamiento: este método retornara la solución encontrada por el algoritmo de búsqueda en caso de existir

4.2.1.5.2. Requerimientos del motor de búsqueda con adversario

- obtener el movimiento “pensado” por el algoritmo de búsqueda.
entrada: el estado del juego desde donde el motor de búsqueda tiene que buscar el movimiento a realizar.

Restricción de la función: los tipos sobre los cuales están definidos el motor de búsqueda, las reglas, el problema y el tipo del estado deben ser iguales o compatibles.
comportamiento: realiza una búsqueda para encontrar el mejor movimiento posible dado el estado de entrada.

- computar evaluación de un estado
entrada: un estado
restricción de la función: el estado debe ser informado.
comportamiento: ejecutar un método de evaluación determinado sobre la entrada

4.2.1.5.3. Requerimientos del motor de búsqueda sin adversario

- Obtener el camino desde el estado inicial hasta el estado solución, tanto el estado inicial como el estado solución deben estar en la lista (excepto que sean el mismo) y todos los estados intermedios deben estar en orden, donde cada uno representa el estado anterior luego de ejecutarse una regla de cambio de estado.

Restricciones de la función: el algoritmo de búsqueda debe haber sido ejecutado, no necesariamente se tiene que haber encontrado una solución
comportamiento: retornar la lista de estados intermedios desde el estado inicial hasta el estado solución, en caso de que el problema no haya tenido solución la lista debe ser vacía.

- Ejecutar el algoritmo de búsqueda sobre el estado inicial, definido en el problema sobre el que realiza la búsqueda el motor.

Restricciones de la función: el problema debe estar definido, esto implica que tenga un estado inicial, que tenga un conjunto de reglas de cambio de estado y que el tipo sobre el cual opera el problema, las reglas y el motor sean iguales o compatibles al tipo del estado.
comportamiento: ejecutar el algoritmo de búsqueda partiendo desde el estado inicial del problema
Resultado: verdadero si y sólo si existe estado s tal que existe un conjunto de reglas R tal que se cumple que el estado inicial deriva mediante R en s y el estado s es exitoso.

4.2.1.5.4. Requerimientos funcionales del problema

- definir el estado a partir del cual se empieza a resolver el problema
entrada: un estado
restricciones de la función: el tipo del estado y el tipo sobre el cual opera el problema deben ser iguales o compatibles
comportamiento: definir el estado parámetro como estado inicial del problema

- definir el conjunto de funciones utilizadas para pasar de un estado a otro (reglas)
entrada: un conjunto de reglas de cambio de estado
restricciones de la función: las reglas deben ser aplicables al tipo de estado sobre el cual opera el problema
comportamiento: establecer las reglas de entrada como las reglas del problema

- obtener un conjunto de estados donde cada uno de estos fue generado por alguna función de cambio de estado (regla)
entrada: un estado
comportamiento: aplica todas las reglas de cambio de estado (aplicables) al estado de entrada, cada regla tiene una función que permite saber si la misma es aplicable a un estado dado.
resultado: un conjunto de estados obtenidos al aplicar todas las reglas al estado entrada.

4.2.1.5.5. Requerimientos funcionales del estado

- Consultar si un estado es final, es decir, si es posible aplicar un cambio de estado de un solo paso sin romper el invariante del mismo.
Entrada: se aplica sobre un estado
resultado: retorna si al estado se le puede aplicar algún cambio (siguiendo las reglas del problema)

- Consultar si un estado puede considerarse como solución al problema de búsqueda para el cual este fue definido
entrada: se aplica sobre un estado
resultado: retorna si el estado es exitoso en relación al problema
- Comparación con otro estado
entrada: se aplica sobre un estado y necesita de parámetro otro estado
resultado: un valor que especifique la relación de orden del estado actual con el de entrada

4.2.1.5.6. Requerimientos funcionales de los estados con evaluación

- Evaluar el estado (asociar un valor al mismo) dependiendo de “cuan bueno” es el estado en relación al problema de búsqueda que se quiere solucionar
resultado: un valor que define cuan bueno es un estado en relación a la resolución del problema

- Constante de valor mínimo de evaluación
resultado: el valor mínimo posible que puede arrojar la evaluación de un estado

- Constante de valor máximo de evaluación
resultado: el valor máximo posible que puede arrojar la evaluación de un estado

4.2.1.5.7. Requerimientos funcionales de los estados para adversario

- Consultar si un estado corresponde a un nivel MAX o un nivel MIN, es decir, a que jugador

pertenece dicho estado
resultado: retornar si el estado esta en nivel MIN o MAX

- Consultar si un estado es exitoso para un nivel (jugador) en particular
resultado: retornar si el estado es exitoso para el nivel (jugador) consultado
nota: se debe cumplir que la funcionalidad de estado "es exitoso" debe ser igual a ganó MIN ó ganó MAX

4.2.1.5.8. Requerimientos funcionales de reglas de cambio de estado

- Verificar si una regla en particular puede ser aplicada a un estado, es decir, verificar si la aplicación de esta regla no invalida las reglas del juego o problema de búsqueda
entrada: un estado
restricciones de la función:el tipo sobre el cual opera la regla debe ser el mismo o compatible al tipo del estado
resultado: retorna si es posible aplicar la regla al estado pasado como entrada
nota: se debe mantener siempre la condición de si el estado es final entonces la regla no es aplicable

- Cambiar el estado aplicando la función asociada a la regla
entrada: un estado
restricciones de la función:el tipo sobre el cual opera[3] la regla debe ser el mismo o compatible al tipo del estado
comportamiento: generar todos los estados resultantes de aplicar la regla al estado entrada
resultado: retornar los estados generados

4.2.1.6. Atributos del sistema

4.2.1.6.1. Portabilidad

La plataforma esta realizada en Java, el cual es un lenguaje portable que puede ser compilado y ejecutado en cualquier plataforma operativa que posea un SDK y una maquina virtual para el mismo. Ninguna parte de la plataforma posee código que "ate" a una librería particular, por lo tanto la portabilidad de esta plataforma es completa.

3Cuando se habla de "el tipo sobre el cual opera X estructura" se hace referencia al uso de genericidad (parámetros de clases). Cuando se habla de tipo compatibles se refiere a polimorfismo, la clase C es compatible con la clase B si ambas heredan o implementan a la clase A.

4.3. Diseño de clases no detallado

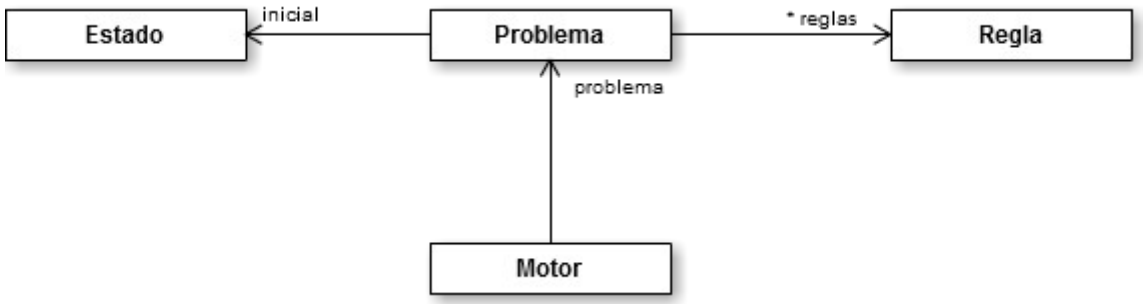

Figura 5 – estructura básica del diseño

La plataforma consta de tres partes básicas, las cuales son: el estado, donde se define la estructura para una instancia del problema a resolver; regla, donde se define una función para, partiendo de un estado, se obtienen todos los estados alcanzables desde el mismo en un solo paso; el problema, el cual mediante un estado obtiene todos los estados alcanzables utilizando una lista de reglas asociadas al problema a resolver; finalmente el motor que a partir de un estado intenta buscar una solución a un problema particular.

4.3.1. Estado

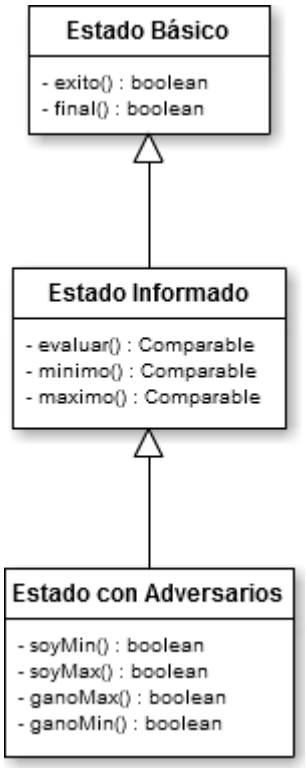

Figura 6 – estructura básica del diseño de estado

Un estado define la representación de una instancia particular de un problema, el problema en sí se define mediante el estado y las reglas de cambio, para nuestra plataforma el estado a su vez es el encargado de definir operaciones referentes a la búsqueda de una solución, estas incluyen saber si un estado es una solución, si el estado ya no puede aceptar cambios, que tan bueno es estado respecto al problema, etc.
Para las clases involucradas en la búsqueda, solo los métodos relacionados a la misma son requeridos del estado, esto también permite que un motor de búsqueda no informada utilice un estado informado (el motor va a ignorar el método de evaluación así como cualquier otro que no necesite).
Un estado define una instancia de un problema determinado, como por ejemplo tener una jarra con 2 litros de agua y otra con 1, es una instancia del problema de las jarras de agua. Por lo tanto es necesario que un estado pueda informar si el mismo es exitoso o no para un determinado problema. Esta información podría estar en el problema a resolver, tiene sentido que un problema pueda determinar si un estado determinado es solución o no, sin embargo al hacer que el problema solo maneje la obtención de sucesores se puede tener un problema único al cual se le asigna un estado inicial y un conjunto de reglas en lugar de tener que escribir una nueva clase por cada problema nuevo, sobre todo cuando solo es un método el que se necesita implementar. Otra razón para que sea el estado el que sea responsable de verificar si es o no solución del problema es la legibilidad de los algoritmos que lo utilizan, ya que leer algo como "si estadoActual.esExitoso..." es mas claro que leer algo como "si problema.esExitoso(estadoActual)". Finalmente se dividen los estados dependiendo del tipo de búsqueda que se realiza, búsquedas donde solo se busca una solución recorriendo el árbol a medida que se obtienen sucesores solo requieren que se pueda saber si un estado es o no exitoso; búsquedas donde el resultado pueda no ser un estado exitoso sino el mejor estado que se puede conseguir en una cantidad fija de pasos, o bien una búsqueda donde los estados sucesores son visitados teniendo en cuenta una valoración que indica (de manera aproximada) cuan cerca está un estado de resolver el problema, requieren que el estado pueda retornar su evaluación; finalmente cuando se realiza una búsqueda para un problema con adversarios, es decir, problemas donde hay dos agentes y hay una solución para cada uno de ellos, es necesario saber cuando un estado es solución para uno y cuando para el otro, y dado que hay dos soluciones totalmente opuestas, la evaluación ya no representa la cercanía a la solución sino que debe representar a cual de las soluciones está mas cerca, por lo que es necesario conocer cuales son los dos valores extremos que puede retornar.

4.3.2. Regla

Figura 7 – estructura básica del diseño de las reglas[4]

Todo problema requiere que se pueda pasar de un estado a otro, en problemas de búsqueda esto se conoce como obtener el/los sucesores de un estado. Si bien es posible implementar esta obtención de sucesores tanto en el problema como en el estado, hacerlo de esta forma es un obstaculo para

4 Cada regla se define para un estado particular, eso es por que necesita acceso a los métodos de lectura y modificación del mismo

hacer una solución genérica que cada problema en particular debe completar con los detalles específicos del mismo. Por lo tanto la forma mas simple de ver esta obtención es por medio de funciones que definen las distintas modificaciones que se le pueden realizar a un estado, cada una de estas modificaciones solo necesita comprobar si se puede realizar y luego si es posible realizarla. Esta visión permite el intercambio de reglas dando una gran flexibilidad a la definición de nuevos problemas reutilizando soluciones anteriores.
En todo problema de búsqueda es necesario conocer cuales son los pasos válidos que permiten pasar de un estado a otro, por ejemplo en el ta-te-ti un jugador solo puede poner una ficha (con la que está jugando) en un casillero vacío siempre que sea su turno y el juego no halla terminado; en el problema del laberinto (visto como un grafo no dirigido) solo se puede ir de el nodo actual (el lugar donde se está parado en el laberinto) a un nodo adyacente, no es posible "saltar" nodos. Todo problema de búsqueda tiene reglas para cambiar de un estado a otro, usualmente si uno resuelve algoritmicamente un problema de búsqueda, se tendrá un motor, un problema y un estado, colocando dentro del problema las distintas funciones para cambiar de estado (las reglas del problema) y cuando se quiera obtener los sucesores de un estado particular, será necesario llamar a cada una de dichas funciones. Esta forma de diseñar la solución tiene la desventaja de crear un problema (clase) "a medida" del problema a resolver, si en algún momento se quiere agregar o quitar una regla será necesario modificar la clase que modela el problema. Por esta razón es que representar cada regla de cambio de estado de manera individual e independiente es una buena alternativa, permite intercambiar, agregar, quitar reglas, permite modificar una única regla de manera independiente al resto del algoritmo, y mas aún permite crear una representación genérica de un problema de búsqueda, donde el buscar los sucesores de un estado se reduce a ejecutar la lista de reglas que dicho problema posee. **Una regla de un problema requiere solo dos funciones, la primera es una función que permita verificar si la misma es aplicable a un estado en particular, la segunda es dado un estado particular obtener todos los estados que se derivan de aplicar dicha regla.**

4.3.3. Problema

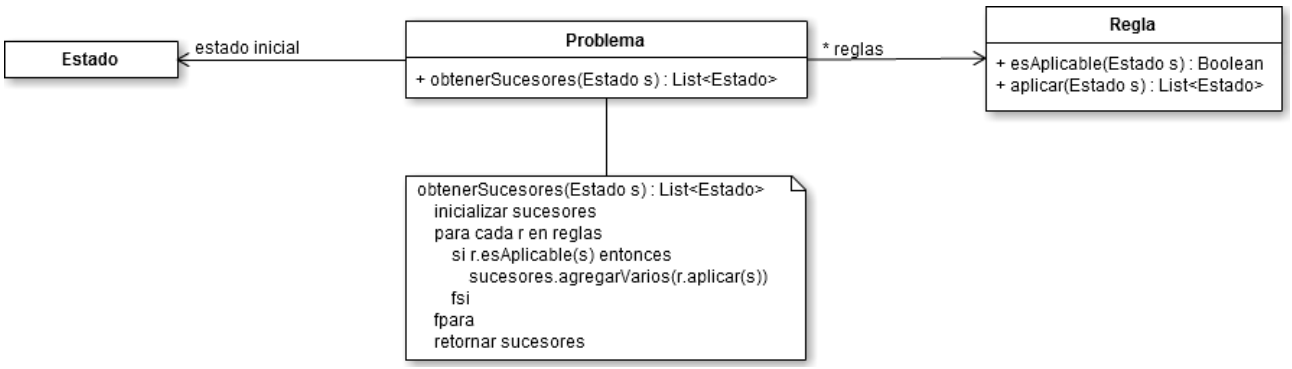

Figura 8 – estructura básica del diseño del problema

Todo problema de búsqueda está representado por un estado inicial del problema y una forma de obtener todos los nuevos estados que se derivan de uno en particular mediante la aplicación de un paso (un cambio de estado embebido en las reglas o descripción del problema). Teniendo en cuenta el diseño del estado y de las reglas, nuestra plataforma ve al problema como una forma de generalizar cualquier problema de búsqueda, definiendo el estado inicial y las reglas del mismo. La

forma en la cual se obtienen los sucesores es simple y no hay obligación de realizar ningún algoritmo extra.

```
ObtenerSucesores(Estado s) : List<Estado>
    inicializar sucesores
    para cada r en reglas
        si r.esAplicable(s) entonces
            sucesores.agregarVarios(r.aplicar(s))
        fsi
    fpara
    retornar sucesores
```

4.3.4. Motores

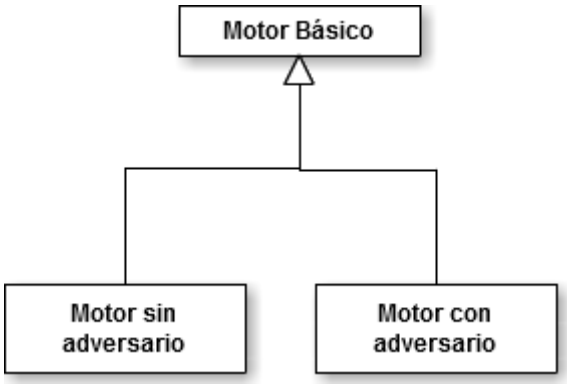

Figura 9 – estructura básica del diseño de los motores

En todo problema de búsqueda es necesario un algoritmo para encontrar la solución al mismo, de este proceso se encarga el motor de búsqueda , el cual generalmente encuentra la misma mediante la generación y recorrido de un árbol. Para problemas de búsqueda donde se desea una solución exacta el motor de búsqueda debe retornar una lista de estados desde el inicial hasta la solución y en general se empieza la búsqueda a partir del estado inicial del problema, el cual no cambia muy a menudo. En el caso de motores de búsqueda en donde el resultado es aproximado o donde el problema se trata de uno con adversarios, el resultado de la búsqueda solo es un estado correspondiente a la mejor opción encontrada o al siguiente movimiento a realizar en el caso de un juego. En este tipo de motores el estado a partir del cual se empieza a realizar la búsqueda tiende a cambiar con mucha rapidez, por lo cual el método de búsqueda requiere de parámetro el estado a partir del cual buscar en lugar de utilizar un estado que permanece prácticamente fijo. De forma general ambos casos necesitan poder definir el problema que resuelven y obtener la solución del problema, los motores que pertenecen al primer caso requieren obtener el camino desde el estado inicial al estado solución y una función que retorne si el problema tuvo o no solución (desde el estado inicial) sin requerir ningún parámetro (ya que la búsqueda la realiza desde el estado inicial

del problema). Los motores que pertenecen al segundo caso requieren una función que permita evaluar un estado (cuando cerca se encuentra de la solución), a su vez el método de búsqueda requiere un estado y devuelve otro representando el siguiente movimiento, finalmente como este tipo de búsqueda utiliza un árbol acotado en profundidad en necesario una función para definir la misma.

4.3.5. Utilización de genericidad

La genericidad permite realizar clases parametrizables por una o mas clases, esto significa que la clase va a ser instanciada para utilizar dentro de la misma a las clases pasadas como parámetro, esto permite realizar cosas como por ejemplo una clase que define una lista y luego parametrizarla con la clase de los elementos que va a contener esta lista. En nuestro caso la genericidad es de mucha utilidad y se explicara la misma para las distintas secciones de la plataforma.

4.3.5.1. Genericidad en Reglas

El algoritmo para obtener sucesores debe poder ver los métodos de "esAplicable" y "aplicar" de cada regla, pero a su vez estos métodos requieren ver métodos específicos de un estado particular (la forma de obtener sucesores para el problema de jarras requiere métodos para modificar a las mismas, mientras que para un ta-te-ti tendrán métodos para colocar una ficha).
Esto lleva a lo siguiente:
una regla se aplica sobre un estado particular, el cual es pasado como parámetro al instanciar la misma, ej : ReglaLlenarJarras extends Regla<Jarras>

Una regla como ya fue explicado con anterioridad debe poder dado un estado obtener todos sus sucesores mediante la aplicación de dicha regla y dado un estado debe poder evaluar si se le puede aplicar esta regla la mismo. Por lo tanto en este caso una regla debería funcionar con cualquier tipo de estado y la genericidad permite especificar esto al decir que una regla es parametrizada por cualquier clase que derive del estado mas básico de todos.
Cabe aclarar que gracias a esto la utilización de reglas es totalmente trasparente del tipo de estado sobre el cual se esta trabajando.

4.3.5.2. Genericidad en el problema

Un problema va a requerir aplicar las reglas a un estado para obtener los sucesores, pero las reglas actúan sobre un estado particular (no es posible mezclar reglas para el problema de las jarras con reglas para el reversi), por lo tanto el problema tiene como parametro el estado sobre el cual trabaja, este a su vez va a ser el parámetro de clase para el conjunto de reglas que usa.

```
Clase Problema<A> {
    Reglas<A> reglas;
    …
}
```

Esto muestra ya la obligación de parametrizar el problema, pero otra explicación es que un problema de búsqueda siempre se define en base a un estado particular, dando esto una cierta obligación conceptual de parametrizar la clase.

Un problema requiere de un estado inicial y de un conjunto de reglas, ambos deben ser compatibles, es decir, que el estado sobre el cual operan las reglas debe ser el mismo al tipo del estado inicial. Esto tiene sentido ya que un problema de búsqueda esta definido sobre un estado en particular. Además la función de obtener un estado que provee el problema de búsqueda debe devolver una lista de estados del mismo tipo que los dos anteriores, todo esto se puede especificar haciendo al problema parametrizable con cualquier clase que extienda al estado mas básico de todos.

4.3.5.3. Genericidad en el motor de búsqueda

Como explicamos anteriormente hay tres clases correspondientes a motores de búsqueda, la primera la mas general de todas debe poder operar sobre cualquier tipo de estado al igual que los motores de búsqueda sin adversario, en el caso de los motores de búsqueda con adversarios es necesario restringir sobre el cual el motor opera a aquellos que corresponde a estados con adversarios. Dado que los motores con adversarios, los motores sin adversarios y cualquier motor extra que se quiera construir son extensiones de un motor mas general, la genericidad permite realizar esta extensión restringiendo el tipo de los estados sobre los cuales puede operar un motor determinado pertenecen a cualquier clase que derive del estado mínimo que requiere dicho motor, por ejemplo un motor con adversario va a utilizar un estado con adversario como clase mínima.

4.4. Diseño [CORREGIR DISEÑO]

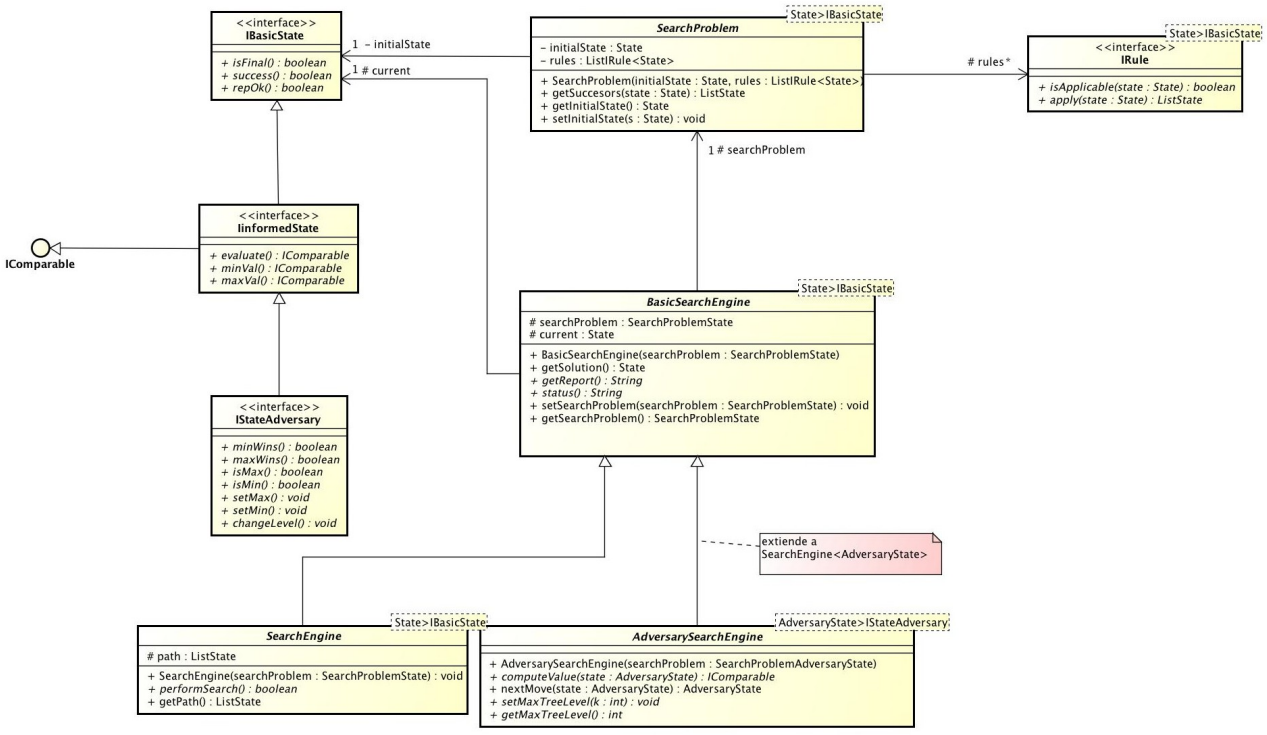

Figura 10 – diagrama de diseño de clases

La especificación del problema de búsqueda se hará con tres componentes, **estado**, **reglas** y **problema de búsqueda**. El estado contiene los métodos para modificar y leer atributos del mismo, permite saber si el mismo es exitoso, así también como si este es un estado final (no hay ninguna acción legal que permita pasar de este estado a otro). Las reglas definen funciones para pasar de un estado al otro y permiten evaluar si la misma es aplicable o no a un cierto estado. Finalmente el problema de búsqueda permite obtener todos los estados sucesores a partir de uno dado (esto lo hace aplicando todas reglas a un estado).
El motor de búsqueda se construye a partir de un motor básico que ofrece todo lo necesario para problemas de búsqueda en general como generación de reporte, consulta de estado del motor, asignación del problema de búsqueda. Y luego dos motores especializan a este, un motor de búsqueda sin adversarios y un motor de búsqueda con adversarios. Tanto el motor básico como el de búsqueda sin adversarios utilizan cualquier estado que derive del estado más básico, el motor de búsqueda con adversarios utiliza cualquier estado que derive del estado con adversarios (los distintos estados se explicaran más adelante). Todo motor utiliza al problema de búsqueda (cada uno con el estado que requiera).
Los estados se van a definir partiendo de un estado básico que ofrece lo mínimo requerido para problemas de búsqueda. Extendiendo a este se encuentra un estado para búsquedas informadas, agregando lo necesario para las mismas y finalmente el estado con adversarios extiende a este último, nuevamente agregando todo lo necesario para búsquedas con adversarios.

4.4.1. Decisiones de diseño[5]

En esta sección se detallaran y explicaran aquellas decisiones de diseño mas importantes o que pueden tener otras soluciones igualmente correctas.
La decisión de que la obtención de sucesores esté del lado del problema de búsqueda se basa en que se puede tener problemas con distintas reglas pero mismo estado.
Se decidió crear un puente para ser utilizado entre cualquier aplicación que necesite utilizar un motor de búsqueda con adversarios, y el motor de búsqueda. En principio para facilitar la interacción entre aplicación-plataforma. Si bien sería mejor solución dividir dicho puente para diferenciar juegos con y sin adversario, así también como tener en cuenta una cantidad de jugadores mayor a dos para los juegos con adversario. Se decidió dejar ese puente sin modificar por cuestiones de tiempo, complejidad y el hecho de que dicho puente no está incluido como un requerimiento, sin embargo es un cambio que debería hacerse en próximas versiones. Este puente se pensó para brindar lo necesario para interactuar con el motor, teniendo en cuenta los requerimientos en común que toda aplicación pueda tener, de manera de maximizar a estos sin especializar demasiado a este puente.
La decisión de tener un problema de búsqueda totalmente implementado es para permitir un desarrollo ágil de problemas y motores, por lo que se implementó el obtener sucesores como un ciclo que por cada regla del problema que sea aplicable agrega los estados generados por dicha a los sucesores.

5Cabe destacar que todas las extensiones e implementaciones de clases (abstractas e interfaces) que usen genericidad se harán definiendo como parámetro de clase a toda clase que extienda a la mínima clase necesaria. Este parámetro será utilizado en todo atributo de la clase que requiera parámetro de clase.

4.4.2. Diseño (clases y responsabilidades)

4.4.2.1. Estado

Se comienza con un estado básico (IBasicState) el cual debe tener las siguientes responsabilidades:

- Contener toda estructura necesaria para un estado de un problema de búsqueda
- Ofrecer los métodos necesarios para interactuar con dichas estructuras
- Ofrecer los métodos necesarios que requiera cualquier problema de búsqueda (es éxito, es final, representación, verificar invariante, comparar a otro)

Un estado para búsquedas informadas (IinformedState) extiende al anterior, y tiene las siguientes responsabilidades:

- Ofrecer un método para evaluar al estado
- Ofrecer los valores mínimos y máximos en el rango de evaluación

Nota: en principio se eligió utilizar un valor entero para la evaluación de estados, este tipo si bien muchas veces es suficiente, se pueden encontrar casos donde no es suficiente, como por ejemplo si en la evaluación se realiza un división el tipo entero ya no es suficiente (salvo que se trunque o redondee el valor obtenido), por otro lado si la evaluación requiere un tipo mas complejo ya no alcanza ni con entero ni con real. Con esto en mente la primer solución sería utilizar la interfaz “Comparable”, sin embargo dado que esta clase es parametrizada, utilizarla significa modificaciones muy complicadas al código o utilizarla sin parámetros lo cual deja el código lleno de advertencia. La solución fue crear una interfaz “IComparable” no parametrizada que ofrece el método “compareTo(IComparable other)” igual al que ofrece “Comparable”.

Un estado para búsquedas con adversarios (IStateAdversary) extiende al anterior, y tiene las siguientes responsabilidades:

- Ofrecer métodos para saber que jugador ganó (ganó max, ganó min)
- Ofrecer métodos para saber a que jugador pertenece un estado (es max, es min)
- Ofrecer métodos para definir que jugador corresponde a un estado (setear max, setear min)

4.4.2.2. Reglas

Una regla (IRule) define una función de cambio de estado, ofrece dos métodos, uno para saber si dado un estado la regla actual es aplicable al mismo, otra para dado un estado obtener todos aquellos estados que se obtengan de aplicar la regla al mismo. Se debe cumplir que si la regla no es aplicable, entonces la función de cambio de estado devuelve una lista sin ningún estado.

4.4.2.3. Problema de búsqueda

El problema de búsqueda (SearchProblem) tiene las siguientes responsabilidades:

- Definir cual es el estado inicial del problema
- Obtener a partir de un estado todos los sucesores del mismo, esto es aplicar todas las reglas del problema al estado.

4.4.2.4. Motor de búsqueda

Se comienza con un motor básico (BasicSearchEngine) que ofrece lo mínimo necesario para un motor de búsqueda y tiene las siguientes responsabilidades:

- Contener el problema de búsqueda sobre el que realiza las búsquedas
- Obtener la última solución al problema
- Obtener el estado actual de la búsqueda en forma de texto
- Obtener un reporte de la última búsqueda realizada

Se sigue por un lado con un motor para problemas sin adversarios (SearchEngine), el cual tiene las siguientes responsabilidades:

- Obtener el camino del estado inicial al estado solución (si existe)
- Realizar la búsqueda y retornar si se encontró o no una solución

Por otro lado se tiene el motor para problemas con adversarios (AdversarySearchEngine), el cual tiene las siguientes responsabilidades:

- Computar el valor de un estado en el árbol de búsqueda
- Encontrar el mejor movimiento a realizar a un estado dado
- Obtener y definir los limites del árbol de búsqueda

4.5. Plataforma y herramientas utilizadas

OS: Mac OS X (64)
Versión 10.6.8

OS: Ubuntu
Versión 10.04. LTS

OS: Windows Ultimate 7 (64)
Versión 7 SP1

Sistema de control de versiones: SVN (Mac OS, Ubuntu), Tortoise (Windows).
Versión: 1.6.16, 1.6.6 y 1.6.??? respectivamente
Utilizado para control de versiones de los archivos usados/generados en el desarrollo del proyecto.

Editor de texto: LibreOffice (Mac OS, windows), OpenOffice (Ubuntu)
Versión: 3.3.3 y 3.2 respectivamente
Utilizado para generar informe del proyecto, manual de usuario y documentaciones varias.

IDE de programación: Eclipse, NetBeans
Versión: 3.7.0 (Indigo) y 7.0.1
Utilizado para el desarrollo de la plataforma y ejemplos de uso de la misma.

En principio Eclipse fue el IDE utilizado por las funciones que incorpora y que ayudan de gran manera al desarrollo de código, estas son: corrección sintáctica y semántica durante la edición de código así como la función de autocompletar cuando se esta escribiendo la llamada a un método, también tiene incorporado la edición y visualización de Javadoc, así como permitir la compilación del mismo; posteriormente cambiamos el IDE a NetBeans ya que el mismo tiene incorporado la utilización de sistema de control de versiones e incorpora herramientas de generación y desarrollo de interfaces gráficas, las cuales fueron de muy utilidad en la etapa de desarrollo de ejemplos y al mismo tiempo NetBeans posee herramientas de "profiling" las cuales resultaron ser esenciales durante el desarrollo de los ejemplos y motores de búsqueda utilizados en los mismos. Contando además con este IDE sigue contando con las funcionalidades de Eclipse.

Editor de diagramas UML: Astah (Community)
Versión: 6.4 (Model Version 34)
Utilizado para la generación de diagramas de clases.

Servicio de almacenamiento de archivos: DropBox
Versión: 1.1.35
Utilizado para almacenar y compartir archivos del proyecto entre los integrantes y el Director del Proyecto. Este servicio se reemplazo por la utilización de Sistema de Control de Versiones y solamente se lo siguió usando para compartir ciertos documentos con el Director.

4.6. Implementación

En esta sección se detallaran ciertos aspectos importantes de implementación, así también como las decisiones de diseño tomadas.

4.6.1. Estados

Se decidió utilizar una interfaz propia llamada "IComparable" en lugar de la interfaz "Comparable" perteneciente a la librería genérica de Java para referirse a evaluaciones de estados, esto se hizo porque la interfaz "Comparable" es parametrizada y esto causaba problemas ya que el estado "IinformedState" debería implementar "Comparable<IinformedState>" esto trae ambigüedad en el algoritmo de búsqueda cuando se comparan las evaluaciones de dos estados ya que es posible tener muchos estados que implementen "IinformedState"[PREGUNTAR NAZA!]. Solucionar el problema anterior no es imposible pero es una complicación que a según nuestro punto de vista es totalmente innecesario. La desventaja de utilizar "IComparable" es la necesidad de crear una clase para cada nuevo valor utilizado para realizar la comparación, por ejemplo un "ComparableInt",

“ComparableFloat” para enteros y reales respectivamente, aunque la cantidad de nuevos valores de comparación necesarios van a ser muy pocos.

4.6.2. Reglas

En un principio se había pensado en implementar una regla especial que permitiera saber si un estado era o no exitoso, esta idea finalmente se descarto ya que esto “rompería”con la semántica de las reglas ya que seria necesario darle significado distinto a las funciones de las mismas dependiendo si es una regla común o una regla para determinar el éxito de un estado. Por otro lado tiene mas sentido que la verificación del éxito de un estado este del lado del estado o del problema, pero no del lado de las reglas.

4.6.3. Problema

Aprovechando el diseño, (de estado y reglas por separado, y el uso de genericidad), el método de obtención de sucesores se realizó de manera genérica y de forma muy simple al recorrer la lista de reglas del problema y para cada una si la misma era aplicable al estado que se paso como parámetro al método de obtención se sucesores entonces se añadía a la lista de sucesores todos los nuevos estados de aplicar dicha regla.

4.6.4. Genericidad

La genericidad es mas útil cuando se aplican restricciones a los parámetros de una clase, esto es muy importante en la implementación de motores de búsqueda.
Un motor restringe el parámetro de clase para que no solo sea un estado sino para que sea cualquier clase que derive del estado mínimo que necesita, esto significa que un motor de búsqueda informado va a restringir el parámetro de clase a cualquier clase que extienda a “IinformedState”, un motor de búsqueda con adversario va a hacer lo mismo pero usando “IStateAdversary”. Cuando un motor extiende a otro es necesario que el parámetro de clase del motor a extender sea una clase que extienda del mínimo estado que el nuevo motor necesita, para este caso podemos ver el ejemplo de un motor informado, la plataforma no tiene ninguna clase para estos pero un motor informado es un motor de búsqueda sin adversario y este es al que debe extender, la forma de realizar la extensión debe ser “MotorInformado<EstadoInformado extends IinformedState> extends SearchEngine<EstadoInformado>”.[REVISAR]

4.7. Testing

Para testear este proyecto se utilizaron técnicas Ad hoc, utilizando a los ejemplos de la plataforma como base para realizar a los mismo. En un principio se utilizo una versión por consola del ejemplo del Reversi lo cual permitía ver una gran cantidad información extra mayormente utilizada para verificar el algoritmo de búsqueda hecho por el motor. Utilizando esta versión la mayoría de los errores encontrados estuvieron del lado del ejemplo mas específicamente del estado del Reversi, del lado del motor utilizado (MinMax con poda alfa-beta) no se encontró ningún error excepto por una ligera falla en el diseño del algoritmo de búsqueda donde el algoritmo de búsqueda se aplicaba a los sucesores del estado inicial en lugar de a este mismo (la búsqueda se realizaba a partir del primer nivel del árbol y no de la raíz). Las desventajas de utilizar la versión por consola del ejemplo era la

dificultad de visualizar el estado del tablero (juego Reversi) con lo cual la siguiente fase de prueba se realizo sobre versiones con interfaz gráfica de los ejemplos (Reversi y Flood-It!), la mayoría de los errores que se encontraban generalmente dentro de la implementación de los estados de los ejemplos, errores en la lógica de los juegos así como errores relacionados a la interfaz gráfica de los ejemplos, si embargo se logro encontrar un error relacionado a una mala elección de motor de búsqueda para el ejemplo de Flood-It!, este caso fue elegir un motor Best-First Search para un motor donde el árbol de búsqueda era muy grande y la búsqueda debía realizarse hasta encontrar un estado exitoso, este error fue solucionado cambiando el motor de búsqueda Depth-First Search guiado por la evaluación de estado y se cambio la generación del tablero del Flood-It! Para que retornara tableros mas fáciles, conjuntamente con este cambio se aplicaron restricciones a la regla de cambio de estado del mismo juego para descartar estados inútiles (cambiar de color sin aumentar el tamaño del área conquistada).

Etapas	Modulo testeado	Números de Errores
Ejemplos (versión consola)	Motor	1(error de diseño de algoritmo de búsqueda)
Ejemplos (versión consola)	Estado	5 (función es final -Reversi-, linkeo de tokens -Reversi-, funciones de recorrido y búsqueda de movidas validas -Reversi-, generación de listas de movidas disponibles, función de valoración demasiado simple -Reversi-)
Ejemplos (versión interfaz gráfica)	Motor	1 (mala elección de algoritmo de búsqueda -Flood-It!-)
Ejemplos (versión interfaz gráfica)	Reglas	1 (generación de estados sucesores inútiles, en reglas de movimiento -Flood-It!-)
Ejemplos (versión interfaz gráfica)	Estados	2 (algoritmo demasiado azaroso para la generación del tablero del Flood-It!, verificación de fichas capturadas -Flood-It!-)
Ejemplos (versión interfaz gráfica)	Todos	Varios errores irrelevantes (errores de lógica, de interfaz gráfica, etc.)

5. Casos de estudio

Para poder probar el uso de la plataforma vamos a generar dos juegos con los cuales mostraremos que partes de la misma se deben implementar y cómo, y dadas otras partes ya implementadas previamente, se mostrara como utilizar a las mismas en un nuevo proyecto. Dado que la plataforma tiene dos motores básicos, uno pensado para búsqueda con adversario y el otro para sin adversario. Los juegos que se desarrollaran para esta plataforma serán uno con adversario y otro sin, además para mostrar la utilización del framework de la sección de motores se construirán cuatro motores de búsqueda (mas si el tiempo lo permite), para poder dar al usuario final una muestra de el uso de la plataforma como base de la plataforma para nuevos motores y el uso de la plataforma para utilizar motores ya implementados.
En general la construcción de nuevas aplicaciones que utilicen esta plataforma se divide siempre en las mismas etapas, extender e implementar uno de los tres estados provistos por la plataforma según correspondan, después el desarrollo o reutilización de un motor de búsqueda que sea aplicable al problema que se esta solucionando y por cada regla de cambio de estado que tenga el problema se debe construir una nueva regla “IRules”. Una buena técnica al momento de desarrollar las reglas es filtrar la generación de estados sucesores inútiles, tal como no realizar ninguna cambio de estado (regla vacía, skip), generación de estados sucesores que se sabe que no puede llevar a nada nuevo, etc. Es importante distinguir cuando la regla skip es una movida inútil y cuando es una movida legal del problema (cuando el problema permite “saltear turnos”).

5.1. Reversi

Es un juego con adversarios, se juega con un tablero de ocho filas por ocho columnas, con fichas blancas y negras (u otros colores). A un jugador se le asigna un color y se dice que lleva las fichas de ese color, lo mismo para el adversario con el otro color, El inicio del juego como se ve en el diagrama se colocan dos fichas blancas y dos negras.

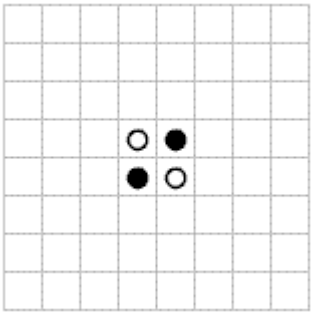

Figura 11 – Tablero inicial del reversi

Empezando por quien lleva las fichas negras los jugadores deben hacer un movimiento por turno, a menos que no se pueda hacer ninguno, pasando en ese caso el turno del jugador contrario. El movimiento consiste en colocar una ficha de forma que flanquee una o varias fichas del color contrario y voltear esas fichas para que pasen a mostrar el propio color. Para que las fichas estén flanqueadas, deben formar una línea continua recta (diagonal u ortogonal) de fichas del mismo color entre dos fichas del color contrario (una de ellas la recién colocada y la otra ya presente). En

el siguiente ejemplo juegan las blancas donde indica la flecha y se puede ver qué fichas se voltean.

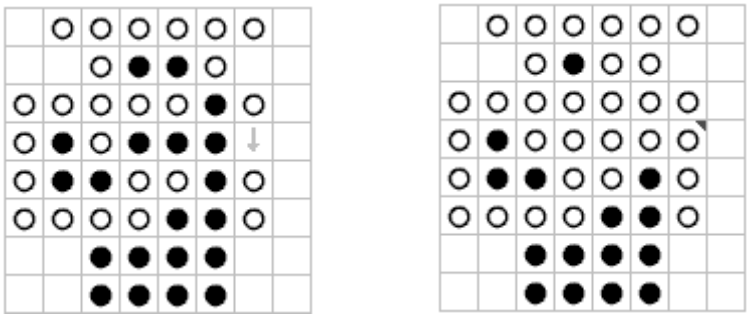

Figura 12 – tableros luego de realizar una movida (ficha blanca)

La partida finaliza cuando ningún jugador puede mover (normalmente cuando el tablero está lleno de fichas) y gana quien en ese momento tenga sobre el tablero más fichas mostrando su color.

Se utilizo una sola regla, ya que en el momento que le toque jugar a un jugador este lo que puede hacer es agregar una ficha (un movimiento) o en el caso que no se pueda realizar movimiento pasar y en el motor utilizamos la técnica de MinMax poda alfa beta [enlace a poda alfa-beta], ya que la técnica MinMax es para búsqueda con adversarios, pero al ser un árbol tan grande, lo que realizamos es usar con poda alfa beta, de esta forma nos garantizamos ¿de ocupar menos espacio?

5.1.1. Detalles de implementación

A continuación se discutirán algunos detalles de implementación importantes y decisiones de diseño al momento de implementar este caso de estudio.

5.1.2. Estado

Para describir el estado se utilizo una matriz cuadrada pero en lugar de utilizar enteros para definir los estados de las casillas se utilizó una clase ReversiToken la cual sirve para definir el estado de una celda (blanca o negra) y contiene una lista con todos las fichas vecinas que posee. La idea de utilizar una ficha conectada con sus vecinos era para hacer mas claro los algoritmos que necesitan validar o realizar una jugada, que en general se hacen por completo utilizando índices. A su vez utilizar fichas vecinas hace que sea mas difícil que el algoritmo se “pase” de la zona del tablero.

Además de utilizar fichas para los contenidos de las celdas se decidió que en lugar de validar las jugadas que se realizaban (poner ficha de color C en posición X,Y) se iba a generar listas de movidas disponibles para cada jugador, luego el control de la validez de las movidas pasa a ser responsabilidad del algoritmo que utilice al estado y da mas flexibilidad sobre como hacer esta verificación. Sería posible hacer que esta generación de movidas disponibles fuera mas inteligente ya que actualmente en cada cambio de estado, el algoritmo vacía las listas y las genera desde cero nuevamente.

5.2.3. Reglas

El reversi tiene una única regla que es “colocar ficha”, esta se hizo mediante un ciclo que por cada movimiento dentro de la lista de movimientos válidos para el jugador actual realice la movida y agregue el nuevo estado generado a la lista de sucesores. No es posible realizar ningún filtrado extra en esta regla ya que no hay movimientos que lleven a un estado igual o parecido.

5.2.4. Motor utilizado

El motor elegido para este caso de estudio fue el MinMax con poda Alfa-Beta, es importante aclarar que este motor no se hizo específicamente para el Reversi (esto iría en contra con la idea tras la plataforma), así que es posible reutilizarlo para cualquier problema de búsqueda siempre que este sea uno de adversarios. Una corrección importante que se le tuvo que realizar al motor fue que al ejecutar el algoritmo MinMax, este empezaba desde los sucesores del estado inicial, y fue necesario cambiarlo para que lo hiciera desde la raíz, este error fue más un error del diseño del algoritmo que de implementación, y el motor funcionaba correctamente incluso antes de realizar esta corrección, el cambio se pudo ver en la cantidad de nodos que visitaban ambas versiones del algoritmo, siendo menor en la versión corregida.

5.2. Flood-It

Se juega con un tablero de catorce filas por catorce columnas, primero hay que situarse en la cuadricula superior izquierda y se empieza seleccionado uno de los colores del menú. El color de la zona cambiara y se extenderá a los espacios adyacentes que tenga el mismo color seleccionado, por lo que podrás cubrir zonas del tablero. El objetivo del juego es cubrir todo el tablero de un mismo color en a lo sumo 25 pasos. Al igual que el caso anterior la única regla de movimiento es elegir un color, por las restricciones de pasos es posible realizar un filtrado de sucesores en la regla de cambio de estado en donde todos los sucesores donde se cambia por el mismo color o la superficie de la zona capturada no cambia pueden ser filtrados.

Para este ejemplo se utilizo un motor de búsqueda con el algoritmo Depth-First search guiado por la evaluación de los estados, esto significa que la lista de sucesores que recorre el algoritmo en cada nivel, se ordena de mayor a menor.

Figura 13 – tablero inicial del flood-It

Figura 14 – tablero luego de 10 pasos

5.2.1. Detalles de implementación

A continuación se discutirán algunos detalles de implementación importantes y decisiones de diseño al momento de implementar este caso de estudio.

5.2.2. Estado

El estado es muy parecido al estado del Reversi, una matriz con valores, nuevamente se decidió utilizar una estructura del tipo "Ficha" mediante la clase Flood-ItToken. En este caso esta decisión tuvo un impacto mucha mas importante en el algoritmo, ya que por el problema mismo (agrandar una zona controlada de celdas del mismo color hasta cubrir todo el tablero) requería que se hiciera una búsqueda similar a un Depth-First Search, - y hay que destacar el similar -, ya que la idea es partir desde la celda en la esquina inferior izquierda e ir buscando todos los adyacentes que tengan el mismo color y a su vez repetir la búsqueda para cada uno de estos, esto es necesario en dos partes del problema, el primero es cuando se necesita verificar el tamaño de la zona controlada, y el segundo caso es cuando se cambia de color.

La verificación de cambio de estado es inexistente, debería utilizarse un color valido cuando se realiza el cambio pero fuera de eso no hay mas que verificar ya que el cambio de color se realiza siempre partiendo desde la celda en la esquina superior izquierda.

Un cambio realizado que es importante destacar es sobre la generación del tablero, inicialmente esta era completamente aleatoria (para cada celda buscaba un color entre los 6 disponibles y lo seteaba). Esta generación resultaba en tener tableros en los cuales era realmente difícil poder ganar y a su vez causaba que la inteligencia artificial (motor de búsqueda) terminara diciendo en casi todos los casos que el tablero no se podía resolver en hasta 25 pasos al mismo tiempo que necesitaba muchísimo tiempo para llegar a esta conclusión (es necesario realizar una búsqueda exhaustiva del árbol para concluir que no hay solución posible). Por lo tanto se decidió cambiar el algoritmo de generación del tablero para que al generar el color de una nueva celda tuviera en cuenta las que había generado antes, el algoritmo era simple:

Si la celda a generar era la de la esquina superior izquierda se generaba con el algoritmo anterior (generación completamente aleatoria)

En caso contrario se utilizaban las celdas de atrás, arriba y diagonal atrás-arriba según estuvieran disponibles para agregarlas a una lista de colores (la cual admite repetidos), finalmente se completaba la lista con colores aleatorios hasta que la longitud de la misma fuera de cuatro.

Finalmente se elegía un índice de forma aleatoria para elegir un color de esta lista.

Este algoritmo hacía que los tableros generados fueran mucho mas simples, contenía mas zonas del mismo color y de mayor tamaño.

5.2.3. Reglas

Al igual que en el caso anterior (Reversi) hay una única regla, “cambiar color”, y es un ciclo sobre valores posibles generan un nuevo estado para cada valor. Sin embargo en este caso hay varios movimientos a descartar, el primero es que cambiar por el mismo color no tiene sentido así que solo con este filtro se reduce el tamaño de sucesores de un estado de 6 a 5 (parece nada para un caso, pero cuando se lo mira desde el punto de vista de generación de un árbol de búsqueda, uno menos es mucha diferencia), el segundo filtro es sobre cambios de color que no agrandan la zona controlada, por ejemplo, si se la zona controlada (de color verde) no tiene ningún adyacente de color amarillo, entonces cambiar de verde a amarillo no tiene sentido, gasta un paso para no lograr nada y las probabilidades que se encuentre una solución siguiendo ese tipo de cambios es cada vez menor (solo se tienen 25 pasos como máximo para completar el tablero), con este filtrado el tamaño del árbol generado se achica considerablemente.

5.2.4. Motor utilizado

Encontrar el motor de búsqueda apropiado para este caso de estudio resulto ser una tarea complicada, sin embargo gracias a esto y a un error de elección cometida al principio se terminó con 3 motores más para la plataforma.

Inicialmente se cometió el error de utilizar un Best-First Search para este caso de estudio, fue simplemente un desastre, la cantidad de nodos que tiene el árbol de búsqueda para el Flood-It es muy grande, y querer meter todos estos en una cola para que el algoritmo los analice es absurdo. Retoques al motor, aumentar el filtrado en la regla de movimiento (al inicio no se filtraba nada), todos cambios que no ayudaban a que el motor funcionara. Esta equivocación, importante, nos dejo un motor extra para la plataforma y una enseñanza sobre lo importante que es pensar bien antes de hacer una elección.

Luego de descartar el uso del Best First Search como motor de búsqueda para el Flood-It se penso en utiliza un Depth First Search, el porqué de la elección fue que el árbol de búsqueda tiene altura 25 como máximo y utilizar busqueda a lo largo iba a eliminar cualquier problema de uso excesivo de memoria. El nuevo motor resultó ser casi inutil, si bien no se encontraban errores de falta de memoria, el motor tardaba demasiado y recién podía dar pistas cuando el tablero estaba a mas de la mitad de ser resuelto. Incluso luego del cambio en la generación de tableros el Depth First Search seguía siendo lento. Finalmente se tomó la decisión de crear un nuevo motor realizando una copia del Depth First Search pero ordenando la lista de sucesores de mayor a menor según una función de

valoración, dado que en principio se había elegido el Best First Search que es un motor de búsqueda informada, el Flood-It ya contaba con dicha función. Este ultimo motor, denominado Guided Depth First Search, resulto tener tiempos de respuesta muy rápidos, por lo que terminó siendo el elegido para la inteligencia artificial del Flood-It.

6. Manual de usuario

A continuación se detallan dos manuales de usuario, el primero es un manual de como utilizar la plataforma, que clases extender, que métodos implementar y como extender partes de la plataforma si se quiere tener una plataforma mas grande para uso futuro. El segundo manual es sobre como probar los casos de prueba implementados.

6.1. Utilización de la plataforma

La plataforma Garoé se puede dividir en cuatro módulos, estado, reglas de cambio de estado, motor de búsqueda y problema. Para un nuevo proyecto que utilice esta plataforma no siempre va a ser necesario realizar implementación para todos los módulos, a forma de ejemplo, si uno quisiera realizar el problema de las jarrar de agua, solo debería implementar un estado y varias reglas, una clase principal para ejecutar el programa y nada mas; si uno quisiera implementar un nuevo motor, solo sería necesario implementar el motor sin prestar atención al resto de los módulos. Por lo tanto, se dará un manual para cada modulo de la plataforma.

6.1.1. Estados

Cuando se requiera implementar un nuevo estado se debe tener en cuenta cual de los tres estados en es el que define de manera mas clara al estado de nuestro problema, la forma de realizar esta elección se puede hacer con tres preguntas:

1) Mi estado corresponde a un problema de búsqueda con adversarios?
2) Mi problema requiere que evalué cuando cerca está mi estado de ser solución?
3) Mi problema requiere algo que se puede generalizar?

Si (1) es verdadero entonces el nuevo estado debe implementar la clase IStateAdversary

Si (1) es falso pero (2) es verdadero entonces el nuevo estado implementara a IinformedState

Si todas las preguntas son falsas entonces el nuevo estado implementara a IBasicState

Si (1) y (2) son falsas pero (3) es verdadera, entonces el camino a seguir es extender la plataforma con un nuevo estado y definir el o los métodos necesarios para luego implementar o extender al mismo para el caso particular que se necesita

Para un nuevo estado el perfil de la clase debe ser:

```
MiEstado extends [StateAdversary|IinformedState|IBasicState]
```

6.1.2. Reglas

Para implementar las reglas de nuestro problema será necesario implementar una regla a la vez, lo importante es saber cuando una regla esta correctamente definida y cuando involucra a mas de una regla del problema. Para poder implementar una nueva regla se debe implemetar la clase IRule pasando como parámetro de la clase el estado particular que vamos a utilizar, ejemplo

```
TransferirAgua implements IRule<Jarras>.
```

Es importante no agrupar reglas, ya que si bien el obtener sucesores del problema es generico y trata a todas las reglas por igual, tener reglas agrupadas dificulta el filtrado de estados inutiles y la implementación de algoritmos de obtención de sucesores mas eficientes, por ejemplo una obtención perezosa de los mismos.

Para comprobar que una regla no es una agrupación de otras, se puede utilizar el siguiente método:

1) Mi regla se puede explicar usando una acción que no involucre implicitamente a otras?
2) Mi regla se puede explicar sin utilizar “hace esto Y esto”?

Si (2) es verdadera, entonces hay reglas agrupadas, ya que para explicar la regla es necesario nombrar varias acciones.

Si (1) es verdadero, entonces se esta utilizando una generalización de varias reglas del problema, por ejemplo en el problema de las jarras decir “regla que cambia el estado de las jarras”, esto involucra implicitamente a todas las reglas de cambio de estado del problema.

Si ambas preguntas son falsas entonces la regla está bien definida.

IMPORTANTE: Es importante tener en cuenta que pueden darse casos donde una regla por mas simple que sea pueda reducirse en otras mas pequeñas, por ejemplo la regla “colocarFicha” en el reversi podría dividirse en una regla por cada celda (es decir, tener reglas como “colocarFicha23” que se refiera a colocar una ficha en la posición 2,3). La habilidad de poder elegir el nivel de granularidad de las reglas se tiene que ver con mucho cuidado, puede ayudar a aumentar la eficiencia o ayudar al filtrado de estados inutiles pero a su vez puede llevar a tener un gran número de reglas que no ayuden en nada.

6.1.3. Problema

La clase que define un problema, “SearchProblem”, fue hecha de tal forma que puede utilizarse para cualquier tipo de problema sin necesidad de implementarla o extenderla. Cuenta con una generación de sucesores generica que utiliza un estado como parámetro y una lista de reglas.

Para utilizar esta clase solo es necesario crear un nuevo objeto de la forma

```
SearchProblem<MiEstado> problema = new SearchProblem<MiEstado>(estadoInicial, reglas)
```

donde MiEstado es la clase del estado que se va a utilizar en el problema, estadoInicial y reglas son el estado inicial del problema y las reglas de cambio de estado respectivamente.

Para extenderla solo es necesario crear una nueva clase de la forma

```
MiProblema<Estado extends EstadoMinimo> extends SearchProblem<Estado>
```

donde Estado especifica cualquier estado que o bien sea de la clase EstadoMinimo o bien extienda al mismo. EstadoMinimo se refiere a lo minimo que necesito de un estado para el problema que estoy extendiendo, si MiProblema está hecho para estados con función de evaluación entonces el estado mínimo necesario será IStateAdversary.

6.1.4. Motores

En principio cualquier nuevo motor que se quiera implementar se puede extender de alguno de los tres que se proveen. Si se desea crear un nuevo motor, ya se una nuevo motor abstracto para la plataforma o un nuevo motor implementado para utilizar el problema de búsqueda, el procedimiento es muy similar al de generar un nuevo problema por lo que se explicará mas brevemente en forma de pasos a seguir:

Elegir el motor abstracto que mejor represente al nuevo motor. Si el nuevo motor no corresponde ni a SearchEngine ni a AdversarySearchEngine, extender BasicSearchEngine. Crear una nueva clase con la forma

```
abstract class MiMotorAbstracto<Estado extends EstadoMinimo> extends Motor<Estado>
```

Donde MiMotorAbstracto es el nuevo motor a agregar a la plataforma, EstadoMinimo es el mínimo estado que se necesita para el nuevo motor y Motor es el motor de la plataforma que se va a extender. En el caso de implementar un nuevo motor solo es necesario remover el “abstract”

6.1.4.B. Juntando todo

Para armar todas las partes y poder utilizar el motor, reglas y estado que se implementaron hay que poder juntar todas las partes, la forma mas simple para hacer esto es la siguiente:

```
MiAplicación {
        …
        MiEstado estado = new MiEstado();
        List<IRule<MiEstado>> reglas = new LinkedList<IRule<MiEstado>>();
        reglas.add(new MiRegla1());
        …
        reglas.add(new MiReglaK())
        SearchProblem<MiEstado> problema = new SearchProblem<MiEstado>(estado, reglas);
        MotorAbstracto<MiEstado> motor = new MiMotor<MiEstado>(problema);
        ...
        ...
}
```

6.1.5. Utilización de la plataforma Garoé como librería

El formato elegido para la distribución de la plataforma Garoé es mediante un archivo JAR (de todas formas el código fuente de la plataforma está disponible en el repositorio[http://code.google.com/p/garoe-platform/]).

Para poder utilizar la plataforma como librería es necesario conocer como está estructurada la misma:

6.1.5.1. Paquete framework (framework)

Este es el paquete que contiene la plataforma en si, las clases son las que se muestran en el grafico de diseño de la plataforma[REFERENCIA], este paquete será necesario siempre que se necesite crear algo nuevo (nuevo motor, estado, regla, problema) para ser utilizado en un nuevo proyecto. Para extender a la plataforma en si, por ejemplo agregar un nuevo motor o estado abstracto, hay dos opciones:

1) Trabajar sobre el código fuente de la plataforma y agregar nuevas clases al paquete "framework" o sobre un nuevo paquete con un nombre apropiado como "extended"

2) Crear un nuevo proyecto agregando el JAR como librería y crear los paquetes que sean necesarios para extender la plataforma.

En ambos casos se debe mantener la politica de distribuir la plataforma como archivo JAR.

6.1.5.2. Paquete engines (engines)

Este paquete trae todos los motores que vienen ya implementados, estos motores pueden ser utilizados sin necesidad de implementar código extra. Al igual que el caso anterior, si se desea implementar un nuevo motor, este debe estar alojado en este paquete si se modifica el código fuente de la plataforma o en un paquete con nombre apropiado como "misMotores" en caso de trabajar con el archivo JAR.

6.1.5.3. Paquete utils (utils)

Este paquete contiene clases auxiliares para el resto de la librería, la mas importante es la interfaz IComparable y sus dos implementaciones, ComparableInt y ComparableFloat, las cuales son utilizadas en las evaluaciones de estados. La clase SimpleGame presenta una clase para poder probar un juego con adversarios, es posible utilizar esta clase para probar como se debería implementar un juego con adversarios, así como utilizar el motor para pedir nuevas jugadas, sin embargo puede no ser muy eficiente ni comodo para muchos casos, como cuando se quiere realizar un juego con interfaz gráfica, en estos casos se recomienda el patrón de diseño "Model View Controler (MVC)" que fue el utilizado para los dos casos de uso implementados.

6.1.5.4. Paquetes ejemplos (example_Flood-It, example_reversi)

Estos paquetes contienen los casos de uso implementados, ambos están implementados con una interfaz gráfica, aunque en el caso del reversi permanece una versión del juego por consola que fue utilizado en las primeras instancias para probar el motor de búsqueda y los modulos del juego (estado y regla), su funcionamiento probablemente no sea del todo correcto y se lo debería tratar como un simple remanente de una etapa de pruebas inicial.

6.1.5.5. Paquete de imágenes (images)

En este paquete se guardan todas las imagenes utilizadas en las distintas interfaces gráficas, el uso del paquete es para que las imagenes queden almacenadas dentro del archivo JAR.

6.1.5.6. Paquetes y clases

Paquete	Clase	Descripción
framework	IBasicState	Estado
framework	IinformedState	Estado informado
framework	IStateAdversary	Estado para problemas con adversario
framework	SearchProblem	Problema de búsqueda
framework	IRule	Regla de cambio de estado
framework	BasicSearchEngine	Motor de búsqueda
framework	SearchEngine	Motor de búsqueda sin adversarios
framework	AdversarySearchEngine	Motor de búsqueda con adversarios
engines	BestFirstSearchEngine	Motor de búsqueda que utiliza algoritmo de búsqueda “Best First Search”
engines	DepthFirstSearchEngine	Motor de búsqueda que utiliza algoritmo de búsqueda “primero en profundidad”
engines	GuidedDepthFirstSearchEngine	Motor de búsqueda que utiliza algoritmo de búsqueda “primero en profundidad” modificado para utilizar función de evaluación y así ordenar la lista de sucesores a visitar
engines	MinMaxABSearchEngine	Motor de búsqueda para problemas de búsqueda con adversarios, que utiliza algoritmo de búsqueda “Min Max con poda Alfa-Beta”
utils	IComparable	Clase utilizada para el valor de retorno de funciones de evaluación
utils	ComparableInt	Instancia de IComparable que utiliza enteros como valor interno
utils	ComparableFloat	Instancia de IComparable que utiliza reales

		como valor interno
utils	Pair	Una clase para trabajar con pares heterogéneos de elementos
utils	SimpleGame	Una clase de prueba para mostrar como se puede utilizar un motor de búsqueda (con adversarios) en una aplicación
example_Flood-It	FlodditApp_gui	Clase principal para correr el ejemplo del Flood-It
example_Flood-It	Flood-ItState	Clase que define el estado del Flood-It
example_Flood-It	Flood-ItToken	Clase que representa a una ficha utilizada dentro del tablero del Flood-It
example_Flood-It	Flood-ItMoveRule	Regla de cambio de estado del Flood-It
example_Flood-It	Flood-ItController	Controlador para la interfaz principal del Flood-It
example_Flood-It	Flood-ItMainGui	Interfaz gráfica principal para el Flood-It
example_Flood-It	IAController	Controlador para la inteligencia artificial (motor de búsqueda) utilizada en el Flood-It
example_Flood-It	IAGui	Interfaz gráfica para la inteligencia del Flood-It (utilizada para mostrar todas las jugadas a realizar para ganar el juego)
example_Flood-It	BoardRender	Renderizador de la tabla gráfica utilizada en la interfaz gráfica principal.
example_Flood-It	AboutFlood-It	Interfaz gráfica para la ventana “Acerca de” del Flood-It
example_Flood-It	WinLoseWindow	Interfaz gráfica para la ventana de “ganaste/perdiste” del Flood-It
example_reversi	ReversiState	Define el estado del juego Reversi
example_reversi	ReversiToken	Representa a una ficha, utilizada en el tablero del Reversi
example_reversi	ReversiMoveRule	Regla de cambio de estado del Reversi
example_reversi	ReversiLogic	Se encarga de manejar la lógica del juego (Reversi)
example_reversi	ReversiController	Controlador para la interfaz principal del Reversi
example_reversi	ReversiAISettingsController	Controlador para la interfaz de preferencias de la inteligencia artificial utilizada en el Reversi

example_reversi	Model	Utilizado por la lógica, representa el modelo del juego reversi (tablero, nombres de jugadores, colores de los jugadores, etc)
example_reversi	ReversiBoardRenderer	Renderizador de la tabla gráfica utilizada en la interfaz gráfica principal.
example_reversi	ReversiGUI	Interfaz gráfica principal del Reversi
example_reversi	SettingsGUI	Interfaz gráfica para las preferencias de juego del Reversi
example_reversi	EndGameDialog	Interfaz gráfica para el dialogo de “perdiste/ganaste/empate” del Reversi
example_reversi	AIgui	Interfaz gráfica para las preferencias de la inteligencia artificial del Reversi
examples	GaroePlayground	Aplicación principal para probar todos los ejemplos de la plataforma. Permite agregar de manera simple nuevos ejemplos a probar
examples	GaroePlaygroundController	Controlador para la interfaz principal del “GaroePlayground”
examples	GaroePlaygroundGUI	Interfaz principal para el “GaroePlayground”
examples	GaroePlaygroundAbout	Interfaz para el “Acerca de” utilizado en “GaroePlayground”
examples	Flood-ItGame	Inicializador del juego Flood-It para utilizarlo desde “GaroePlayground”
examples	ReversiGame	Inicializador del juego Reversi para utilizarlo desde “GaroePlayground”
Images	---	Contiene todas las imágenes utilizadas en las aplicaciones con interfaz gráfica.

7. Conclusión

Esta sección estará dividida en dos subsecciones, una donde nosotros describiremos lo que hemos aprendido durante el desarrollo de esta plataforma. La otra será un resumen de todo el informe, conteniendo lo que vale la pena destacar de esta plataforma y las ventajas que tiene su utilización visto desde nuestro punto de vista gracias a la experiencia que obtuvimos al utilizar a la misma para desarrollar los casos de prueba.

Lo que aprendimos

Durante el desarrollo de esta plataforma fue necesario comprender como funcionan los problemas de búsqueda desde el punto de vista del diseño e implementación de los mismo, buscar las características en común que tienen distintos tipos de problemas, como aquellos que solo preguntan si un estado es final o si es exitoso, aquellos que utilizan una función de valoración para determinar cuan cerca está un estado de resolver el problema o cuan "bueno" es en relación al problema a resolver, así como poder comprobar que todas las soluciones a problemas de búsqueda tienen algún esquema similar a "Estado" + "Problema" + "Reglas de cambio de estado" + "Motor de búsqueda", que no necesariamente están separados en clases distintas pero si en la idea general del diseño, es decir que puede que las reglas formen parte del problema (o del estado) pero que se las puede identificar claramente.

La necesidad de implementar varios motores de búsqueda nos permitió entender de forma mucho mas clara los distintos algoritmos que utilizamos (MinMax con Poda, búsqueda en profundidad y a lo ancho) y como realizar modificaciones a los mismos según nuestras necesidades (búsqueda en profundidad guiada por evaluación de estados), así como afianzar los conocimientos sobre búsqueda aprendidos durante el cursado de la carrera.

El tener que realizar un diseño de una plataforma que debe cumplir con ser lo más genérica posible y la utilización de un lenguaje orientado a objetos como Java, nos permitió mejorar nuestro conocimiento sobre diseño y utilización del paradigma de POO, ya que cuando uno debe realizar algún programa que hace algo el foco se encuentra en que se obtenga algún resultado esperado, en cambio cuando no hay un programa sino una base sobre la cual se van a escribir programas uno debe pensar en que cualquier programa que pertenezca a la categoría de problemas de búsqueda (en nuestro caso) pueda ser escrito de forma clara, simple y dando independencia a todas las partes del programa (en nuestro caso podemos nombrar el estado, motor, problema, etc).
Hubo muchos problemas que aparecieron durante el desarrollo que no estaban ligados al conocimiento del lenguaje o dificultades en la realización del diseño, sino que estaban relacionados con problemas mas prácticos como las distintas versiones que se iban generando del proyecto y como mantener todos los archivos actualizados a la última versión cuando se utilizan varias computadoras para trabajar, así como poder identificar cual de todas las versiones del proyecto contenía un archivo particular en donde un cambio había generado algún error. En un principio se opto por la utilización de Dropbox [AGREGAR REFERENCIA] pero el problema seguía estando, solo que ahora era posible compartir el problema con todas las computadoras utilizadas. Finalmente el cambio de Dropbox por un sistema de versionado (SVN[AGREGAR REFERENCIA]) junto a "Google Code" para la administración de datos del proyecto y como servidor para el repositorio hizo que el desarrollo fuera mucho más dinámico y eliminó cualquier problema de numerosas

versiones, con este cambio era mucho mas simple encontrar que cambio era el que había generado un problema en particular y poder asegurarse de que siempre se estuviera trabajando con la última versión de los archivos.
Junto con la aceptación de “Google Code” y SVN para control de versiones, se cambió de IDE (NetBeans) con lo cual se pudo experimentar con otro IDE, con otras funcionalidades, con sus ventajas y desventajas. El nuevo IDE, si bien contiene algunas desventajas en cuanto al IDE anterior (Eclipse) permitió mayor velocidad de desarrollo para interfaces gráficas, un trabajo mas cómodo mezclando la utilización de control de versión y la integración de versionado dentro del IDE.

Experiencia de utilizar la plataforma

Al finalizar el desarrollo de la plataforma, se comenzó con el desarrollo de los ejemplos de uso para la misma, el trabajo realizado, se puede resumir en:

1) Implementar estado, regla para Reversi
2) Implementar Motor de búsqueda MinMax con poda Alfa-Beta
3) Implementar un juego completo de Reversi
4) Corrección del algoritmo de MinMax
5) Implementar estado, regla para Flood-it
6) Implementar Motor Best First Search
7) Implementar un juego completo de Flood-it
8) Comprobación de que el motor elegido no funcionaba con el Flood-it
9) Modificación de la regla de cambio de estado para Flood-it
10) Implementación de Motor Depth First Search
11) Comprobación de que el motor elegido era muy lento
12) Nueva modificación de la regla de cambio de estado para Flood-it
13) Modificación de la generación del tablero del Flood-it
14) Implementación de Motor Guided Depth First Search

Los puntos importantes a destacar son la implementación de estados, reglas y motores, así como la utilización de los mismos por una aplicación. Estos puntos que son los que están directamente relacionados con la plataforma, mostraron que el uso de la plataforma provee las mismas ventajas que trae el uso de un patrón de diseño para resolver un problema, es decir diseñar e implementar la solución del mismo. Cuando fue necesario implementar a los distintos motores, el esfuerzo estuvo solamente sobre el algoritmo de búsqueda y el diseño de la plataforma no se interpuso en ningún momento, lo mismo sucedió cuando se implementaron las reglas para los problemas, para cada regla solo fue necesario completar ambas funciones definidas en la interfaz IRule, lo mismo sucedió a su vez en la implementación de los estados. Con lo que podemos afirmar que efectivamente esta plataforma simplifica en gran manera el desarrollo de aplicaciones que utilizan algoritmos de búsqueda, y que en ningún momento se enfrento a alguna situación donde la plataforma nos causaba alguna limitación o dificultad extra.

8. Bibliografía

[REF#1] - http://homepage.ufp.pt/~jtorres/ensino/ia/alfabeta.html – Ejemplos Gráficos MinMax con poda Alfa-Beta

[REF#2] - Técnicas de Diseño de algoritmos (material de lectura de la materia diseño de algoritmos)

[REF#3] – Filminas de la materia diseño de algoritmos
[REF#4]- http://es.wikipedia.org/wiki/Java_(lenguaje_de_programaci%C3%B3n)
[REF#5]- Desing Patterns, Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides.