Universidad Francisco Marroquín

Estructuras de Datos

Enrique Andrés Bolaños

Carné: 20180521

Algoritmos de búsqueda y ordenamiento

Expectiminimax

El algoritmo expectiminimax es una variación del algoritmo minimax, para uso en sistemas de inteligencia artificial que juegan “juegos de suma cero” para dos jugadores, como el backgammon, en el que el resultado depende de una combinación de elementos de habilidad y azar del jugador, como las tiradas de dados. Además de los nodos "min" y "max" del árbol minimax tradicional, esta variante tiene nodos "chance" ("mover por naturaleza"), que toman el valor esperado de un evento aleatorio que ocurre. En términos de la teoría de juegos, un árbol expectiminimax es el árbol de juegos de un juego extenso de información perfecta, pero incompleta.

En el método tradicional de minimax, los niveles del árbol se alternan de max a min hasta que se alcanza el límite de profundidad del árbol. En un árbol expectiminimax, los nodos "casuales" se intercalan con los nodos máximo y mínimo. En lugar de tomar el máximo o el mínimo de los valores de utilidad de sus hijos, los nodos aleatorios toman un promedio ponderado, siendo el peso la probabilidad de que se alcance al niño.

El entrelazado depende del juego. Cada "turno" del juego se evalúa como un nodo "max" (que representa el turno del jugador de IA), un nodo "min" (que representa el turno de un oponente potencialmente óptimo), o un nodo "casual" (que representa un efecto aleatorio o Jugador 1)

Por ejemplo, considere un juego en el que cada ronda consiste en un solo lanzamiento de dados, y luego las decisiones tomadas primero por el jugador de IA, y luego por otro oponente inteligente. El orden de los nodos en este juego alternaría entre "chance", "max" y luego "min".

Ventajas:

1. Es más rápido que el normal minimax
2. Permite optimizar la búsqueda

Class	Search algorithm
Worst-case performance	$\mathcal{O}(b^m n^m)$, where n is the number of distinct dice throws
Best-case performance	$\mathcal{O}(b^m)$, in case all dice throws are known in advance

Pseudocode:

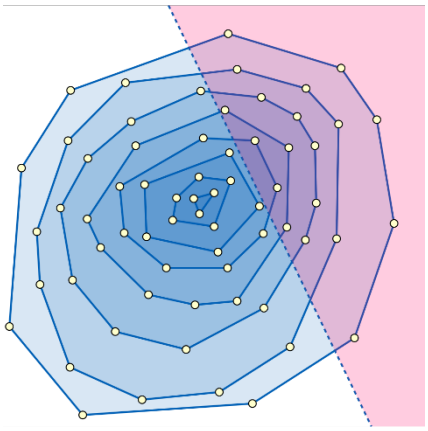
```
function expectiminimax(node, depth)
    if node is a terminal node or depth = 0
        return the heuristic value of node
    if the adversary is to play at node
        // Return value of minimum-valued child node
        let α := +∞
        foreach child of node
            α := min(α, expectiminimax(child, depth-1))
    else if we are to play at node
        // Return value of maximum-valued child node
        let α := -∞
        foreach child of node
            α := max(α, expectiminimax(child, depth-1))
    else if random event at node
        // Return weighted average of all child nodes' values
        let α := 0
        foreach child of node
            α := α + (Probability[child] * expectiminimax(child, depth-1))
    return α
```

Fractional cascading

En informática, la cascada fraccional es una técnica para acelerar una secuencia de búsquedas binarias para el mismo valor en una secuencia de estructuras de datos relacionadas. La primera búsqueda binaria en la secuencia toma una cantidad logarítmica de tiempo, como es estándar en las búsquedas binarias, pero las búsquedas sucesivas en la secuencia son más rápidas. La versión original de la cascada fraccional, presentada en dos artículos por Chazelle y Guibas en 1986 (Chazelle y Guibas 1986a; Chazelle y Guibas 1986b) combinó la idea de la cascada, originada en la búsqueda de rango de estructuras de datos de Lueker (1978) y Willard (1978), con la idea de muestreo fraccional, que se originó en Chazelle (1983). Los autores posteriores introdujeron formas más complejas de cascada fraccionada que permiten mantener la estructura de datos a medida que los datos cambian mediante una secuencia de eventos discretos de inserción y eliminación.

Aplicaciones

Las aplicaciones típicas de la cascada fraccional involucran estructuras de datos de búsqueda de rango en geometría computacional. Por ejemplo, considere el problema de los informes de rango del plano medio: es decir, intersectando un conjunto fijo de n puntos con un plano medio de consulta y listando todos los puntos en la intersección



Alpha–beta pruning

La poda alfa-beta es un algoritmo de búsqueda que busca disminuir el número de nodos que son evaluados por el algoritmo minimax en su árbol de búsqueda. Es un algoritmo de búsqueda adversarial utilizado comúnmente para juegos de dos jugadores (Tic-tac-toe, Chess, Go, etc.). Deja de evaluar un movimiento cuando se ha encontrado al menos una posibilidad que demuestra que el movimiento es peor que un movimiento examinado previamente. Tales movimientos no necesitan ser evaluados más a fondo. Cuando se aplica a un árbol minimax estándar, devuelve el mismo movimiento que minimax, pero elimina las ramas que no pueden influir en la decisión final.

```
function alphabeta(node, depth, α, β, maximizingPlayer) is
    if depth = 0 or node is a terminal node then
        return the heuristic value of node
    if maximizingPlayer then
        value := −∞
        for each child of node do
            value := max(value, alphabeta(child, depth − 1, α, β, FALSE))
            α := max(α, value)
            if α ≥ β then
                break (* β cut-off *)
        return value
    else
        value := +∞
        for each child of node do
            value := min(value, alphabeta(child, depth − 1, α, β, TRUE))
            β := min(β, value)
            if α ≥ β then
                break (* α cut-off *)
        return value
```

El algoritmo mantiene dos valores, alfa y beta, que representan la puntuación mínima que el jugador maximizador tiene asegurada y la puntuación máxima que el jugador minimizador tiene asegurada respectivamente. Inicialmente, alfa es infinito negativo y beta es infinito positivo, es decir, ambos jugadores comienzan con la peor puntuación posible. Siempre que el puntaje máximo que el jugador minimizador (es decir, el jugador "beta") tenga asegurado sea menor que el puntaje mínimo que el jugador que maximiza (es decir, el jugador "alfa") tiene asegurado (es decir, beta ≤ alfa), el maximizador el jugador no necesita considerar más descendientes de este nodo, ya que nunca será alcanzado en el juego real.

Class	Search algorithm
Worst-case performance	$O(b^d)$
Best-case performance	$O\left(\sqrt{b^d}\right)$

Tree sort

Una clasificación de árbol es un algoritmo de clasificación que crea un árbol de búsqueda binaria a partir de los elementos que se van a ordenar y luego atraviesa el árbol (en orden) para que los elementos salgan en orden ordenado. Su uso típico es ordenar elementos en línea: después de cada inserción, el conjunto de elementos visto hasta ahora está disponible en orden ordenado.

Agregar un elemento a un árbol de búsqueda binario es en promedio un proceso O (log n) (en notación O grande). Agregar n elementos es un proceso O (n log n), lo que hace que la clasificación en árbol sea un proceso de "clasificación rápida". Agregar un elemento a un árbol binario desequilibrado requiere O (n) tiempo en el peor de los casos: cuando el árbol se parece a una lista enlazada (árbol degenerado). Esto resulta en el peor de los casos de tiempo O (n^2) para este algoritmo de clasificación. Este peor caso ocurre cuando el algoritmo opera en un conjunto ya ordenado, o uno que está casi ordenado, invertido o casi invertido. Sin embargo, el tiempo esperado de O (n log n) se puede lograr al mezclar la matriz, pero esto no ayuda para elementos iguales.

El comportamiento del caso más desfavorable se puede mejorar mediante el uso de un árbol de búsqueda binaria con equilibrio automático. Usando un árbol de este tipo, el algoritmo tiene un O (n log n) peor desempeño, por lo que es óptimo para un orden de comparación. Sin embargo, los árboles requieren que la memoria se asigne en el montón, lo que es un acierto significativo en el rendimiento en comparación con quicksort y heapsort. Cuando se utiliza un árbol de distribución como árbol de búsqueda binario, el algoritmo resultante (llamado splaysort) tiene la propiedad adicional de que es una clasificación adaptativa, lo que significa que su tiempo de ejecución es más rápido que O (n log n) para las entradas que están casi ordenadas.

```haskell
data Tree a = Leaf | Node (Tree a) a (Tree a)

insert :: Ord a => a -> Tree a -> Tree a
insert x Leaf = Node Leaf x Leaf
insert x (Node t y s)
    | x <= y = Node (insert x t) y s
```

```
    | x > y  = Node t y (insert x s)

flatten :: Tree a -> [a]
flatten Leaf = []
flatten (Node t x s) = flatten t ++ [x] ++ flatten s

treesort :: Ord a => [a] -> [a]
treesort = flatten . foldr insert Leaf
```

Tournament sort

La clasificación de torneos es un algoritmo de clasificación. Mejora la ordenación de selección ingenua al usar una cola de prioridad para encontrar el siguiente elemento en la ordenación. En la selección de selección ingenua, se requieren operaciones O (n) para seleccionar el siguiente elemento de n elementos; en una clasificación de torneo, toma O (log n) operaciones (después de construir el torneo inicial en O (n)). La clasificación de torneos es una variación de heapsort.

Las clasificaciones de selección de reemplazo de torneo se utilizan para recopilar las ejecuciones iniciales para algoritmos de clasificación externos. Conceptualmente, se lee un archivo externo y sus elementos se insertan en la cola de prioridad hasta que se llena. Luego, el elemento mínimo se extrae de la cola y se escribe como parte de la primera ejecución. El siguiente elemento de entrada se lee y se inserta en la cola, y el mínimo se selecciona de nuevo y se agrega a la ejecución. Existe un pequeño truco: si el nuevo elemento que se está insertando en la cola es menor que el último elemento agregado a la ejecución, el valor de clasificación del elemento se incrementa, por lo que formará parte de la próxima ejecución. En promedio, una ejecución será 100% más larga que la capacidad de la cola de prioridad.

Las clasificaciones de torneos también se pueden usar en las combinaciones de N-way.

```
import Data.Tree


-- | Adapted from `TOURNAMENT-SORT!` in the Stepanov and Kershenbaum report.
tournamentSort :: Ord t
       => [t]  -- ^ Input: an unsorted list
       -> [t]  -- ^ Result: sorted version of the input
tournamentSort alist
        = go (pure<$>alist) -- first, wrap each element as a single-tree forest
 where go [] = []
       go trees = (rootLabel winner) : (go (subForest winner))
        where winner = playTournament trees


-- | Adapted from `TOURNAMENT!` in the Stepanov and Kershenbaum report
playTournament :: Ord t
         => Forest t -- ^ Input forest
         -> Tree t   -- ^ The last promoted tree in the input
playTournament [tree] = tree
playTournament trees = playTournament (playRound trees [])
```

```haskell
-- | Adapted from `TOURNAMENT-ROUND!` in the Stepanov and Kershenbaum report
playRound :: Ord t
       => Forest t -- ^ A forest of trees that have not yet competed in round
       -> Forest t -- ^ A forest of trees that have won in round
       -> Forest t -- ^ Output: a forest containing promoted versions
                   --   of the trees that won their games
playRound [] done = done
playRound [tree] done = tree:done
playRound (tree0:tree1:trees) done = playRound trees (winner:done)
 where winner = playGame tree0 tree1


-- | Adapted from `TOURNAMENT-PLAY!` in the Stepanov and Kershenbaum report
playGame :: Ord t
         => Tree t  -- ^ Input: ...
         -> Tree t  -- ^ ... two trees
         -> Tree t  -- ^ Result: `promote winner loser`, where `winner` is
                    --   the tree with the *lesser* root of the two inputs
playGame tree1 tree2
    | rootLabel tree1 <= rootLabel tree2  = promote tree1 tree2
    | otherwise                           = promote tree2 tree1


-- | Adapted from `GRAB!` in the Stepanov and Kershenbaum report
promote :: Tree t  -- ^ The `winner`
        -> Tree t  -- ^ The `loser`
        -> Tree t  -- ^ Result: a tree whose root is the root of `winner`
                   --   and whose children are:
                   --   * `loser`,
                   --   * all the children of `winner`
promote winner loser = Node {
    rootLabel = rootLabel winner,
    subForest = loser : subForest winner}


main :: IO ()
main = print $ tournamentSort testList
 where testList = [-0.202669, 0.969870, 0.142410, -0.685051, 0.487489, -0.339971, 0.832568,
0.00510796, -0.822352, 0.350187, -0.477273, 0.695266]
```

Spreadsort

Spreadsort es un algoritmo de clasificación inventado por Steven J. Ross en 2002. Combina conceptos de clasificaciones basadas en la distribución, como la ordenación de radix y ordenación por depósito, con conceptos de partición de ordenaciones comparativas como quicksort y mergesort. En los resultados experimentales, se demostró que era altamente eficiente, a menudo superaba a los algoritmos tradicionales, como el ordenamiento rápido, particularmente en distribuciones que exhibían estructura y clasificación de cadenas. Existe una implementación de código abierto con análisis de rendimiento y puntos de referencia [2], y documentación HTML [3].

Quicksort identifica un elemento de pivote en la lista y luego divide la lista en dos sublistas, aquellos elementos menores que el pivote y aquellos mayores que el pivote. Spreadsort generaliza esta idea al dividir la lista en n / c particiones en cada paso, donde n es el número total de elementos en la lista y c es una constante pequeña (en la práctica, generalmente entre 4 y 8 cuando las comparaciones son lentas o mucho más grandes) en situaciones donde son rápidos). Utiliza técnicas basadas en la distribución para lograr esto, primero ubicando el valor mínimo y máximo en la lista, y luego dividiendo la región entre ellos en n / c contenedores de igual tamaño. Cuando el almacenamiento en caché es un

problema, puede ayudar tener un número máximo de contenedores en cada paso de división recursiva, lo que hace que este proceso de división tome varios pasos. Aunque esto causa más iteraciones, reduce las fallas de caché y puede hacer que el algoritmo se ejecute más rápido en general.

```
// spreadsort sorting example

//

//  Copyright Steven Ross 2009-2014.

//

// Distributed under the Boost Software License, Version 1.0.

//    (See accompanying file LICENSE_1_0.txt or copy at

//          http://www.boost.org/LICENSE_1_0.txt)

//  See http://www.boost.org/libs/sort for library home page.

#include <boost/sort/spreadsort/spreadsort.hpp>
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <algorithm>
#include <vector>
#include <string>
#include <fstream>
#include <sstream>
#include <iostream>
using namespace boost::sort::spreadsort;

#define DATA_TYPE int


//Pass in an argument to test std::sort
int main(int argc, const char ** argv) {
  size_t uCount,uSize=sizeof(DATA_TYPE);
  bool stdSort = false;
  unsigned loopCount = 1;
  for (int u = 1; u < argc; ++u) {
    if (std::string(argv[u]) == "-std")
      stdSort = true;
    else
      loopCount = atoi(argv[u]);
  }
  std::ifstream input("input.txt", std::ios_base::in | std::ios_base::binary);
  if (input.fail()) {
    printf("input.txt could not be opened\n");
    return 1;
  }
  double total = 0.0;
  std::vector<DATA_TYPE> array;
  input.seekg (0, std::ios_base::end);
    size_t length = input.tellg();
  uCount = length/uSize;
  //Run multiple loops, if requested
  for (unsigned u = 0; u < loopCount; ++u) {
```

```
    input.seekg (0, std::ios_base::beg);
    //Conversion to a vector
    array.resize(uCount);
    unsigned v = 0;
    while (input.good() && v < uCount)
      input.read(reinterpret_cast<char *>(&(array[v++])), uSize );
    if (v < uCount)
      array.resize(v);
    clock_t start, end;
    double elapsed;
    start = clock();
    if (stdSort)
      std::sort(array.begin(), array.end());
    else
      boost::sort::spreadsort::spreadsort(array.begin(), array.end());
    end = clock();
    elapsed = static_cast<double>(end - start);
    std::ofstream ofile;
    if (stdSort)
      ofile.open("standard_sort_out.txt", std::ios_base::out |
                 std::ios_base::binary | std::ios_base::trunc);
    else
      ofile.open("boost_sort_out.txt", std::ios_base::out |
                 std::ios_base::binary | std::ios_base::trunc);
    if (ofile.good()) {
      for (unsigned v = 0; v < array.size(); ++v) {
        ofile.write(reinterpret_cast<char *>(&(array[v])), sizeof(array[v]) );
      }
      ofile.close();
    }
    total += elapsed;
    array.clear();
  }
  input.close();
  if (stdSort)
    printf("std::sort elapsed time %f\n", total / CLOCKS_PER_SEC);
  else
    printf("spreadsort elapsed time %f\n", total / CLOCKS_PER_SEC);
  return 0;
}
```