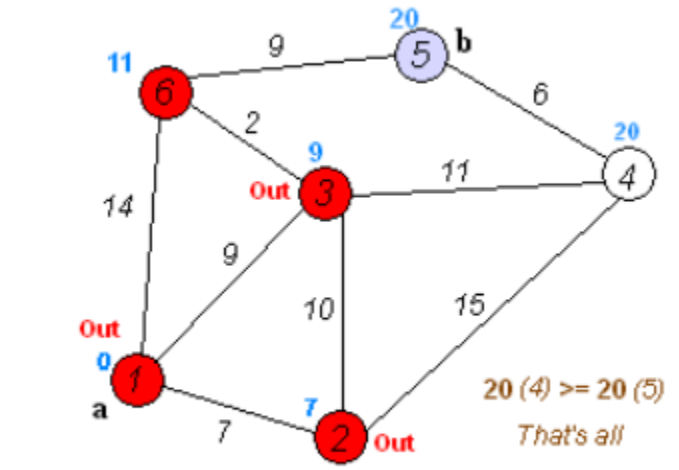
Реальная задача 2: “Библиотека для работы с графом”
В модуле graph.c реализовал функции:
struct Graph* graph_init(void); - инициализации графа
int graph_add_edge(struct Graph *graph, int start, int end, double weight); - добавления ребра
int graph_del_edge(struct Graph* graph, int start, int end); - удаления ребра
void print_graph(struct Graph* graph); - печати графа
int graph_add_vertex(struct Graph *graph, int vert); - добавления вершины
int graph_del_vertex(struct Graph *graph, int vert); - удаления вершины
int graph_check_vertex(struct Graph * graph, int vertex); - проверка наличия вершины (доп.)
int graph_search_edge(struct Graph * graph, int start, int end, int reload_flag); - проверки наличия ребра (доп.)
void graph_kill(struct Graph * graph); - деинициализации графа

В отдельном модуле(graph_algo.h) реализовал функцию
struct Graph* dijkstra(struct Graph * graph, int start, int end);
Она ищет кратчайший путь от вершины start до вершины end и возвращает граф, содержащий этот самый путь. (Возврщает в графе все кратчайшие пути, которые были вычислены в процессе.)

Задача:
Найти кратчайший путь в графе:



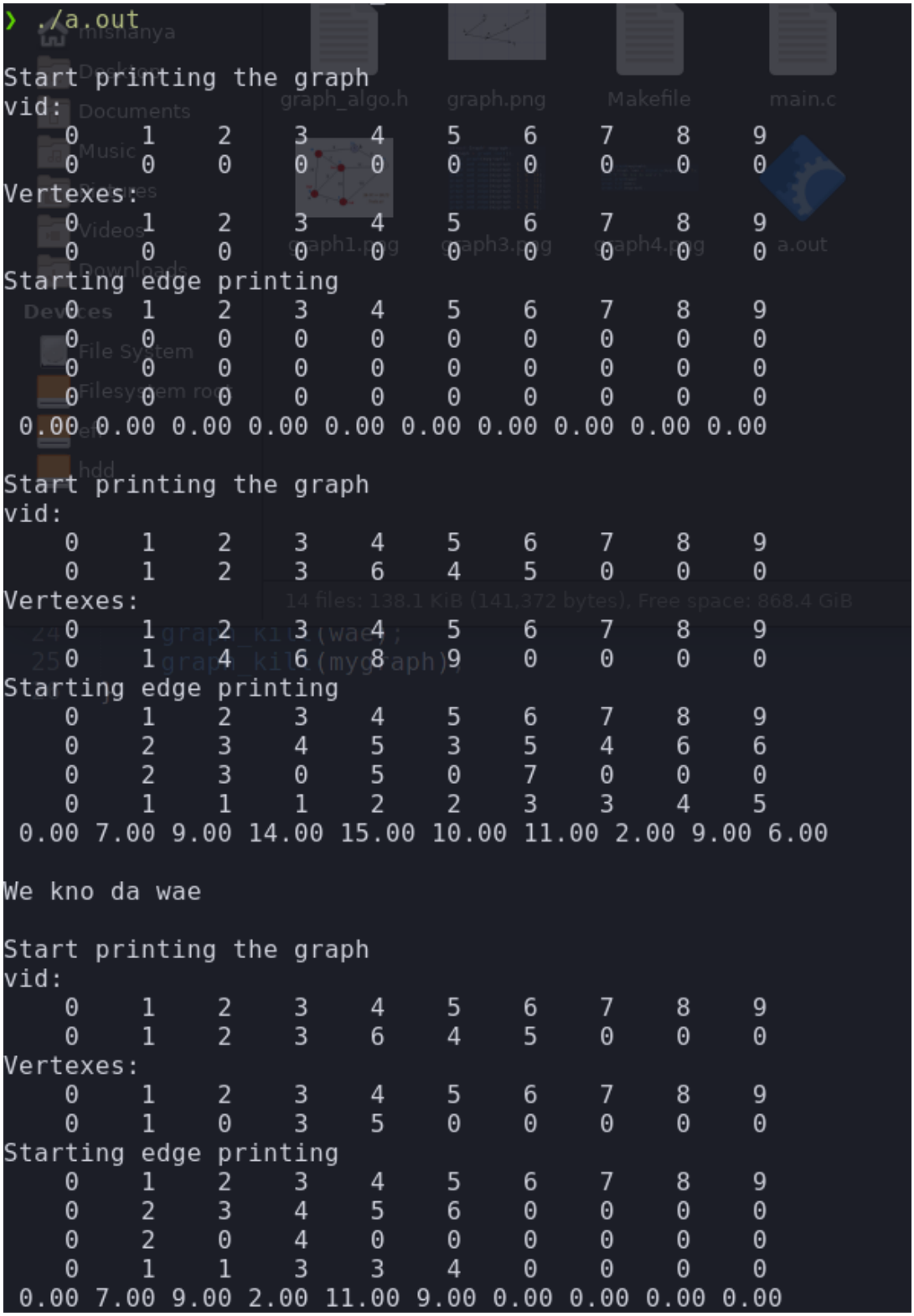
```
struct Graph* mygraph;
mygraph = graph_init();
print_graph(mygraph);
graph_add_edge(mygraph, 1, 2, 7);
graph_add_edge(mygraph, 1, 3, 9);
graph_add_edge(mygraph, 1, 6, 14);
graph_add_edge(mygraph, 2, 4, 15);
graph_add_edge(mygraph, 2, 3, 10);
graph_add_edge(mygraph, 3, 4, 11);
graph_add_edge(mygraph, 3, 6, 2);
graph_add_edge(mygraph, 6, 5, 9);
graph_add_edge(mygraph, 4, 5, 6);
```

Здесь мы создаём граф

```
struct Graph *wae = dijkstra(mygraph, 1, 5);
printf("\nWe kno da wae\n");
print_graph(wae);
graph_kill(wae);
graph_kill(mygraph);
```

Здесь мы печатаем полученный граф, вычисляем путь, печатаем путь, затем удаляем всё

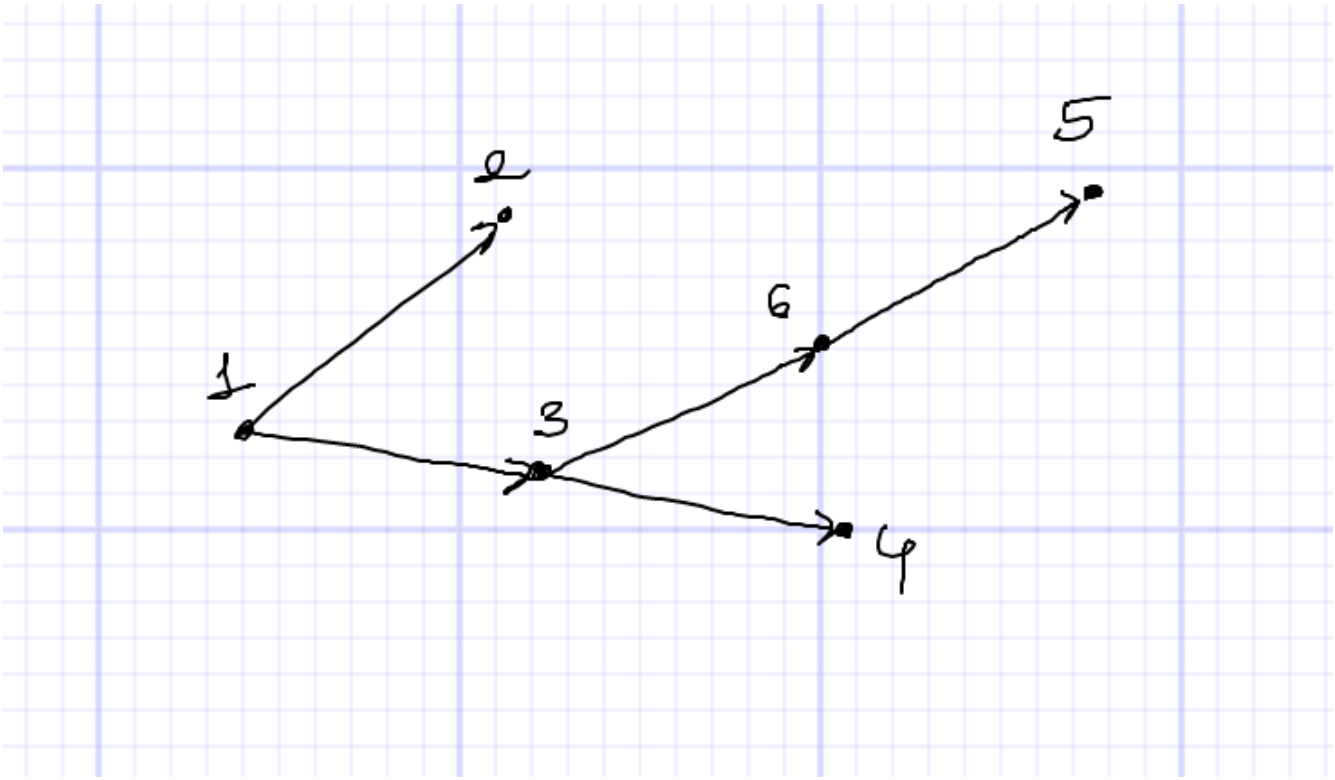
Получаем такой вывод:



```
❯ ./a.out

Start printing the graph
vid:
    0    1    2    3    4    5    6    7    8    9
    0    0    0    0    0    0    0    0    0    0
Vertexes:
    0    1    2    3    4    5    6    7    8    9
    0    0    0    0    0    0    0    0    0    0
Starting edge printing
    0    1    2    3    4    5    6    7    8    9
    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0
 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00

Start printing the graph
vid:
    0    1    2    3    4    5    6    7    8    9
    0    1    2    3    6    4    5    0    0    0
Vertexes:
    0    1    2    3    4    5    6    7    8    9
    0    1    4    6    8    9    0    0    0    0
Starting edge printing
    0    1    2    3    4    5    6    7    8    9
    0    2    3    4    5    3    5    4    6    6
    0    2    3    0    5    0    7    0    0    0
    0    1    1    1    2    2    3    3    4    5
 0.00 7.00 9.00 14.00 15.00 10.00 11.00 2.00 9.00 6.00

We kno da wae

Start printing the graph
vid:
    0    1    2    3    4    5    6    7    8    9
    0    1    2    3    6    4    5    0    0    0
Vertexes:
    0    1    2    3    4    5    6    7    8    9
    0    1    0    3    5    0    0    0    0    0
Starting edge printing
    0    1    2    3    4    5    6    7    8    9
    0    2    3    4    5    6    0    0    0    0
    0    2    0    4    0    0    0    0    0    0
    0    1    1    3    3    4    0    0    0    0
 0.00 7.00 9.00 2.00 11.00 9.00 0.00 0.00 0.00 0.00
```

Как можно видеть, вывод совершенно правильный, но, как я думаю, вам это не очень очевидно, поэтому запускаю рука-рисователь, чтобы вы могли увидеть полученный путь.

А вот и наш граф – результат поиска:



Как мы видим, кратчайший путь от вершины 1 до вершины 5: 1, 3, 6, 5
Также в результирующем графе присуствуют все веса рёбер этого графа, поэтому, кратчайшее расстояние = 20.0, а значит, задача успешно решена.

А теперь немного о том, как же он тут хранится, возможно в комментариях к коду это изложено не совсем понятно.

В массиве vid сопостовляются номера вершин, заданные пользователем и “служебные номера”, которые представляют собой индекс массива vert.
На i-том месте, в массиве vert, хранится индекс в массиве adj1 конца ребра, исходящего из вершины со служебным номером i. В массиве adj2 хранится индекс в массиве adj1 конца следующего ребра, также исходящего из вершны i и так, пока не будут перечислены все рёбра, исходящие из вершины i. Если рёбер из вершины i нет, то в массиве vert, на i-м месте, находится значение 0. Таким образом, если ребро только одно, то в массиве adj2[vert[i]] будет 0.
В массиве source хранится служебный номер начала каждого ребра, это нужно для быстрой работы функции удаления вершины(и не только). В массиве wght хранятся веса всех рёбер.
Как я думаю, использовав эту реализацию у меня получилось сократить использование памяти для хранения графа, по сравнению с матрицей смежности.