Galvenokārt Dijkstras algoritms (<https://www.youtube.com/watch?v=EFg3u_E6eHU> ), bet šķautnes svars ir nevis viens skaitlis, bet skaitļu pāris – biļešu skaits un ceļa garums. Taisot kaimiņu sarakstu (<https://clevercode.lv/theory/document/66>), liek iekšā pair<int,<pair<int,int>>, kur vispirms ir virsotnes numurs, un pēc tam pāris (0,1), ja virsotnes ir vienā krāsā, vai (1,1), ja dažādās krāsās. Salīdzinot pāri visvienkāršāk pārbaudīt, vai vai nu pirmais skaitlis (biļešu skaits) ir mazāks, vai pirmie skaitļi ir vienādi un otrais skaitlis (ceļa garums) ir mazāks. To, cik katra veida biļešu vajag, var aprēķināt no kopējā skaita, zinot, ka tās vajadzēs pirkt pamīšus (tātad biļešu skaits atšķirsies par vai nu 1 vai 0), un zinot, kādā krāsā ir sākotnējā virsotne.

Lai Dijkstras algoritms izpildītos ātrāk, lai noteiktu kura neapmeklētā virsotnei ir vistuvāk sākumam, var izmantot set vai priority queue, kur ieliek pāri (ceļa garums (šajā gadījumā – pāris (biļešu skaits un ceļa garums)) un virsotne). Set ir sakārtots, bet priority queue nodrošina, ka augšā ir lielākais elements (šeit vajag mazāko, tāpēc vai nu vajag rakstīt min priority queue <https://www.geeksforgeeks.org/priority-queue-in-cpp-stl/> vai vienkārši reizināt visu ar -1).

Dijkstra ar priority queue: <https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-using-priority_queue-stl/> klasi “Graph” taisīt nevajag; galvenais metode Graph::shortestPath.   
Rindiņu  
list<pair<int, int> >::iterator i;  
for (i = adj[u].begin(); i != adj[u].end(); ++i)  
vietā var vienkārši *for (auto v : adj[u])* vai *for (pair<int, pair<int, int>> v : adj[u])* un vēlāk *(\*i).first* vietā izmantot *v.first*.