Лабораторная работа №1

Тема: Сложность алгоритмов и их оптимизация

Цель работы: Получить навыки вычисления сложности алгоритмов и оптимизации различными методами.

Алгоритм: Поиск в глубину (DFS)

Импорты

```
import random
import time
import tracemalloc
from tqdm import tqdm
import pandas as pd
```

Рекурсивный DFS

```
def dfs_recursive(graph, start, visited=None):
    if visited is None:
        visited = set()

    visited.add(start)

    for neighbor in graph[start]:
        if neighbor not in visited:
            dfs_recursive(graph, neighbor, visited)

    return visited
```

Временная сложность

Для ориентированного/неориентированного графа:

- каждый узел посещается один раз — `O(V)`
- каждое ребро рассматривается один раз — `O(E)`

Итого: `O(V+E)`

Пространственная сложность

- рекурсивный стек: глубина до `V`
- множество посещённых: `O(V)`

Итого: `O(V)`

Итеративный DFS

```
def dfs_iterative(graph, start):
    visited = set()
    stack = [start]

    while stack:
        node = stack.pop()
        if node not in visited:
            visited.add(node)
            stack.extend(graph[node])

    return visited
```

Алгоритмическая оптимизация

Для DFS реальной алгоритмической оптимизации нет, так как асимптотика оптимальна `O(V+E)`.

Но возможны улучшения:

- Убрать рекурси, стек вручную (итеративный DFS)

Избавляемся от overhead рекурсии и ограничений глубины стека.

Преимущества:

- устойчив к глубине графа
- быстрее, т.к. нет рекурсивных вызовов

Генерация графа

```
def generate_graph(n, edges_per_node=5):
    graph = {i: [] for i in range(n)}
    for i in range(n):
        for _ in range(edges_per_node):
            graph[i].append(random.randint(0, n-1))
    return graph
```

Бенчмаркинг

```
def benchmark_recursive_n(graph, n_runs=1000):
    import tracemalloc
    import time

    times = []
    peaks = []

    for i in range(n_runs):
        tracemalloc.start()
```

```
        t0 = time.time()

        try:
            dfs_recursive(graph, 0)
        except RecursionError as e:
            tracemalloc.stop()
            continue

        t1 = time.time()
        current, peak = tracemalloc.get_traced_memory()
        tracemalloc.stop()

        run_time = t1 - t0
        times.append(run_time)
        peaks.append(peak)

    if len(times) == 0:
        return {'avg_time': None, 'avg_peak_mem': None}

    return {'avg_time': sum(times)/len(times), 'avg_peak_mem':
sum(peaks)/len(peaks)/1024}

def benchmark_iterative_n(graph, n_runs=1000):
    import tracemalloc
    import time

    times = []
    peaks = []

    for i in range(n_runs):
        tracemalloc.start()
        t0 = time.time()

        dfs_iterative(graph, 0)

        t1 = time.time()
        current, peak = tracemalloc.get_traced_memory()
        tracemalloc.stop()

        run_time = t1 - t0
        times.append(run_time)
        peaks.append(peak)

    if not times:
        return {'avg_time': None, 'avg_peak_mem': None}

    return {'avg_time': sum(times)/len(times), 'avg_peak_mem':
sum(peaks)/len(peaks)/1024}

n_values = [500, 1_000, 10_000]
results = []
```

```python
for n in tqdm(n_values):
    graph = generate_graph(n)

    rec = benchmark_recursive_n(graph, n_runs=1000)
    it = benchmark_iterative_n(graph, n_runs=1000)

    results.append({
        'graph_size': n,
        'recursive_avg_time': rec['avg_time'],
        'recursive_avg_peak_mem_KB': rec['avg_peak_mem'],
        'iterative_avg_time': it['avg_time'],
        'iterative_avg_peak_mem_KB': it['avg_peak_mem']
    })
```

```
100%|██████████| 3/3 [00:54<00:00, 18.27s/it]
```

Таблица результатов

```python
df = pd.DataFrame(results)
df
```


```
{"summary":"{\n  \"name\": \"df\",\n  \"rows\": 3,\n  \"fields\": [\n
{\n      \"column\": \"n_runs\",\n      \"properties\": {\n
\"dtype\": \"number\",\n        \"std\": 5346,\n        \"min\": 500,\
n        \"max\": 10000,\n        \"num_unique_values\": 3,\n
\"samples\": [\n          500,\n          1000,\n          10000\n
],\n        \"semantic_type\": \"\",\n        \"description\": \"\"\n
}\n    },\n    {\n      \"column\": \"recursive_avg_time\",\n
\"properties\": {\n        \"dtype\": \"number\",\n        \"std\":
0.004227788306291441,\n        \"min\": 0.0022176814079284666,\n
\"max\": 0.008196676969528199,\n        \"num_unique_values\": 2,\n
\"samples\": [\n          0.008196676969528199,\n
0.0022176814079284666\n        ],\n        \"semantic_type\": \"\",\n
\"description\": \"\"\n      }\n    },\n    {\n      \"column\":
\"recursive_avg_peak_mem_KB\",\n      \"properties\": {\n
\"dtype\": \"number\",\n        \"std\": 8.463365897759548,\n
\"min\": 54.29497265625,\n        \"max\": 66.2639794921875,\n
\"num_unique_values\": 2,\n        \"samples\": [\n
66.2639794921875,\n          54.29497265625\n        ],\n
\"semantic_type\": \"\",\n        \"description\": \"\"\n      }\
n    },\n    {\n      \"column\": \"iterative_avg_time\",\n
\"properties\": {\n        \"dtype\": \"number\",\n        \"std\":
0.010520863612324377,\n        \"min\": 0.0002978625297546387,\n
\"max\": 0.018640203952789307,\n        \"num_unique_values\": 3,\n
\"samples\": [\n          0.0002978625297546387,\n
0.0005396101474761963\n        ],\n        \"semantic_type\": \"\",\n
\"description\": \"\"\n      }\n    },\n    {\n      \"column\":
\"iterative_avg_peak_mem_KB\",\n      \"properties\": {\n
\"dtype\": \"number\",\n        \"std\": 429.99346614242876,\n
```

```
\"min\": 48.513328125,\n        \"max\": 794.48440625,\n
\"num_unique_values\": 3,\n        \"samples\": [\n
48.513328125,\n          50.9202578125\n        ],\n
\"semantic_type\": \"\",\n        \"description\": \"\"\n      }\
n    }\n  ]\n}","type":"dataframe","variable_name":"df"}
```