Лабораторная работа №13_17

По теме 6.1-6.7: «Алгоритмы, внешняя память, геометрия, нечёткие множества, теория игр, машинное обучение и нейронные сети.»

Выполнил студент Гогадзе Арсений Давидович,

1 курса ВШКМиС

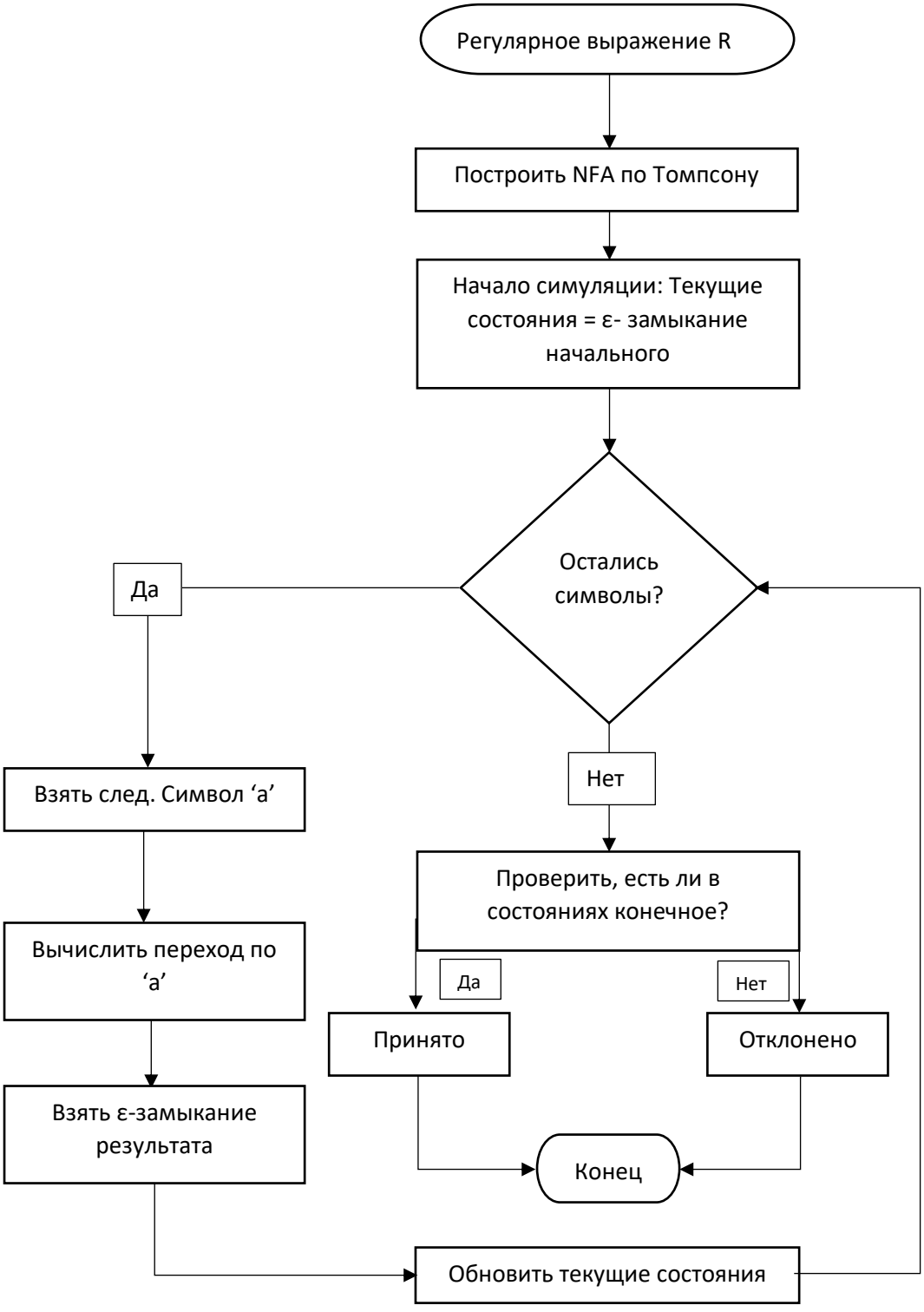
Группы ПИ04

Москва, 2025

Предметная область №1: «Автоматы и регулярные выражения — NFA»

Краткое описание области: Конечные автоматы используются для распознавания шаблонов в строках; существуют DFA (детерминированные) и NFA (недетерминированные). Thompson's construction даёт способ превратить регулярное выражение в NFA.

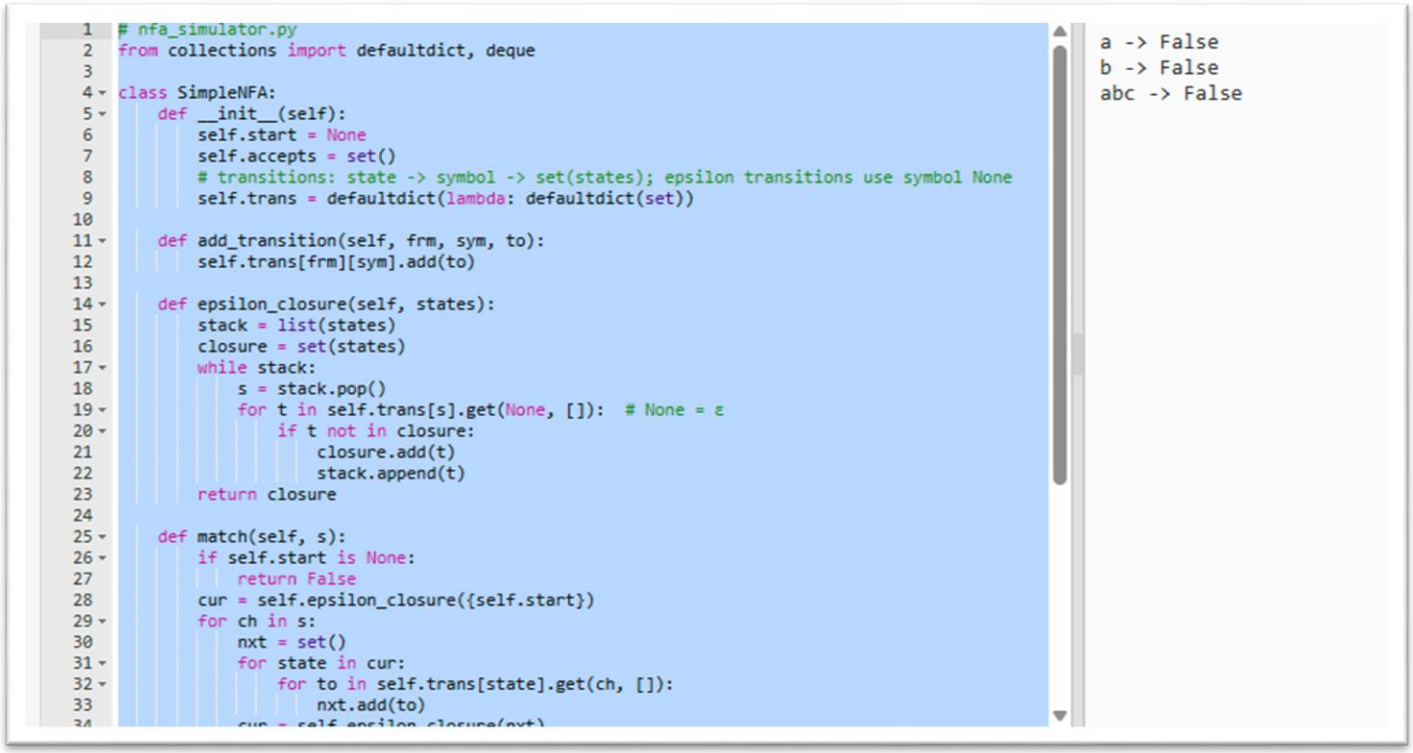
Блок-схема:



Пошаговый анализ:

1. Инициализировать множество текущих состояний стартовым состоянием (и его ε-замыканием).
2. Для каждого входного символа: для каждого текущего состояния взять переходы по этому символу; собрать новые состояния; расширить их ε-переходами (ε-closure).
3. После обработки всех символов проверить, встретилось ли какое-то принимающее состояние. Если да — строка принимается.

Результат:



```python
# nfa_simulator.py
from collections import defaultdict, deque

class SimpleNFA:
    def __init__(self):
        self.start = None
        self.accepts = set()
        # transitions: state -> symbol -> set(states); epsilon transitions use symbol None
        self.trans = defaultdict(lambda: defaultdict(set))

    def add_transition(self, frm, sym, to):
        self.trans[frm][sym].add(to)

    def epsilon_closure(self, states):
        stack = list(states)
        closure = set(states)
        while stack:
            s = stack.pop()
            for t in self.trans[s].get(None, []):  # None = ε
                if t not in closure:
                    closure.add(t)
                    stack.append(t)
        return closure

    def match(self, s):
        if self.start is None:
            return False
        cur = self.epsilon_closure({self.start})
        for ch in s:
            nxt = set()
            for state in cur:
                for to in self.trans[state].get(ch, []):
                    nxt.add(to)
            cur = self.epsilon_closure(nxt)
```

```
a -> False
b -> False
abc -> False
```

Оценка временной сложности:

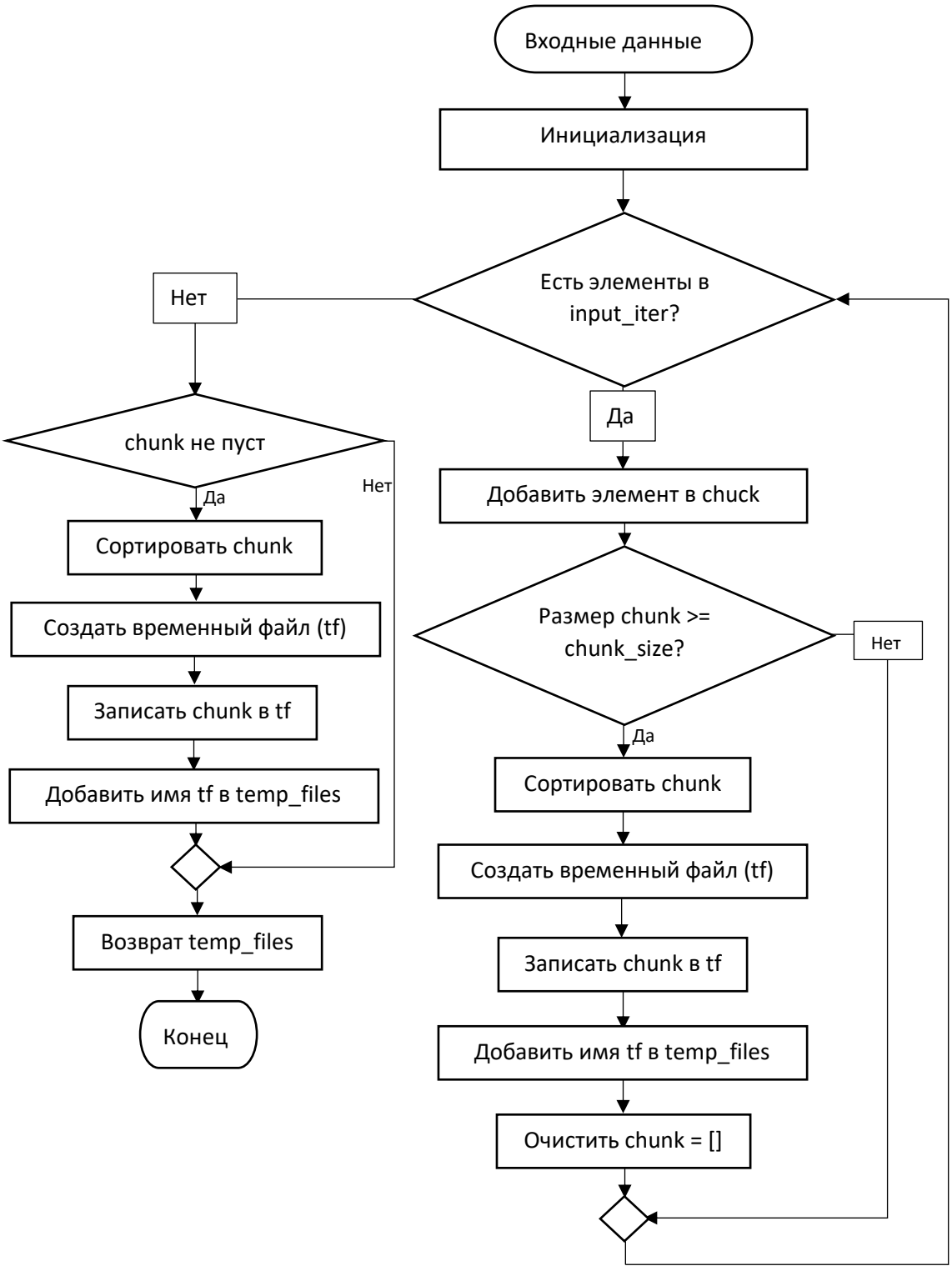
 Предел: для строки длины m и автомата с n состояниями, каждый шаг обработки символа требует обхода текущего множества состояний и вычисления ε-замыкания — в худшем случае O(n). Итого O(n·m).

 На практике для небольших NFA это быстро; Thompson's construction даёт NFA размер O(|r|).

Предметная область №2: «Алгоритмы во внешней памяти — External Merge Sort.»

Краткое описание области: Когда данные не помещаются в RAM, используются внешние алгоритмы: делим на блоки (чанки), сортируем каждый чанок во внутренней памяти, затем делаем k-ary (многостороннее) слияние по файлам. Сложность выражают через число блоков и операций I/O.

Блок-схема:



Пошаговый анализ:

1. Разделение входного файла на ceil(N/M) чанков, каждый размером ≤ M.

2. В памяти сортировка каждого чанка (любым подходящим алгоритмом).
3. Запись отсортированных чанков во временные файлы.
4. Открытие всех временных файлов и одновременное k-way слияние: поддерживать min-кучу (heap) первой строки из каждого файла; извлекать минимальный элемент, записывать в выход, читать следующий элемент из того же файла и помещать в кучу; повторять до опустошения.
5. Удаление временных файлов.

Результат:

main.py | numbers_sorted.txt

```
336         print(f"\nТест: {size} строк, лимит памяти: {mem_limit / (1024*1024):.1f} MB")
337
338         # Генерация файла
339         input_file = f'benchmark_{size}.txt'
340         output_file = f'benchmark_{size}_sorted.txt'
341
342         generate_large_file(input_file, size)
343
344         # Замер памяти и времени
345         process = psutil.Process()
346         start_time = time.time()
347         start_memory = process.memory_info().rss
348
349         # Сортировка
350         sorter = ExternalMergeSort(memory_limit=mem_limit)
351         sorter.sort_file(input_file, output_file)
352         sorter.cleanup()
353
354         end_time = time.time()
355         end_memory = process.memory_info().rss
356
357         print(f"  Время: {end_time - start_time:.2f} сек")
358         print(f"  Пиковая память: {(end_memory - start_memory) / (1024*1024):.2f} MB")
359
360         # Удаление временных файлов
361         os.remove(input_file)
362         os.remove(output_file)
363
```

input

```
Выберите тест:
  1. Сортировка чисел
  2. Сортировка строк
  3. Сортировка CSV
  4. Бенчмарк

Ваш выбор (1-4, Enter для в◆ех): 1

=== Тест сортиров◆ числовых данных ===
Временная директория: /tmp/external_sort_r_uj4zv8
Начало сортировки файла: numbers_input.txt
Размер файла: 0.33 MB
  Создана часть 1: 50000 строк, 336.3 KB
Создано 1 отсортированных частей
Сортировка завершена. Результат в: numbers_sorted.txt
Первые 10 чисел: [11, 18, 22, 23, 23, 24, 62, 73, 74, 98]
Проверка сортировки: True
Временная директория уд◆ена: /tmp/external_sort_r_uj4zv8

=== Все тесты завершены ===
```

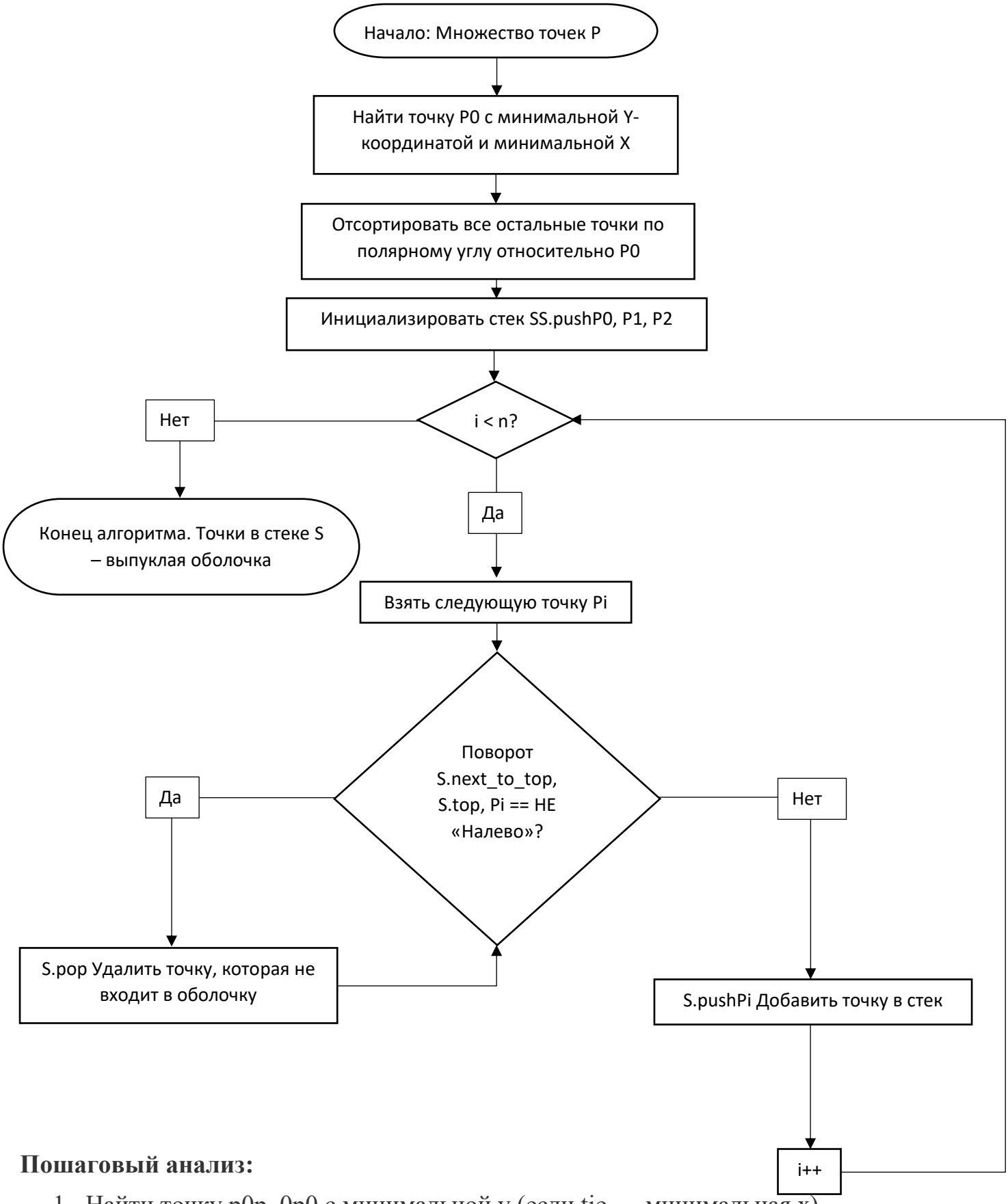
Оценка сложности:

- I/O-число операций: примерно O((N/B)*log(M/B)(N/B)) в модели внешней памяти (M — внутренняя память, B — размер блока). Для практической реализации время доминирует за счёт чтения/записи блоков.

Предметная область №3: «Вычислительная геометрия — Graham Scan»

Краткое описание области: Вычислительная геометрия решает задачи о точках, многоугольниках, диаграммах Вороного и т.д. Graham Scan строит выпуклую оболочку множества точек за O*(n*log n): сортировка по углу + проход стека.

Блок-схема:



Пошаговый анализ:

1. Найти точку p0p_0p0 с минимальной y (если tie — минимальная x).

2. Отсортировать остальные точки по полярному углу относительно p0p_0p0.
3. Инициализировать стек первыми 2–3 точками.
4. Перебирать остальные точки: пока последние два в стеке и новая точка делают неповорот влево (не CCW), убирать вершину стека; затем пушить новую точку.
5. В стеке останутся вершины выпуклой оболочки в порядке обхода.

Результат:

main.py

```
484
485 if __name__ == "__main__":
486     print("=== Алгоритм Грэхема (Graham Scan) для построения выпуклой оболочки ===")
487
488     # Демонстрационный пример
489     print("\n--- Демонстрационный пример ---")
490
491     # Генерируем случайные точки
492     points = generate_random_points(50, seed=42)
493
494     # Строим выпуклую оболочку
495     hull = GrahamScan.convex_hull(points)
496
497     print(f"Всего точек: {len(points)}")
498     print(f"Точек в выпуклой оболочке: {len(hull)}")
499     print(f"Выпуклая оболочка: {hull}")
500
501     # Визуализация
502     plot_convex_hull(points, hull, "Graham Scan - Демонстрация")
503
504     # Бенчмарк
505     benchmark_graham_scan()
506
507     # Тестирование граничных случаев
508     test_edge_cases()
509
510     # Интерактивная демонстрация
511     # interactive_demo()
```

input

```
=== Алгоритм Грэхема (Graham Scan) для построения выпуклой оболочки ===

--- Демонстрационный пример ---
Всего точек: 50
Точек в выпуклой оболочке: 12
Выпуклая оболочка: [(80.94, 0.65), (89.22, 8.69), (95.72, 33.66), (99.75, 50.95), (98.95, 64.00), (97.11, 86.08), (53.62, 97.31), (21.10, 94.29), (10.96, 62.74), (2.65, 1
/home/main.py:305: UserWarning: FigureCanvasAgg is non-interactive, and thus cannot be shown
  plt.show()
=== Бенчмарк алгоритма Грэхема ===
Точек:    100 -> Выпуклая оболочка:   12 точек Время:   0.25 мс
Точек:    500 -> Выпуклая оболочка:   19 точек Время:   2.20 мс
Точек:   1000 -> Выпуклая оболочка:   19 точек Время:   4.08 мс
Точек:   5000 -> Выпуклая оболочка:   27 точек Время:  11.29 мс
Точек:  10000 -> Выпуклая оболочка:   28 точек Время:  21.39 мс

=== Тестирование граничных случаев ===
1 точка: [(10.00, 20.00)] -> hull: [(10.00, 20.00)]
2 точки: [(10.00, 20.00), (30.00, 40.00)] -> hull: [(10.00, 20.00), (30.00, 40.00)]
3 точки (треугольник): hull: [(0.00, 0.00), (10.00, 0.00), (5.00, 10.00)]
Коллинеарные точки: hull: [(0.00, 0.00), (15.00, 15.00)]
Прямоугольник + внутренние точки: hull имеет 4 вершин


...Program finished with exit code 0
Press ENTER to exit console.
```

Оценка сложности:

 Сортировка точек по углу: O(n log n).

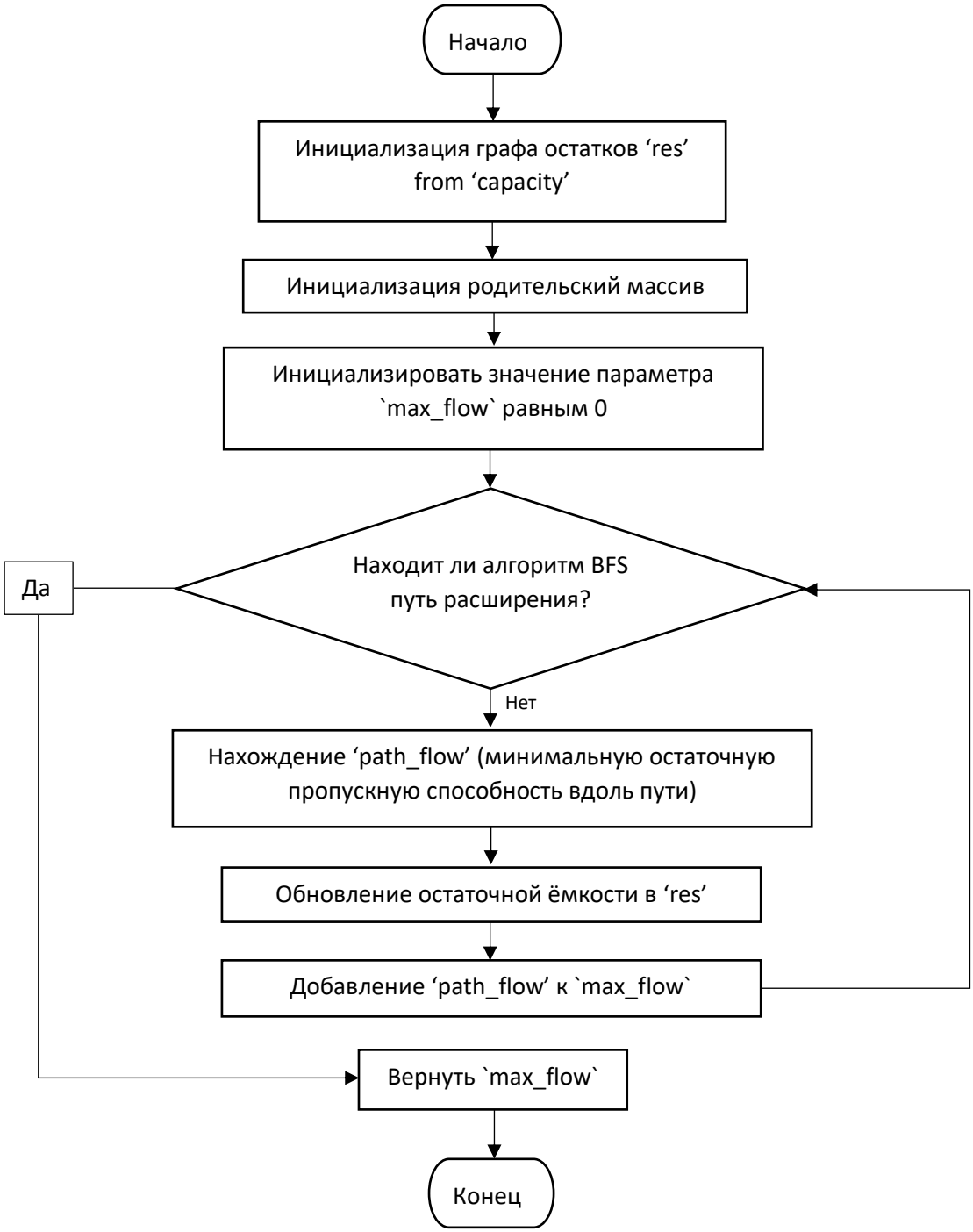
 Проход стека: O(n).
Итого O(n log n).

Предметная область №4: «Теория расписаний / потоки — Ford–Fulkerson / Edmonds–Karp»

Краткое описание области: Алгоритмы потоков ищут максимальный поток из источника в сток. Ford–Fulkerson ищет увеличивающие пути (любым методом),

Edmonds–Karp — частный случай с BFS (короткие по ребрам пути), что даёт полиномиальную оценку.

Блок-схема:



Пошаговый анализ:

1. Построить остаточную сеть равную исходной (capabilities).
2. Выполнять BFS, чтобы найти путь из s в t в остаточной сети (ребро с положительной пропускной способностью).

3. Если путь найден — вычислить минимальную пропускную способность на пути (path_flow), обновить остаточные ёмкости по пути (убавить в прямом направлении, прибавить в обратном), увеличить max_flow на path_flow.
4. Повторять, пока не останется увеличивающих путей.

Результат:

main.py

```
        print("Неверный выбор, использую BFS")
        max_flow = network.max_flow_bfs(source, sink)
        algo_name = "BFS"

    print(f"\n{algo_name}: максимальный поток = {max_flow}")

    # Показываем потоки
    print("\nПотоки на ребрах:")
    for edge in network.edges:
        if edge.flow > 0 or edge.capacity > 0:
            print(f"  {edge.u} → {edge.v}: {edge.flow:.1f}/{edge.capacity:.1f}")

    # Визуализация
    visualize = input("\nВизуализировать сеть? (y/n): ").strip().lower()
    if visualize == 'y':
        network.visualize(f"Максимальный поток = {max_flow}")


if __name__ == "__main__":
    print("=== Алгоритм Форда-Фалкерсона для поиска максимального потока ===")

    # Демонстрация на классическом примере
    solve_max_flow_problem()

    # Демонстрация приложений
    solve_bipartite_matching()
    solve_transport_problem()
```

input

```
=== Алгоритм Форда-Фалкерсона для поиска максимального потока ===

=== Пример 1: Классическая сеть ===
Исходная сеть:
  0 → 1: capacity = 16
  0 → 2: capacity = 13
  1 → 2: capacity = 10
  1 → 3: capacity = 12
  2 → 1: capacity = 4
  2 → 4: capacity = 14
  3 → 2: capacity = 9
  3 → 5: capacity = 20
  4 → 3: capacity = 7
  4 → 5: capacity = 4
/home/main.py:307: UserWarning: FigureCanvasAgg is non-interactive, and thus cannot be shown
  plt.show()

Максимальный поток из 0 в 5: 23.0

Потоки на ребрах:
  0 → 1: 12.0/16.0
  0 → 2: 11.0/13.0
  1 → 2: 0.0/10.0
  1 → 3: 12.0/12.0
  2 → 1: 0.0/4.0
  2 → 4: 11.0/14.0
```

Оценка сложности:

- Edmonds–Karp гарантирует O(VE^2) временной сложности (BFS даёт кратчайшие пути, число увеличений потоков ≤ O(VE)). Ford–Fulkerson без ограничения метода поиска может иметь экспоненциальный худший случай.