S. 70.

Szabó 928 Attila 11. o. t.
Leőwey Klára Gimnázium, Pécs
szabo.attila94@gmail.com

A program C nyelven, a gcc 4.6.1 verziójával készült (Linux alatt).

A program azt vizsgálja, hogy adott minimális élhossz mellett mekkora maximális élhossz szükséges ahhoz, hogy a kettő közti hosszúságú éleket használva el lehessen jutni A-ból B-be: ezt az élhosszúságok rendezett sorozatán bináris kereséssel hatékonyan meg lehet oldani. A feladatban ilyen minimum–maximum különbségek minimumát keressük, ezért ez a módszer alkalmas a feladat megoldására. A folyamat gyorsítására csak azokat az eseteket vizsgáljuk, amikor a pillanatnyilag ismert legkisebb különbségnél kisebbet állíthatunk elő.

A `N, M, A, B` változók jelentése nyilvánvaló, `W` a különböző élhosszúságok száma: ennek a keresés hatékonyításában van szerepe; a `graph` tömb első dimenziója az éllistát jelzi, második dimenziója szerint a 0. elem az él hossza, az 1. és 2. elem a két végpont. A `neighbor` és `neigh_index` tömbök szomszédsági listát tárolnak: a `neighbor` tömb első dimenziója sorfolytonosan tárolja az egyes elemek szomszédjait, a második dimezió szerinti 0. elem a távolság, az 1. elem a szomszéd sorszáma; a `neigh_index` tömb az egyes csúcsok szomszédjainak kezdőcímét tárolja. A `lengths` tömb elemei nagyság szerinti sorrendben tárolja az élek hosszait; az `aux` tömb többféle célra használt segédtömb; a `found` tömb a mélységi keresés során használt segédtömb, amely csak azért globális, hogy ne kelljen minden híváskor újra allokálni (ez jelentősen lassítaná a programot).

A `readin` függvény a standard bemenetről olvassa be a bemeneti adatokat, illetve az élek beolvasása során elkészíti a `neigh_index` tömb első változatát, amely az egyes csúcsok szomszédjainak számát tárolja.

A `preprocess` függvény elkészíti `neigh_index` végleges változatát, kitölti `neighbor`-t, majd az `aux` tömbbe összegyűjtött élhosszúságokat a `heapsort()` eljárással rendezi, és eltávolítja belőle az ismétlődéseket, így alakul ki a `lengths` tömb. A `heapsort` függvény rendezi nagyság szerinti sorrendbe az `aux` tömböt: annak viszonylag kis mérete esetleg egyszerű rendezési módszert is alkalmazhatóvá tenne. A csere segédváltozója a `swap`, a rendezett tömb előállítása során a „kupacból” műveletet a kupac határán leállítja a viszonylag bonyolult kilépési feltétel.

A `connected` függvény meghatározza, hogy a gráfban el lehet-e jutni A-ból B-be, `lengths[min_index]` és `lengths[max_index]` közötti hosszúságú éleket használva (a `lengths` tömb elemein kívüli szélsőértékeket fölösleges vizsgálni): a választ `char` változóban, logikai alapértékként (0 vagy 1) adja vissza. Ha `max_index` értelmetlen (kisebb, mint `min_index`, vagy nagyobb, mint `W-1`), a megfelelő választ adja. Különben mélységi keresést végez (mivel a verem tömbös programozása egyszerűbb, mint a soré), amelynek során a kezdetben hamisra (0) állított `found` tömbben jelöli, ha eljutott egy adott csúcsba. A keresés addig tart, amíg vagy a teljes elérhető gárfrészletet nem jártuk be, vagy el nem értük B-t.

A `min_diff` függvény határozza meg a feladat tulajdonképpeni megoldását. Minden lehetséges minimális élhosszat kipróbálunk (indexük `i`), nagyság szerinti csökkenő sorrendben: ez véletlenszerű élhosszaknál jobb stratégiának tűnik, mint a növekvő sorrend. A `mdiff` változó tartalmazza a pillanatnyilag ismert legkisebb különbséget, kezdetben irreálisan soknak, $2 \cdot 10^9$-nek választjuk. Egy élhossz vizsgálatát azzal kezdjük, hogy meghatározzuk azt a legnagyobb `high` indexet, amelyre $lengths[high] - lengths[i] < mdiff$: ezeken az indexeken lehetséges, hogy kisebb különbséget találjunk, mint a már ismert legkisebb. Ezután ellenőrizzük, hogy `high`-t, mint maximális indexet választva elérhető-e A-ból B: ha nem, az adott minimális élhosszal nem kapható jobb megoldás, a következőre lépünk. Ha igen, bináris keresés segítségével megkeressük azt a legkisebb indexet, amellyel a gráfban még vezet út A és B között. Mivel ez a kezdeti ellenőrzés miatt biztosan jobb különbséget ad, mint az addigi optimum, `mdiff`-et helyettesíti `lengths` két megfelelő elemének különbségével. Mindkét bináris keresés során olyan x-et keresünk, hogy egy bizonyos tulajdonság $x - 1$-re nem teljesül, x-re viszont igen: ezt a bináris keresés olyan átalakított változatával tesszük meg, amely a két szélsőérték közti `mi`–`ma` intervallumot felezi úgy, hogy `mi`-re ne teljesüljön a feltétel, `ma`-ra viszont igen: a keresés akkor áll le, amikor az intervallum két végpontjának távolsága 1: ekkor `ma` az említett x-szel egyezik meg.

A `connected` függvény futásideje a mélységi keresésével egyezik meg, azaz $O(N + M)$-mel becsülhető felülről (feltéve, hogy minden csúcs és él megvizsgálásra kerül). A `min_diff` függvény egy minimális különbség vizsgálata során $O(\log W)$ műveletet végez `high` meghatározására, utána legfeljebb $O(\log W)$ lépéses bináris keresés következhet, melynek minden lépése egy `connected` műveletet igényel. Egy ilyen vizsgálat futásideje tehát $O(\log W(N + M))$-mel becsülhető felülről; mivel W ilyen vizsgálatra kerül sor, a teljes futásidőt $O(W \log W(N + M))$-mel becsülhető felülről: a becslés elég durva, ugyanis a keresésből a felesleges műveleteket kizáró vizsgálat nagymértékben csökkenti a nagy időigényű `connected` eljárások számát. `min_diff` becsült futásidőfüggvénye mellett a többi lépésé elhanyagolható ($O(N)$, $O(M)$, illetve a kupacrendezés $O(M \log M)$ futásidejű). A program futásideje tehát felülről becsülhető $O(W \log W(N+M))$-mel. A tesztelésre használt gépen (4 GB RAM, kb. 2 GHz Dual Core processzor) a program maximális véletlenszerű tesztesetekre 6–7 századmásodperc alatt futott le (elképzelhetőek rosszabb eredményt produkáló gráfok).