**Taller 7**

Backtracking General Thread

Paralelizar un algoritmo de backtracking utilizando threads en Java puede mejorar significativamente el rendimiento, especialmente en problemas con múltiples soluciones. Aquí hay un ejemplo básico de cómo paralelizar un algoritmo de backtracking usando threads en Java:

Primero, crea una interfaz para representar el problema de backtracking:

|  |
| --- |
| public interface BacktrackingProblem {     boolean isSolution();     void applyMove(int move);     void undoMove(int move);     List<Integer> getPossibleMoves();     Object getSolution(); } |

Crea una clase para ejecutar el backtracking en paralelo:

|  |
| --- |
| import java.util.ArrayList; import java.util.List; import java.util.concurrent.ConcurrentLinkedQueue; import java.util.concurrent.atomic.AtomicBoolean;  public class ParallelBacktracking implements Runnable {     private BacktrackingProblem problem;     private AtomicBoolean foundSolution;     private ConcurrentLinkedQueue<Object> solutions;      public ParallelBacktracking(BacktrackingProblem problem, AtomicBoolean foundSolution, ConcurrentLinkedQueue<Object> solutions) {         this.problem = problem;         this.foundSolution = foundSolution;         this.solutions = solutions;     }      @Override     public void run() {         if (foundSolution.get()) {             return;         }          if (problem.isSolution()) {             solutions.add(problem.getSolution());             foundSolution.set(true);             return;         }          List<Integer> moves = problem.getPossibleMoves();          for (int move : moves) {             if (foundSolution.get()) {                 return;             }              problem.applyMove(move);             ParallelBacktracking child = new ParallelBacktracking(problem, foundSolution, solutions);             child.run();             problem.undoMove(move);         }     } } |

Utiliza la clase ParallelBacktracking en un problema específico de backtracking:

|  |
| --- |
| import java.util.List; import java.util.concurrent.ConcurrentLinkedQueue; import java.util.concurrent.atomic.AtomicBoolean;  public class Main {     public static void main(String[] args) {         BacktrackingProblem problem = new MyBacktrackingProblem(); // Reemplaza esto con la instancia de tu problema de backtracking         AtomicBoolean foundSolution = new AtomicBoolean(false);         ConcurrentLinkedQueue<Object> solutions = new ConcurrentLinkedQueue<>();          List<Integer> initialMoves = problem.getPossibleMoves();         List<Thread> threads = new ArrayList<>();          // Crea un thread para cada movimiento inicial posible         for (int move : initialMoves) {             problem.applyMove(move);             ParallelBacktracking parallelBacktracking = new ParallelBacktracking(problem, foundSolution, solutions);             Thread thread = new Thread(parallelBacktracking);             threads.add(thread);             thread.start();             problem.undoMove(move);         }          // Espera a que todos los threads terminen         for (Thread thread : threads) {             try {                 thread.join();             } catch (InterruptedException e) {                 e.printStackTrace();             }         }          // Imprime las soluciones encontradas         while (!solutions.isEmpty()) {             System.out.println(solutions.poll());         }     } } |

Este ejemplo crea un thread para cada movimiento inicial posible, permitiendo que diferentes ramas del árbol de backtracking se exploren en paralelo. Asegúrate de reemplazar MyBacktrackingProblem con tu implementación específica de BacktrackingProblem.

Ten en cuenta que este enfoque puede crear muchos threads, lo que puede llevar a un rendimiento deficiente si el número de threads excede la cantidad de núcleos de CPU disponibles. Para evitar esto, considera utilizar un ThreadPoolExecutor o un `ForkJoinPool` para limitar el número de threads que se ejecutan en paralelo.

Aquí tienes un ejemplo utilizando ForkJoinPool:

Modifica la clase ParallelBacktracking para que extienda RecursiveAction:

|  |
| --- |
| import java.util.List; import java.util.concurrent.ConcurrentLinkedQueue; import java.util.concurrent.atomic.AtomicBoolean; import java.util.concurrent.RecursiveAction;  public class ParallelBacktracking extends RecursiveAction {     // ... (El resto del código de ParallelBacktracking permanece igual) } |

Actualiza el método run de la clase ParallelBacktracking para llamar a compute() en lugar de run():

|  |
| --- |
| // ...      @Override     public void run() {         compute();     }      @Override     protected void compute() {         if (foundSolution.get()) {             return;         }          if (problem.isSolution()) {             solutions.add(problem.getSolution());             foundSolution.set(true);             return;         }          List<Integer> moves = problem.getPossibleMoves();         List<ParallelBacktracking> tasks = new ArrayList<>();          for (int move : moves) {             if (foundSolution.get()) {                 return;             }              problem.applyMove(move);             ParallelBacktracking child = new ParallelBacktracking(problem, foundSolution, solutions);             tasks.add(child);             problem.undoMove(move);         }          invokeAll(tasks);     }  // ... |

En la clase Main, utiliza ForkJoinPool en lugar de crear y unir threads manualmente:

|  |
| --- |
| import java.util.List; import java.util.concurrent.ConcurrentLinkedQueue; import java.util.concurrent.ForkJoinPool; import java.util.concurrent.atomic.AtomicBoolean;  public class Main {     public static void main(String[] args) {         BacktrackingProblem problem = new MyBacktrackingProblem(); // Reemplaza esto con la instancia de tu problema de backtracking         AtomicBoolean foundSolution = new AtomicBoolean(false);         ConcurrentLinkedQueue<Object> solutions = new ConcurrentLinkedQueue<>();          List<Integer> initialMoves = problem.getPossibleMoves();         ForkJoinPool forkJoinPool = new ForkJoinPool();          List<ParallelBacktracking> tasks = new ArrayList<>();          // Crea un task para cada movimiento inicial posible         for (int move : initialMoves) {             problem.applyMove(move);             ParallelBacktracking parallelBacktracking = new ParallelBacktracking(problem, foundSolution, solutions);             tasks.add(parallelBacktracking);             problem.undoMove(move);         }          // Ejecuta los tasks en paralelo usando ForkJoinPool         forkJoinPool.invokeAll(tasks);          // Imprime las soluciones encontradas         while (!solutions.isEmpty()) {             System.out.println(solutions.poll());         }     } } |

De esta manera, el ForkJoinPool manejará automáticamente la creación, ejecución y unión de threads, permitiendo un mejor control sobre la paralelización y la utilización de recursos del sistema.

Backtracking General Thread – Camino mas corto

Para paralelizar un algoritmo de backtracking de camino más corto utilizando threads en Java con un enfoque de divide y vencerás, puedes emplear ForkJoinPool y RecursiveTask. Aquí hay un ejemplo básico de cómo hacerlo:

Crea una interfaz para representar el problema del camino más corto:

|  |
| --- |
| import java.util.List;  public interface ShortestPathProblem {     boolean isSolution();     void applyMove(int move);     void undoMove(int move);     List<Integer> getPossibleMoves();     List<Integer> getCurrentPath();     int getCurrentPathLength(); } |

Crea una clase para ejecutar el backtracking en paralelo utilizando divide y vencerás:

|  |
| --- |
| import java.util.List; import java.util.concurrent.RecursiveTask;  public class ParallelShortestPath extends RecursiveTask<List<Integer>> {     private ShortestPathProblem problem;     private int bestPathLength;      public ParallelShortestPath(ShortestPathProblem problem, int bestPathLength) {         this.problem = problem;         this.bestPathLength = bestPathLength;     }      @Override     protected List<Integer> compute() {         if (problem.isSolution()) {             return problem.getCurrentPath();         }          List<Integer> shortestPath = null;         List<Integer> moves = problem.getPossibleMoves();         List<ParallelShortestPath> tasks = new ArrayList<>();          for (int move : moves) {             problem.applyMove(move);              if (problem.getCurrentPathLength() < bestPathLength) {                 ParallelShortestPath child = new ParallelShortestPath(problem, bestPathLength);                 tasks.add(child);             }              problem.undoMove(move);         }          for (ParallelShortestPath task : tasks) {             task.fork();         }          for (ParallelShortestPath task : tasks) {             List<Integer> path = task.join();             if (path != null && (shortestPath == null || path.size() < shortestPath.size())) {                 shortestPath = path;                 bestPathLength = path.size();             }         }          return shortestPath;     } } |

Utiliza la clase ParallelShortestPath en un problema específico de camino más corto:

|  |
| --- |
| import java.util.List; import java.util.concurrent.ForkJoinPool;  public class Main {     public static void main(String[] args) {         ShortestPathProblem problem = new MyShortestPathProblem(); // Reemplaza esto con la instancia de tu problema de camino más corto         int initialBestPathLength = Integer.MAX\_VALUE;          ForkJoinPool forkJoinPool = new ForkJoinPool();          ParallelShortestPath parallelShortestPath = new ParallelShortestPath(problem, initialBestPathLength);         List<Integer> shortestPath = forkJoinPool.invoke(parallelShortestPath);          // Imprime el camino más corto encontrado         if (shortestPath != null) {             System.out.println("Camino más corto: " + shortestPath);             System.out.println("Longitud del camino: " + shortestPath.size());         } else {             System.out.println("No se encontró ningún camino.");         }     } } |

Este ejemplo crea un ForkJoinPool y utiliza RecursiveTask para explorar diferentes ramas del árbol de backtracking en paralelo, siguiendo un enfoque de divide y vencerás. Los subproblemas se dividen en tareas más pequeñas y se combinan al final para encontrar el camino más corto.

Asegúrate de reemplazar `MyShortestPathProblem` con tu implementación específica de ShortestPathProblem. Ten en cuenta que este enfoque puede no ser óptimo para todos los problemas del camino más corto, ya que la paralelización puede depender de la naturaleza específica del problema. Además, para problemas más grandes, es posible que desees ajustar el número de threads utilizados en función del rendimiento y los recursos del sistema.

Este ejemplo utiliza el enfoque de divide y vencerás, dividiendo el problema en subproblemas más pequeños y asignándolos a diferentes tareas para ser ejecutadas en paralelo. El ForkJoinPool se encarga de administrar la ejecución de estas tareas, fusionándolas de nuevo para obtener el resultado final.

Al utilizar este enfoque, es posible que puedas aprovechar al máximo los recursos disponibles en tu sistema y, en última instancia, mejorar el rendimiento de tu algoritmo de backtracking para encontrar el camino más corto. Sin embargo, ten en cuenta que la eficacia de este enfoque puede variar según la naturaleza específica del problema y la arquitectura del sistema. Por lo tanto, es importante realizar pruebas de rendimiento y ajustar el enfoque según sea necesario para obtener los mejores resultados posibles.

**Solicitud**

Demostrar la paralelización del backtracking de camino más corto indicando la mejora del rendimiento utilizando Thread de Java