ALTRI PARADIGMI DI PROGRAMMAZIONE

A.A. 2020/2021

Laurea triennale in Informatica

13: Esempi svolti 3

TIC TAC TOE

Mettiamo a frutto la classe che abbiamo scritto per gestire il gioco del TicTacToe.

Per la piccola dimensione dello spazio delle possibili mosse, il TicTacToe è un gioco facile da *risolvere*, cioè è relativamente semplice calcolare tutte le possibili mosse esaminando così tutte le possibili partite.

Come primo approccio, usiamo il metodo che ritorna tutte le possibili mosse per implementare un attraversamento breadth-first:

partendo dalla prima mossa, calcoliamo tutte le mosse successive possibili. Da esse, calcoliamo nuovamente tutte le mosse possibili e così via, scartando le situazioni in cui la partita si conclude.

```
private static List< Game > breadth(List< Game > games) {
  List< Game > res = new ArrayList<>();
  for (Game g : games) res.addAll(g.moves());
  return res;
}
```

```
Game root = Game.setup();
List< Game > first = root.moves();
List< Game > second = breadth(first);
List< Game > third = breadth(second);
List< Game > fourth = breadth(third);
```

A partire dalla quinta mossa, è possibile che un giocatore abbia vinto o che la partita sia finita in parità.

Dobbiamo quindi filtrare le partite terminate da quelle che possono proseguire.

```
class CountStatus {
  public final List< Game > games;
  public final int wonO, wonX, ties;

  CountStatus(List< Game > games, int wonO,
      int wonX, int ties) {
    this.games = games;
    this.wonO = wonO;
    this.wonX = wonX;
    this.ties = ties;
  }
}
```

```
private static CountStatus breadth(CountStatus status) {
  int wO = 0, wX = 0, ts = 0;
  List< Game > res = new ArrayList<>();
  for (Game g : breadth(status.games))
    switch (g.status) {
      case PLAYER_O_WON: wO++; break;
      case PLAYER_X_WON: wX++; break;
      case TIE: ts++; break;
      case ONGOING: res.add(g);
    }
  return new CountStatus(res, wO + status.wonO,
      wX + status.wonX, ts + status.ties);
}
```

```
CountStatus fifth = breadth(new CountStatus(fourth));
CountStatus sixth = breadth(fifth);
CountStatus seventh = breadth(sixth);
CountStatus eighth = breadth(seventh);
CountStatus ninth = breadth(eighth);
```

Una partita è lunga al massimo nove mosse; dopo nove mosse quindi tutte le partite sono necessariamente terminate e possiamo stampare i risultati.

```
System.out.println(
  "Won O: %d Won X: %d Tied: %d (%d)"
    .formatted(ninth.wonO, ninth.wonX, ninth.ties,
      (end - start)));
```

RICERCA CONCORRENTE

Come realizzare una versione della stessa ricerca che sfrutti la concorrenza per ottenere (speriamo) lo stesso risultato in minor tempo?

Sopratutto, come realizzarla in modo da dare alla libreria standard il compito di gestire efficacemente la distribuzione del lavoro?

La risposta è, ovviamente, implementanto l'attraversamento dell'albero come uno `Spliterator`.

Con tale componente possiamo costruire uno Stream e aprofittare della gestione automatica della concorrenza.

```
public class GameIterator implements Spliterator< Game > {

  Queue< Game > current;

  public GameIterator(Game seed) {
    current = new LinkedList< Game >(seed.moves());
  }
```

Teniamo una coda delle posizioni pronte ad essere ritornate. Ogni volta che analizziamo una posizione, accodiamo le mosse legali a partire da quella posizione.

In questo modo, produciamo uno stream di tutte le mosse possibili.

```
@Override
public boolean tryAdvance(Consumer< ? super Game > action) {
  boolean result = false;
  if (current.isEmpty()) {
    result = false;
  } else {
    Game res = current.remove();
    current.addAll(res.moves());
    action.accept(res);
    result = true;
  }
  return result;
}
```

Nel caso ci venga richiesto di separare l'iterator, prendiamo metà delle posizioni correnti e le mettiamo nel nuovo iteratore.

```java
@Override
public Spliterator< Game > trySplit() {
  Spliterator< Game > res = null;
  if (current.size() > 1) {
    List< Game > dest = new ArrayList<>();
    int len = current.size() / 2;
    for (int i = 0; i <= len; i++)
      dest.add(current.remove());
    res=new GameIterator(dest.iterator());
  }
  return res;
}
```

Abbiamo bisogno di un nuovo costruttore per preparare un nuovo iteratore a partire da un insieme di elementi.

```
private GameIterator(Iterator< Game > current) {
  this.current = new LinkedList< Game >();
  while (current.hasNext())
    this.current.offer(current.next());
}
```

Infine, dobbiamo implementare una stima della dimensione (che non possiamo calcolare) e dichiarare le caratteristiche di questo iteratore.

```
@Override
public long estimateSize() {
  return -1;
}

@Override
public int characteristics() {
  return Spliterator.IMMUTABLE | Spliterator.DISTINCT |
    Spliterator.NONNULL;
}
```

Per fare lo stesso calcolo di tutte le mosse possibili, ci serve un contenitore per le somme parziali.

```
class Score {
  int wonX = 0, wonO = 0, ties = 0;

  Score(int wonX, int wonO, int ties) {
    this.wonO = wonO;
    this.wonX = wonX;
    this.ties = ties;
  }

  static Score ONGOING = new Score(0, 0, 0);
  static Score WON_X = new Score(1, 0, 0);
  static Score WON_O = new Score(0, 1, 0);
  static Score TIE = new Score(0, 0, 1);
```

Dotiamo questo contenitore di una operazione associativa, in modo da poterlo usare in una riduzione.

Ci servirà anche un metodo per calcolare il valore corrispondente a ciascuna partita completata.

```
static Score sum(Score a, Score b) {
  return new Score(a.wonX + b.wonX, a.wonO + b.wonO,
    a.ties + b.ties);
}

static Score score(Game game) {
  return switch (game.status()) {
    case PLAYER_O_WON -> Score.WON_O;
    case PLAYER_X_WON -> Score.WON_X;
    case TIE -> Score.TIE;
    case ONGOING -> Score.ONGOING;
  };
}
```

Ora possiamo costruire lo stream e calcolare la somma.

```
Score res = StreamSupport
  .stream(new GameIterator(Game.setup()), true)
  .parallel()
  .filter(g -> g.status() != GameStatus.ONGOING)
  .collect(Collectors.reducing(
    new Score(0, 0, 0),
    Score::score, Score::sum)
  );
```

Possiamo introdurre un "trucco" per sommare i risultati senza passare per un oggetto.

Osserviamo che i risultati non sono mai maggiori di 2^20.

```
long res = StreamSupport
  .stream(new GameIterator(Game.setup()), true)
  .parallel()
  .filter(g -> g.status() != GameStatus.ONGOING)
  .mapToLong( g -> switch (g.status()) {
    case PLAYER_O_WON -> 0x1L;
    case PLAYER_X_WON -> 0x100000L;
    case TIE -> 0x10000000000L;
    case ONGOING -> 0x0L;
  }).sum();
```

```
System.out.println(
  "Won O: %d Won X: %d Ties: %d (%d)"
    .formatted(
      res & 0xfffff,
      (res >> 20) & 0xfffff,
      (res >> 40) & 0xfffff,
      (end - start)));
```

LINK INTERESSANTI

Maurice Naftalin & Dmitry Vyazelenko, Devoxx 2019:

What lies beneath

https://www.youtube.com/watch?v=5AFgNuGwLos