دانشکده مهندسی

گروه مهندسی کامپیوتر

گرایش هوش مصنوعی

الگویتم NSGA-II

استاد:

دکتر مجتبی روحانی

دانشجو:

علی گلی

آذر ماه

سال ۱۴۰۲

۱ الگوریتم

۱.۱ الگوریتم NSGA-II

NSGA-II، یک الگوریتم ابتکاری ژنتیک چند هدفه (MOGA) است که برای حل مسائل بهینه‌سازی با چندین هدف طراحی شده است. MOGA یا Multi-Objective Genetic Algorithm به معنای الگوریتم ژنتیکی چند هدفه است، که هدف اصلی آن بهینه‌سازی چندین معیار یا هدف به صورت همزمان است. NSGA-II یکی از معروف‌ترین و مؤثرترین الگوریتم‌های MOGA است.

نام NSGA-II مخفف Non-dominated Sorting Genetic Algorithm II است. این الگوریتم از مراحل مختلفی مانند مرتب‌سازی غیرمسلط (Non-dominated Sorting)، جمعیت غنی‌سازی (Crowding Distance Assignment) و عملگرهای ژنتیکی چندین هدفه استفاده می‌کند تا به جواب‌های بهتر و پارتو (پارتوی بهینه) در فضای چندین هدف برسد.

الگوریتم NSGA-II بر اساس مفهوم غیرمسلطی بودن (non-dominance) عمل می‌کند. یعنی، حلقه جستجو به دنبال جواب‌هایی است که هیچ‌کدام از آنها نسبت به یکدیگر در تمام اهداف بهتر نباشند. این انتخاب به معنای عدم وجود جوابی که در تمام اهداف بهتر از دیگر جواب‌ها باشد. با استفاده از NSGA-II، می‌توان به صورت همزمان به چندین هدف اصلی در مسائل بهینه‌سازی پاسخ داد و پیشنهادات متعددی برای تعادل بین اهداف مختلف ارائه کرد.

۱.۲ متد NSGA-II

الگوریتم NSGA-II یک الگوریتم ژنتیک چند هدفه است که برای حل مسائل بهینه‌سازی با چندین هدف طراحی شده است. در زیر، مراحل کلی کار NSGA-II توضیح داده شده‌اند:

- **مقدمه (Initialization):**

تولید یک جمعیت اولیه از پارتوها (نقاط در فضای جستجو) با استفاده از تولید تصادفی یا روش‌های خاص.

- **ارزیابی (Evaluation):**

ارزیابی هر پارتو در جمعیت بر اساس تعدادی از هدف‌های بهینه‌سازی.

- **غیرمسلطی سازی (Non-dominated Sorting):**

دسته‌بندی پارتوها به گروه‌های مسلط و غیرمسلط بر اساس رابطه مسلطی (در یک یا چند هدف)، به طوری که هیچ پارتویی در یک گروه نسبت به پارتوهای دیگر بهتر نباشد.

- **جمعیت فرعی (Crowding Distance Assignment):**

محاسبه فاصله جمعیت به عنوان یک معیار تنوع بین پارتوهای مسلط در هر گروه.

- **انتخاب (Selection):**

انتخاب پارتوهایی که بر اساس معیارهای غیرمسلطی سازی و فاصله جمعیت از جمعیت فرعی به عنوان والدین برگزیده می‌شوند.

- **تولید نسل بعدی (Crossover and Mutation):**

استفاده از عملگرهای ژنتیکی مانند جهش (Mutation) و چند نقطه‌ای یا ترکیب (Crossover) بر روی والدین برگزیده برای تولید نسل بعدی.

- **جمعیت جدید (Environmental Selection):**

ادغام پارتوهای تولید شده با جمعیت اصلی و اعمال مراحل مرتب‌سازی مسلطی و محاسبه فاصله جمعیت به منظور حفظ تنوع.

- **تکرار (Iterations):**

تکرار مراحل ۳ تا ۷ تا رسیدن به شرایط خاتمه (پایان) یا دستیابی به جواب بهینه‌تر.

۱.۳ کد NSGA-II

۱.۳.۱ متد های evaluation

۱.۳.۱.۱ متد Kursawe

این متد دارای ۳ مقدار تصمیم است (n_var) و ۲ تابع برای بهینه سازی است. بازه انتخاب شده از ۵- تا ۵ است.

```python
class Kursawe(Problem):
    def __init__(self):
        super().__init__(n_var=3, n_obj=2, n_constr=0, xl=np.array([-5, -5
, -5]), xu=np.array([5, 5, 5]))

    def _evaluate(self, x, out, *args, **kwargs):
        l = []
        for i in range(2):
            l.append(-10 * np.exp(-0.2 * np.sqrt(np.square(x[:, i]) + np.
square(x[:, i + 1]))))
        f1 = np.sum(np.column_stack(l), axis=1)
        f2 = np.sum(np.power(np.abs(x), 0.8) + 5 * np.sin(np.power(x, 3
)), axis=1)
        out["F"] = np.column_stack([f1, f2])

    def _calc_pareto_front(self, *args, **kwargs):
        return Remote.get_instance().load("pymoo", "pf", "kursawe.pf")

```

۱.۳.۱.۲ متد Schaffer

این متد طبق فرمول تکلیف نوشته شده و شامل ۲ تابع بهینه سازی است و مقادیر تصمیم گیری ۲ بعدی است.

```python
class Schaffer(Problem):
    def __init__(self):
        super().__init__(n_var=2, n_obj=2, n_constr=0, xl=np.array([-10, -10]), xu=np.array([10**5, 10**5]))

    def _evaluate(self, x, out, *args, **kwargs):
        f1, f2 = [], []
        for i in range(len(x)):
            f1.append(x[i][0]**2+x[i][1]**2)
            f2.append(((x[i][1]-2)**2) +((x[i][0]-2)**2))
        out["F"] = np.column_stack([f1, f2])
```

۱.۳.۱.۳ متد zdt1

این تابع طبق فرمول نوشته شده است و آرایه تصمیم گیر ۳۰ بعدی است شامل ۲ تابع بهینه ساز

```python
class zdt1(Problem):
    def __init__(self, **kwargs):
        super().__init__(n_var=30, n_obj=2, xl=0, xu=1, vtype=float, **kwargs)

    def _evaluate(self, x, out, *args, **kwargs):
        f1 = x[:, 0]
        g = 1 + 9.0 / (self.n_var - 1) * np.sum(x[:, 1:], axis=1)
        f2 = g * (1 - np.power((f1 / g), 0.5))

        out["F"] = np.column_stack([f1, f2])

    def _calc_pareto_front(self, n_pareto_points=100):
        x = np.linspace(0, 1, n_pareto_points)
        return np.array([x, 1 - np.sqrt(x)]).T

```

۱.۳.۱.۴ متد zdt2

این متد بر اساس فرمول تکلیف نوشته شده است و ابعاد همانند تابع قبلی است.

```python
class zdt2(Problem):
    def __init__(self, **kwargs):
        super().__init__(n_var=30, n_obj=2, xl=0, xu=1, vtype=float, **
kwargs)

    def _evaluate(self, x, out, *args, **kwargs):
        f1 = x[:, 0]
        c = np.sum(x[:, 1:], axis=1)
        g = 1.0 + 9.0 * c / (self.n_var - 1)
        f2 = g * (1 - np.power((f1 * 1.0 / g), 2))

        out["F"] = np.column_stack([f1, f2])

    def _calc_pareto_front(self, n_pareto_points=100):
        x = np.linspace(0, 1, n_pareto_points)
        return np.array([x, 1 - np.power(x, 2)]).T

```

۱.۳.۱.۵ تابع zdt3

این تابع نیز همانند توابع قبلی نوشته شده است.

```python
class zdt3(Problem):
    def __init__(self, **kwargs):
        super().__init__(n_var=30, n_obj=2, xl=0, xu=1, vtype=float, **
kwargs)

    def _evaluate(self, x, out, *args, **kwargs):
        f1 = x[:, 0]
        c = np.sum(x[:, 1:], axis=1)
        g = 1.0 + 9.0 * c / (self.n_var - 1)
        f2 = g * (1 - np.power(f1 * 1.0 / g, 0.5) - (f1 * 1.0 / g) * np.
sin(10 * np.pi * f1))

        out["F"] = np.column_stack([f1, f2])

    def _calc_pareto_front(self, n_points=100, flatten=True):
        regions = [[0, 0.0830015349],
                   [0.182228780, 0.2577623634],
                   [0.4093136748, 0.4538821041],
                   [0.6183967944, 0.6525117038],
                   [0.8233317983, 0.8518328654]]

        pf = []

        for r in regions:
            x1 = np.linspace(r[0], r[1], int(n_points / len(regions)))
            x2 = 1 - np.sqrt(x1) - x1 * np.sin(10 * np.pi * x1)
            pf.append(np.array([x1, x2]).T)

        if not flatten:
            pf = np.concatenate([pf[None,...] for pf in pf])
        else:
            pf = np.row_stack(pf)

        return pf

```

۱.۳.۱.۶ تابع zdt4

این تابع شامل ۲ تابع بهینه سازی است و آرایه تصمیم گیری ۱۰ بعدی است. همچنین مقادیر x بین ۰ تا ۱ می‌باشد.

```python
class zdt4(Problem):
    def __init__(self, **kwargs):
        super().__init__(n_var=10, n_obj=2, xl=0, xu=1, vtype=float, **kwargs)
        self.xl = -5 * np.ones(self.n_var)
        self.xl[0] = 0.0
        self.xu = 5 * np.ones(self.n_var)
        self.xu[0] = 1.0
        self.func = self._evaluate

    def _calc_pareto_front(self, n_pareto_points=100):
        x = np.linspace(0, 1, n_pareto_points)
        return np.array([x, 1 - np.sqrt(x)]).T

    def _evaluate(self, x, out, *args, **kwargs):
        f1 = x[:, 0]
        g = 1.0
        g += 10 * (self.n_var - 1)
        for i in range(1, self.n_var):
            g += x[:, i] * x[:, i] - 10.0 * np.cos(4.0 * np.pi * x[:, i])
        h = 1.0 - np.sqrt(f1 / g)
        f2 = g * h

        out["F"] = np.column_stack([f1, f2])
```

۱.۳.۱.۷ متد zdt6

این تابع نیز همانند قبلی نوشته شده است.

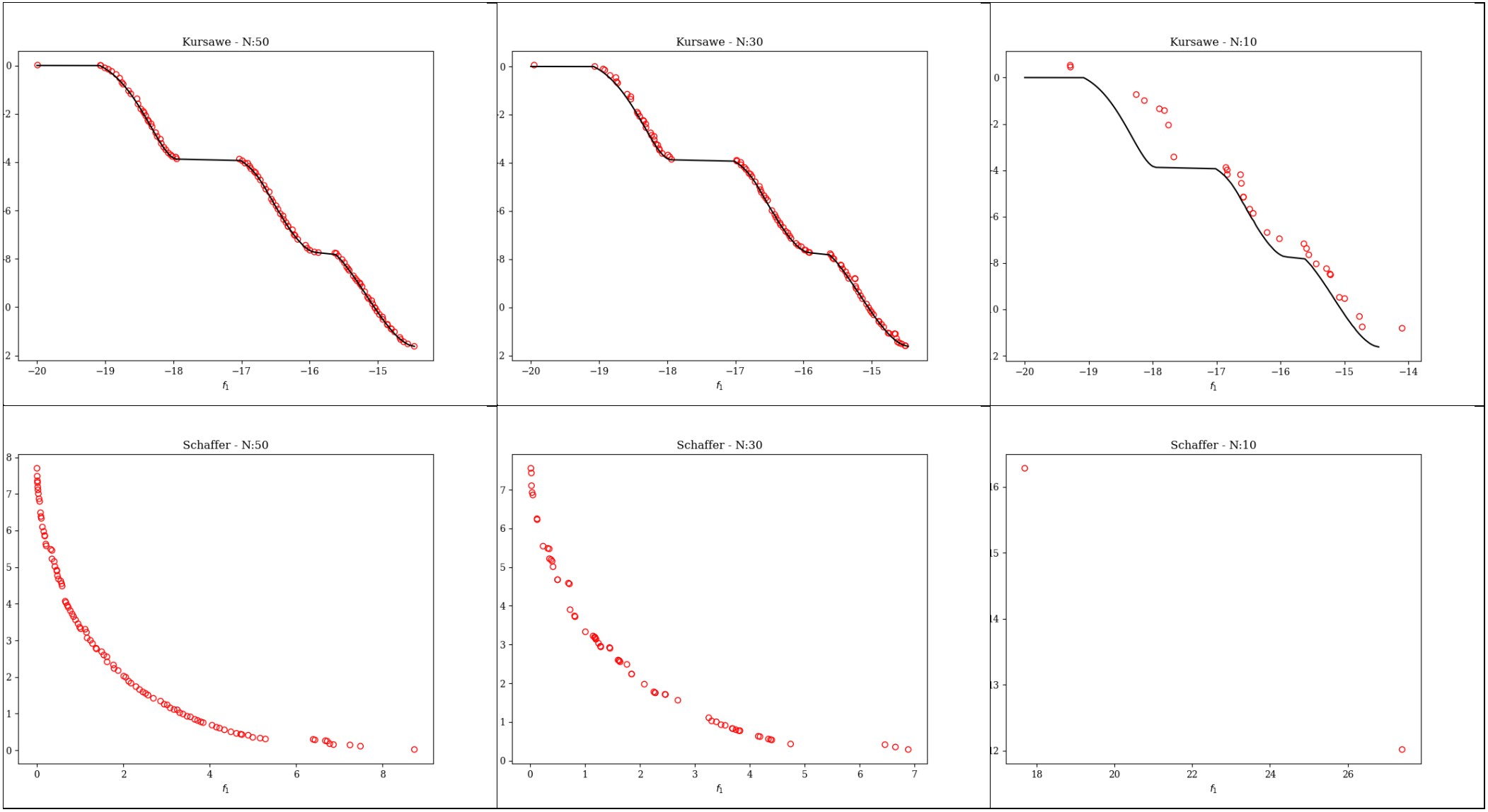
```python
class zdt6(Problem):
    def __init__(self, **kwargs):
        super().__init__(n_var=10, n_obj=2, xl=0, xu=1, vtype=float, **kwargs)

    def _evaluate(self, x, out, *args, **kwargs):
        f1 = 1 - np.exp(-4 * x[:, 0]) * np.power(np.sin(6 * np.pi * x[:, 0]), 6)
        g = 1 + 9.0 * np.power(np.sum(x[:, 1:], axis=1) / (self.n_var - 1.0), 0.25)
        f2 = g * (1 - np.power(f1 / g, 2))

        out["F"] = np.column_stack([f1, f2])

    def _calc_pareto_front(self, n_pareto_points=100):
        x = np.linspace(0.2807753191, 1, n_pareto_points)
        return np.array([x, 1 - np.power(x, 2)]).T
```

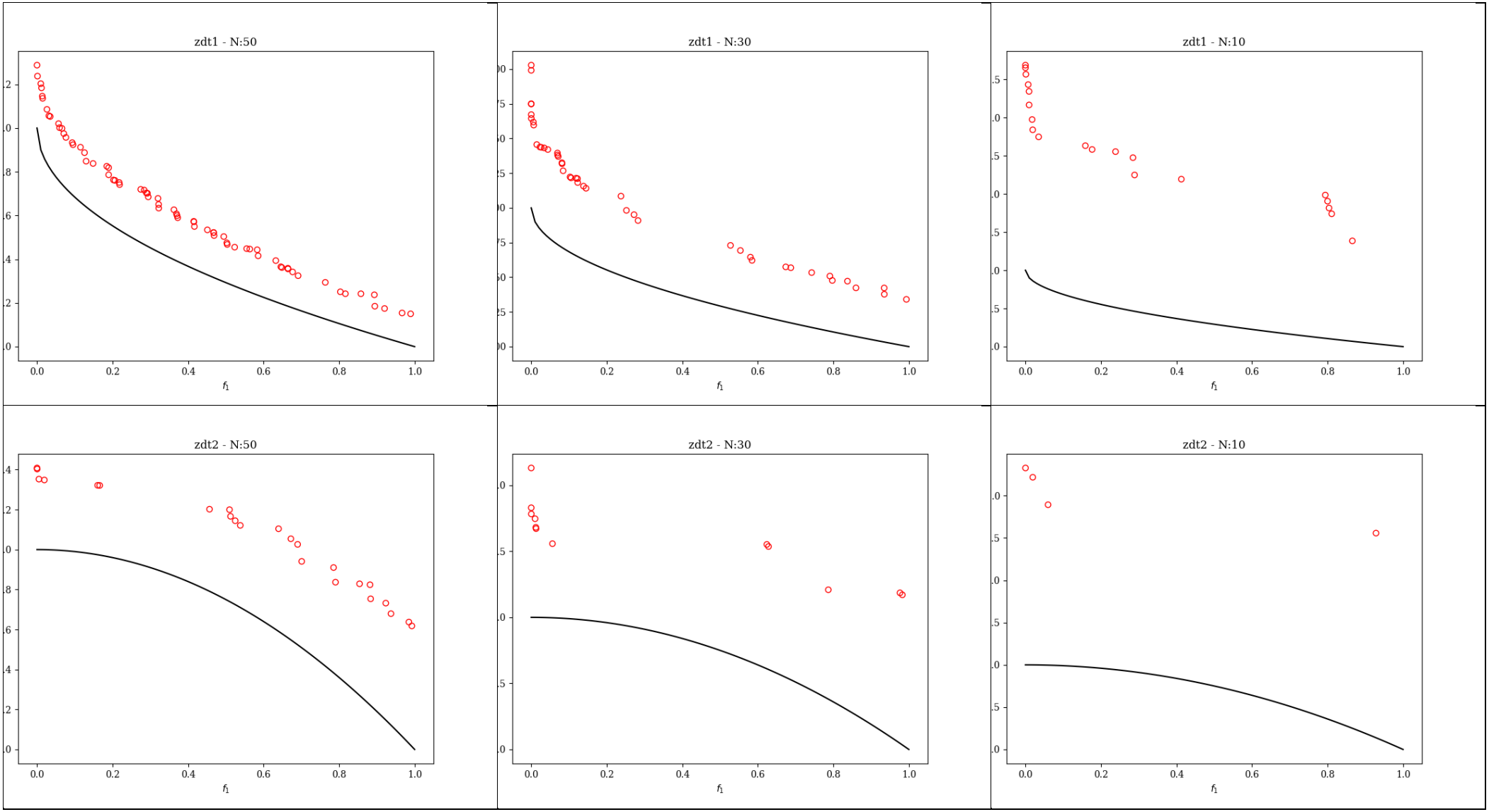
۱.۳.۲ پیاده سازی کد NSGA-II

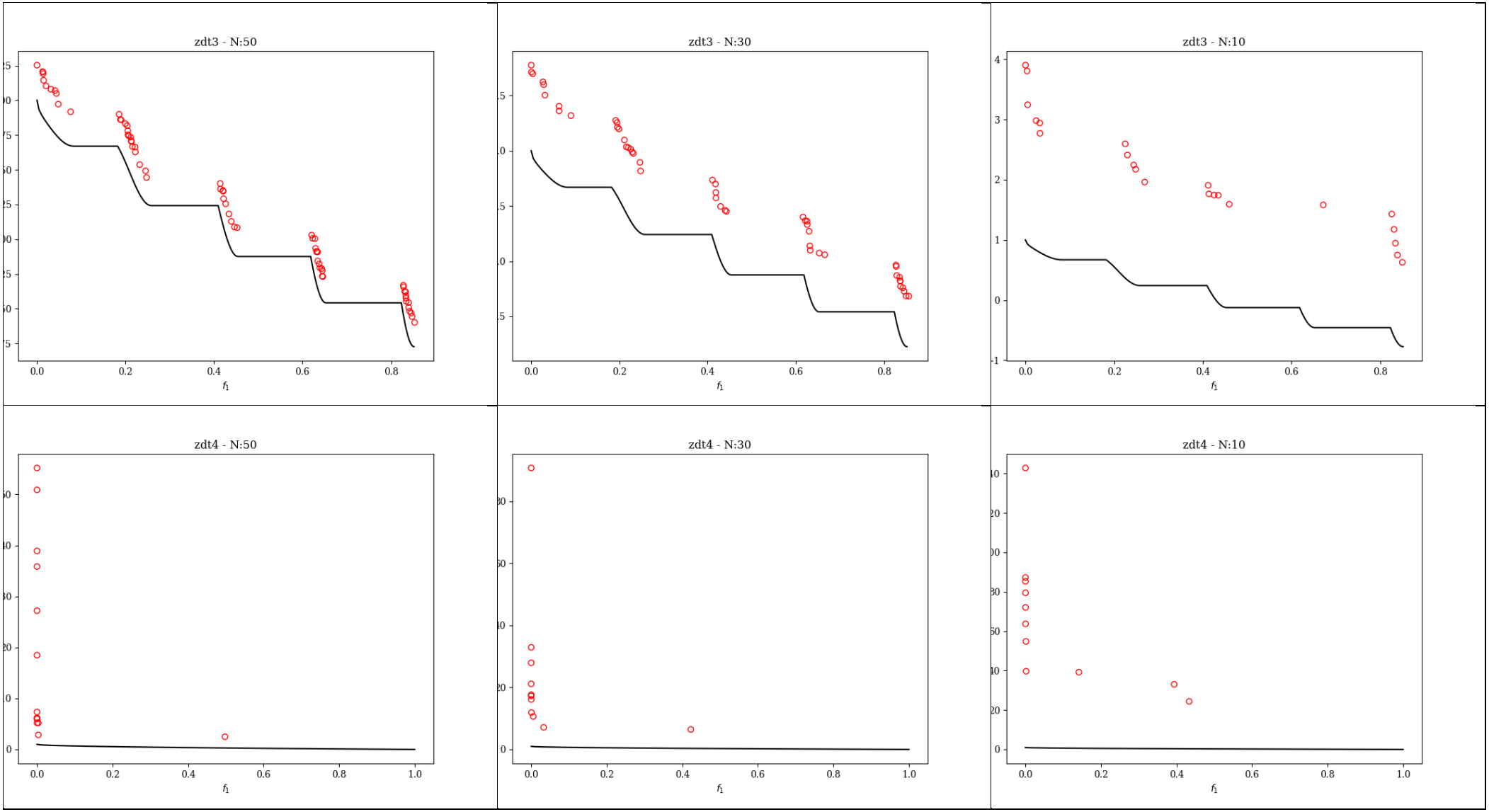
```
PROBLEM_CLASSES = [Kursawe(), Schaffer(), zdt1(), zdt2(), zdt3(), zdt4(), zdt6()]

for pr in PROBLEM_CLASSES:
    for N in [10, 30, 50]:
        problem = pr
        algortihm = NSGA2(pop_size=100)
        res = minimize(problem, algortihm, ('n_gen', N), seed=1)

        plot = Scatter()
        plot.title = f"{problem.__class__.__name__} - N:{N}"
        plot.add(problem.pareto_front(), plot_type='line', color='black')
        plot.add(res.F, facecolor='none', edgecolor='red')
        plot.show()

```

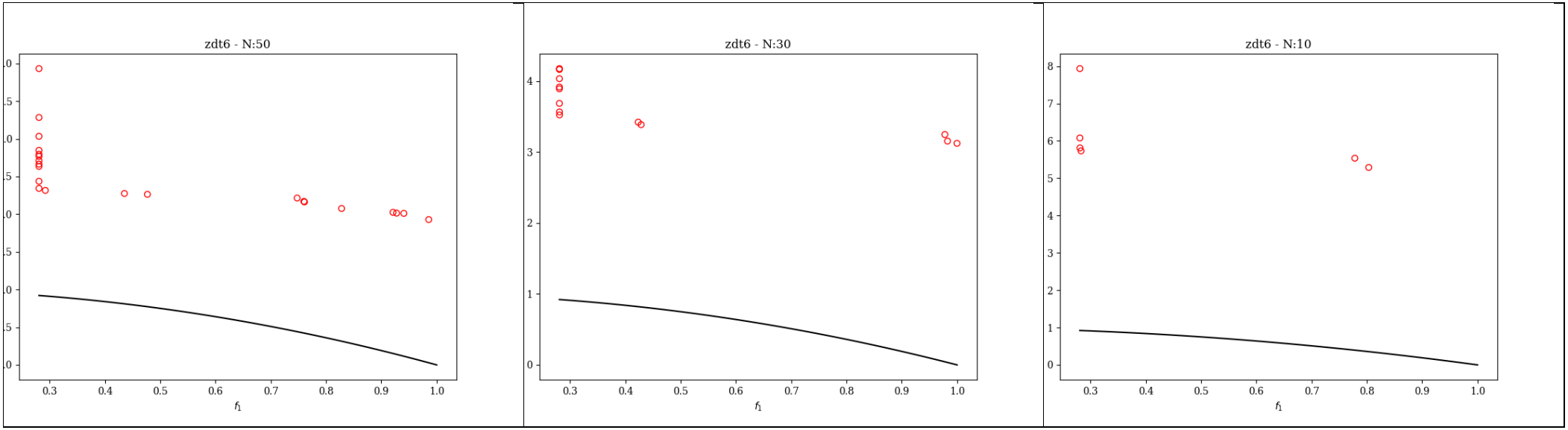
۱.۴ نتیجه کد ها

۱.۴.۱ نتیجه کد ها در صورت معیار توقف N



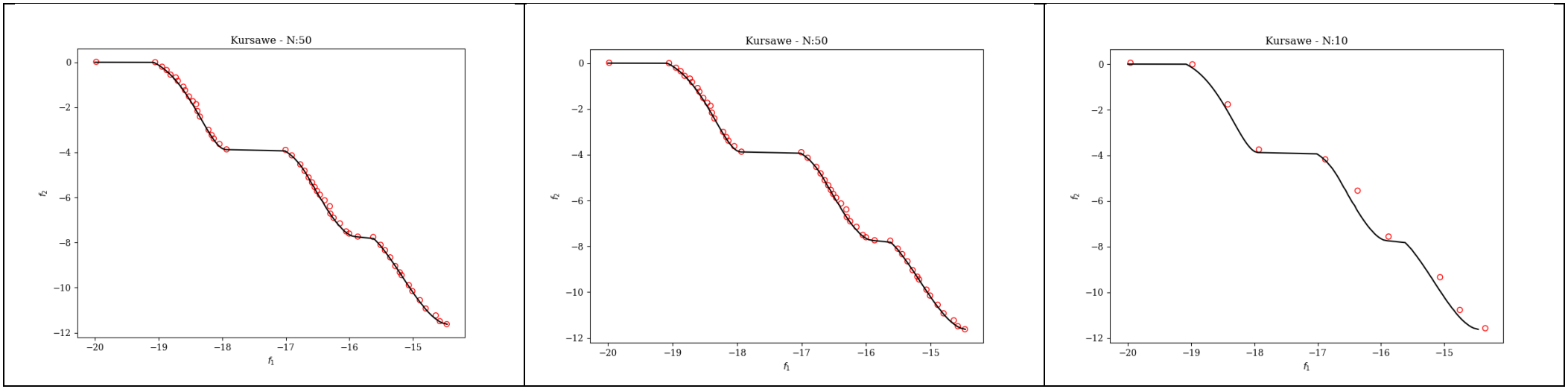

zdt1 - N:50
zdt1 - N:30
zdt1 - N:10
zdt2 - N:50
zdt2 - N:30
zdt2 - N:10
f_1

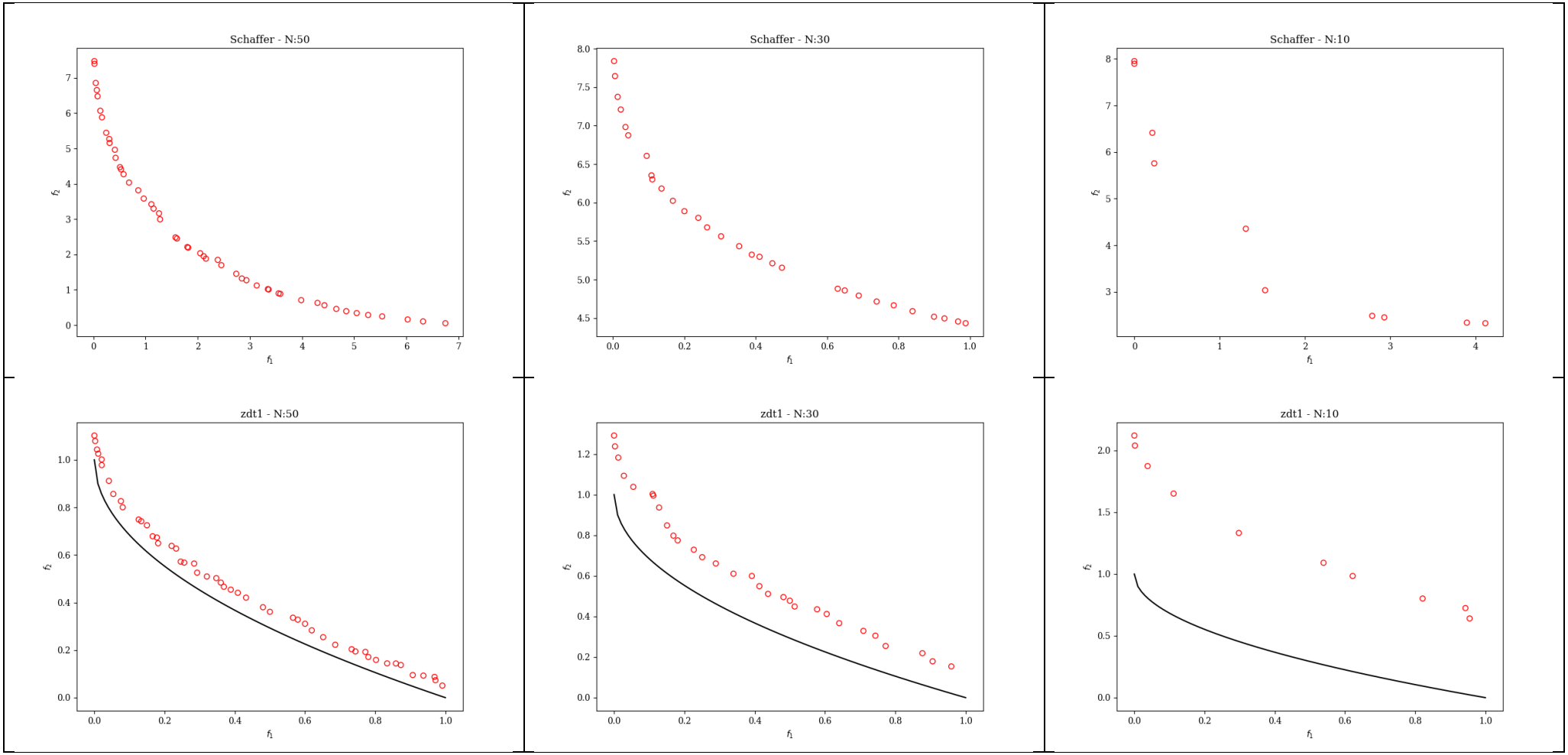

zdt3 - N:50
zdt3 - N:30
zdt3 - N:10
zdt4 - N:50
zdt4 - N:30
zdt4 - N:10
f_1

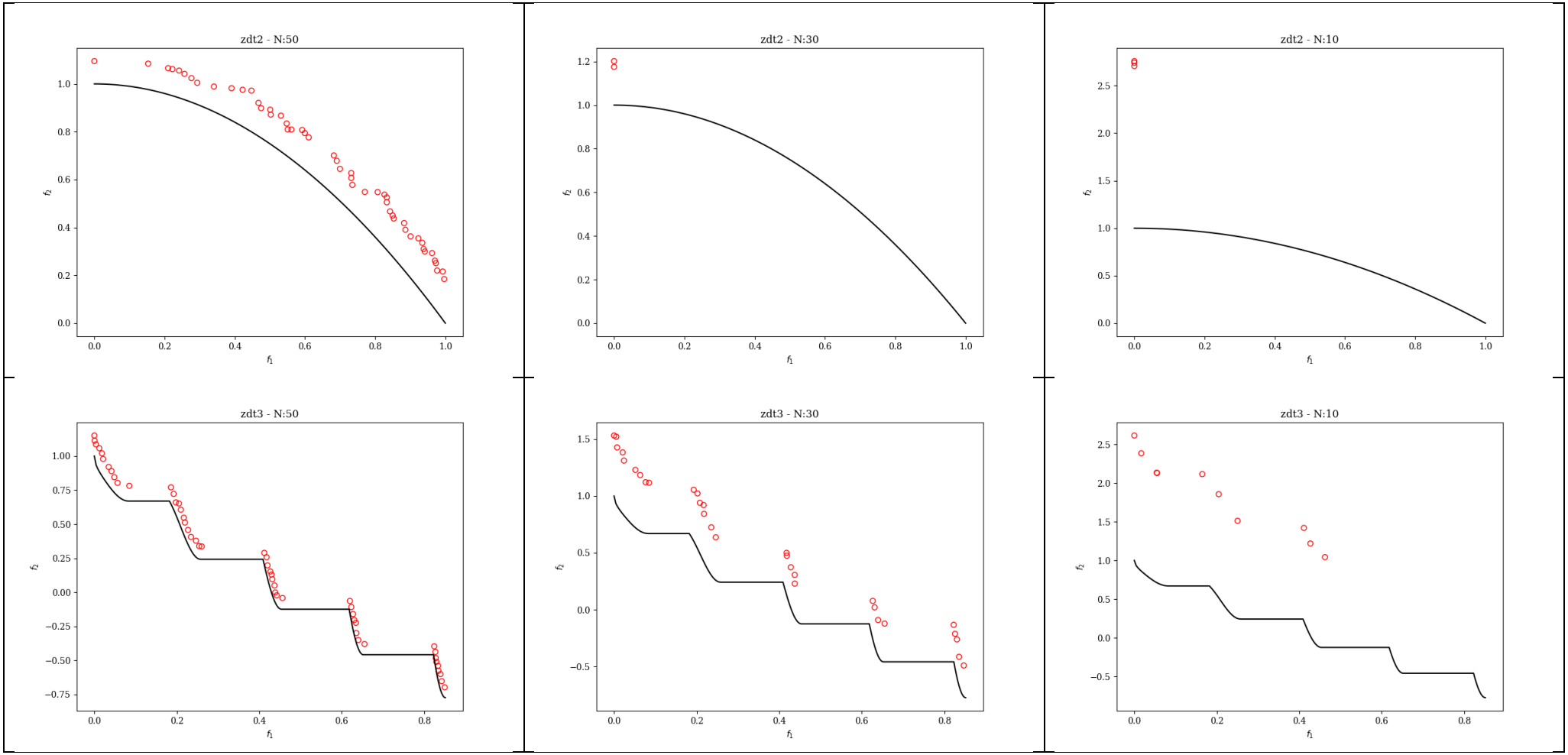


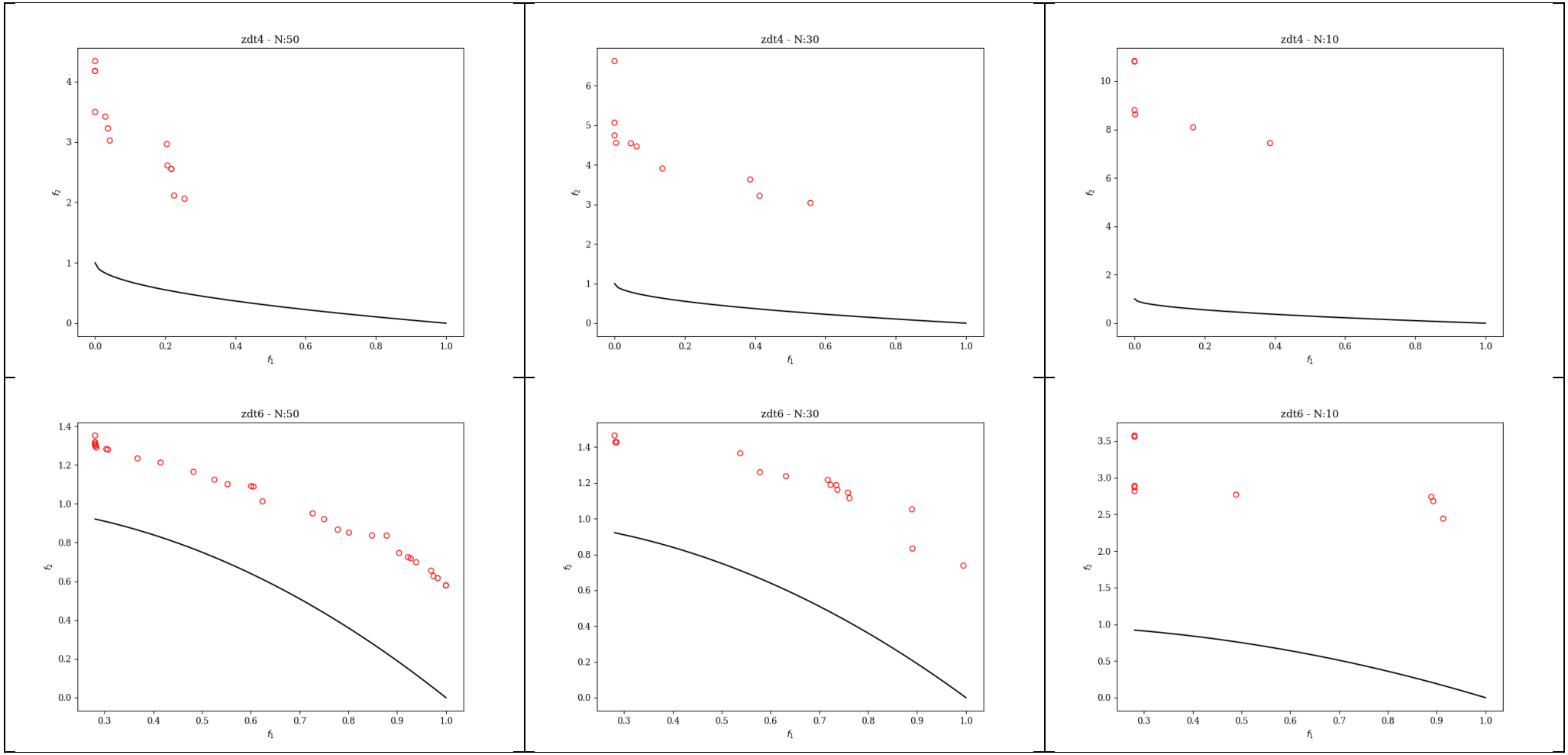
از آنجا که معیار توقف هربار افزایش می‌یابد باعث می‌شود که نزدیک تر به پارتو شویم و پراکندگی بیشتری را کشف کنیم. در بعضی متد های مانند Kursawe همان معیار توقف پایین نیز ما را به پارتو نزدیک می‌نماید و برای تابع zdt3 نیز قابل قبول است.

۱.۴.۲ نتیجه کد ها در صورت جمعیت برابر N




Schaffer - N:50
Schaffer - N:30
Schaffer - N:10
zdt1 - N:50
zdt1 - N:30
zdt1 - N:10
f_1
f_2


zdt2 - N:50
zdt2 - N:30
zdt2 - N:10
zdt3 - N:50
zdt3 - N:30
zdt3 - N:10
f_1
f_2



در جمعیت های کوچک diveristy رعایت نمی‌شود و نمی‌توانیم پراکندگی زیادی را مشاهده نماییم. در جمعیت های بالا فضای جستجو بالا می‌رود ولی هزینه محاسبه نیز بیشتر می‌شود. جمعیت های کوچک به لحاظ محاسباتی ارزان تر هستند ولی تمامی دگرگونی در فضای جستجو را نشان نمی‌دهد.