به نام خدا

گزارش نهایی پروژه سیستم‌های بی‌درنگ

پروژه شماره ۱۸ – گروه ۳۷

سینا بیرامی ۴۰۰۱۰۵۴۳۳

علی هاشمیان ۴۰۱۱۰۶۶۸۵

مقدمه

با گسترش روزافزون محاسبات لبه و نیاز فزاینده به پردازش داده‌ها در نزدیکی منبع تولید، چالش‌های جدیدی در زمینه مدیریت منابع و اجرای وظایف مطرح شده است. یکی از این چالش‌های کلیدی، زمان‌بندی کارآمد وظایف پیچیده و به هم وابسته در محیط‌های لبه با منابع محدود و ناهمگن می‌باشد. این وظایف، که اغلب به صورت گراف وظیفه (DAG) مدل‌سازی می‌شوند، نیازمند راهکارهایی هستند که نه تنها محدودیت‌های زمانی اجرای آن‌ها را رعایت کنند، بلکه بتوانند معیارهای حیاتی دیگری نظیر مصرف انرژی و کیفیت خدمات (QoS) را نیز بهینه سازند.

پروژه حاضر با هدف پاسخگویی به این نیاز، به طراحی و پیاده‌سازی یک سامانه زمان‌بندی بی‌درنگ برای گراف‌های وظیفه (DAGs) در محیط لبه می‌پردازد. هدف اصلی، توسعه راهکاری است که ضمن رعایت محدودیت‌های زمانی، بتواند به طور همزمان مصرف انرژی را بهینه کرده و کیفیت خدمات (QoS) ارائه شده به کاربران را ارتقا بخشد. برای دستیابی به این هدف، از شبیه‌ساز CloudSimPlus به منظور مدل‌سازی دقیق محیط لبه و اجزای آن بهره گرفته شده است.

وظایف در این سامانه به صورت گراف‌های جهت‌دار بدون دور (DAG) مدل می‌شوند که نمایانگر وابستگی‌های اجرایی میان آن‌هاست. در این پروژه، ابتدا الگوریتم پایه CPOP (Critical Path On Processor) به عنوان نقطه شروع و خط مبنا پیاده‌سازی شد. سپس، یک الگوریتم زمان‌بندی نوین مبتنی بر بهینه‌سازی ازدحام ذرات (PSO) طراحی و پیاده‌سازی گردید که به طور خاص برای بهینه‌سازی دوگانه مصرف انرژی و کیفیت خدمات هدف‌گذاری شده است.

ارزیابی نهایی عملکرد الگوریتم پیشنهادی (PSO) از طریق مقایسه با الگوریتم Random-CPOP (که در آن اولویت‌بندی با CPOP و تخصیص ماشین به صورت تصادفی است) و بر اساس معیارهایی چون مصرف انرژی، کیفیت خدمات ، Miss Ratio و Makespan، برای سناریوهای مختلف با تعداد دستگاه‌های لبه و اندازه‌های متفاوت گراف وظیفه، صورت پذیرفته است. این گزارش به تشریح کامل مراحل مدل‌سازی، پیاده‌سازی الگوریتم‌ها، و تحلیل نتایج به دست آمده می‌پردازد.

راه اندازی محیط

برای آغاز فرآیند توسعه و پیاده‌سازی، پیش‌نیازهای اولیه شامل نصب و پیکربندی صحیح محیط توسعه جاوا (JDK) و ابزار مدیریت پروژه Maven بر روی سیستم توسعه‌دهنده مد نظر قرار گرفت.

نخستین گام عملی، ایجاد یک پروژه جدید Maven با استفاده از دستور استاندارد archetype:generate بود. این دستور ساختار اولیه پروژه را به همراه فایل pom.xml برای مدیریت وابستگی‌ها ایجاد می‌کند. دستور دقیق استفاده شده به شرح زیر است:

```
mvn archetype:generate -DgroupId=com.mycompany.app -DartifactId=my-cloudsim-app
DarchetypeArtifactId=maven-archetype-quickstart -DarchetypeVersion=1.4 -
DinteractiveMode=false
```

پس از ایجاد ساختار پروژه، وابستگی اصلی این پروژه، یعنی کتابخانهCloudSimPlus ، به فایل pom.xmlاضافه گردید. این کتابخانه ابزارها و کلاس‌های لازم برای شبیه‌سازی محیط‌های ابری و لبه را فراهم می‌آورد و هسته اصلی شبیه‌سازی‌های این پروژه را تشکیل می‌دهد. قطعه کد زیر به بخش <dependencies> فایل pom.xml افزوده شد:

```
<dependency>
  <groupId>org.cloudsimplus</groupId>
  <artifactId>cloudsim-plus</artifactId>
  <version>6.4.0</version>
</dependency>
```

بدین ترتیب، با تعریف این وابستگی، Maven به طور خودکار کتابخانه CloudSimPlus و نیازمندی‌های آن را در زمان ساخت پروژه دانلود و مدیریت می‌کند.

در نهایت، برای کامپایل، بسته‌بندی و اجرای پروژه از دستورات استاندارد Maven استفاده شد. این دستورات به ترتیب عبارتند از:

1) کامپایل پروژه:

```
mvn compile
```

این دستور کد منبع جاوا را به بایت‌کد کامپایل می‌کند.

2) بسته‌بندی پروژه:

```
mvn package
```

این دستور پروژه کامپایل شده را به یک فایل JAR (یا فرمت دیگری بسته به پیکربندی) بسته‌بندی می‌کند.

3) اجرای پروژه:

```
mvn exec:java -Dexec.mainClass="com.mycompany.app.App"
```

این دستور برنامه را با تعیین کلاس اصلی (در اینجا com.mycompany.app.App) اجرا می‌کند.

با انجام این مراحل، محیط توسعه و اجرای پروژه برای پیاده‌سازی‌های بعدی آماده گردید.

برای اجرای بخش اصلی پروژه که شامل اجرای همزمان 2 الگوریتم و مشاهده نمودارها هست، لازم است کلاس FinalComparisonRunner اجرا شود.

مدل‌سازی محیط Edge در CloudSimPlus

برای دستیابی به اهداف پروژه، اولین گام اساسی، ایجاد مدلی قابل فهم و کارآمد از محیط محاسبات لبه در بستر شبیه‌ساز CloudSimPlus بود. این مدل‌سازی شامل تعریف دقیقی از اجزای زیرساختی و نحوه تعامل آن‌ها، و همچنین چگونگی نمایش وظایف و ساختار وابستگی میان آن‌هاست.

همانطور که خواسته شده بود، شبیه‌سازی‌ها برای 15 حالت مختلف، شامل تعداد لبه 10 یا 20 یا 30، همچنین تعداد وظایف 100، 200، 300، 400 یا 500 هست.

پیاده‌سازی این اجزا عمدتاً در کلاس SimulationUtils متمرکز شده است. اجزای اصلی این مدل عبارتند از:

شبیه‌سازی زیرساخت لبه

در این شبیه‌سازی، محیط لبه به صورت مجموعه‌ای از منابع محاسباتی در نظر گرفته شده است که از طریق CloudSimPlus مدل شده‌اند. اجزای اصلی این مدل عبارتند از:

مراکز داده (Datacenters): متد createSimpleDatacenter در کلاس SimulationUtils مسئول ایجاد و پیکربندی DatacenterSimple است. این متد، میزبان‌های فیزیکی را نیز درون این مرکز داده ایجاد و مدیریت می‌کند.

میزبان‌ها (Hosts): در همان متد createSimpleDatacenter، نمونه‌هایی از HostSimple با مشخصات تعریف‌شده (شامل تعداد واحدهای پردازشی، حافظه و...) ایجاد و به لیست میزبان‌های مرکز داده اضافه می‌شوند.

ماشین‌های مجازی (VMs): برای ایجاد لیستی از ماشین‌های مجازی (VmSimple) با خصوصیات یکسان، متد createVmList در کلاس SimulationUtils پیاده‌سازی شده است. تعداد این ماشین‌ها به عنوان ورودی به این متد داده می‌شود.

کارگزار (Broker): در کلاس SingleSimulationRunner و در متد runSimulation، یک نمونه از DatacenterBrokerSimple ایجاد می‌شود تا وظیفه ارسال ماشین‌های مجازی و Cloudletها به مرکز داده را بر عهده بگیرد.

در شبیه‌سازی ما هر VM نماینده یک edge است.

مدل‌سازی وظایف و گراف وابستگی (DAG)

نحوه تعریف و نمایش وظایف محاسباتی و ارتباطات میان آن‌ها، بخش مهم دیگری از مدل‌سازی است:

گره‌های وظیفه (Task Nodes): ساختار اصلی هر وظیفه، شامل شناسه، Cloudlet مرتبط، مهلت زمانی و لیست والدین و فرزندان، در کلاس داخلی TaskNode واقع در فایل App.java تعریف شده است. هر TaskNode یک نمونه از CloudletSimple را نیز در خود جای می‌دهد که نشان‌دهنده بار محاسباتی است.

تولید گراف وظایف (DAG Generation): منطق تولید گراف‌های وظیفه تصادفی، شامل ایجاد وظایف با طول‌های محاسباتی مختلف و برقراری وابستگی‌ها بین آن‌ها به صورت یک گراف جهت‌دار بدون دور، در کلاس TaskDagGenerator و به طور خاص در متد generateRandomDAG پیاده‌سازی شده است. همچنین، متد deepCopyDag در همین کلاس برای ایجاد کپی‌های مستقل از گراف‌ها برای شبیه‌سازی‌های دسته‌ای استفاده می‌شود.

این مدل‌سازی، با فراهم آوردن یک محیط کنترل‌شده و قابل تکرار، امکان پیاده‌سازی، آزمایش و مقایسه الگوریتم‌های زمان‌بندی مختلف را در شرایط گوناگون محیط لبه میسر می‌سازد.

پیاده‌سازی ساخت DAG

در ساخت DAG تعداد یال ها عددی بین n/2 و n(n-1)/2 انتخاب میشود.

ساخت هر یال هم به صورت رندوم با تنها قید کوچکتر بودن شماره والد انجام میشود تا دور به وجود نیاید.

```
3 usages
public static List<App.TaskNode> generateRandomDAG(int numTasks) {
    List<App.TaskNode> dag = new ArrayList<>();
    Random rand = new Random();

    for (int i = 0; i < numTasks; i++) {
        long length = 8000 + (i * 10);
        Cloudlet cl = new CloudletSimple(length,  pesNumber: 2);
        cl.setSizes(1024);
        App.TaskNode task = new App.TaskNode(i, cl);
        dag.add(task);
    }

    int minEdges = numTasks / 2;
    int maxEdges = numTasks * (numTasks - 1) / 2;
    int edgeCount = rand.nextInt( bound: maxEdges - minEdges + 1) + minEdges;

    Set<String> addedEdges = new HashSet<>();
    int added = 0;

    while (added < edgeCount) {
        int parent = rand.nextInt( bound: numTasks - 1);
        int child = rand.nextInt( bound: numTasks - parent - 1) + parent + 1;

        String edgeKey = parent + "," + child;
        if (addedEdges.contains(edgeKey)) continue;
```

```
        App.TaskNode parentNode = dag.get(parent);
        App.TaskNode childNode = dag.get(child);

        childNode.addDependency(parentNode);
        addedEdges.add(edgeKey);
        added++;
    }

    assignRealisticDeadlines(dag);

    return dag;
}
```

پیاده‌سازی الگوریتم CPOP برای زمان‌بندی گراف وظیفه

پس از برپایی و مدل‌سازی محیط شبیه‌سازی، گام بعدی در فاز نخست پروژه، پیاده‌سازی یک الگوریتم زمان‌بندی پایه بود تا به عنوان معیاری برای سنجش راهکارهای آتی عمل کند. برای این منظور، الگوریتم شناخته‌شده CPOP (Critical Path On Processor) انتخاب گردید. این الگوریتم، با تمرکز بر مسیر بحرانی گراف وظایف، تلاش می‌کند تا زمان کلی اتمام کارها (Makespan) را به حداقل برساند. در پیاده‌سازی حاضر، یک نسخه از این الگوریتم که در آن انتخاب ماشین مجازی (پردازنده) به صورت تصادفی صورت می‌گیرد، توسعه داده شده است. این رویکرد، که "Random-CPOP" نامیده می‌شود، امکان بررسی تاثیر اولویت‌بندی CPOP را به تنهایی فراهم می‌آورد.

فرآیند پیاده‌سازی این الگوریتم در پروژه، مجموعه‌ای از مراحل کلیدی را در بر می‌گیرد که عمدتاً در کلاس SingleSimulationRunner و با بهره‌گیری از توابع کمکی در SimulationUtils و App.TaskNode وجود دارد:

بخش اصلی الگوریتم CPOP، مفهوم رتبه (Rank) است. برای هر وظیفه در گراف، دو نوع رتبه محاسبه می‌شود تا اهمیت و جایگاه آن در توالی کلی اجرای وظایف مشخص گردد:

رتبه بالایی (Upward Rank - rankU): این مقدار، تخمینی از طولانی‌ترین مسیر از یک وظیفه مشخص تا انتهای گراف (یک وظیفه خروجی) ارائه می‌دهد. محاسبه rankU به صورت بازگشتی و با در نظر گرفتن هزینه اجرای خود وظیفه و بیشترین rankU در میان فرزندانش صورت می‌گیرد. این منطق در متد computeRankU از کلاس SimulationUtils پیاده‌سازی شده است. وظایفی که rankU بالاتری دارند، به نوعی در گلوگاه‌های انتهایی مسیرهای طولانی قرار گرفته‌اند.

رتبه پایینی (Downward Rank - rankD): به طور مشابه، rankD طولانی‌ترین مسیر از ابتدای گراف (یک وظیفه ورودی) تا وظیفه مورد نظر را (با احتساب هزینه اجرای خود وظیفه و بیشینه rankD والدین) منعکس می‌کند. این محاسبه نیز به صورت بازگشتی در متد computeRankD

از کلاس SimulationUtils انجام می‌شود. مقدار بالای rankD نشان‌دهنده این است که وظیفه در ادامه یک مسیر طولانی اولیه قرار دارد.

```
2 usages
public static double computeRankU(App.TaskNode node) {
    if (node.rankU >= 0) return node.rankU;
    if (node.children.isEmpty()) return node.rankU = node.cloudlet.getLength();
    double max = 0;
    for (App.TaskNode child : node.children) {
        max = Math.max(max, computeRankU(child));
    }
    return node.rankU = node.cloudlet.getLength() + max;
}

2 usages
public static double computeRankD(App.TaskNode node) {
    if (node.rankD >= 0) return node.rankD;
    if (node.parents.isEmpty()) return node.rankD = node.cloudlet.getLength();
    double max = 0;
    for (App.TaskNode parent : node.parents) {
        max = Math.max(max, computeRankD(parent));
    }
    return node.rankD = node.cloudlet.getLength() + max;
}
```

محاسبه دقیق این رتبه‌ها برای تمام گره‌های گراف، پیش از آغاز فرآیند زمان‌بندی و در ابتدای متد runSimulation از کلاس SingleSimulationRunner انجام می‌شود.

```
dag.forEach(SimulationUtils::computeRankU);
dag.forEach(SimulationUtils::computeRankD);
```

پس از آنکه "رتبه‌های بالایی و پایینی" برای هر وظیفه در گراف محاسبه شد، گام بعدی تعیین اولویت هر وظیفه برای زمان‌بندی است. در الگوریتم CPOP، این اولویت با ترکیب همین دو رتبه (بالایی و پایینی) برای هر وظیفه به دست می‌آید. به بیان ساده‌تر، وظیفه‌ای که هم در بخش آغازین یک مسیر طولانی قرار دارد (رتبه پایینی بالا) و هم در بخش پایانی یک مسیر طولانی دیگر واقع شده (رتبه بالایی بالا)، از اهمیت ویژه‌ای برخوردار می‌شود.

این محاسبه‌ی اولویت در متد getCPOPScore که درون کلاس App.TaskNode تعریف شده، انجام می‌پذیرد. نتیجه‌ی این متد، یک امتیاز عددی است که به عنوان شاخص اصلی برای

مرتب‌سازی و انتخاب وظایف در فرآیند زمان‌بندی به کار می‌رود. وظایفی که امتیاز CPOP بالاتری کسب می‌کنند، به عنوان کاندیداهای اصلی برای قرارگیری روی "مسیر بحرانی" گراف شناخته می‌شوند و در نتیجه، در صف زمان‌بندی زودتر از سایرین مورد توجه قرار خواهند گرفت. این رویکرد به الگوریتم کمک می‌کند تا بر روی بخش‌هایی از گراف که بیشترین تأثیر را بر زمان کلی اجرا دارند، تمرکز نماید.

```
1 usage
public double getCPOPScore() { return rankU + rankD; }
```

فرآیند اصلی زمان‌بندی در یک حلقه تکرارشونده در متد runSimulation پیاده‌سازی شده است. این حلقه تا زمانی که تمام وظایف گراف، زمان‌بندی و برای اجرا ارسال شوند، ادامه می‌یابد:

1. غربالگری وظایف آماده: در هر گام، سیستم ابتدا وظایفی را شناسایی می‌کند که تمام پیش‌نیازهای آن‌ها (وظایف والد) به اتمام رسیده و آماده اجرا هستند. این بررسی از طریق متد isReady در کلاس App.TaskNode صورت می‌پذیرد.

```
while (scheduled.size() < dag.size()) {
    for (App.TaskNode node : dag.stream()
            .filter(n -> !scheduled.contains(n))
            .filter(n -> n.isReady(scheduled))
```

```
1 usage
public boolean isReady(Set<TaskNode> done) { return done.containsAll(parents); }
```

2. اولویت‌بندی و انتخاب: لیست وظایف آماده سپس بر اساس امتیاز CPOP (به صورت نزولی) مرتب می‌شود. این اطمینان می‌دهد که وظایف با بالاترین اولویت ابتدا برای تخصیص در نظر گرفته می‌شوند.

```
Comparator.comparingDouble(App.TaskNode::getCPOPScore).reversed()
```

3. تخصیص (تصادفی) به ماشین مجازی: در این مرحله، برای هر وظیفه از لیست مرتب‌شده، یک ماشین مجازی به صورت تصادفی از میان ماشین‌های مجازی موجود انتخاب می‌شود. این بخش، وجه تمایز پیاده‌سازی "Random-CPOP" است. پس از انتخاب، وظیفه (در

قالب Cloudlet) به ماشین مجازی منتخب تخصیص داده شده و از طریق کارگزار برای اجرا به شبیه‌ساز ارسال می‌گردد.

```
} else {
    selectedVm = vmList.get(rand.nextInt(vmList.size()));
}
```

پس از اینکه تمام وظایف در این چرخه پردازش و ارسال شدند، شبیه‌سازی CloudSimPlus با فراخوانی متد simulation.start() آغاز می‌شود و تا زمان تکمیل همه Cloudletها ادامه می‌یابد. نتایج به‌دست‌آمده از این شبیه‌سازی پایه، شامل معیارهایی نظیر Makespan، مصرف انرژی و کیفیت خدمات، در بخش‌های آتی گزارش مورد تحلیل قرار خواهند گرفت و به عنوان خط مبنایی برای ارزیابی الگوریتم‌های پیشرفته‌تر عمل خواهند کرد.

پیاده‌سازی الگوریتم بهینه‌سازی مبتنی بر ازدحام ذرات (PSO)

برای دستیابی به اهداف چندگانه پروژه (کاهش انرژی، افزایش QoS و کاهش makespan)، یک الگوریتم زمان‌بندی مبتنی بر بهینه‌سازی ازدحام ذرات (PSO) طراحی و در کلاس PSOScheduler پیاده‌سازی شد.

مبانی نظری PSO

PSO یک الگوریتم بهینه‌سازی فرابتکاری است که از رفتار اجتماعی دسته‌ای از پرندگان یا ماهی‌ها الهام گرفته شده است. در این الگوریتم، مجموعه‌ای از ذرات (Particles) در فضای جستجو حرکت می‌کنند. هر ذره نمایانگر یک راه‌حل بالقوه برای مسئله است. حرکت هر ذره تحت تأثیر سه عامل قرار دارد: بهترین موقعیت شخصی تجربه شده توسط خود ذره (pBest)، بهترین موقعیت کلی یافت شده توسط کل ازدحام (gBest)، و اینرسی یا سرعت فعلی ذره.

```
6 usages
private static class Particle {
    6 usages
    int[] vmMapping;
    5 usages
    Map<Integer, List<Integer>> vmOrderMap;
    4 usages
    double[][] velocity;
    3 usages
    int[] pBestMapping;
    2 usages
    Map<Integer, List<Integer>> pBestOrderMap;
    3 usages
    double pBestFit;
```

پارامترها:

```
1 usage
private final int swarmSize = 200;
1 usage
private final int maxIter = 100;
1 usage
private final double inertia = 0.9;
1 usage
private final double c1 = 1.4;
1 usage
private final double c2 = 1.1;
```

مدل‌سازی مسئله برای PSO

هر ذره در الگوریتم PSO، یک نگاشت کامل از وظایف به VMها را نشان می‌دهد. در پیاده‌سازی فعلی، این نگاشت شامل دو بخش است:

vmMapping[]: آرایه‌ای که مشخص می‌کند هر وظیفه به کدام ماشین مجازی (VM) اختصاص داده شده است.

به‌عنوان مثال vmMapping[3] = 1 یعنی وظیفه 3 باید روی VM شماره 1 اجرا شود.

vmOrderMap: ساختاری که در آن برای هر VM، لیستی از شناسه‌های وظایف مرتب‌شده‌ای که روی آن اجرا می‌شوند تعریف شده است. این ترتیب، مفهوم execution order درون هر VM را مشخص می‌کند.

```
2 usages
private Map<Integer, List<Integer>> generateVmOrderMap(int[] mapping) {
    Map<Integer, List<Integer>> map = new HashMap<>();
    for (int vm = 0; vm < vmCount; vm++) map.put(vm, new ArrayList<>());
    Random rnd = new Random();
    List<Integer> taskIds = new ArrayList<>();
    for (int i = 0; i < taskCount; i++) taskIds.add(i);
    Collections.shuffle(taskIds);
    for (int taskId : taskIds) {
        int vm = mapping[taskId];
        map.get(vm).add(taskId);
    }
    return map;
}
```

تابع برازش (Fitness Function)

مهم‌ترین بخش الگوریتم PSO، تابع برازش (Fitness) است که کیفیت هر راه‌حل (هر ذره) را ارزیابی می‌کند. تابع fitness یکی از هسته‌های اصلی الگوریتم است که کیفیت هر ذره را بر اساس نگاشت اختصاص داده شده ارزیابی می‌کند. دقت شود از آنجایی که امکان شبیه‌سازی واقعی نبود و این کار بسیار زمانبر می‌شد، از روش تخمینی استفاده کردیم.

مراحل محاسبه Fitness به شرح زیر هستند:

محاسبه حجم کاری کلی روی هر VM با توجه به vmOrderMap

با استفاده از سرعت پردازشی VMها، تخمینی از Makespan و Energy تولید می‌شود

هر وظیفه، بر اساس deadlineاش، مورد ارزیابی قرار می‌گیرد و از این طریق QoS وظیفه محاسبه می‌شود:

$$QoS = 1 - \max\left(0, \frac{\text{FinishTime} - \text{Deadline}}{\text{ExecutionTime}}\right)$$

در نهایت مقدار نهایی Fitness محاسبه می‌شود بر اساس ترکیبی وزنی:

$$\text{Fitness} = 0.5 \times \left(\frac{\text{Makespan}}{10} \right) - 0.3 \times \text{QoS}$$

البته توجه شود که در فرمول بالا در واقع انرژی هم تاثیر می‌گذارد چون در مساله‌ی ما، در واقع ضریبی از Makespan در نظر گرفته می‌شود.

```
private double fitness(int[] mapping, Map<Integer, List<Integer>> orderMap) {
    double[] vmWork = new double[vmCount];

    for (int vm = 0; vm < vmCount; vm++) {
        for (int taskId : orderMap.get(vm)) {
            long len = originalDag.get(taskId).cloudlet.getLength();
            vmWork[vm] += len;
        }
    }

    double cap = 2000.0;
    double makespan = 0;
    // double energy = 0;
    for (int vm = 0; vm < vmCount; vm++) {
        double exec = vmWork[vm] / cap;
        makespan = Math.max(makespan, exec);
```

```
    double qos = 0;
    for (int i = 0; i < taskCount; i++) {
        double ft = vmWork[mapping[i]] / cap;
        double d = originalDag.get(i).deadline;
        double l = originalDag.get(i).cloudlet.getLength() / cap;
        qos += 1 - Math.max(0, (ft - d) / l);
    }

    qos = Math.max(0, qos / taskCount);
    return 0.5 * (makespan / 10.0) - 0.3 * qos;
}
```

فرآیند بهینه‌سازی

فرآیند کلی که در متد schedule کلاس PSOScheduler پیاده‌سازی شده، به شرح زیر است:

1. مقداردهی اولیه ذرات (initSwarm):

- تولید تصادفی mapping وظایف به VMها

- ساخت vmOrderMap با لیست‌های ترتیب وظایف در هر VM

- مقداردهی اولیه سرعت‌ها (velocity) برای ذرات

```
1 usage
private List<Particle> initSwarm() {
    List<Particle> swarm = new ArrayList<>();
    Random rnd = new Random();

    for (int i = 0; i < swarmSize; i++) {
        int[] mapping = new int[taskCount];
        double[][] velocity = new double[taskCount][vmCount];
        for (int t = 0; t < taskCount; t++) {
            mapping[t] = rnd.nextInt(vmCount);
            for (int v = 0; v < vmCount; v++) {
                velocity[t][v] = rnd.nextDouble( origin: -1,  bound: 1);
            }
        }

        var orderMap :Map<Integer, List<Integer>> = generateVmOrderMap(mapping);
        double fit = fitness(mapping, orderMap);

        swarm.add(new Particle(mapping, orderMap, velocity, fit));
    }
    return swarm;
}
```

2. حلقه تکرار بهینه‌سازی (برای maxIter بار):

- از طریق تابع fitness کیفیت هر ذره سنجیده می‌شود

- به‌روزرسانی pBest و gBest

- به‌روزرسانی سرعت‌ها بر اساس قواعد PSO استاندارد:

$$v_{i,j}^{\text{new}} = w \cdot v_{i,j}^{\text{old}} + c_1 \cdot r_1 \cdot (\text{pBest}_i - x_{i,j}) + c_2 \cdot r_2 \cdot (\text{gBest} - x_{i,j})$$

- تصمیم‌گیری مجدد VM برای هر task با Softmax Sampling

- به‌روزرسانی random ترتیب اجرا درون هر VM

```
for (int iter = 0; iter < maxIter; iter++) {
    for (Particle p : swarm) {
        double fit = fitness(p.vmMapping, p.vmOrderMap);
        if (fit < p.pBestFit) {
            p.pBestFit = fit;
            p.pBestMapping = p.vmMapping.clone();
            p.pBestOrderMap = p.deepCopyMap(p.vmOrderMap);
        }
        if (fit < gBestFitness) {
            gBestFitness = fit;
            gBestMapping = p.vmMapping.clone();
            gBestOrderMap = p.deepCopyMap(p.vmOrderMap);
        }
    }

    fitnessHistory.add(gBestFitness);
```

```
fitnessHistory.add(gBestFitness);

for (Particle p : swarm) {
    for (int t = 0; t < taskCount; t++) {
        for (int v = 0; v < vmCount; v++) {
            double r1 = Math.random(), r2 = Math.random();
            double cog = c1 * r1 * ((p.pBestMapping[t] == v) ? 1 : 0);
            double soc = c2 * r2 * ((gBestMapping != null && gBestMapping[t] == v) ? 1 : 0);
            p.velocity[t][v] = inertia * p.velocity[t][v] + cog + soc;
        }

        double[] expVals = Arrays.stream(p.velocity[t]).map(Math::exp).toArray();
        double sum = Arrays.stream(expVals).sum();
        double rnd = Math.random(), cum = 0;
        for (int v = 0; v < vmCount; v++) {
            cum += expVals[v] / sum;
            if (rnd <= cum) {
                p.vmMapping[t] = v;
                break;
            }
        }
    }

    p.vmOrderMap = generateVmOrderMap(p.vmMapping);
}
```

3. اجرای واقعی هر ۲۰ تکرار (برای ارزیابی تعریف‌شده):

SingleSimulationRunner.runSimulation اجرا می‌شود تا رفتار واقعی ذره ارزیابی شود.

```
if (iter % 20 == 0 && gBestMapping != null) {
    List<App.TaskNode> dagWithPos = applyToDag(gBestMapping, gBestOrderMap);
    var res :SimulationResult = SingleSimulationRunner.runSimulation(dagWithPos, vmCount);
    System.out.printf("Iter %d → Makespan=%.2f | QoS=%.2f%%\n", iter, res.makespan(), res.qosScore() * 100);
}
```

4. پس از اتمام تکرار، بهترین نگاشت (gBest) روی DAG اصلی اعمال می‌شود و آماده اجرا است.

```
return applyToDag(gBestMapping, gBestOrderMap);
```

اعمال خروجی PSO روی DAG

تابع applyToDag وظیفه دارد نگاشت نهایی PSO را روی وظایف DAG اعمال کند.

- ثبت preferredVm برای هر وظیفه

- برای هر وظیفه‌ای که در vmOrderMap اومده، مقدار executionOrder درون آن VM نوشته می‌شود.

```
2 usages
private List<App.TaskNode> applyToDag(int[] mapping, Map<Integer, List<Integer>> orderMap) {
    List<App.TaskNode> copy = TaskDagGenerator.deepCopyDag(originalDag);
    for (int i = 0; i < taskCount; i++) {
        copy.get(i).setPreferredVm(mapping[i]);
    }

    for (Map.Entry<Integer, List<Integer>> entry : orderMap.entrySet())
        for (int order = 0; order < entry.getValue().size(); order++) {
            int taskId = entry.getValue().get(order);
            copy.get(taskId).setExecutionOrder(order);
        }

    return copy;
}
```

سپس، این DAG به تابع SingleSimulationRunner.runSimulation فرستاده می‌شود تا در محیط واقعی CloudSim اجرا و ارزیابی شود.

طراحی شبیه‌سازی و ارزیابی

برای مقایسه عملکرد دو الگوریتم، یک چارچوب ارزیابی جامع در کلاس FinalComparison طراحی شد.

سناریوهای شبیه‌سازی

شبیه‌سازی‌ها برای تمام ترکیبات پارامترهای زیر انجام شدند:

تعداد دستگاه‌های لبه (VM): ۱۰، ۲۰، ۳۰

تعداد وظایف در گراف: ۱۰۰، ۲۰۰، ۳۰۰، ۴۰۰، ۵۰۰

برای هر ترکیب (مثلاً ۲۰۰ وظیفه و ۱۰ دستگاه لبه)، یک گراف وظیفه پایه تولید می‌شود. سپس دو کپی از این گراف ایجاد می‌گردد تا هر دو الگوریتم روی ساختار وابستگی یکسان آزمایش شوند و مقایسه عادلانه باشد.

```
public static void main(String[] args) {
    List<Integer> taskSizes = List.of(100, 200, 300, 400, 500);
    List<Integer> edgeCounts = List.of(10, 20, 30);

    List<ComparisonResult> results = new ArrayList<>();

    for (int taskCount : taskSizes) {
        List<App.TaskNode> baseDag = TaskDagGenerator.generateRandomDAG(taskCount);
        for (int edge : edgeCounts) {
            System.out.printf("\n=== Running TaskCount=%d | Edge=%d ===\n", taskCount, edge);
```

```
// Run Random-CPOP
List<App.TaskNode> dagCPOP = TaskDagGenerator.deepCopyDag(baseDag);
SimulationResult cpopResult = SingleSimulationRunner.runSimulation(dagCPOP, edge);
results.add(new ComparisonResult(edge, taskCount, algorithm: "CPOP",
        cpopResult.makespan(), cpopResult.energy(),
        cpopResult.deadlineMissRatio(), cpopResult.qosScore()));
System.out.println("Random-CPOP finished");

// Run PSO
List<App.TaskNode> dagPSO = TaskDagGenerator.deepCopyDag(baseDag);
PSOScheduler pso = new PSOScheduler();
List<App.TaskNode> optimizedDag = pso.schedule(dagPSO, edge);
SimulationResult psoResult = SingleSimulationRunner.runSimulation(optimizedDag, edge);
results.add(new ComparisonResult(edge, taskCount, algorithm: "PSO",
        psoResult.makespan(), psoResult.energy(),
        psoResult.deadlineMissRatio(), psoResult.qosScore()));
System.out.println("PSO finished");
```

معیارهای ارزیابی

عملکرد الگوریتم‌ها بر اساس چهار معیار کلیدی زیر سنجیده شد:

مصرف انرژی (Energy Consumption): این معیار، مجموع انرژی مصرفی توسط تمام میزبان‌های فیزیکی (Hosts) در طول شبیه‌سازی است. برای محاسبه دقیق، از مدل توان PowerModelHostSimple در CloudSimPlus استفاده شد که توان مصرفی هر میزبان را بر اساس درصد بهره‌برداری (CPU utilization) آن در هر لحظه مدل‌سازی می‌کند. این معیار، بهره‌وری الگوریتم در استفاده از منابع سخت‌افزاری را می‌سنجد. مقدار کمتر به معنای عملکرد بهتر است.

```java
double totalEnergy = 0.0;
double time = simulation.clock();
for (Host h : datacenter.getHostList()) {
    double util = 0;
    for (Vm vm : h.getVmList()) {
        util += vm.getCpuPercentUtilization(time);
    }
    double avgUtil = h.getVmList().isEmpty() ? 0 : util / h.getVmList().size();
    double power = h.getPowerModel().getPower(avgUtil);
    totalEnergy += power * time;
}
```

کیفیت خدمات (Quality of Service - QoS): یک امتیاز ترکیبی بین ۰ و ۱ که بر اساس میزان رعایت مهلت‌های زمانی (deadlines) وظایف محاسبه می‌شود. امتیاز بالاتر به معنای QoS بهتر است. امتیاز QoS برای هر وظیفه محاسبه و در نهایت میانگین آن برای تمام وظایف گزارش می‌شود.

```java
double qos = 0;
for (App.TaskNode task : dag) {
    double finish = task.cloudlet.getFinishTime();
    double deadline = task.deadline;
    double execTime = task.cloudlet.getActualCpuTime();

    qos += 1 - Math.max(0, (finish - deadline) / execTime);
}

qos = Math.max(0, qos / dag.size());
```

زمان اتمام کل (Makespan): زمان تکمیل آخرین وظیفه در گراف. Makespan کارایی کلی و سرعت الگوریتم در به پایان رساندن مجموعه کارها را نشان می‌دهد. مقدار کمتر به معنای عملکرد بهتر است.

```java
double makespan = finished.stream()
        .mapToDouble(Cloudlet::getFinishTime)
        .max()
        .orElse(0);
```

نرخ نقض مهلت زمانی (Deadline Miss Ratio): این معیار یک شاخص حیاتی برای سیستم‌های بی‌درنگ است و به صورت درصد وظایفی که پس از مهلت زمانی (Deadline) از پیش تعیین‌شده

خود به اتمام رسیده‌اند، محاسبه می‌شود. این یک معیار باینری (موفق/ناموفق) برای هر وظیفه است. مقدار کمتر به معنای پایبندی بیشتر به محدودیت‌های زمانی و عملکرد بهتر است.

```
long missed = dag.stream().filter(App.TaskNode::isDeadlineViolated).count();
double missRatio = (double) missed / dag.size();
```

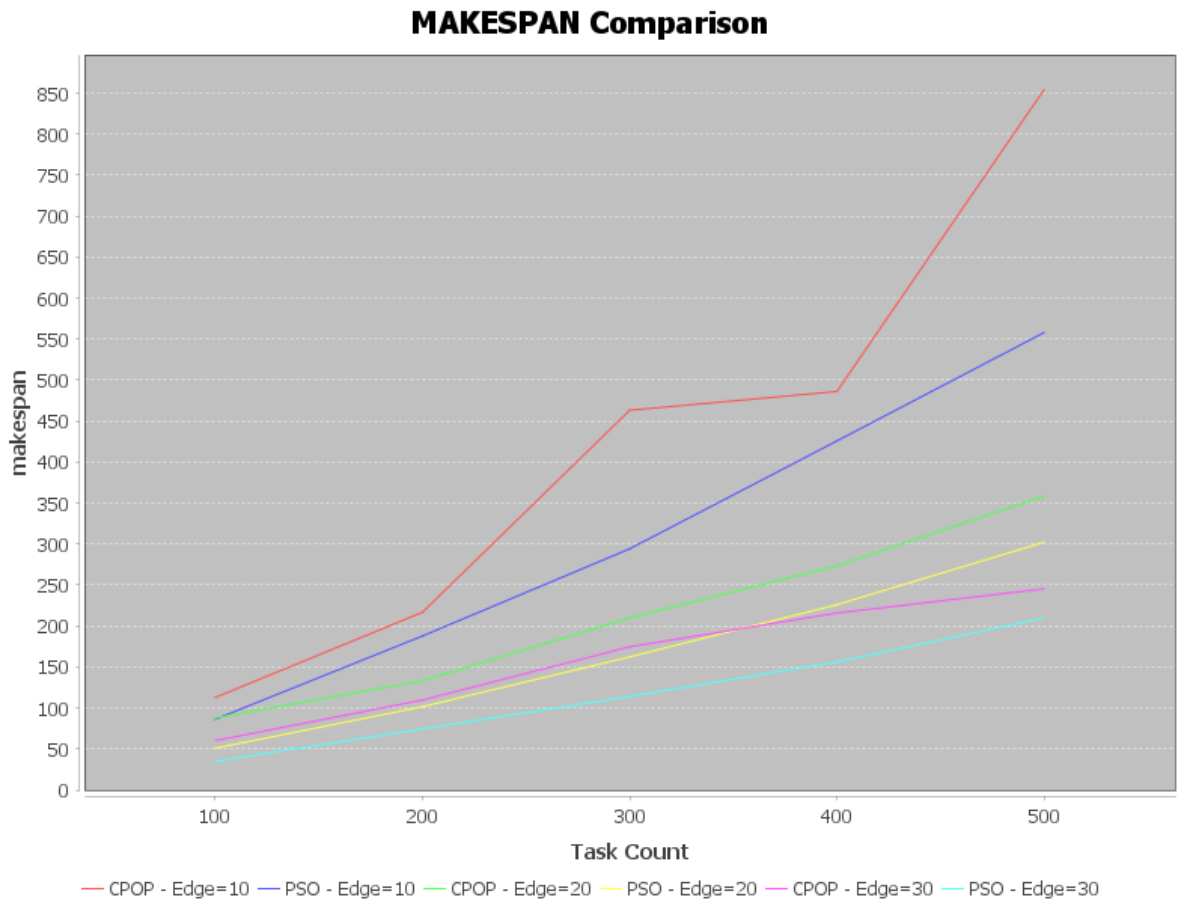
نتایج و تحلیل

پس از اجرای کامل شبیه‌سازی‌ها برای تمامی ۱۵ سناریو متفاوت با تعداد مختلف وظایف (Tasks) و تعداد یال‌های گراف (Edges)، نتایج به‌دست‌آمده در قالب چهار شاخص اصلی یعنی Makespan، مصرف انرژی، نرخ نقض مهلت زمانی (Deadline Miss Ratio) و امتیاز کیفیت خدمت (QoS) تحلیل شدند. الگوریتم PSO در اغلب سناریوها عملکرد بهتری نسبت به Random-CPOP دارد، اما در برخی شرایط خاص نیز رفتار غیرمنتظره‌ای از خود نشان می‌دهد که در ادامه به آن پرداخته می‌شود.

نمودار مقایسه‌ای زمان اتمام کل (Makespan)

نمودار Makespan به طور کلی نشان‌دهنده برتری الگوریتم PSO نسبت به Random-CPOP است. با افزایش تعداد وظایف، هر دو الگوریتم شاهد رشد Makespan هستند، اما این رشد در PSO کنترل‌شده‌تر و آهسته‌تر است. دلیل اصلی این برتری در PSO، توانایی آن در انجام نگاشت وظایف به ماشین‌های مجازی بر اساس یادگیری تجربی و با هدف کاهش گلوگاه‌ها و استفاده حداکثری از موازی‌سازی است.

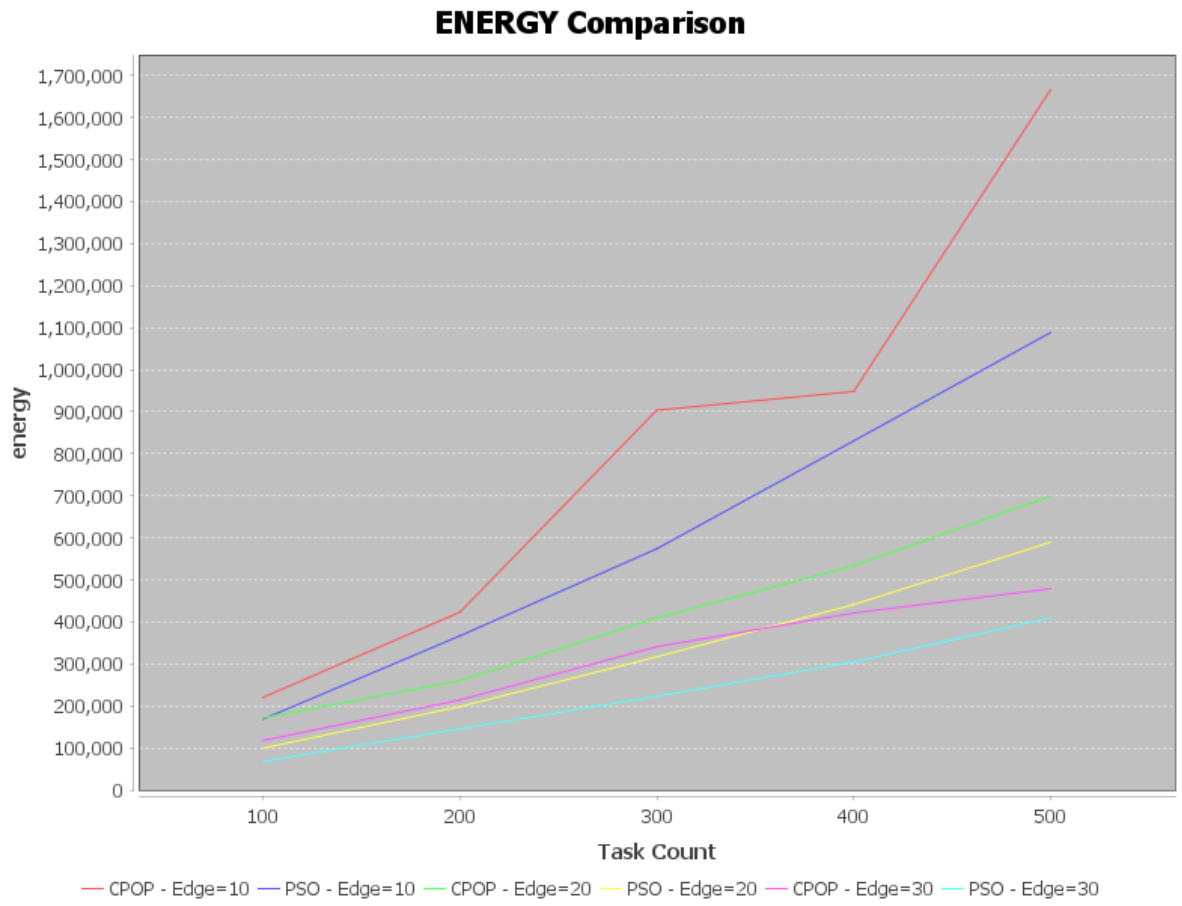
در حالی که CPOP وظایف را بر اساس اولویت‌های منطقی اجرا می‌کند ولی نگاشت آن به VMها کاملاً تصادفی است، PSO از دیدی جامع به مسئله نگاه کرده و تلاش می‌کند تا هر وظیفه را روی VMی قرار دهد که تضمین کند اجرای وظیفه سریع‌تر، کاراتر و با تداخل کمتر باشد. نتیجه این تصمیم‌گیری هوشمندانه، کاهش محسوس زمان اجرای کل پروژه در بسیاری از سناریوهاست.



نمودار مقایسه‌ای مصرف انرژی

در شاخص مصرف انرژی نیز PSO در اغلب سناریوها عملکرد بهتری نسبت به Random-CPOP از خود نشان می‌دهد. الگوریتم ما با بهره‌گیری از تابع هدف ترکیبی در PSO، می‌آموزد که نه تنها Makespan را کاهش دهد، بلکه وظایف را به گونه‌ای توزیع کند که از اجرای VMها در حالت‌های غیربهینه (زیر بار یا بیش‌ازحد بارگذاری شده) جلوگیری شود.

در مقابل، Random-CPOP با اختصاص وظایف به ماشین‌ها به صورت تصادفی، معمولاً موجب استفاده نامتوازن از منابع می‌شود: برخی VMها بیش‌فعال و برخی غیرفعال باقی می‌مانند که منجر به مصرف انرژی ناکارآمد و اتلاف منابع می‌شود.

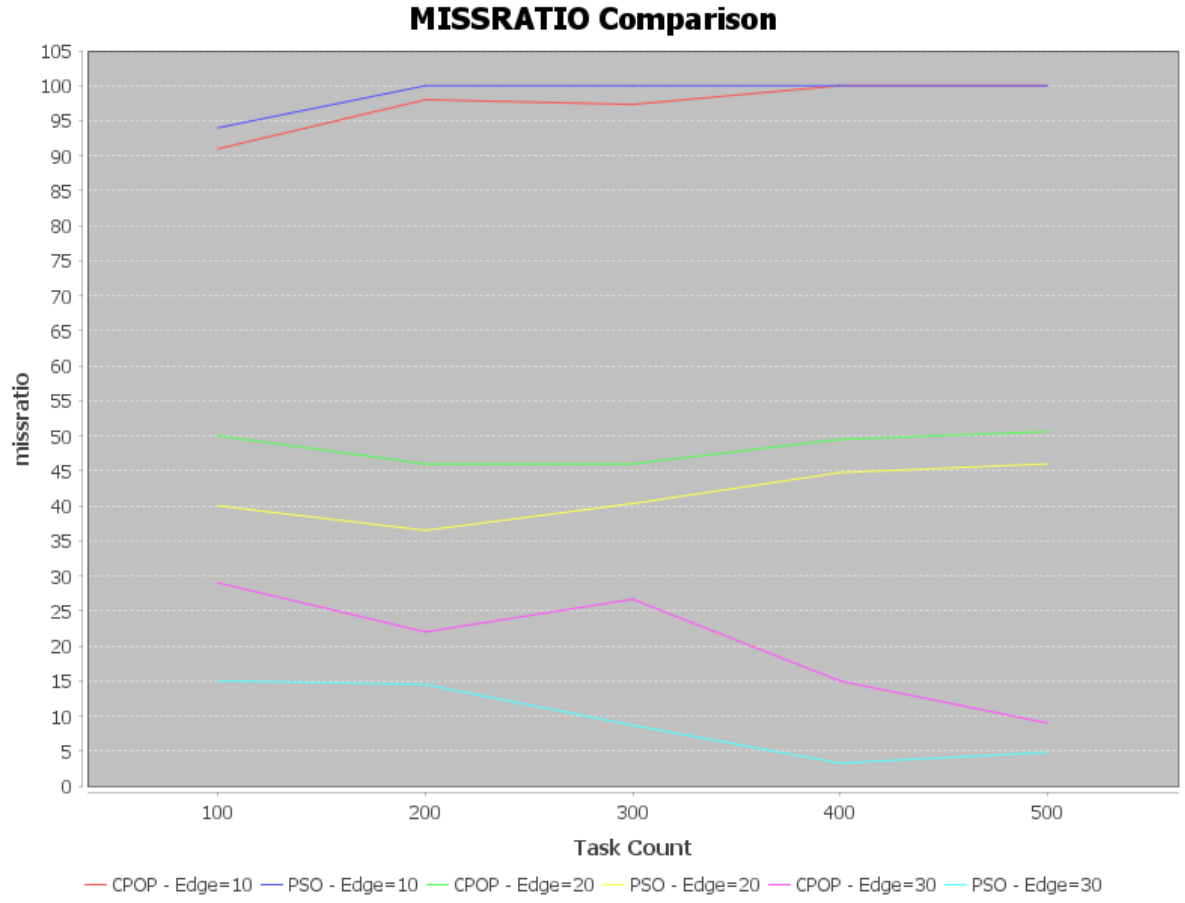


نمودار مقایسه‌ای نرخ نقض مهلت زمانی

از نتایج به‌دست‌آمده مشخص است که در برخی موارد – به‌ویژه در سناریوهایی با تعداد یال‌های کم (Edge=10) – الگوریتم PSO نرخ بسیار بالای نقض مهلت زمانی دارد، حتی گاهی تا 100 درصد، برخلاف انتظار، این عملکرد ضعیف PSO در این موارد خاص، به دلیل وابستگی های بین وظایف در DAG اتفاق می‌افتد.

نسخه فعلی PSO ترتیب اجرایی وظایف را تنها درون هر ماشین مجازی (VM) مشخص می‌کند و از یک ترتیب کلی بین همه وظایف استفاده نمی‌کند. به همین دلیل در سناریوهایی که بسیاری از وظایف می‌توانند به صورت موازی اجرا شوند، ممکن است PSO وظایف مهم و اولیه را در اولویت قرار ندهد و باعث شود وظایف فرزند دیرتر اجرا شوند و از deadline عبور کنند. این در حالی است که CPOP، به واسطه استفاده از رتبه‌بندی وظایف (rankU + rankD) در سطح DAG، در بسیاری از موارد ترتیب اجرای مناسب‌تری را ایجاد کرده و نرخ نقض مهلت زمانی کمتری دارد، حتی با نگاشت تصادفی.

البته گاهی با افزایش تعداد یال‌ها، وابستگی‌های DAG پیچیده‌تر شده و مزایای PSO در نگاشت بهینه VM بهتر خود را نشان می‌دهد. در این شرایط، PSO موفق می‌شود نرخ نقض مهلت زمانی را به‌شدت کاهش دهد و بسیاری از وظایف را به موقع خاتمه دهد.



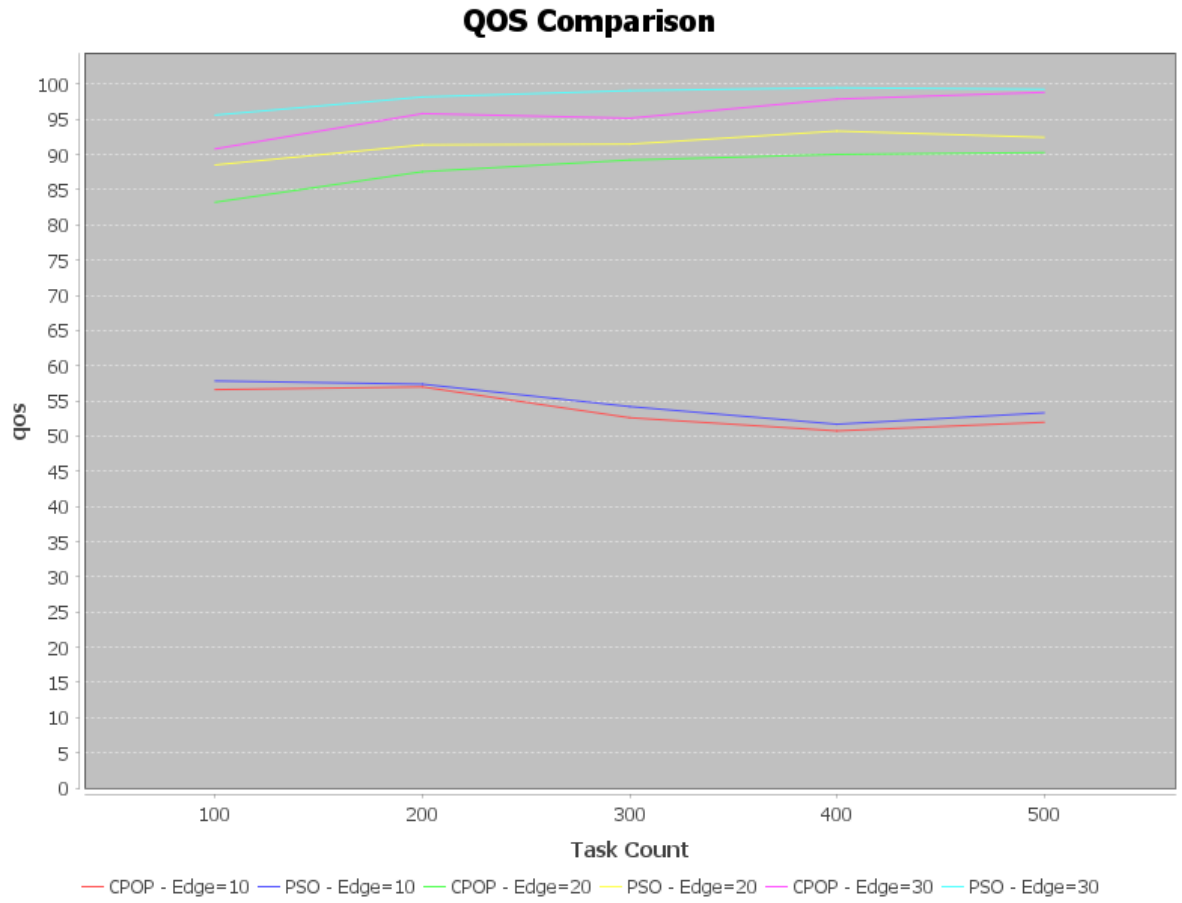
نمودار مقایسه‌ای کیفیت خدمات (QoS)

امتیاز QoS یکی از معیارهای مهم ارزیابی عملکرد سیستم‌های واقعی است زیرا نه تنها توجه به عبور نکردن از deadline دارد، بلکه میزان تأخیر نسبی (حتی اگر از deadline عبور نکند) را نیز لحاظ می‌کند. نتایج نشان می‌دهند که الگوریتم PSO معمولاً امتیاز QoS بالاتری نسبت به Random-CPOP دارد، به‌ویژه در سناریوهایی که وابستگی در گراف بیشتر است.

این نتیجه نشان می‌دهد که PSO نه تنها توانسته بسیاری از وظایف را قبل از deadline خاتمه دهد، بلکه وظایفی که تأخیر داشته‌اند نیز معمولاً با تأخیر کمتر پایان یافته‌اند. این موضوع به

دلیل ساختار تابع برازش PSO است که مستقیماً تلاش می‌کند زمان پایان واقعی وظیفه را به deadline آن نزدیک نگه دارد و تأخیر کلی را به حداقل برساند.

در مقابل، کیفیت خدمات در Random-CPOP به شدت با کاهش edge افت پیدا می‌کند، زیرا این الگوریتم هیچ مکانیزم فعالی برای کنترل تأخیرها و حفظ QoS ندارد.



نتیجه‌گیری کلی

الگوریتم PSO در اکثر معیارها – شامل زمان اجرا، بهره‌وری انرژی و کیفیت خدمت – عملکرد بهتری نسبت به الگوریتم Random-CPOP از خود نشان داده است، به‌خصوص زمانی‌که ساختار گراف پیچیده‌تر و وابستگی‌ها بیشتر باشند. با این حال، در مواردی به دلیل رعایت وابستگی های DAG، ضعف PSO در ترتیب‌دهی نمایان می‌شود و می‌تواند باعث افت شدید در معیارهایی مانند Miss Ratio شود.

در مجموع، استفاده از PSO در سناریوهای واقع‌گرایانه‌تر که گراف وظایف دارای همبستگی و وابستگی است، توصیه می‌شود، زیرا در این حالت مزایای یادگیری و بهینه‌سازی PSO به بهترین شکل ممکن مشاهده می‌شود.

دیگر کلاس‌های پروژه

همچنین پروژه شامل کلاس های دیگری هم هستند که در ادامه توضیح خلاصه ای از کاربرد آنها آورده‌ایم:

FitnessPlotter: از این کلاس برای مشاهده روند بهبود fitness در PSO استفاده شد تا بهترین مجموعه پارامتر ها و fitness function یافت شود.

```
1 usage
public FitnessPlotter(String title, List<Double> fitnessValues) {
    super(title);

    XYSeries series = new XYSeries( key: "Best Fitness");
    for (int i = 0; i < fitnessValues.size(); i++) {
        series.add( x: i + 1, fitnessValues.get(i));
    }

    XYSeriesCollection dataset = new XYSeriesCollection(series);

    JFreeChart chart = ChartFactory.createXYLineChart(
            title,
            xAxisLabel: "Iteration",
            yAxisLabel: "Fitness",
            dataset,
            PlotOrientation.VERTICAL,
            legend: true,  tooltips: true,  urls: false
    );

    ChartPanel panel = new ChartPanel(chart);
    panel.setPreferredSize(new java.awt.Dimension( width: 800,  height: 600));
    setContentPane(panel);
}
```

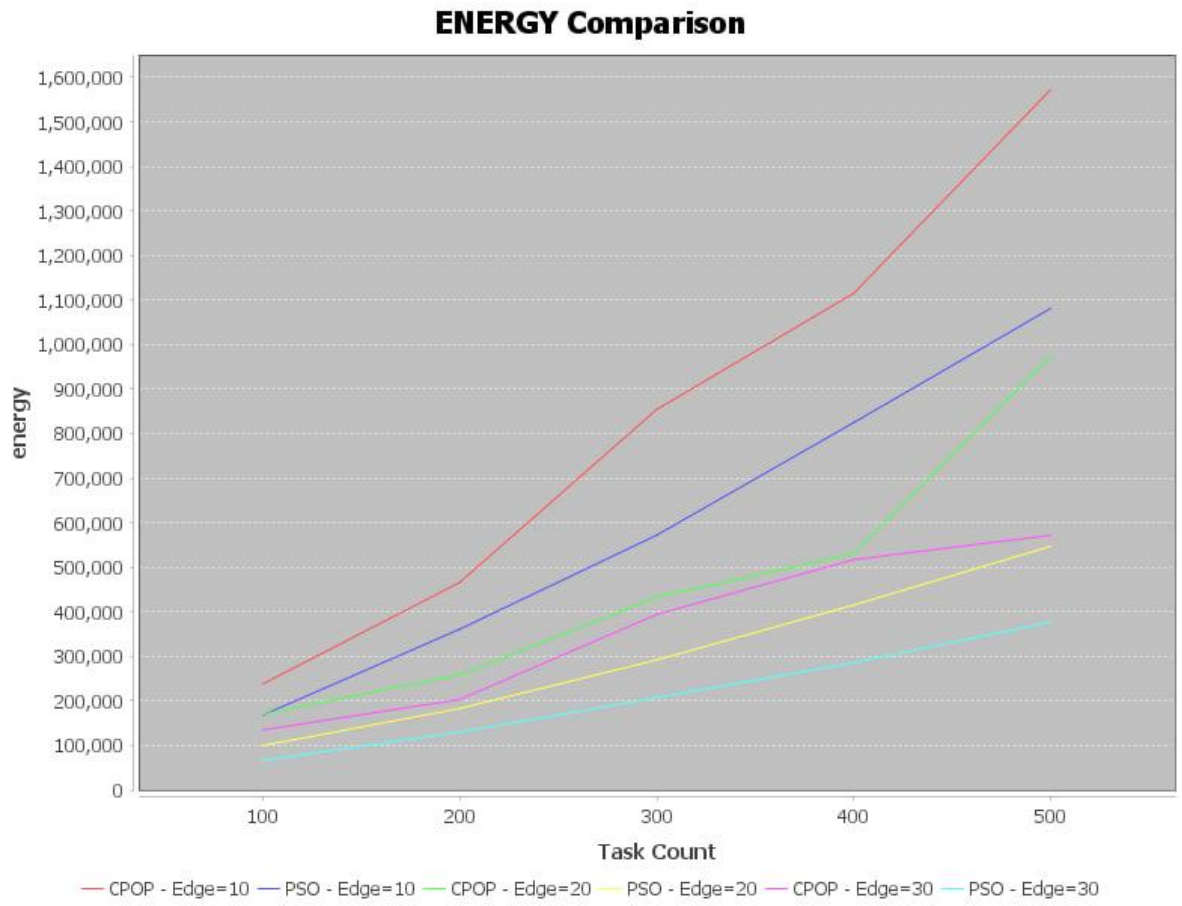
PSOBatchRunner: از این کلاس برای اجرای PSO روی ۹ سناریو با تعداد edge برابر ۱۰ و ۲۰ و ۳۰ و تعداد تسک ۱۰۰ و ۳۰۰ و ۵۰۰ استفاده شد تا با مشاهده نتایج بتوان پیاده سازی و انتخاب پارامترهای PSO را بهبود بخشید.

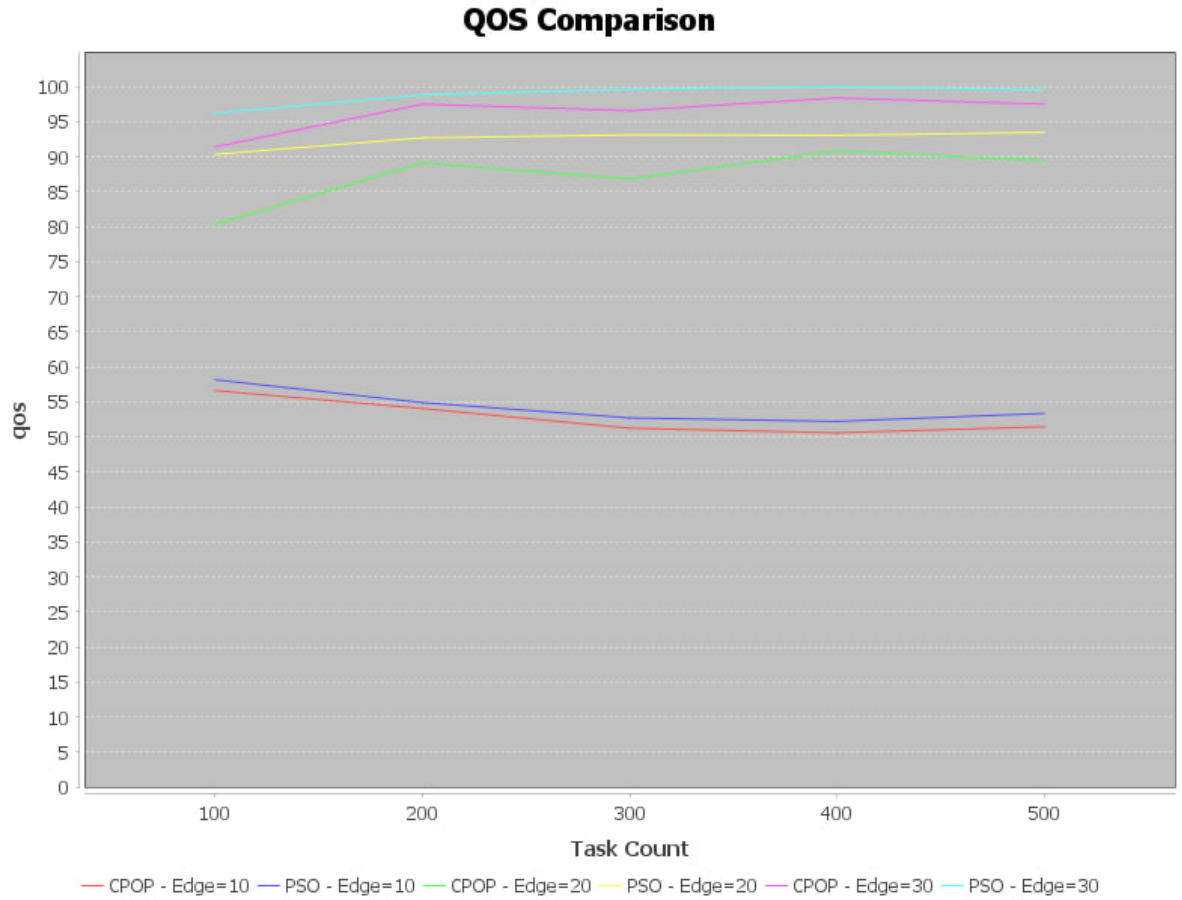
```java
public class PSOBatchRunner {
    public static void main(String[] args) {
        List<Integer> edgeOptions = List.of(10, 20, 30);
        List<Integer> taskCounts = List.of(100, 300, 500);

        for (int taskCount : taskCounts) {
            List<App.TaskNode> dag = TaskDagGenerator.generateRandomDAG(taskCount);

            for (int edge : edgeOptions) {
                PSOScheduler pso = new PSOScheduler();
                List<App.TaskNode> scheduledDag = pso.schedule(dag, edge);
                FitnessPlotter.show( title: "Fitness over Iterations (tasks: " + taskCount + " edge: " + edge + ")", pso.getFitnessHistory());
                var result : SimulationResult = SingleSimulationRunner.runSimulation(scheduledDag, edge);

                System.out.printf("Edge: %d, Tasks: %d → Makespan: %.2f, QoS: %.2f%%, Miss: %.2f%%, Energy: %.2f\n",
                        edge, taskCount,
                        result.makespan(),
                        result.qosScore() * 100,
                        result.deadlineMissRatio() * 100,
                        result.totalExecutionTime()
```

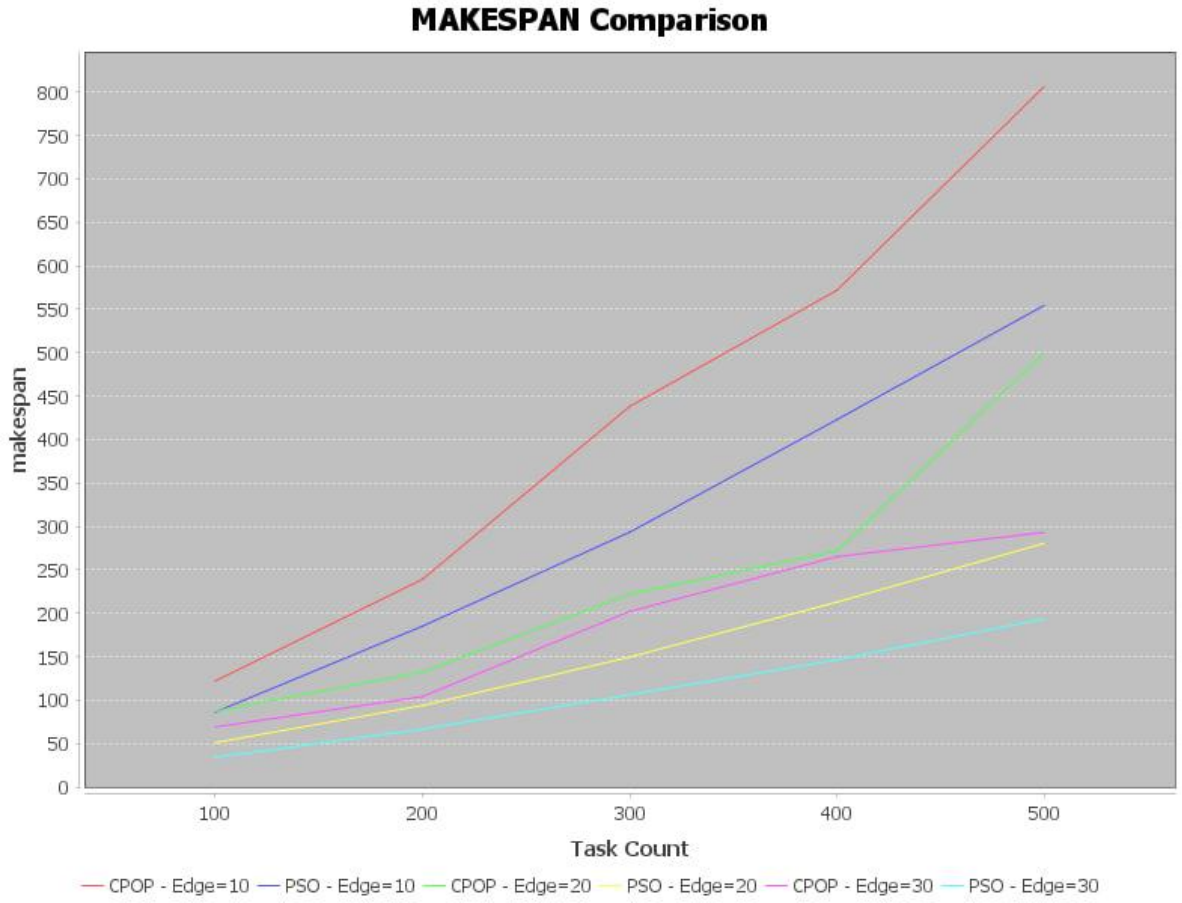
ChartPlotter: این کلاس برای مشاهده نمودار های نهایی خروجی مقایسه زمانبندی حاصل از الگوریتم Baseline و PSO نوشته شده است.

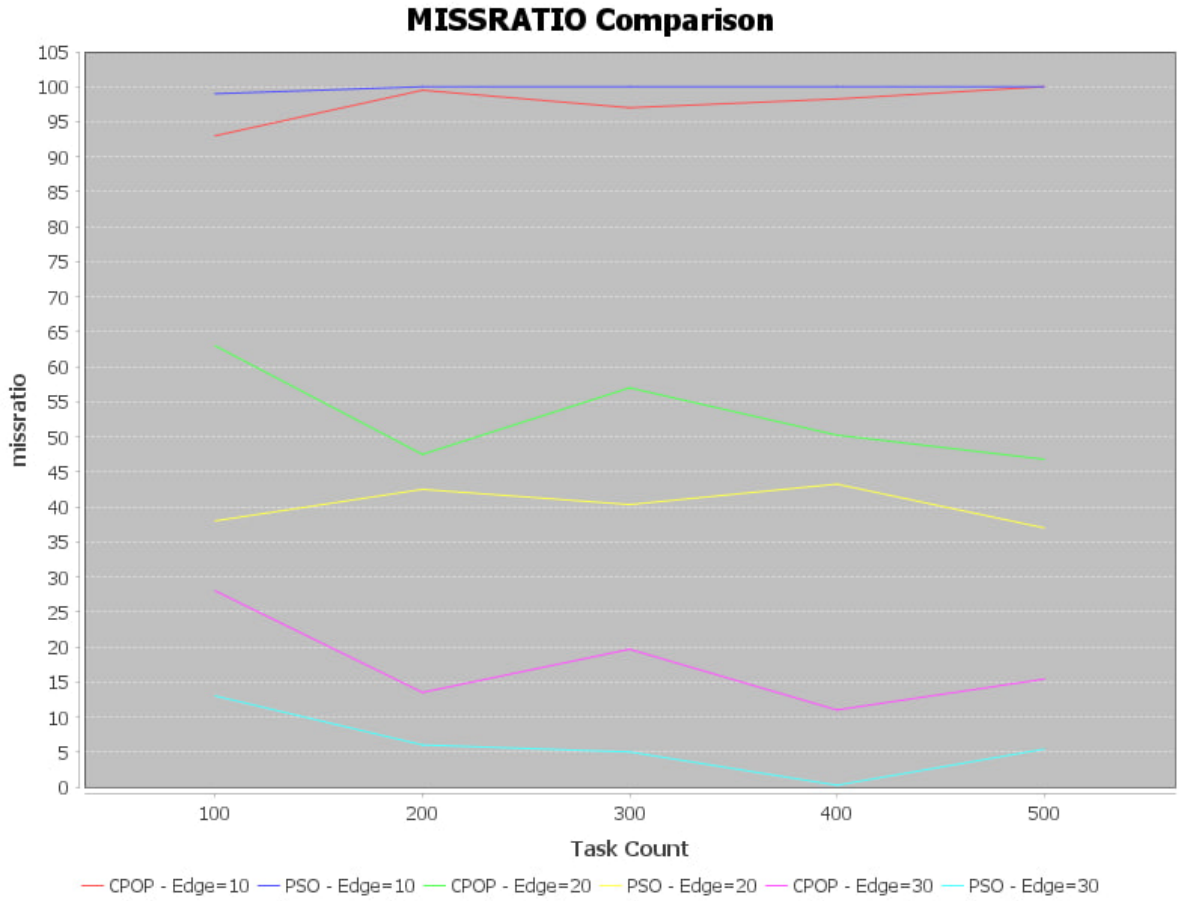
ابتکار در ترکیب دو الگوریتم CPOP و PSO

با توجه به اینکه در الگوریتم Random-CPOP نگاشت وظایف بر روی VM ها به صورت رندوم انجام میشود و از طرف دیگر در الگوریتم PSO بهینه سازی روی Mapping تمرکز دارد (زیرا با هر بار تغییر در mapping زمانبندی و ترتیب اجرا کاملا باید تغییر کند و عملا در ترتیب اجرا بهینه سازی ذرات وجود ندارد.) تصمیم گرفتیم که در یک الگوریتم ترکیبی از PSO برای mapping و از CPOP برای Scheduling این وظایف بر روی VM های نگاشت شده استفاده کنیم. نتایج زمانبندی با این الگوریتم در مقایسه با الگوریتم پایه یعنی Random-CPOP بسیار عالی بود که در ادامه نمودار های آن آمده است.


ENERGY Comparison
1,600,000
1,500,000
1,400,000
1,300,000
1,200,000
1,100,000
1,000,000
900,000
800,000
700,000
600,000
500,000
400,000
300,000
200,000
100,000
0
energy
100
200
300
400
500
Task Count
CPOP - Edge=10
PSO - Edge=10
CPOP - Edge=20
PSO - Edge=20
CPOP - Edge=30
PSO - Edge=30


QOS Comparison
100
95
90
85
80
75
70
65
60
55
50
45
40
35
30
25
20
15
10
5
0
qos
100
200
300
400
500
Task Count
CPOP - Edge=10
PSO - Edge=10
CPOP - Edge=20
PSO - Edge=20
CPOP - Edge=30
PSO - Edge=30


MAKESPAN Comparison
makespan
800
750
700
650
600
550
500
450
400
350
300
250
200
150
100
50
0
100
200
300
400
500
Task Count
CPOP - Edge=10
PSO - Edge=10
CPOP - Edge=20
PSO - Edge=20
CPOP - Edge=30
PSO - Edge=30


MISSRATIO Comparison
missratio
105
100
95
90
85
80
75
70
65
60
55
50
45
40
35
30
25
20
15
10
5
0
100
200
300
400
500
Task Count
CPOP - Edge=10
PSO - Edge=10
CPOP - Edge=20
PSO - Edge=20
CPOP - Edge=30
PSO - Edge=30