به نام خدا

سیستم‌های بی‌درنگ

پروژه درس - زمان‌بندی منصفانه در سیستم‌های ابری

استاد :

دکتر سپیده صفری

نویسنده :

محمدهومان کشوری - هیربد بهنام

شماره دانشجویی :

۹۹۱۷۱۳۳۳ - ۹۹۱۰۵۶۶۷

۱ فاز یک

قسمت اول - Task Generator

در این قسمت دو کلاس Task ، generate_tasks را داریم که اولین کلاس به خود تسک‌ها اشاره می‌کند و دومی به ساخت تسک‌ها مطابق FFT [۱] و GE [۲] اشاره می‌کند.

در generate_tasks ۳ تابع داریم که مطابق زیر هستند :

```python
    @staticmethod
    def FFT(m: int) -> dict[int, Task]:
        total_graph_nodes = int(m * math.log2(m) + 2 * m - 1)
        all_nodes: dict[int, Task] = {}
        for node in range(1, total_graph_nodes + 1):
            all_nodes[node] = Task(node)

        for i in range(1, m - 1):
            for j in range(2 ** (i - 1), 2 ** (i)):
                for k in range(j * 2, j * 2 + 2):
                    all_nodes[j].children.append(k)
                    all_nodes[k].fathers.append(j)

        base_pointer = 2 * m
        forward = -1
        for i in range(0, int(math.log2(m))):
            for j in range(1, m + 1, 2 ** (i)):
                forward *= -1
                for k in range(j, j + 2 ** (i)):
                    all_nodes[base_pointer + k - 1].fathers.append(
                        base_pointer + k - m - 1
                    )
                    all_nodes[base_pointer + k - 1].fathers.append(
                        base_pointer + k - m - 1 + forward * 2 ** (i)
                    )
                    all_nodes[
                        base_pointer + k - m - 1 + forward * 2 ** (i)
                    ].children.append(base_pointer + k - 1)
                    all_nodes[base_pointer + k - m - 1].children.append(
                        base_pointer + k - 1
                    )
            base_pointer += m

        return all_nodes

```

تابع بالا یک عدد m را گرفته و مطابق آن یک دیکشنری از تسک‌ها تولید کرده و آنرا به عنوان خروجی می‌دهد. روابط بین تسک‌ها FFT DAG است.

```python
    @staticmethod
```

[۱] Fast Fourier Transform
[۲] Gaussian Elimination

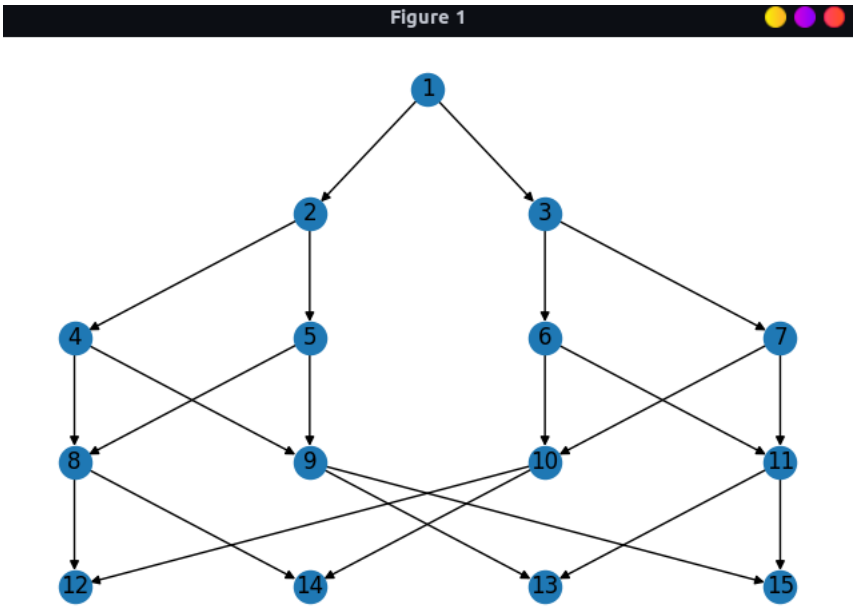
```
    def GE(m: int) -> dict[int, Task]:
        total_graph_nodes = int((m * m + m - 2) / 2)
        all_nodes: dict[int, Task] = {}
        for node in range(1, total_graph_nodes + 1):
            all_nodes[node] = Task(node)

        base_pointer = 1
        while m > 1:
            for i in range(base_pointer + 1, base_pointer + m):
                all_nodes[i].fathers.append(base_pointer)
                all_nodes[base_pointer].children.append(i)

            for i in range(base_pointer + 1, base_pointer + m):
                if i >= total_graph_nodes:
                    break
                all_nodes[i + m - 1].fathers.append(i)
                all_nodes[i].children.append(i + m - 1)

            base_pointer += m
            m -= 1

        return all_nodes

```

این تابع نیز مانند تابع قبلی یک عدد m را گرفته و مطابق آن تسک‌ها را و روابط میان آنها را ساخته و یک دیکشنری از تسک‌ها را خروجی می‌دهد.

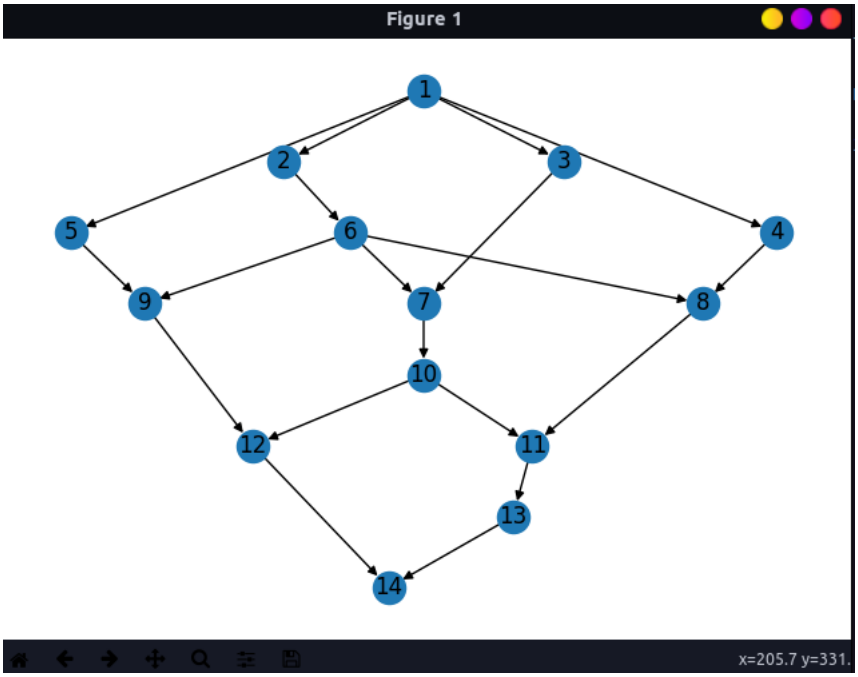
```
    @staticmethod
    def draw_graph(graph: dict[int, Task]):
        G = nx.DiGraph()

        for node, attrs in graph.items():
            G.add_node(node)
            for child in attrs.children:
                G.add_edge(node, child)

        pos = nx.drawing.nx_agraph.graphviz_layout(G, prog="dot")
        nx.draw(G, pos, with_labels=True, arrows=True)
        plt.show()

```

به واسطه این تابع نیز می‌توانیم تسک‌ها رسم کنیم. چند مثال از ورودی و خروجی متناظر آنها را مشاهده کنیم. ورودی برنامه و خروجی‌های متناظر آن :

```
python3 task_generator.py FFT 4

```




```
python3 task_generator.py GE 5

```



در کلاس Task نیز هر تسک ابتدا که تعریف می‌شود یک id و لیست پدران و فرزندان متناظر با آن نیز تعریف می‌شود و یک لیست که نشان می‌دهد هر تسک چقدر زمان اجرا نیاز دارد و یک دیکشنری که نشان می‌دهد هر تسک برروی هر هسته چقدر به communication cost اضافه شد.

```
def __init__(self, id: int):
    self.id = id
    self.fathers: list[int] = []
    self.children: list[int] = []
    # This should be used like this:
    # First index is the ID of the task which we want to comminate with (the child ID)
    # The second index is the core ID which this task is currently on
    # The third index is the core ID which child task should run on
    self.communication_cost: dict[int, list[list[int]]] = {}
    self.computation_times: list[int] = []

```

در توابع زیر نیز عملا هزینه ارتباط و زمان محاسبه تسک به صورت رندوم برای هر cpu تعیین می‌شود.

```
def populate_cpu_dependant_variables(self, cpu_count: int):
    for _ in range(cpu_count):
        self.computation_times.append(random.randint(10, 100))
        self.computation_times[-1] = min(
            max(1, int(random.normalvariate(self.computation_times[-1], 5))),
            self. computation_times[-1] + 25,
        )
    self.generate_random_communication_cost(cpu_count)

def generate_random_communication_cost(self, cpu_count: int):
    self.communication_cost.clear()
    for child_id in self.children:
        costs = [[0] * cpu_count for _ in range(cpu_count)]
        for cpu1 in range(cpu_count):
            for cpu2 in range(cpu_count):
                if cpu1 == cpu2:
                    continue
                costs[cpu1][cpu2] = random.randint(5, 25)
        self.communication_cost[child_id] = costs

```

قسمت دوم - HEFT

در ابتدا با توجه به فایل‌های PDF که آقای طوقانی برای بچه‌ها ایمیل کرده بودند باید الگوریتم HEFT را پیاده سازی کنیم به جای MinMax. برای این کار نیاز است که در ابتدا rank هر کدام از تسک‌های DAG را بدست بیاوریم. برای این کار کد زیر را نوشتیم:

```
    @staticmethod
    def rank(tasks: dict[int, Task]) -> list[tuple[float, Task]]:
        # task-id -> rank
        task_ranks: dict[int, float] = {}
        for id in range(1, len(tasks) + 1):
            if id in task_ranks:
                continue
            task_ranks[id] = HEFT.calculate_rank(tasks, task_ranks, id)
        result: list[tuple[float, Task]] = []
        for id, rank in sorted(
            task_ranks.items(), key=lambda item: item[1], reverse=True
        ):
            result.append((rank, tasks[id]))
        return result

    @staticmethod
    def calculate_rank(
        tasks: dict[int, Task], task_ranks: dict[int, float], wanted:
    int
    ) -> float:
        if wanted in task_ranks:
            return task_ranks[wanted]
        current_task = tasks[wanted]
        if len(current_task.children) == 0:
            return current_task.average_computation()
        current_rank = current_task.average_computation() + max(
            map(
                lambda child_task_id: current_task.average_communication
    (child_task_id)
                + HEFT.calculate_rank(tasks, task_ranks, child_task_id),
                current_task.children,
            )
        )
        task_ranks[wanted] = current_rank
        print(f"{wanted} -> {current_rank}")
        return current_rank

```

به صورت خلاصه کاری که انجام می‌دهیم این است که تابع calculate_rank را بر روی اولین تسک DAG صدا میزنیم و در نتیجه به صورت بازگشتی تمامی rankها بدست می‌آید و آن را در آرایه‌ای که بر اساس rank مرتب شده است قرار می‌دهیم. در ادامه نیز تابع زمان بندی اصلی را باید تعریف کنیم. اکثر ایده‌هایی که گرفتیم از این پروژه برداشته شده بود. این تابع را در زیر مشاهده می‌کنید:

```
    @staticmethod
    def is_interval_occupied_in_time(
        intervals: list[tuple[int, int]], interval: tuple[int, int]
    ) -> bool:
        # https://stackoverflow.com/a/3269471/4213397
        return any(
            map(lambda i: i[0] <= interval[1] and interval[0] <= i[1],
intervals)
        )

    @staticmethod
    def find_gap(
        scheduled_tasks: dict[int, ScheduledTask],
        cpu_id: int,
        fastest_start_time: int,
        computation_cost: int,
    ) -> int:
        """
        Finds the first time which we can schedule a task on a specific
core
        """
        occupied_intervals = sorted(
            map(
                lambda task: (task.start_time, task.
computation_finish_time),
                filter(
                    lambda item: item.ran_cpu_id == cpu_id,
scheduled_tasks.values()
                ),
            ),
            key=lambda item: item[0],  # sort by start time
        )
        candidate_start_time = fastest_start_time
        # This is a horrible way to do it but whatever
        while HEFT.is_interval_occupied_in_time(
            occupied_intervals,
            (candidate_start_time, candidate_start_time +
computation_cost),
        ):
            candidate_start_time += 1
        return candidate_start_time

    @staticmethod
    def schedule(tasks: dict[int, Task], cpus: int) -> dict[int,
ScheduledTask]:
        ranks = HEFT.rank(tasks)
        # task_id -> task
        scheduled_tasks: dict[int, ScheduledTask] = {}
```

```
        for task in map(lambda rank: rank[1], ranks):
            # cpu_id -> (start, finish) for each CPU core
            cpu_runtimes: list[tuple[int, int]] = [(0, 0)] * cpus
            for cpu_id in range(cpus):
                processor_ready = 0  # when does this core can become available for scheduling this task
                for parent_id in task.fathers:
                    assert parent_id in scheduled_tasks  # sanity check
                    communication_cost = tasks[parent_id].communication_cost[task.id][
                        scheduled_tasks[parent_id].ran_cpu_id
                    ][cpu_id]
                    start_delay = (
                        communication_cost
                        + scheduled_tasks[parent_id].computation_finish_time
                    )
                    processor_ready = max(processor_ready, start_delay)
                # Now calculate when we can schedule this task
                cpu_start_time = HEFT.find_gap(
                    scheduled_tasks,
                    cpu_id,
                    processor_ready,
                    task.computation_times[cpu_id],
                )
                cpu_runtimes[cpu_id] = (
                    cpu_start_time,
                    cpu_start_time + task.computation_times[cpu_id],
                )
            # Now check what CPU yields the fastest one
            best_cpu_id = min(
                range(len(cpu_runtimes)), key=lambda x: cpu_runtimes[x][1]
            )
            scheduled_tasks[task.id] = ScheduledTask(
                task, best_cpu_id, cpu_runtimes[best_cpu_id][0]
            )
        return scheduled_tasks

```

تابع اصلی در اینجا schedule است که تعداد تسک‌ها و CPU coreها را می‌گیرد و زمان بندی را انجام می‌دهد. خروجی تابع یک دیکشنری از task_id به زمان‌هایی است که یک تسک شروع به انجام شدن می‌کند و پایان میابد و همچنین CPU که بر روی آن اجرا شده است. تعریف کلاس ScheduledTask را نیز می‌توانید در زیر ببینید:

```
class ScheduledTask:
    def __init__(self, task: Task, cpu_id: int, start_time: int):
        self.task_id = task.id
        self.start_time = start_time
        self.computation_finish_time = start_time + task.
```

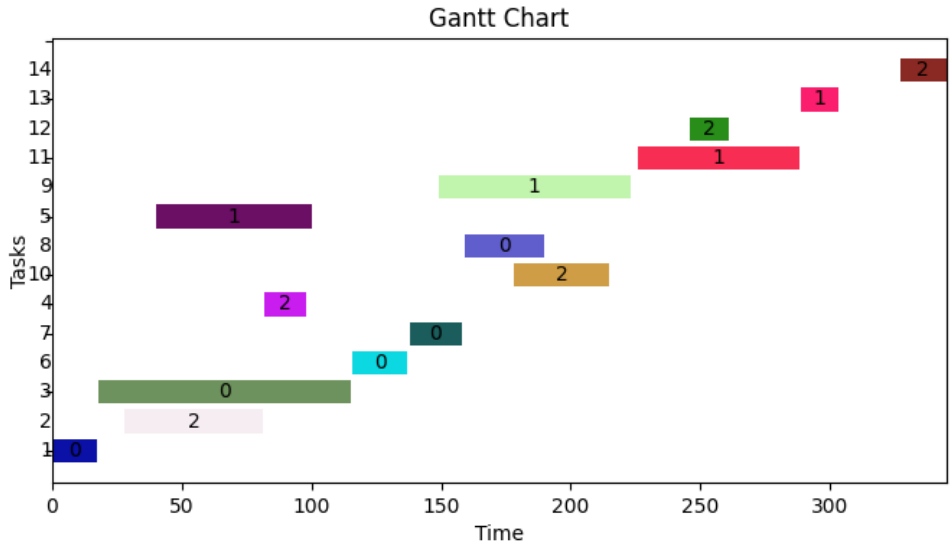
```
    computation_times[cpu_id]
        self.ran_cpu_id = cpu_id

```

در تابع زمانبندی که ما نوشتیم عملا برای هر واحد زمانی شبیه سازی انجام نمی‌شود. بلکه برای هر تسک به ترتیب rank حساب می‌شود که چه زمانی می‌تواند اجرا شود. این روش به نظر ما خیلی بهتر از زمان بندی برای هر ثانیه بود چرا که تسک‌های ما preemptive نیستند.

در داخل تابع schedule کاری که می‌کنیم این است که در ابتدا تسک‌ها را بر اساس رنک سورت می‌کنیم و هر تسک را چک می‌کنیم که زمانی که کارش در هر هسته تمام می‌شود کی است و در نهایت کمترین زمان را انتخاب می‌کنیم و آن تسک را بر روی آن هسته زمان بندی می‌کنیم. دقت کنید که همان طور که گفته شد در صورتی که بخواهیم اطلاعات را از پدرهای این تسک که بر روی هسته‌ی دیگری اجرا شدند بر روی هسته‌ای دیگر بیاوریم مجبور هستیم که کمی سربار پرداخت کنیم و این موضوع در زمان شروع تسک‌های فرزند لحاظ شده است.

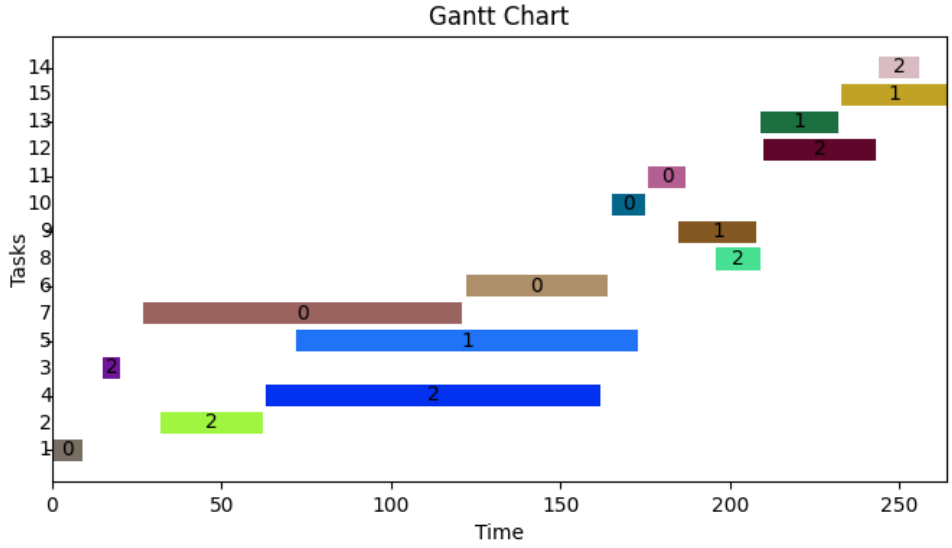
قسمت سوم - تست

در این قسمت دو ورودی کوچک و مختلف را تست و زمانبندی می‌کنیم. در ابتدا به کمک gaussian elimination گرافی با پارامتر ۵ می‌سازیم و آن را زمان بندی می‌کنیم. نتیجه‌ی آن در شکل زیر آمده است:



دقت کنید که محور عمودی نشان دهنده‌ی شماره تسکی است که در حال اجرا است و اعداد نوشته شده بر روی هر کدام از تسک‌ها هسته‌ای است که در آن اجرا می‌شود تسک. به عنوان مثال در شکل بالا همان طور که مشاهده میشود تسک ۳ بلافاصله بعد از اجرای تسک ۱ در همان هسته‌ی ۰ اجرا می‌شود چرا که نیازی به انتقال داده وجود ندارد. همچنین تسک شماره ۲ نیز با کمی تاخیر اجرا می‌شود که ناشی از انتقال داده است.

در ادامه Fast Fourier Transform با پارامتر ۴ را اجرا می‌کنیم. شکل آن در زیر آمده است:



در ابتدا که ما این عکس را دیدیم فکر کردیم که زمان بند ما باگ دارد چرا که مقدار زیادی stack time بین تسک ۱ و ۳ وجود دارد و اصلا بر روی یک هسته یکسان اجرا نمی‌شوند. اما این زمان بندی درست است چرا که اگر اجرا را بر روی هسته ۰ ادامه می‌دادیم، زمان اجرای نهایی بسیار بیشتر می‌شد و در اینجا میصرفید که داده‌ها را بر روی یک هسته‌ی دیگه بفرستیم و بر روی یک هسته‌ی سریع‌تر اجرا کنیم تسک بعدی را.

۲ فاز ۲

قسمت صفرم - DAG

در ابتدا برای این فاز نیاز است که تسک‌های DAGهای مختلف را از هم تمیز دهیم. به همین منظور یک کلاس به نام DAG تعریف می‌کنیم. کاری که این کلاس می‌کند صرفا نگه داشتن تسک‌ها و متادیتاهای مربوط به یک DAG است. به عنوان مثال زمان ریلیز و rank تسک‌ها و غیره. کد این کلاس در زیر آمده است:

```
class DAG:
    def __init__(self, id: int, tasks: dict[int, Task], cpus: int):
        self.id = id
        self.tasks = tasks
        self.lowerbound = HEFT.calculate_makespan(tasks, cpus)
        self.release_time = random.randint(0, 1000)
        self.ranks = HEFT.rank(tasks)

```

قسمت اول - MinMax

در ابتدا ما الگوریتم MinMax را طراحی کردیم. برای طراحی این الگوریتم کاری که کردیم این بود که یک لیست از DAGهای داخل سیستم به همراه تعداد CPUها را می‌گیرد و آن‌ها را زمان بندی می‌کند. روشی که این الگوریتم کار می‌کند بدین صورت است در ابتدا یک متغیر به اسم current_time را نگه می‌داریم که نشان می‌دهد که زمان شبیه‌سازی ما چند است. تا زمانی که تمامی DAGهای ما زمان بندی نشده‌اند کاری که می‌کنیم این است آن DAGهایی را پیدا می‌کنیم که زمان بندی نشده‌اند و ریلیز شده‌اند. سپس از بین آن‌ها آن را پیدا می‌کنیم که کمترین lowerbound را داشته باشد و آن را با HEFT زمان بندی می‌کنیم. دقت کنید که در نهایت نیز current_time را برابر زمانی قرار می‌دهیم که آخرین تسک تمام می‌شود. کد این زمان بند در زیر آمده است:

```
    def schedule(dags: dict[int, DAG], cpus: int) -> dict[dict[int,
ScheduledTask]]:
        dags_list = list(dags.values())
        result: dict[dict[int, ScheduledTask]] = {}
        current_time = 0
        while len(dags_list) != 0:
            # Always advance time in order to check for dynamic
scheduling
            current_time += 1
            # Check if we can run any DAG
            runnable_dags = list(filter(lambda dag: dag.release_time <=
current_time, dags_list))
            if len(runnable_dags) == 0:
                continue
            # If we have reached here, it means that we have at least
one runnable dag
            candidate_index = min(range(len(runnable_dags)), key=lambda
dag_index: runnable_dags[dag_index].lowerbound)
            # Schedule the task
            scheduled_dag = HEFT.schedule(runnable_dags[candidate_index
].tasks, cpus)
            dags_list = list(filter(lambda dag: dag.id != runnable_dags[
candidate_index].id, dags_list))
            # Fix the scheduled tasks based on current time
```

```
            time_to_advance = 0
            for scheduled_task in scheduled_dag.values():
                time_to_advance = max(time_to_advance, scheduled_task.
computation_finish_time)
                scheduled_task.start_time += current_time
                scheduled_task.computation_finish_time += current_time
            result[runnable_dags[candidate_index].id] = scheduled_dag
            # Now advance the time
            current_time += time_to_advance
        return result

```

قسمت دوم - FDWS و Hybrid Rank

این دو الگوریتم بسیار شبیه هم هستند و به همین منظور صرفا یک تابع برای هر دو الگوریتم نوشتیم و با یک boolean مشخص می‌کنیم که کدام الگوریتم باید اجرا بشود. بر خلاف الگوریتم MinMax که یک تایم نگه می‌داشتیم که نشانگر زمان حال بود و بعد از هر مرحله آن را زیاد می‌کردیم در این الگوریتم قرار نیست این کار را انجام دهیم و زمان حال را بر اساس آخرین تسک زمان بندی شده در می‌آوریم. به همین منظور در ابتدا یک حلقه تعریف می‌کنیم که تا زمانی ادامه پیدا می‌کند که تمامی تسک‌های تمامی DAGها را زمان بندی کرده باشیم. در این حلقه کاری که اتفاق می‌افتد این است که باید حساب کنیم که با توجه به زمان بندی‌های قبلی که انجام دادیم زمان حال چیست که بتوانیم تشخیص دهیم که کدام DAGها از ریلیز تایمشان گذشته است. اما نکته‌ای که وجود دارد این است که مقدار اولیه این عدد زمانی که هیچ تسکی را schedule نکردیم برابر است با اولین زمان ریلیز تایم DAG. اما این کار یک باگ ریز دارد و آن هم این است که اگر پردازنده به idle time بخورد کلا زمان بندی متوقف می‌شود چرا که لیست ready_queue خالی می‌شود. به همین منظور در ابتدا زمان بندی را تا جایی ادامه می‌دهیم که تمامی تسک‌های همه‌ی DAGها زمان بندی شده باشد. همچنین مقدار اولیه currenct_time بر اساس DAGهایی که زمان بندی تسک‌های آن‌ها مانده است مقدارش حساب می‌شود.

حال که زمان این iteration را پیدا کردیم کافی است که DAGهایی را پیدا کنیم که با توجه به زمان حال ریلیز شده باشند. از بین این DAGها تسکی را بر می‌داریم که بیشترین رنک را دارد و اجرا نشده است و آن‌ها را در یک لیست قرار می‌دهیم. حال اینجا بر اساس اینکه داریم الگوریتم FDWS یا Hybrid Rank را اجرا می‌کنیم لیست تسک‌ها (عملا ready queue)را صعودی یا نزولی بر اساس رنک مرتب می‌کنیم. البته دقت کنید که در اینجا مجبور هستیم که به رنک‌های تمامی تسک‌ها یک offset که برابر زمان ریلیز آن DAG است را اضافه کنیم. حال در ادامه باید هر یک از این تسک‌ها را زمان بندی کنیم.

برای زمانبندی این تسک‌ها از قسمت خوبی از کد زمانبند HEFT که در فاز قبل زدیم استفاده می‌کنیم. تنها فرق بزرگی که این کد با کد HEFT دارد این است که تابع HEFT.find_gap باید با تمامی تسک‌هایی که تا الان زمان بندی شده است صدا شود به جای اینکه با تمامی تسک‌هایی که مربوط به یک DAG هستند. همچنین متغیر processor_ready نیز به جای ۰ مقدار ریلیز تایم خود DAG را دارد که با این کار تسک پدر را در زمان ریلیز شدن DAG زمان بندی می‌کنیم.

کد کامل را در زیر می‌توانید مشاهده کنید:

```
    def schedule(dags: dict[int, DAG], cpus: int, is_fdws: bool) -> dict
[dict[int, ScheduledTask]]:
        dags = copy.deepcopy(dags)
        # dag_id -> task_id -> task
        result: dict[int, dict[int, ScheduledTask]] = {}
        for dag_id in dags.keys():
            result[dag_id] = {}
        while any(map(lambda dag: len(dag.ranks) != 0, dags.values())):
            # At first get the current time
            current_time = min(map(lambda dag: dag.release_time, filter(
lambda dag: len(dag.ranks) != 0, dags.values()))) # default is the
release time of the first DAG
            if len(FDWS_RANK_HYBRID.reduce_task_list(result)) != 0:
```

```
                current_time = max(current_time, max(map(lambda task:
task.computation_finish_time, FDWS_RANK_HYBRID.reduce_task_list(
result))))
            # Fill the ready queue
            # (rank, dag_id, task)
            ready_queue: list[tuple[float, int, Task]] = []
            for dag_id, dag in filter(lambda d: d[1].release_time <=
current_time, dags.items()):
                if len(dag.ranks) != 0:
                    (rank, task) = dag.ranks.pop(0)
                    # TODO: check + release_time
                    ready_queue.append((rank + dag.release_time, dag_id,
 task))
            # Now sort the ready queue based on rank
            ready_queue = sorted(ready_queue, key=lambda x: x[0],
reverse=is_fdws)
            # For each task in queue, schedule it by EFT
            # Just like EFT.schedule
            for _, dag_id, task in ready_queue:
                # cpu_id -> (start, finish) for each CPU core
                cpu_runtimes: list[tuple[int, int]] = [(0, 0)] * cpus
                for cpu_id in range(cpus):
                    processor_ready = dags[dag_id].release_time  # when
does this core can become available for scheduling this task
                    for parent_id in task.fathers:
                        assert parent_id in result[dag_id]  # sanity
check
                        communication_cost = dags[dag_id].tasks[
parent_id].communication_cost[task.id][
                            result[dag_id][parent_id].ran_cpu_id
                        ][cpu_id]
                        start_delay = (
                            communication_cost
                            + result[dag_id][parent_id].
computation_finish_time
                        )
                        processor_ready = max(processor_ready,
start_delay)
                    # Now calculate when we can schedule this task
                    cpu_start_time = HEFT.find_gap(
                        FDWS_RANK_HYBRID.reduce_task_list(result),
                        cpu_id,
                        processor_ready,
                        task.computation_times[cpu_id],
                    )
                    cpu_runtimes[cpu_id] = (
                        cpu_start_time,
                        cpu_start_time + task.computation_times[cpu_id],
```

```
                    )
                # Now check what CPU yields the fastest one
                best_cpu_id = min(
                    range(len(cpu_runtimes)), key=lambda x: cpu_runtimes
[x][1]
                )
                result[dag_id][task.id] = ScheduledTask(
                    task, best_cpu_id, cpu_runtimes[best_cpu_id][0]
                )
        return result

```

قسمت سوم - بررسی و ارزیابی

حال با استفاده از کد زیر با استفاده از هر سه زمان‌بند، تسک‌ها را زمان‌بندی می‌کنیم و سپس به واسطه پارامترهای unfairness و makespan زمان‌بندی‌ها را ارزیابی می‌کنیم.

```
    if len(sys.argv) < 5:
        print("Format should be as follows : grath generation method +
num_of_tasks + cpu cores + num_of_graphs")
        exit(1)
    generation_method = sys.argv[1]
    number_of_tasks = int(sys.argv[2])
    cpu_cores = int(sys.argv[3])
    num_of_graphs = int(sys.argv[4])
    # scheduling_method = sys.argv[5]
    dags = dict()
    dags_copy = dict()
    task_generator = generate_tasks()
    for i in range(num_of_graphs) :
        if generation_method == "GE":
            graph = task_generator.GE(number_of_tasks)
        elif generation_method == "FFT":
            graph = task_generator.FFT(number_of_tasks)
        else:
            print("Method should be : FFT - GE")
            exit(1)
        print(f"dag {i} Currently has", len(graph), "tasks")
        for task in graph.values():
            task.populate_cpu_dependant_variables(cpu_cores)
        dags_copy[i] = copy.deepcopy(graph)
        dags[i] = DAG(i,graph,cpu_cores)
    makespan = 0
    makespan, unfairness = calculate_slowdown(MinMax.schedule(dags=copy.
deepcopy(dags), cpus=cpu_cores),
                                              copy.deepcopy(
dags_copy), cpu_cores)
    print(f"makespan MinMax {makespan}")
    print(f"unfairness MinMax {unfairness}")
    makespan, unfairness = calculate_slowdown(FDWS_RANK_HYBRID
```

```
                                                    .schedule(dags=copy.deepcopy(dags), cpus=cpu_cores, is_fdws=True),
                                                     copy.deepcopy(dags_copy), cpu_cores)
    print(f"makespan fdws {makespan}")
    print(f"unfairness fdws {unfairness}")
    makespan, unfairness = calculate_slowdown(FDWS_RANK_HYBRID
                                                    .schedule(dags=copy.deepcopy(dags), cpus=cpu_cores, is_fdws=False),
                                                     copy.deepcopy(dags_copy), cpu_cores)
    print(f"makespan rank_hybd {makespan}")
    print(f"unfairness rank_hybd {unfairness}")

```

در کد بالا می‌بینیم که ابتدا گراف‌های ما ساخته شده‌اند و سپس برای زمان‌بندی به ۳ الگوریتم داده شده‌اند. حال در کد زیر عملا زمان‌بندی گراف‌ها را مشاهده می‌کنیم و پارامترهای محاسبه‌شده آنها را :

```
def calculate_slowdown(scheduledTasks : dict[dict[int, ScheduledTask]],
                       dags : dict[int, dict[int, Task]],
                       cpus : int) -> [int, int]:
    print("
    --------------------------------------------------------------------------------
    ")
    pprint(scheduledTasks)
    start = 10000000000000000000
    finish = 0
    unfairness = 0
    graph_things = {}
    for key in scheduledTasks.keys() :
        print("**********************************")
        graph_start = min(list(map(lambda x : scheduledTasks[key][x].start_time,
                                   scheduledTasks[key])))
        graph_end   = max(list(map(lambda x : scheduledTasks[key][x].computation_finish_time,
                                   scheduledTasks[key])))
        finish = max(finish, graph_end)
        start = min(start, graph_start)
        print(graph_start, graph_end)
        heft_sched = HEFT.schedule(dags[i], cpus)
        print(heft_sched)
        heft_start = min(list(map(lambda x : heft_sched[x].start_time, heft_sched)))
        heft_end   = max(list(map(lambda x : heft_sched[x].computation_finish_time, heft_sched)))
        M_Multi = graph_end - graph_start
        M_Own   = heft_end - heft_start
        slowdown = M_Own / M_Multi
```

```
        graph_things[key] = {'slowdown' : slowdown}
        print(M_Multi, M_Own)
    print("================================")
    average_slowdown = sum(list(
        map(lambda x : graph_things[x]['slowdown'],  graph_things)))/len
   (graph_things.keys())
    for key in graph_things.keys() :
        unfairness += abs(graph_things[key]['slowdown'] -
   average_slowdown)
    return (finish - start, unfairness)

```

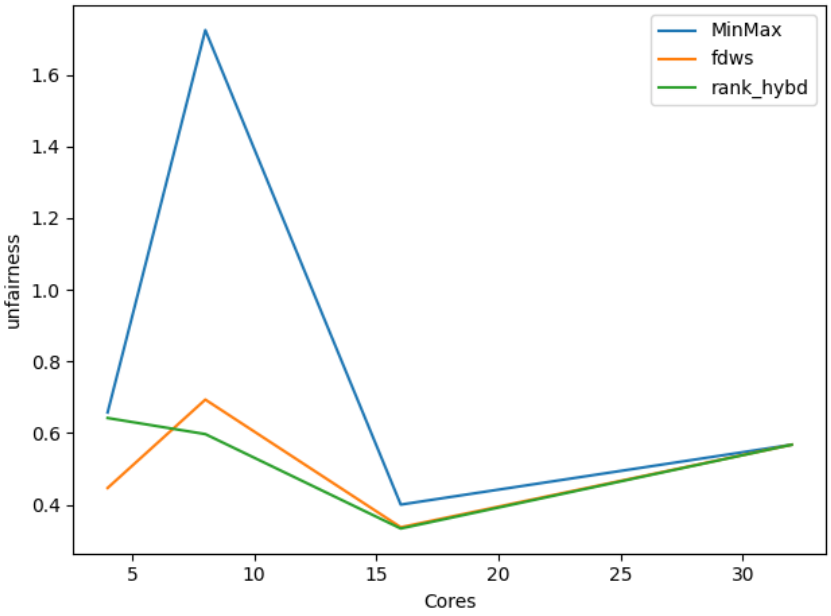
حال یک تستر برای برنامه درست می‌کنیم که عملا ۱۲ نمودار مختلف برای ما رسم می‌کند که در زیر آنها را گذاشته‌ایم. کد تستر در قسمت زیر آمده است :

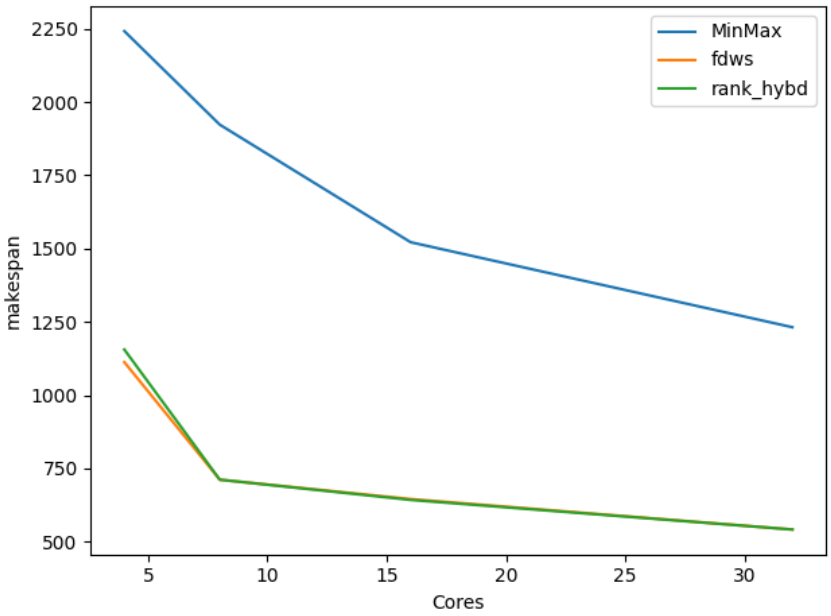
```
cores = [4, 8, 16, 32]
graphs = [4, 8, 16, 32]
tasks = [5, 10, 15]
algorithms = ['MinMax', 'fdws', 'rank_hybd']

for task in tasks:
    for metric in ['unfairness', 'makespan']:

        fig, axs = plt.subplots(1, 1)
        data = {alg: {} for alg in algorithms}
        for i, core in enumerate(cores):

            res = subprocess.run(['python', 'scheduler.py', 'GE', f'{
   task}', f'{core}', '8'], capture_output=True, text=True)
            splited_res = res.stdout.split('\n')


            for r in splited_res:
                if metric in r:
                    val = float(r.split(' ')[2])
                    alg = r.split(' ')[1]
                    data[alg][core] = val
        for alg, values in data.items():
            y = [values[i] for i in cores]
            plt.plot(cores, y, label=alg)


        plt.xlabel("Cores")
        plt.legend()
        plt.ylabel(f"{metric}")

        plt.tight_layout()
        plt.savefig(f'{metric}_cores_task_{task}.png')

```

```
for task in tasks:
    for metric in ['unfairness', 'makespan']:

        fig, axs = plt.subplots(1, 1)
        data = {alg: {} for alg in algorithms}
        for i, graph in enumerate(graphs):

            res = subprocess.run(['python', 'scheduler.py', 'GE', f'{task}', '8', f'{graph}'], capture_output=True, text=True)
            splited_res = res.stdout.split('\n')


            for r in splited_res:
                if metric in r:
                    val = float(r.split(' ')[2])
                    alg = r.split(' ')[1]
                    data[alg][graph] = val
        for alg, values in data.items():
            y = [values[i] for i in graphs]
            plt.plot(graphs, y, label=alg)


        plt.xlabel("Graphs")
        plt.legend()
        plt.ylabel(f"{metric}")

        plt.tight_layout()
        plt.savefig(f'{metric}_graphs_task_{task}.png')

```

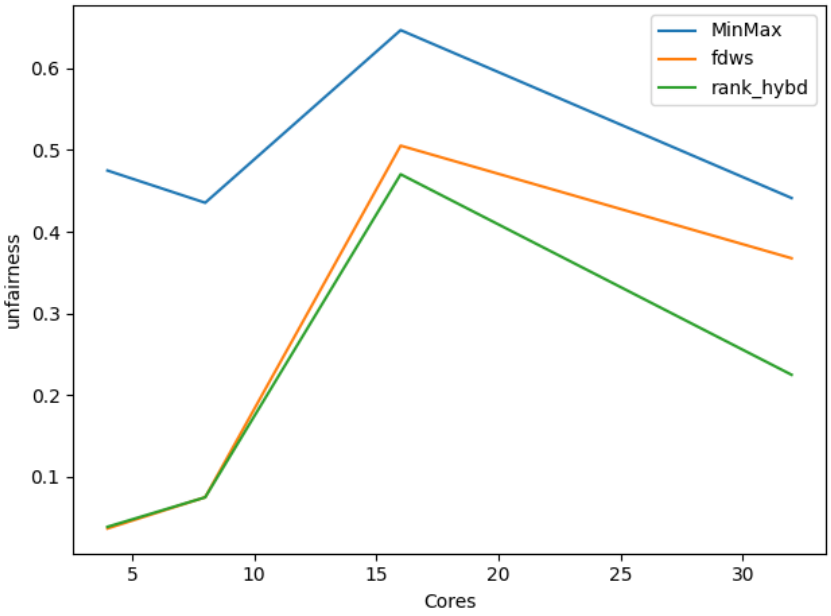
حال در قسمت زیر نتایج را مشاهده می‌کنیم :

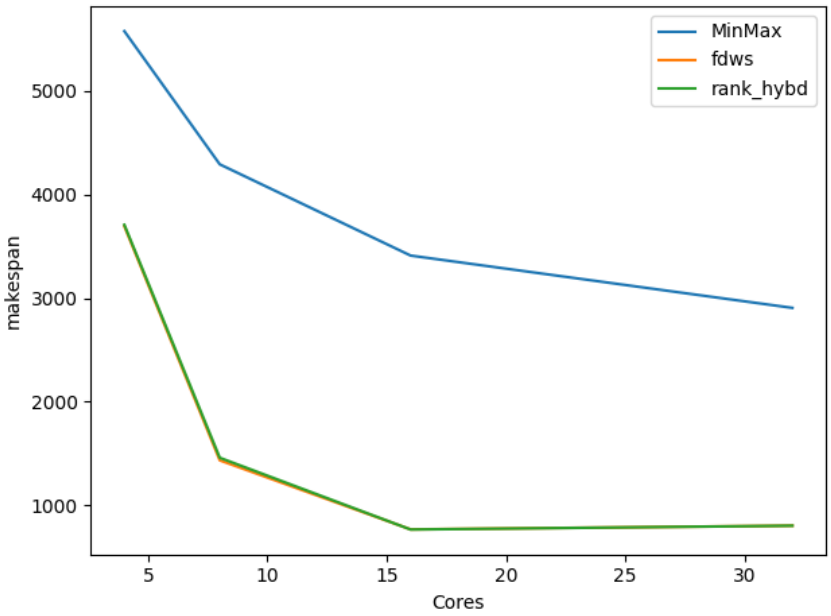
۱۴ تسک


MinMax
fdws
rank_hybd
unfairness
1.6
1.4
1.2
1.0
0.8
0.6
0.4
5
10
15
20
25
30
Cores

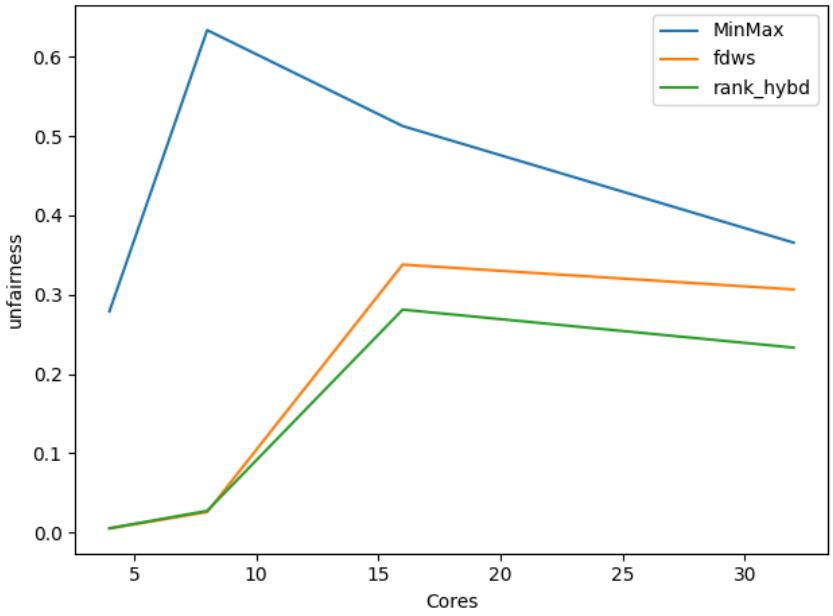

MinMax
fdws
rank_hybd
makespan
2250
2000
1750
1500
1250
1000
750
500
5
10
15
20
25
30
Cores

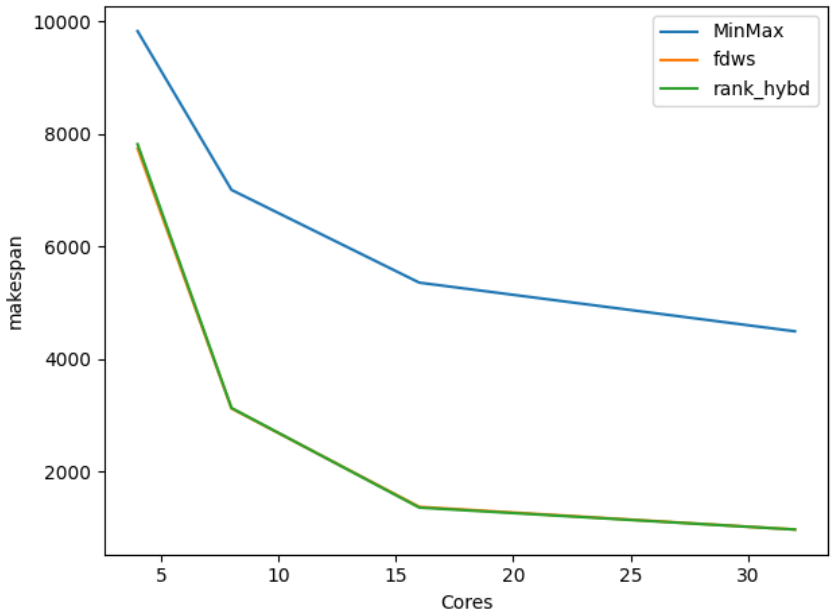
۵۴ تسک


MinMax
fdws
rank_hybd
unfairness
0.6
0.5
0.4
0.3
0.2
0.1
5
10
15
20
25
30
Cores

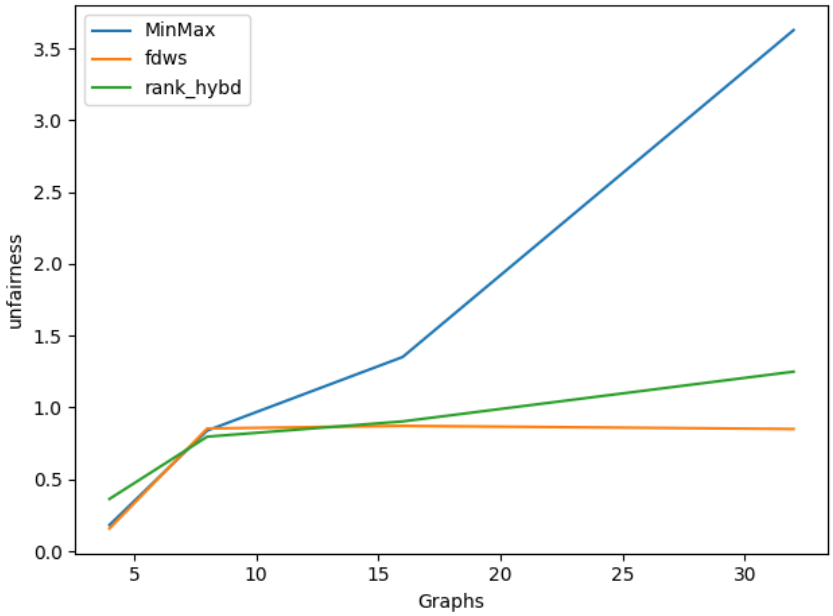

MinMax
fdws
rank_hybd
makespan
5000
4000
3000
2000
1000
5
10
15
20
25
30
Cores

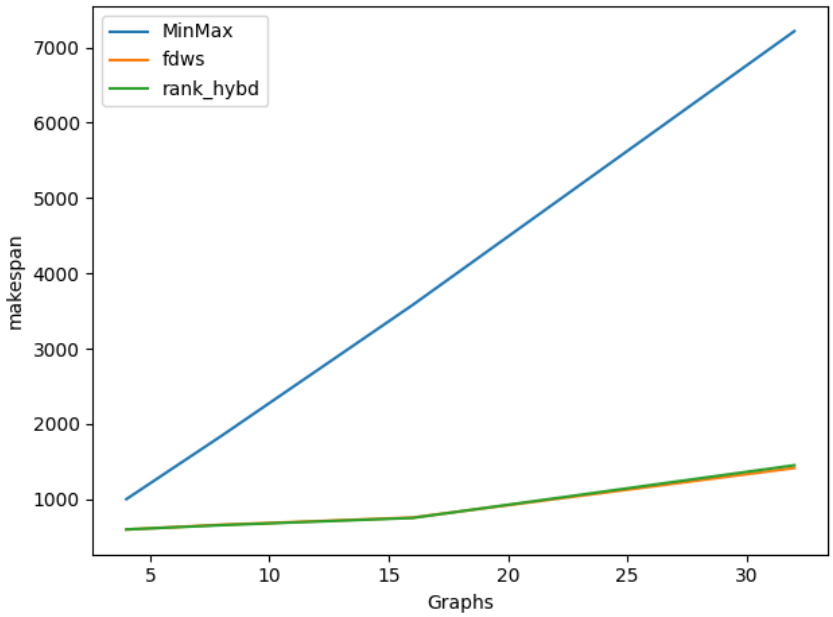
۱۱۹ تسک


MinMax
fdws
rank_hybd
unfairness
0.0
0.1
0.2
0.3
0.4
0.5
0.6
5
10
15
20
25
30
Cores

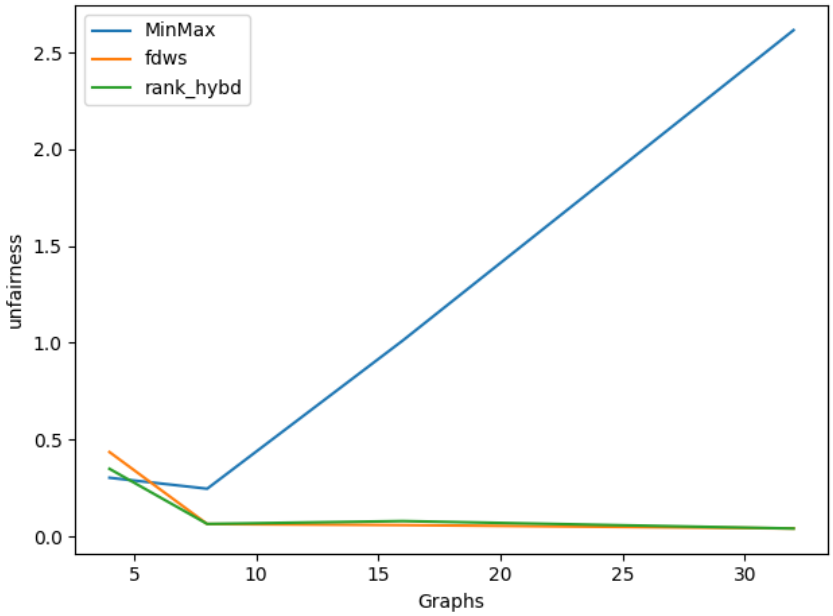

MinMax
fdws
rank_hybd
makespan
2000
4000
6000
8000
10000
5
10
15
20
25
30
Cores

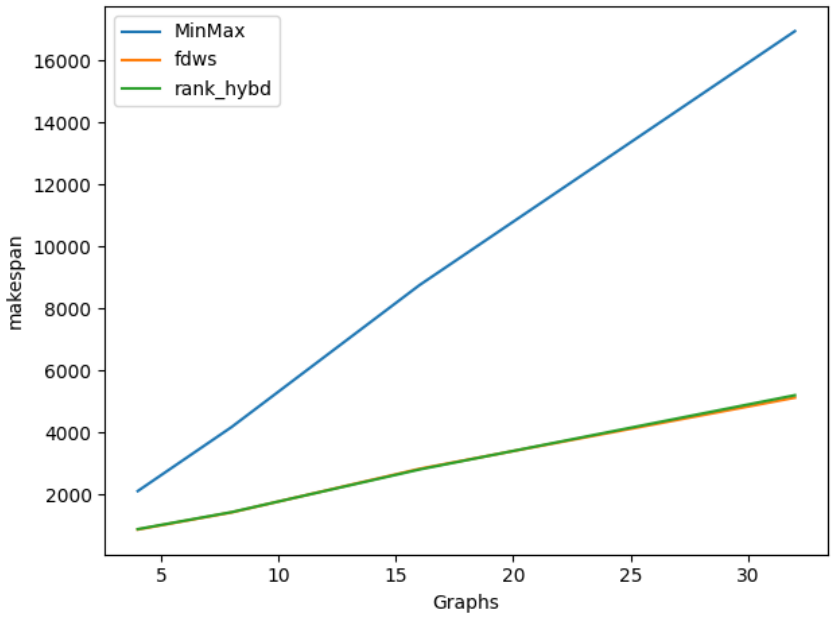
۱۴ تسک


MinMax
fdws
rank_hybd
unfairness
Graphs
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
5
10
15
20
25
30


MinMax
fdws
rank_hybd
makespan
Graphs
1000
2000
3000
4000
5000
6000
7000
5
10
15
20
25
30

۵۴ تسک





همانطور که در شکل‌های بالا مشاهده می‌شود، عملا مین مکس از نظر fairness بدترین عملکرد را داشته و نیز makespan آن از دو الگوریتم دیگر بیشتر است.