دانشگاه صنعتی شریف - تابستان 1404

گزارش پروژه شبیه سازی

استاد:

دکتر بردیا صفایی

دانشجو:

مهدیار احمدی زاده ۹۹۱۷۰۳۳۷

هدف پروژه:

این پروژه با هدف شبیه‌سازی یک شبکه هوشمند انرژی طراحی شده است. در این شبکه، منابع تولید انرژی شامل نیروگاه، پنل خورشیدی و باتری و موارد دیگر قرار دارند و در مقابل، مصرف‌کنندگان با نیازها و اولویت‌های متفاوت از این انرژی استفاده می‌کنند. یک کنترلر مرکزی وظیفه مدیریت درخواست‌های مصرف‌کنندگان را بر عهده دارد و این کار را از طریق الگوریتم‌های صف‌بندی مختلف مانند WRR، NPPS، FIFO و EDF انجام می‌دهد. هدف اصلی پروژه، ارزیابی و مقایسه عملکرد این الگوریتم‌ها در شرایط گوناگون است؛ به‌طوری‌که شاخص‌هایی همچون زمان انتظار، میزان بهره‌برداری از منابع تجدیدپذیر و تأخیر در تأمین انرژی به‌صورت دقیق اندازه‌گیری و تحلیل شوند.

روند انجام پروژه:

این پروژه شامل ۴ فایل می باشد .

1. Model در این فایل کلاس هایی که داریم تعریف کردیم:
    a. مشتری های ما که همان خانه ها می باشد
    b. منابع انرژي که میتواند باتری یا موارد دیگر باشد
    c. درخواست هایی که خانه ها برای انرژي دارند
    d. کنترلر مرکزی که جواب درخواست ها را میدهد.
2. Main که تعاریف اولیه اونجا می باشد و مدل خود را اونجا طراحی میکنیم
3. Simulation که در اینجا بخش منطق اصلی پروژه قرار دارد و درخواست ها اینجا درست میشود و اختصاص درخواست ها اینجا قرار دارد.
4. Analysis در این بخش خروجی هایی که تولید می شود را مقایسه میکنیم و خروجی هایی که از ما خواسته شده است را نشان می دهیم.

اگر بخواهم بگم روند انجام به چه شکلی می باشد ابتدا مدل وقتی تعریف شد ابتدا سراغ تولید درخواست ها می رویم که با توزیع پواسون درست می شوند و سپس بر اساس ترتیب ورودی تایم مرتب می شوند سپس بر اساس یکی از ۴ الگوریتم گفته شده یکی از درخواست ها انتخاب میشود و زمان مربوط به آن محاسبه میشود. در این بخش محاسبه اگر منبع تجدید پذیر یا تجدید ناپذیر باشد فرق میکند.

بعد از انجام آن خروجی هر کدام وارد فایل json مربوطه در بخش output میشوند و برای هر کدام میانگین زمان انتظار و عملکرد و میزان منبع تجدید پذیر محاسبه میشود.

بعد از محاسبه همه آنها نمودار آنها کشیده میشود.

توضیح کد:

فایل model

```
class Consumer:

   def __init__(self, name, priority=1, weight=1):

       self.name = name

       self.priority = priority

       self.weight = weight

       self.available_weight = weight

   def __str__(self):

       return
self.name+'-'+str(self.weight)+'-'+str(self.priority)+'-'+str(self.available_weight)

   def reset_available_weight(self):

       self.available_weight=self.weight
```

در اینجا مصرف کننده های ما مشخص شده اند که برای مثال میتونیم یک خانه تعریف کنیم که اسم دارد و یک اولویت و یک وزن دارد. اولویت آن برای الگوریتم NPPS مورد استفاده میباشد و وزن آن برای الگوریتم wrr مورد استفاده قرار میگیرد و الگوریتم wrr نیاز داریم یکی یکی وزن آنرا کم کنیم به همین علت available-weight قرار دادیم . دو تابع هم داریم در این کلاس که اولی آنها برای دیباگ استفاده شده و مورد دوم هم هر موقع در الگوریتم wrr هر موقع available-weight نیاز میشد آنرا برابر مقدار اولیه میگذاشتیم.

```
class Request:

   def __init__(self, consumer, amount, arrival_time, deadline=None):

       self.consumer = consumer

       self.amount = amount

       self.arrival_time = arrival_time

       self.deadline = deadline

       self.start_time = None

       self.finish_time = None

       self.status = "pending"

   def __str__(self):

                                                                   return
self.consumer.name+'-'+str(self.consumer.weight)+'-'+str(self.consumer.priority
)+'-'+str(self.start_time)+'-'+str(self.arrival_time)+'-'+str(self.deadline)
```

در این بخش هم هر درخواست داریم که هر درخواست باید مشخص باشد از کدام مصرف کننده می آید و میزان منبع انرژی مورد نیاز آن چقدر می باشد و زمان رسیدن و ددلاین و کی شروع میشود و کی تموم میشود باید مشخص باشد و همچنین نوع آنهم

باید مشخص باشد که آیا انجام شده یا خیر و اگر انجام نشده به چه دلیل انجام نشده است.

```
class Controller:

   def __init__(self):

       self.queue = []

   def add_request(self, request):

       self.queue.append(request)

   def clear(self):

       self.queue = []
```

این هم کلاس کنترلر مرکزی ما می باشد که یک صف دارد که به درخواست ها جواب دهد و کلا هم دوتا تابع دارد یکی اضافه کردن درخواست و یکی هم خالی کردن کل لیست.

```
class Producer:

   def __init__(self, name,prob, capacity, p_type="renewable"):

       self.name = name

       self.capacity = capacity

       self.available = capacity

       self.type = p_type   # "renewable" یا "nonrenewable"

       self.prob = prob

   def provide(self, amount):

       if self.available >= amount:

           self.available -= amount

           return amount

       else:

           taken = self.available
```

```
            self.available = 0
            return taken
    def reset(self):
        self.available = self.capacity
```

این هم کلاس تامین کننده های انرژی ما می باشد که نام دارد و میزان توان اولیه و توان باقی مانده و همچنین اینکه مدل آن تجدید پذیر یا تجدید پذیر نبودن آن می باشد و همچنین اینکه چند درصد از تامین های برق از باشد را مشخص میکند همچنین دو تا تباع داریم که یکی از آنها ریست می باشد و یکی هم اینکه چقدر برق میتوانیم از این منبع بگیریم.

فایل main

```
producers = [
    Producer("Solar", 0.2, capacity=80, p_type='renewable'),
    Producer("Battery", 0.4, capacity=50, p_type='renewable'),
    Producer("Diesel", 0.4, capacity=5000, p_type='nonrenewable'),
]
if (sum(p.prob for p in producers) !=1):
    raise ValueError("sum of prob is not valid")


consumers = [
    Consumer("House1", priority=2, weight=5),
    Consumer("House2", priority=1, weight=1),
    Consumer("House3", priority=3, weight=3),
```

```
]



controller = Controller()

sim = Simulation(producers, consumers, controller, sim_time=40)

algorithms = ["FIFO", "NPPS", "WRR", "EDF"]



results = compare_algorithms(sim, algorithms, lambda1=1.5, lambda2=2.0)



for algo, metrics in results.items():

   print(f"\nalgorithm: {algo}")

   print(f"\tavg Waiting Time: {metrics['avg_wait']:.2f}")

   print(f"\tavg Response Time: {metrics['avg_resp']:.2f}")

   print(f"\tthroughput: {metrics['throughput']:.2f} req/unit time")

   renew_pct = renewable_usage(metrics["energy_usage"], ["Solar", "Battery"])

   print(f"\trenewable Usage: {renew_pct:.2f}%")



plot_comparison(results)
```

در این صفحه خیلی منطق انجام نشده است و فقط مقدار دهی های اولیه انجام شده است و تابع های مختلف و شی های مختلف صدا زده شده است و خروجی در کنسول چاپ شده است.

فایل analysis

این فایل تابع های مختلفی دارد که هر کدام در پایین توضیح دادم.

```
def average_waiting_time(requests):
    # print(len(requests))
    waits = []
    for r in requests:
        if r.start_time:
            waits.append(r.start_time - r.arrival_time)
        else:
            waits.append(r.deadline - r.arrival_time)
    return sum(waits) / len(waits)
```

در این بخش همه درخواست ها را داریم و میخواهیم زمان انتظار آنها را بدست بیاریم به این شکل که اگر انجام شده باشند زمان بین شروع درخواست و آمدن آنرا حساب میکنیم و اگر انجام نشده باشد میزان بین آمدن و ددلاین آن را حساب میکنیم و در آخر میانگین میگیریم.

```
def average_response_time(requests):
    responses = []
    for r in requests:
        if r.finish_time:
            responses.append(r.finish_time - r.arrival_time)
        else:
            responses.append(r.deadline - r.arrival_time)
    return sum(responses) / len(responses)
```

در این بخش هم میزان زمانی که در سیستم بوده است را حساب میکنیم که اگر انجام شده باشد زمان پایان اجرا و اگر انجام نشده باشد زمان ددلاین از زمان شروع کم میکنیم و میانگین آنها را در آخر حساب میکنیم.

```
def throughput(requests):

   completed = sum(1 for r in requests if r.finish_time)

   return completed / len(requests)
```

در این بخش می خواهیم عملکرد بررسی کنیم و تعداد درخواست های که پایان یافتن به کل را حساب میکنیم.

```
def renewable_usage(energy_usage, renewable_names):

    renewable = sum(energy_usage[name] for name in renewable_names if name in energy_usage)

   total = sum(energy_usage.values())

   return (renewable / total * 100)
```

در این بخش میزان انرژی که تجدید پذیر هستند را جدا حساب میکنیم و میزان کل انرژی ها هم حساب میکنیم تا بتوانیم میزان استفاده از این منابع مشخص باشد.

```
def requests_to_json(requests):

   data = []

   for r in requests:

       data.append({

           "arrival_time": r.arrival_time,

           "amount": r.amount,

           "consumer": r.consumer.name,

           "deadline": r.deadline,

           "start_time": r.start_time,

           "finish_time": r.finish_time,
```

```
            "status": r.status
        })
    return json.dumps(data, indent=4)
```

در این بخش کل درخواست ها را گرفته و در یک حلقه همه محتویات آن را در json میگذاریم تا بتوانیم آنها را در فایل خروجی بگذاریم.در پوشه output خروجی قابل مشاهده می باشد.

```
def plot_comparison(results):
    algos = list(results.keys())
    waits = [results[a]["avg_wait"] for a in algos]
    resps = [results[a]["avg_resp"] for a in algos]
    thrpts = [results[a]["throughput"] for a in algos]
    plt.figure(figsize=(12,4))
    plt.subplot(1,3,1)
    plt.bar(algos, waits)
    plt.title("average waiting time")
    plt.subplot(1,3,2)
    plt.bar(algos, resps)
    plt.title("average response time")
    plt.subplot(1,3,3)
    plt.bar(algos, thrpts)
    plt.title("throughput")
    plt.show()
```

این تابع با کمک gpt درست شده به این شکل که خروجی های result مشخص شده است و برای هر کدام از آن ۳ معیار یک نمودار میکشیم.

```
def compare_algorithms(sim, algorithms, lambda1, lambda2):
    results = {}
    sim.requests = []
    sim.generate_requests(chi=2)
    asly_requests = copy.deepcopy(sim.requests)
    asly__producer = copy.deepcopy(sim.producers)
    for algo in algorithms:
        sim.producers = copy.deepcopy(asly__producer)
        sim.requests = copy.deepcopy(asly_requests)
        res = sim.run(algorithm=algo, lambda1=lambda1, lambda2=lambda2)
        json_output = requests_to_json(res)
        with open("./output/"+algo+".json", "w") as f:
            f.write(json_output)
        results[algo] = {
            "avg_wait": average_waiting_time(res),
            "avg_resp": average_response_time(res),
            "throughput": throughput(res),
            "energy_usage": sim.energy_usage.copy()
        }
    return results
```

در این بخش در واقع شروع عملیات شبیه سازی است که ابتدا درخواست ها درست میشوند و برای هر کدام از الگوریتم ها مراحل تخصیص درخواست انجام میشود و خروجی های گرفته شده را به عنوان ورودی به بقیه تابع ها میدهیم تا مقایسه خروجی بتوانیم انجام دهیم.

فایل simulation

در این بخش دو کلاس داریم و دو تابع که در پایین توضیح داده شده است.

```
def exponential_distribution(lmbda, size):
    return np.random.exponential(1/lmbda, size)


def poisson_distribution(lmbda, size):
    return np.random.poisson(lmbda, size)
```

این دو تابع برای درست کردن توزیع پوآسون و نمایی می باشد و چون سینتکس آنرا بلد نبودم از gpt کمک گرفتم.

```
class Simulation:
    def __init__(self, producers, consumers, controller, sim_time=50):
        self.producers = producers
        self.consumers = consumers
        self.controller = controller
        self.sim_time = sim_time
        self.requests = []
        self.energy_usage = {p.name: 0 for p in producers}
```

این فایل یک کلاس simulation دارد که به عنوان کل شبیه سازی می باشد و کل عملیات در آن انجام میشود و در آن کنترلر و مصرف کننده و تامین کننده و درخواست هارا دارد.

```
def generate_requests(self, chi=2):
    arrival_times = poisson_distribution(chi, int(self.sim_time/1))
    for t in range(self.sim_time):
        num_reqs = arrival_times[t]
```

```
        # print(t,num_reqs)

        for _ in range(num_reqs):

            consumer = random.choice(self.consumers)

            amount = random.randint(1, 5)

            deadline = t + random.randint(5, 15)

            req = Request(consumer, amount, t, deadline)

            self.requests.append(req)
```

این تابع درخواست ها را درست میکند به این شکل که ابتدا بر اساس میزان زمان کل سرویس توزیع پواسون صدا زده میشود و مقدار ورودی آنهم میدهیم و سپس بر اساس تعداد درخواست ها در هر ثانیه یک درخواست جدید درست میشود که درخواست ساخته شده لحظه ورود آن زمان حال حاضر می باشد و میزان نیاز منبع انرژی بین ۱ تا ۵ در نظر گرفته میشود و میزان زمان ددلاین هم بین ۱ تا ۱۵ رندوم اختصاص داده میشود.

حال در تابع run اختصاص وظایف انجام میوشد.

```
def run(self, algorithm, lambda1, lambda2, t_delay=0, C=0):

    print(f"algorithm : {algorithm} ")

    current_time = 0

    processed_requests = []

    pending = []
```

در اینجا تابع تعریف میشود و مقادیر الگوریتم که نوع الگوریتم حال حاضر مشخص میکند و lambda2 , lambda1 متغیر هایی هستند که در ادامه توضیح میدهم.

میزان c , t_delay دو متغیر هستند که در خود پروژه گفته شده است.

سپس یک متغیر زمان داریم که بر اساس زمان جلو میرود و دو آرایه داریم یکی برای درخواست های تمام شده و یکی هم برای درخواست هایی که آمده اند ولی انجام نشده اند.

```
all_requests = sorted(self.requests, key=lambda r: r.arrival_time)

for r in all_requests:

   r.start_time = None

   r.finish_time = None

   r.status = "pending"
```

بعدش درخواست هارا بر اساس زمان مرتب میکنیم چون مهم هست بر اساس زمان جلو میرویم درخواست اول درخواستی باشد که اولین باشد از لحاظ زمان چون باعث میشود خیلی اردر زمانی کم شود.

```
while (all_requests or pending) and current_time <= self.sim_time:

   # print(len(all_requests),len(pending),len(processed_requests))

   while all_requests and all_requests[0].arrival_time <= current_time:

       pending.append(all_requests.pop(0))
```

بعدش یک حلقه میزنیم در کل درخواست هایی که هنوز انجام نشده اند.

در ابتدا بر اساس زمان حال حاضر درخواست هایی که آمده اند یعنی زمان رسیدن انها قبل از زمان حال حاضر می باشد را پیدا میکنیم و در متغیر pending ها میگذاریم.

```
for r in pending:

   # print(r.deadline)

   if  r.deadline <= current_time:

       r.status = "dropped_deadline"

       if r not in processed_requests:

           processed_requests.append(r)

pending = [r for r in pending if r.status == "pending"]
```

سپس در این بخش این لیست درخواست هارا میبینیم و آنهایی که ددلاین آنها گذشته اند را از داخل لیست حذف میکنیم.

```
if pending:
      #پایین تر توضیح دادم
else:
   current_time = current_time + 1
```

بعدش باید از بین درخواست ها بر اساس الگوریتم یکی را انتخاب کنیم فقط اگر هیچ درخواستی نبود به زمان حال حاضر یکی اضافه میکنیم.

```
if algorithm == "FIFO":
   req = min(pending, key=lambda r: r.arrival_time)
```

اگر fifo باشد اون درخواست انتخاب میکنیم که زمان رسیدن کمتری دارد.

```
elif algorithm == "NPPS":
   req = max(pending, key=lambda r: r.consumer.priority)
```

اگر npps باشد اون را انتخاب میکنیم که اولویت بالاتری دارد.

```
elif algorithm == "EDF":
   req = min(pending, key=lambda r: r.deadline)
```

اگر edf باشد اون را انتخاب میکنیم که ددلاین کمتری دارد.

```
elif algorithm == "WRR":
   valid_requests = [r for r in pending if r.consumer.available_weight > 0]
   if not valid_requests:
       for c in pending:
           c.consumer.reset_available_weight()
       continue
   req = max(valid_requests, key=lambda r: r.consumer.available_weight)
```

```
    req.consumer.available_weight -= 1
```

اگر wrr باشد و آن را انتخاب میکنیم که میزان avalable_weight بیشتری دارد و اگر هیچ کدام نبود میزان available_weight آپدیت میکنیم و برابر میزان اولیه میگذاریم و دوباره اینکارو میکنیم.

```
pending.remove(req)
```

بعد از اینکار ها درخواستی که باید اجرا شود مشخص می باشد پس آنرا از بین درخواست های اجرا نشده حذف میکنیم.

```
available_producers = [p for p in self.producers if p.available >= req.amount]

if not available_producers:

    req.status = "dropped_no_capacity"

    processed_requests.append(req)

    continue



weights_prods = [p.prob for p in available_producers]

producer = random.choices(available_producers, weights=weights_prods, k=1)[0]
```

بعدش از بین منابع اونی که میزان باقی مانده انرژی به این درخواست میخورد انتخاب میکنیم و انتخاب آن رندوم می باشد و میزان احتمال هر کدام از اول مشخص می باشد.

اگر هم هیچ منبعی نباشد بدون تخصیص انرژی برمیگردد.

```
service_time_ctrl = exponential_distribution(lambda1, 1)[0]

start_ctrl = max(req.arrival_time, current_time)

finish_ctrl = start_ctrl + service_time_ctrl + t_delay

# print(producer.type)

if producer.type == "renewable":

    service_time_src = exponential_distribution(lambda2, 1)[0]
```

```
else:
   service_time_src = 0.001


start_src = finish_ctrl + C
finish_src = start_src + service_time_src


if finish_src > self.sim_time:
   req.status = "dropped_sim_time"
   processed_requests.append(req)
   continue
```

در ادامه زمان اجرا درخواست درمیآوریم که ابتدا زمان در کنترلر که توزیع نمایی میباشد را مشخص میکنیم و در ادامه اگر تجدید پذیر باشد زمان نمایی دوم برای منبع حساب میکنیم و با جمع آنها و میزان دو متغیر ورودی زمان کل پایان درخواست مشخص می شود.

اگر هم از زمان کل بیرون زد بدون پایان یافتن درخواست شبیه سازی تموم میشود.

```
producer.available = producer.available - req.amount
self.energy_usage[producer.name]= self.energy_usage[producer.name] + req.amount
req.start_time = start_src
req.finish_time = finish_src
req.status = "processed"
current_time = finish_src
processed_requests.append(req)
```

در ادامه هم زمان هایی که بدست آوردیم و منبعی که انتخاب کرده ایم را مشخص میکنیم تا برای آنالیز به مشکل نخوریم.

```
for r in pending:
   if r.status == "pending":
       r.status = "dropped_remaining"
   if r not in processed_requests:
       processed_requests.append(r)
for r in all_requests:
   if r.status == "pending":
       r.status = "dropped_remaining"
   if r not in processed_requests:
       processed_requests.append(r)
```

در آخر هم درخواست هایی که انجام نشده اند را پایان میزنیم بدون اجرا که بتوانیم آنالیز انجام دهیم.