Ātru algoritmu konstruēšana un analīze

1. Programmēšanas darbs

Autors: Roberts Groza rg11080

Apraksts

Praktiskais darbs izstrādāts ar programmēšanas valodu JAVA. Nodevums ietver sevī programmas kodu *PoohAdventure.java*, uzkompilētas java klases un *input.txt* failu.

Lai programma strādātu korekti, *input.txt* jāatrodas tajā pašā mapē, kurā atrodas uzkompilētās JAVA klases. Pirms programmas izpildes *input.txt* aizpilda ar ieejas datiem (whitespace simboliem nav nozīmes, tie var būt tabi, newlines un spaces, kā nosacījumos prasīts).

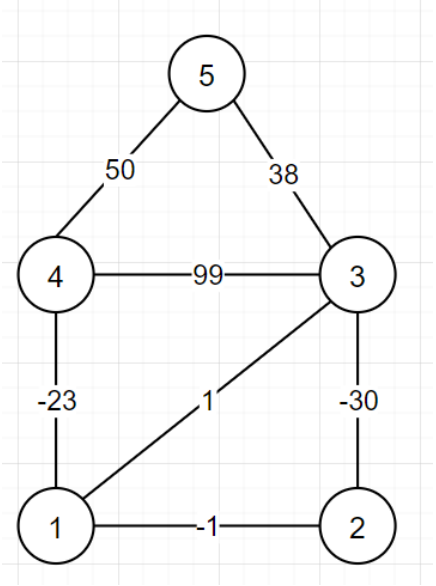
Programmu var palaist no vēlamās komandrindas, ja ir uzstādīta java, ar komandu: *java PoohAdventure*.

Veiksmīgas izpildes gadījumā vajadzētu izdrukāt konsolē “Success!”. Un failā *output.txt* jābūt izdrukātam uzdevuma risinājumam.

```
C:\code\aaka-projekts>java PoohAdventure
Success!
```

Situācijas apraksts

Vispirms mēģināsim izpildīt uzdevumu ar mazāku grafu:



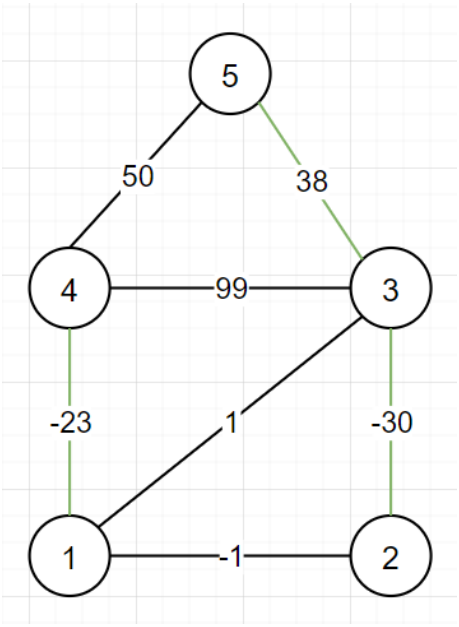
Pirmais cikliskais ceļš, kas krīt acīs ir 1 -> 2 -> 3 -> 1. Šajā ceļā noteikti meduspods ir jāliek uz ceļa 2 -> 3.

Tā kā uz ceļa 2 -> 3 jau stāv meduspods, mēs varam aizmirst par visiem maršrutiem, kas iet caur 2 -> 3, jo attiecīgi tajos maršrutos jau ir medus, piemēram 2 -> 3 -> 4 -> 1 -> 2 mums jau ir medus. Tātad tehniski varam “aizmirst” par šo šķautni, lai Pūks nejauši neveidotu lieki maršrutus, kas iet caur 2 -> 3.

Apskatam maršrutu 1 -> 3 -> 4 un redzam, ka nākamais pods jāliek šķautnē 1 -> 4. Tāpat kā ar 2 -> 3, arī par visiem ceļiem, kas ir caur šo šķautni varam aizmirst, tātad varam pagaidām aizmirst par šķautni.

Maršrutā 3 -> 4 -> 5 ņemam šķautni 38.

Mūsu grafs tagad izskatās tā, ja iekrāsojam jau paņemtās šķautnes zaļas:



Redzam, ka pāri palikušās šķautnes vairs neveido ciklus – tātad nav ciklisku maršrutu, jo nav ciklu un vairs nav jāliek podi, lai apmierinātu 1. Uzdevuma nosacījumu.

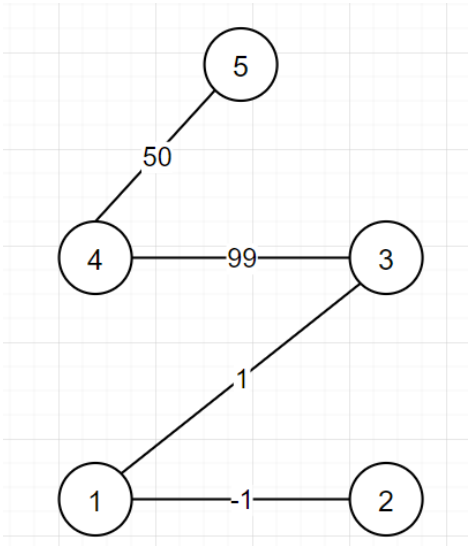
Vēl mēs varam ievērot, ka melnās šķautnes veido *Maximum spanning tree*, kas ir blakusprodukts, kas tiek izmantots manā uzdevuma risinājumā.

Otra lieta, ko var novērot, ir tā, ka, ja Pūks uzliks medus podu 1 -> 2, sarežģītību summa paliks mazāka. **Katra virsotne ar negatīvu svaru samazina kopējo šķautņu sarežģītību summu**, kas nozīmē, ka, lai apmierinātu 2. Uzdevuma nosacījumu Pūkam ir jāieliek pods arī visos pāri palikušajos koku dobumos ar negatīvu sarežģītību.

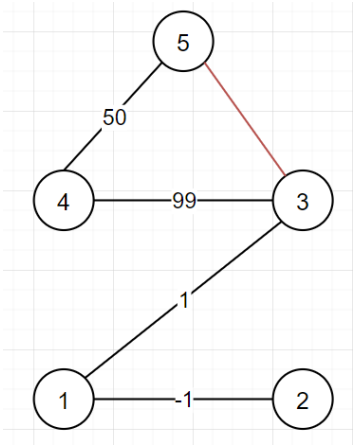
Tātad mūsu apskatītajā grafā mazākā sarežģītību summa būs 38 – 23 – 30 – 1 = -16 un Pūkam jāizvēlas šķautnes 1 -> 2, 1 -> 4, 2 -> 3, 3 -> 5.

Risinājums

Jau iepriekš apskatītajā grafā mēs ieraudzījām, ka mums palika spanning tree.

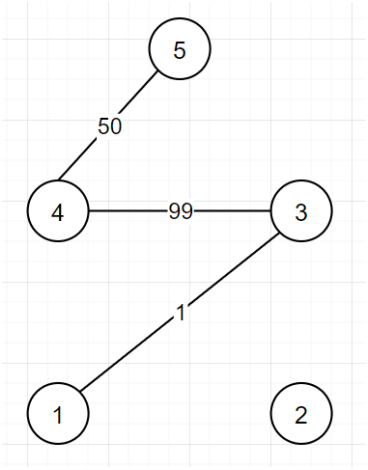


Šī uzdevuma gadījumā *spanning tree* ir noderīgs, tāpēc, ka tajā nav ciklu. Un, ja nav ciklu, tad nav ciklisku maršrutu. Ja mums būtu ceļš no 3 -> 5, tad mums būtu cikls, kas nozīmē, ka tur būtu vajadzīgs meduspods, ja šķautnes sarežģītība < 50 (te palīdzībā nāk *Maximum*).



Ja 3 -> 5 būtu lielāks par 50 un mazāks par 99, tad, 4 -> 5 būtu meduspods un viņu varētu ignorēt dēļ tā, ka visi ceļi caur to apmierinātu uzdevuma nosacījumus. Tātad mūs interesē *maximum spanning tree*. Šajā grafā pieliekot šķautni starp jebkurām 2 virsotnēm, veidosies cikls, kas prasīs meduspodu. Protams vēl meduspods jāuzliek uz 1 -> 2, lai dabūtu vēl mazāku sarežģītību summu. Būtībā mēs atrodam lielāko grūtības pakāpju summu, no kuras Pūks var izvairīties, lai neizveidotu ciklisku maršrutu.

Kā iepriekš noskaidrojām šī grafa mazākā šķautņu sarežģītību summa ir -16. Varam novērot, ka visu šķautņu sarežģītību summa ir 134, un iegūtā *maximum spanning tree* šķautņu summa ir 149. Ja izņemam negatīvās šķautnes no *maximum spanning tree*, jo tas būtu Pūkam izdevīgi, mēs iegūstam 150. Un 134 – 150 = - 16, kas ir mūsu sarežģītību summa, un pāri palikušās grafa šķautnes ir tās, uz kurām nevajag likt medu.



Algoritms

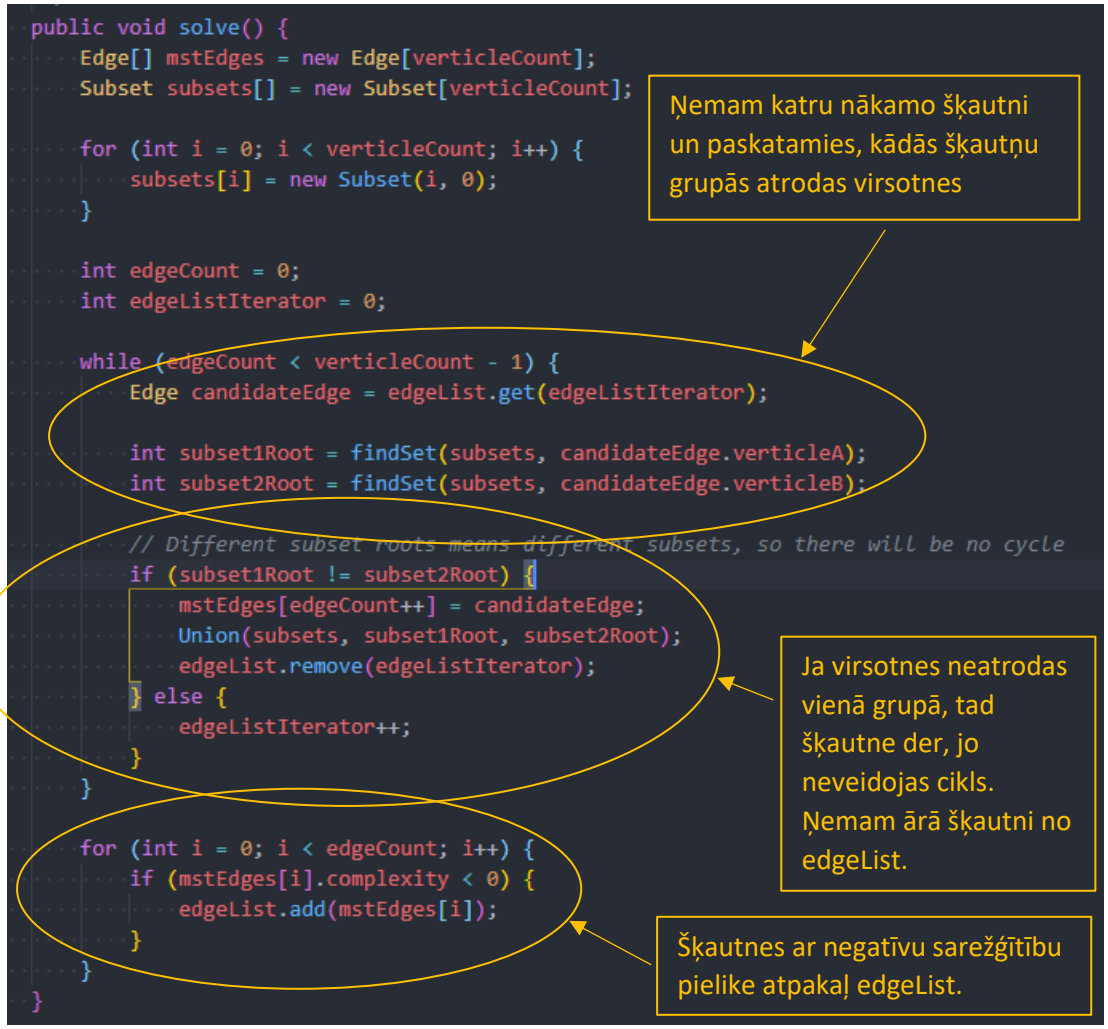
Lai atrastu *Maximum spanning tree*, es izmantoju *Kruskal's* algoritmu. Tikai tā kā es meklēju *Maximum* nevis *Minimum*, es kursā apskatītajā algoritmā šķautnes sakārtoju nevis augošā secībā, bet gan dilstošā secībā.

Uzdevuma risināšanas algoritms ir sekojošs:

1. Paralēli lasot failu kārtoju šķautnes sarakstā dilstošā secībā (Šķautnes glabāju JAVA LinkedList datu struktūrā, kas ir saistīts saraksts).
2. Atrodu visas šķautnes, kas veido *Maximum spanning tree* and *Kruskal's* algoritmu.
3. No kopējā grafa šķautņu saraksta izņemam visas šķautnes, kas veido *Maximum spanning tree* un kuru sarežģītība nav negatīva (kā noskaidrojām, negatīvās šķautnes ir izdevīgi pievienot, lai dabūtu mazāko iespējamo summu, pat ja tas nav nepieciešams). **Rezultātā no sākotnējā šķautņu saraksta noņemtās šķautnes, kur nav jāliek meduspods, mums atstāj tikai tās šķautnes, kurās jāliek meduspods.**
4. Izdrukājam palikušo šķautņu kopskaitu, summu un izdrukājam visas pāri palikušās šķautnes.

Kruskal's algoritma implementācija

Kruskal's algoritms implementēts metodē *solve()*. Mainīgajā *subsets* tiek glabāta informācija par to, kādā jau MST piederošo šķautņu grupā atrodas virsotnes. Klasē Subset ir mainīgie *root* un *rank*. Pēc *root* nosaka, kurai grupai pieder virsotne, ja virsotnēm ir vienāds *root*, tas nozīmē, ka virsotnes atrodas jau vienā grupā. *Root* ir grupas sākumpunkts. *Rank* tiek izmantots, lai veicot *union*, lielākā grupa absorbētu mazāko. Metode *findSet(Subset subsets[], int verticle)* atrod, kurai grupai pieder virsotne *verticle*. Metode *union(Subset subsets[], int subset1Root, int subset2Root*) apvieno dotās grupas, kuras apzīmē ar saknēm *subset1Root* un *subset2Root*, vienā.




```
public void solve() {
    Edge[] mstEdges = new Edge[verticleCount];
    Subset subsets[] = new Subset[verticleCount];

    for (int i = 0; i < verticleCount; i++) {
        subsets[i] = new Subset(i, 0);
    }

    int edgeCount = 0;
    int edgeListIterator = 0;

    while (edgeCount < verticleCount - 1) {
        Edge candidateEdge = edgeList.get(edgeListIterator);

        int subset1Root = findSet(subsets, candidateEdge.verticleA);
        int subset2Root = findSet(subsets, candidateEdge.verticleB);

        // Different subset roots means different subsets, so there will be no cycle
        if (subset1Root != subset2Root) {
            mstEdges[edgeCount++] = candidateEdge;
            Union(subsets, subset1Root, subset2Root);
            edgeList.remove(edgeListIterator);
        } else {
            edgeListIterator++;
        }
    }

    for (int i = 0; i < edgeCount; i++) {
        if (mstEdges[i].complexity < 0) {
            edgeList.add(mstEdges[i]);
        }
    }
}
```

Pēc šīs funkcijas izpildes *edgeList* masīvā atrodas tikai tās šķautnes, kurās jāliek meduspods.

Sarežģītības novērtējums

Apzīmēsim ar *n* virsotņu skaitu grafā un ar *m* šķautņu skaitu grafā.

Ieejas faila apstrāde un *edgeList* aizpildīšana:

```
80          while (fileReader.hasNextInt()) {
81              int verticleA = fileReader.nextInt() - 1;
82              int verticleB = fileReader.nextInt() - 1;
83              int complexity = fileReader.nextInt();
84
85              boolean added = false;
86
87              for (int i = 0; i < edgeList.size(); i++) {
88                  if (edgeList.get(i).complexity <= complexity) {
89                      edgeList.add(i, new Edge(verticleA, verticleB, complexity));
90                      added = true;
91                      break;
92                  }
93              }
94
95              if (!added) {
96                  edgeList.addLast(new Edge(verticleA, verticleB, complexity));
97              }
98          }
```

While cikls (80. rindiņa) izpildīsies *m* reizes. Sliktākajā gadījumā, ja ieejas failā visas šķautnes sakārtotas augošā secībā, *for* cikls (87. rindiņa) izpildīsies katru reizi tik reizes, cik *edgeList* ir šķautņu. Šķautņu skaits *edgeList* pieaug katrā iterācijā par 1 - Pirmajā iterācijā 1, otrajā 2, trešajā iterācijā 3, i-tajā iterācijā i. Tātad while & if konstrukcijas sarežģītība ir $O(m^2)$, jo pēdējā iterācijā for cikls tiks izpildīts m reizes. Šķautnes pievienošana saistītajā sarakstā, ja zinam, kur jāpievieno ir O(1). Tātad *fillEdgeList* sarežģītība ir $O(m^2)$.

Implementētā *Kruskal's* algoritma sarežģītība O(m * log m). 179. Rindiņā for cikls sliktākajā gadījumā izpildīsies m reizes. Tātad *solve* sarežģītība ir O(m * log m).

PrintResult metodē ir 2 for cikli, kas katrs pa reizei iziet cauri visam masīvam - O(m).