Алгоритм Дейкстры с кучей Фибоначчи.

Сравнение кучи Фибоначчи с std::priority_queue.

Были реализованы стандартная куча Фибоначчи и стандартный алгоритм Дейкстры, работающий с произвольной кучей, названия основных операций которой совпадают с названиями таковых у **std::priority_queue**: **push()**, **top()**, **pop()**.

Для реализации алгоритма Дейкстры **std::priority_queue** использовалась с компаратором **std::greater<T>**.

Были произведены 2 типа тестов:

Hill тесты - n раз **push()**, затем n раз **pop()**;

Saw тесты - t раз **push()**, затем **pop()**, повторить k раз.

Первое - тесты простого сценария использования, а второе - тесты, приближенные к сценарию использования в алгоритме Дейкстры.

На Hill тестах **std::priority_queue** проигрывала на этапе **push()**, но на больших объёмах входных данных значительно выигрывала на этапе **pop()**, позволяя обходить Фибоначчиеву куч.

На Saw тестах, за счёт относительно малого количества операций **pop()**, **std::priority_queue** проигрывала в 2-3 раза практически на всех тестах, не считая, разве что, тестов с очень маленькими объёмами данных - t и k < 10.

Результаты некоторых Hill тестов:

Hill тесты	куча Фибоначчи, $*10^{-6}$с	**std::priority_queue**, $*10^{-6}$с
n = 9	5 (1 + 4)	6 (4 + 2)
n = 72	43 (7 + 36)	65 (25 + 40)
n = 6038	4296 (390 + 3906)	5937 (1445 + 4492)
n = 946563	1913254 (70629 + 1842625)	1719935 (446304 + 1273631)
n = 20000000	90575161 (1437659 + 89137502)	55335229 (15062561 + 40272668)

(Большее количество тестов в FibData/fib_full_test.txt)

Результаты некоторых Saw тестов:

Saw тесты	куча Фибоначчи, $*10^{-6}$с	**std::priority_queue**, $*10^{-6}$с
t = 5, k = 4	3	7
t = 3578, k = 7	2991	8133
t = 8, k = 7168	9139	25258
t = 75, k = 90	662	1763
t = 6500, k = 6500	4469499	10940141

(Большее количество тестов в FibData/fib_full_test.txt)

Здесь, как и в алгоритме Дейкстры, использовались **std::pair<int, int>** и **std::pair<double, int>**, в котроых второй элемент - номер зхаписи или номер вершины.

Для тестов алгоритма Дейкстры был написан генератор случайных графов. Сначала он создаёт подвешенное дерево, в котором у каждой вершины, кроме листов и, может быть, последнего не листа, создаётся хотя бы pl детей, а вероятность создания каждого следующего - pD. После этого каждая пара вершин, ещё не соединённых ребром, соединяется в вероятностью pE. В графе получается n вершин.

Результаты некоторых тестов алгоритма Дейкстры:

Тесты	куча Фибоначчи, $* 10^{-6}$с	**std::priority_queue**, $* 10^{-6}$с
pD = 60%, pE = 60% n = 4384, pl = 1	96630	102116
pD = 90%, pE = 90% n = 1183, pl = 1	14466	15208
pD = 50%, pE = 00% n = 3815, pl = 1	3720	4977
pD = 90%, pE = 04% n = 4902, pl = 1	25877	32015
pD = 00%, pE = 50% n = 4908, pl = 1	104814	113284
pD = 04%, pE = 90% n = 537, pl = 1	3009	3868
pD = 92%, pE = 92% n = 4834, pl = 7	147898	145707
pD = 90%, pE = 90% n = 5000, pl = 5	139543	152922
pD = 00%, pE = 100% n = 5000, pl = 1	149383	158422
pD = 00%, pE = 00% n = 5000, pl = 1	2820	8783

(Большее количество тестов в DijData/dij_full_test.txt)

Как видно из результатов, алгоритм Дейкстры с кучей Фиббоначи в большинстве случаев работает в среднем на несколько (5-7) процентов быстрее, чем с **std::priority_queue**, но на близких к бамбуку графах прирост скорости заметно выше: больше 20% на почти дереве, больше 30% на дереве и больше 200% на бамбуке.