Acercamiento Voraz a problema de Gatos y Ratones

Emanuel Fonseca, Nicolás Prantl y Germán Márquez

Universidad Católica del Uruguay

Septiembre 2022

Acercamiento voraz a problema de Gatos y Ratones

Análisis del problema y desarrollo de solución voraz

A la hora de diseñar una solución voraz al problema planteado debemos considerar qué aspecto del problema debe ser "optimizado".
Los algoritmos voraces resuelven un problema iniciando desde un punto de vista general y trabajando de manera top-down, solucionando el problema a medida que se llega a subproblemas menores. Un algoritmo codicioso siempre toma la decisión que se ve mejor en ese momento. Hace una elección localmente óptima con la esperanza de que esta elección conduzca a una solución globalmente óptima.
Considerando el problema planteado, la optimización del algoritmo **maximizará la cantidad de ratones comidos**. Como cada gato puede comer un solo ratón, la solución ideal será que el número de ratones comidos sea igual, o lo más aproximado posible, al número de gatos.

Con esto en mente, debemos identificar los posibles problemas que pueden surgir al intentar alcanzar la solución óptima. Debido a que los gatos comen un único ratón pero pueden elegir entre varios de ellos de acuerdo al alcance definido, los principales factores que pueden afectarnos son el **Overlap y Aislamiento**. Gatos próximos entre sí tendrán alcances solapados, lo cual conduce a que tengan acceso compartido a comer grupos de ratones similares y por lo tanto, cuando un gato coma un ratón, este ratón debe ser subsecuentemente incomible por el resto. Este overlap también puede causar aislamiento. Si un ratón fuese por ejemplo alcanzable por un único gato pero este elige otro ratón en su rango, este ratón ha quedado aislado e incomible, lo cual puede llevar a una solución no-óptima.

Desarrollo de implementación al problema

Debido a que cada gato puede comer un único ratón, nuestro algoritmo debe definir un orden en que los gatos coman los ratones. La idea es sencilla, el gato con menos cantidad de ratones que tenga para comer, será el próximo en comer.

Luego de haber comido un ratón, el gato pasará a estar deshabilitado de comer nuevamente y el ratón no podrá ser comido una segunda vez. Esto define pues la **subestructura** de nuestro algoritmo óptimo - tras elegir a un gato que come a un ratón, la siguiente iteración del algoritmo operará sobre el resto de los gatos que todavía no han podido comer, y habrá un ratón menos disponible. El algoritmo continuará hasta que no hayan gatos que posean ratones disponibles a su alcance para poder comer.

La selección que proponemos pues para cada iteración del algoritmo es permitir **comer al gato con menor cantidad de ratones alcanzables**. La razón de esta selección es simple - **está elección nos da el mayor grado de seguridad** en minimizar overlap/aislamiento-. Los gatos en mayor riesgo a ser deshabilitados por sus vecinos son aquellos con menor cantidad de ratones de los cuales elegir, y por lo tanto elegimos que coman primero, creando tras esta elección subproblemas un nuevo gato pasa a ser el de mayor riesgo, eligiéndolo en cada iteración hasta llegar a que todos los gatos han comido o que la disposición del array inherentemente iba a dejar a algunos gatos sin comer.

Comparación con Algoritmo Fuerza Bruta

Resolver un algoritmo mediante fuerza bruta es simple, requiere solamente calcular todas las alternativas y elegir la mejor entre ellas. Esta simpleza es sin embargo la razón por la cual este acercamiento es raras veces el mejor - no se tiene ninguna técnica inteligente para obtener el mejor resultado. Frente a problemas extremadamente simples este acercamiento es fácil de programar, más cuando se introduce el menor grado de dificultad, variación o múltiples elecciones a un problema, la cantidad de casos posibles crece exponencialmente, y en igual manera crece el costo de calcular esta cantidad de soluciones.

Volviendo a nuestro problema de gatos y ratones, la fuerza bruta aplicada al problema planteado necesitaría que se calculase todas las posibles combinaciones de gatos-ratones a la hora de comer, seleccionando finalmente la mejor opción - como mencionamos anteriormente, aquellas donde la cantidad de ratones comidos sea lo más próxima o igual a el número de gatos.

Consideramos entonces la cantidad de selecciones sobre la que trabajaría un algoritmo bruto. En el peor de los casos, suponemos que cada gato tendrá su alcance lleno de ratones. Con el alcance mínimo de valor 1, cada gato tendría pues dos ratones entre los cuales elegir a su alrededor. No bastará considerar las elecciones en que todos los gatos eligen a su ratón izquierdo o a su ratón derecho, sino que el algoritmo bruto deberá calcular todos los casos combinatorios combinando ambas selecciones para todos los gatos. Este conjunto de posibilidades representa un conjunto de partes, por lo que el **orden del algoritmo bruto será $O(2^n)$.**

Nótese además que una vez calculadas todas las alternativas posibles, el algoritmo debería recorrer este enorme volumen de casos buscando cual es el mejor de todos (o al menos que satisfaga la condición de ser una respuesta óptima) agregando una considerable iteración final.

Por el otro lado, el acercamiento voraz nos presenta un volumen de cálculos mucho menor. La operativa de este algoritmo (selecciones óptimas locales, iteración o recursión sobre subproblemas optimizables) termina requiriendo una menor cantidad de iteraciones sobre el arreglo que calcular alternativas combinatorias, teniendo el agregado de devolver una única solución que podemos confiar es óptima en vez de tener que buscarla sobre una enorme cantidad de soluciones posibles.

Descripción de la Implementación

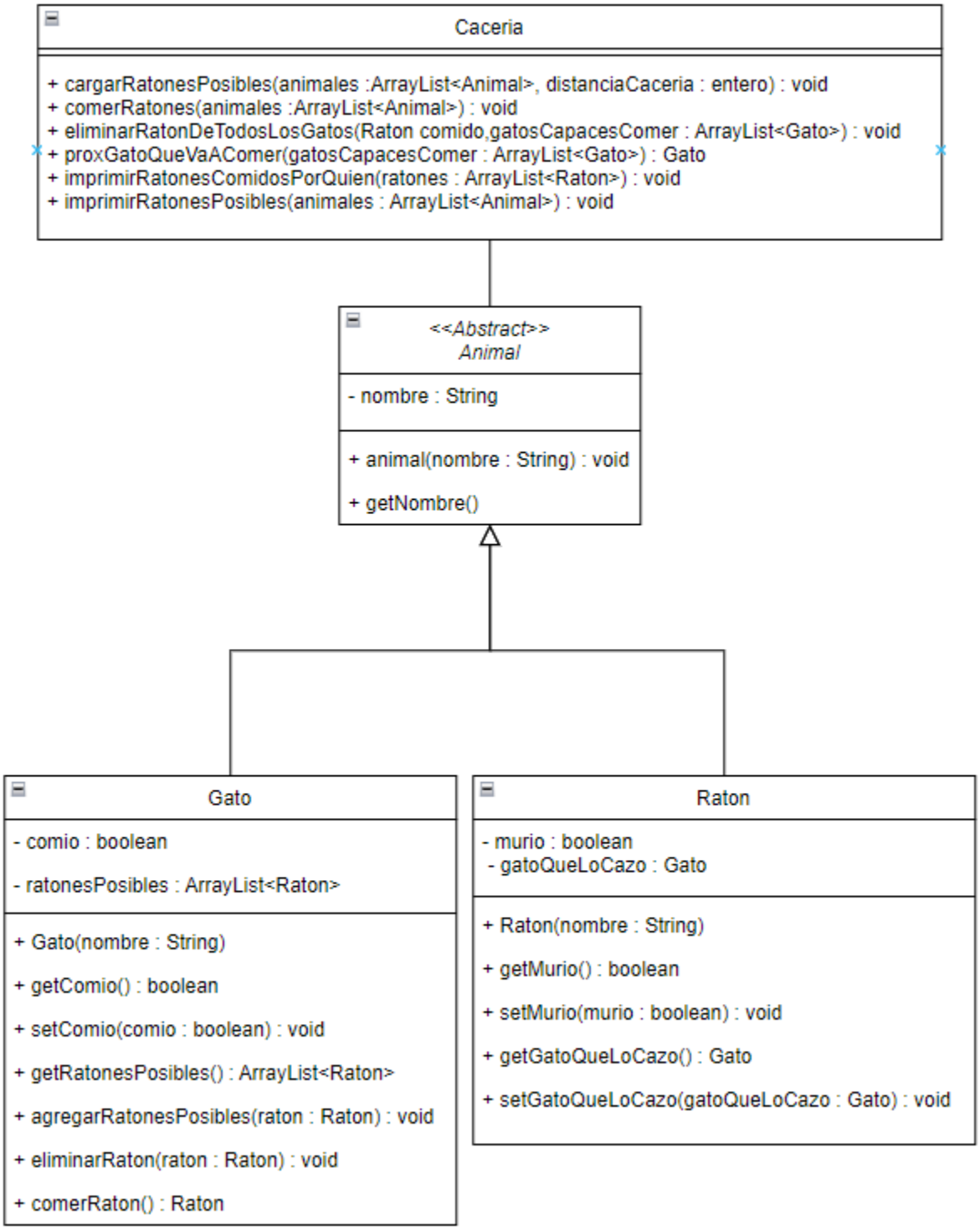

Fig 1. Diagrama de clases UML

Como se explicaba en las secciones anteriores, nuestro algoritmo permite que el gato con menor cantidad de ratones disponibles para comer será el que tenga prioridad en el orden de comida. A la hora de implementar este algoritmo, esto plantea la necesidad de conocer dicha información, para lo cual implementamos el método de “cargar ratones posibles”. Este método se ejecuta primero en nuestro algoritmo, encargándose de recorrer el array de animales en búsqueda de los gatos. En cada gato encontrado, el algoritmo comienza una iteración anidada que recorre el rango de la distancia de cacería definida a ambos lados del gato, guardando todos los ratones encontrados en el array perteneciente a cada gato, “ratones posibles”. Al terminar la iteración principal todos los gatos del array conocerán que ratones están a su alcance al empezar la cacería de ratones.

La cacería de ratones se maneja desde el algoritmo “comer ratones”, que trabaja sobre dos arrays que instancia, una lista de gatos por comer, y una lista de ratones a ser comidos. El algoritmo itera sobre el array entero una vez, guardando en ambos arrays los gatos que tengan ratones por comer en su array interno por un lado, y los ratones que se pudiesen comer por el otro, para simplificar su acceso y edición.

Tras terminar la separación se inicia la lógica de elegir gatos y comer ratones. Un loop mientras que trabajará hasta que no queden gatos en el array de “gatos por comer” emplea el método “próximo gato que va a comer” para elegir el la lista de gatos el que tenga menor cantidad de ratones disponibles, luego realizando la representación de la comida mediante métodos de “comer ratón (gato)”, “set murió (ratón)” y “set gato que lo comió (ratón)”. De esta manera se representa correctamente que el ratón ha sido comido.

Finalmente, un último método “eliminar ratón de todos los gatos” se encarga de remover el recientemente comido ratón de la lista de ratones alcanzables de todos los gatos que no hayan comido todavía. La necesidad de actualizar esta información es lo que justifica la separación de gatos y ratones en arrays separados para facilitar su acceso. Luego, un método de impresión se encarga de imprimir por terminal que ratones fueron comidos y por qué gatos.

Finalmente es necesario clarificar cómo opera el método “próximo gato que va a comer”, mencionado previamente. Este método simplemente itera a través de la lista de gatos que esperan comer, recordando aquel que tenga la menor cantidad de ratones disponibles y devolviéndolo al final para que se pueda operar con él. Sin embargo, este método además aprovecha la iteración a través de los gatos para revisar que, durante la comida de ratones, no siga en la lista un gato que ha quedado deshabilitado, cuya lista de ratones disponibles se encuentre vacía - evitando así operar sobre gatos incapaces de comer de manera innecesaria.

Obsérvese pues que nuestra solución, que requiere el uso de una iteración “para cada” anidada dentro de una iteración “mientras” tendrá un **orden de tiempo de ejecución cuadrático**.

Conclusiones

Hemos visto al desarrollar este ejercicio cómo se diferencia el planteamiento de un algoritmo voraz a uno de fuerza bruta, repasando los pasos necesarios para diseñar y reconocer situaciones donde estos algoritmos son aplicables y útiles. A la vez, el implementar nuestra solución es una forma de ver como se requiere un grado mayor de complejidad a la hora de aplicar una solución de este tipo, en vez de un algoritmo de fuerza bruta que intercambia eficiencia por simplicidad. Hemos notado como el Orden de tiempo de ejecución de nuestra solución voraz es cuadrático mientras que el orden del algoritmo de fuerza bruta es exponencial, por lo cual podemos asegurar que el algoritmo voraz será más eficiente.

Bibliografía

Thomas Cormen (2022). "*Introduction to Algorithms*". MIT Press, 3rd Edition.