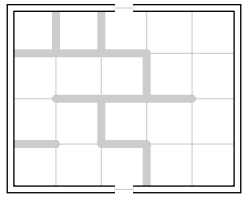
Tomasz Idziaszek Treść zadania | **Juliusz Straszyński** Opracowanie | **Marek Sommer** Program

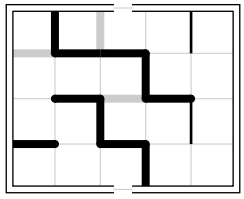
Dostępna pamięć: 128 MB. *OI, etap III, dzień pierwszy, 13.04.2016*

Żywopłot

Królewski ogrodnik Bajtazar ma za zadanie wyhodować w królewskim ogrodzie labirynt z żywopłotu. Ogród można podzielić na $m \times n$ kwadratowych pól. Otoczony jest murem, w którym na środku północnej i południowej ściany są wejścia. Na każdej krawędzi dzielącej dwa pola można zbudować kawałek żywopłotu – z cisu lub tui. Król bardziej lubi cis, więc chciałby mieć w swoim ogrodzie jak najwięcej kawałków żywopłotu z cisu. Niestety, cis wymaga lepszej gleby, więc nie wszędzie go można posadzić.

Aby żywopłot tworzył labirynt, musi spełniać dodatkowy warunek: do każdego pola musi być możliwość dojścia z obu wejść i, co więcej, tylko na jeden sposób. (Z danego pola można przejść bezpośrednio na pole sąsiadujące, jeśli na dzielącej te pola krawędzi nie znajduje się kawałek żywopłotu. Dwa sposoby dojścia uznajemy za różne, jeśli przechodzą przez różne zbiory pól.)





W lewej części powyższego rysunku przedstawiono przykładowy ogród dla $m = 4$ i $n = 5$, zawierający 31 krawędzi. Wyróżniono w nim 13 krawędzi, na których można posadzić żywopłot z cisu.

Na prawej części rysunku przedstawiono przykładowy labirynt składający się z 12 kawałków żywopłotu, z których 10 jest żywopłotem z cisu, a 2 są żywopłotem z tui. Nie istnieje labirynt zawierający więcej kawałków z cisu. Twoim zadaniem będzie napisanie programu, który pomoże Bajtazarowi w zaprojektowaniu labiryntu.

Wejście

W pierwszym wierszu standardowego wejścia znajdują się dwie liczby całkowite m i n oznaczające rozmiar ogrodu ($2 \leqslant m, n$ oraz n jest liczbą nieparzystą). Kolejne m wierszy zawiera po $n-1$ znaków, opisujących pionowe krawędzie (czytane rzędami, od lewej do prawej). Znak `C` *oznacza, że na danej krawędzi można posadzić żywopłot z cisu, a znak* `T` *oznacza, że można posadzić żywopłot z tui. Kolejne $m-1$ wierszy zawierające po n znaków opisuje poziome krawędzie (również czytane rzędami, od lewej do prawej).*

Wyjście

W pierwszym wierszu standardowego wyjścia należy wypisać dwie liczby całkowite: liczbę posadzonych kawałków żywopłotu tworzących labirynt oraz maksymalną liczbę kawałków żywopłotu

z cisu. W kolejnych $2m-1$ wierszach należy opisać krawędzie labiryntu (w kolejności jak na wejściu). Należy wypisać znak `Z`*, jeśli krawędź zawiera żywopłot, lub znak* `.` *(kropka) w przeciwnym wypadku.*

Jeśli istnieje wiele rozwiązań spełniających warunki króla, należy wypisać dowolne z nich.

Przykład

Dla danych wejściowych:	*jednym z poprawnych wyników jest:*
`4 5`	`12 10`
`CCTT`	`Z..Z`
`TTCT`	`..Z.`
`TCTT`	`.Z.Z`
`TTCT`	`..Z.`
`CCCTT`	`.ZZ..`
`TCCCT`	`.Z.Z.`
`CTCTT`	`Z.Z..`

Wyjaśnienie do przykładu: *Dane wejściowe opisują ogród z lewej części rysunku; wynik opisuje labirynt z prawej części rysunku.*

Testy „ocen”:

`1ocen`: *$m = 4$, $n = 3$, w każdym miejscu można posadzić cis;*

`2ocen`: *$m = 100$, $n = 99$, na pionowych krawędziach można posadzić cis, na poziomych można posadzić tuję;*

`3ocen`: *$m = 1000$, $n = 999$, w każdym miejscu można posadzić tuję.*

Ocenianie

Zestaw testów dzieli się na następujące podzadania. Testy do każdego podzadania składają się z jednej lub większej liczby osobnych grup testów.

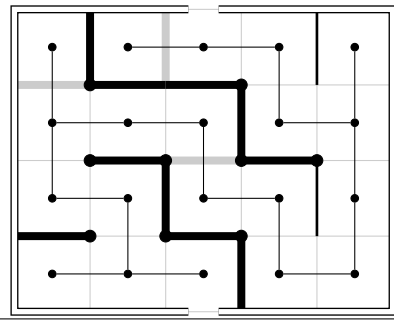
Jeśli Twój program wypisze poprawny pierwszy wiersz, a dalsza część wyjścia nie będzie poprawna, uzyska 52% punktów za dany test. W szczególności, aby uzyskać 52% punktów za test, wystarczy wypisać tylko jeden wiersz wyjścia.

Podzadanie	**Warunki**	**Liczba punktów**
1	$n \cdot m \leqslant 12$	*25*
2	$n, m \leqslant 100$	*25*
3	$n, m \leqslant 1000$	*50*

Rozwiązanie

Przypomnijmy definicję labiryntu: z każdego pola da się dojść do obu wejść na dokładnie jeden sposób. Sytuację z zadania możemy przedstawić jako graf nieskierowany, którego wierzchołki odpowiadają polom, a krawędzie łączą pary sąsiednich pól,

które *nie są* oddzielone kawałkiem żywopłotu. Graf taki jest więc labiryntem, gdy dla każdego wierzchołka istnieje dokładnie jedna ścieżka do wierzchołka oznaczonego wejściem oraz dokładnie jedna do wierzchołka oznaczonego wyjściem. Jest to tożsame z tym, że graf musi być spójny i bez cykli, bo każdy cykl w grafie spójnym oznaczałby istnienie takiego wierzchołka, który ma co najmniej dwie ścieżki do wejścia/wyjścia. Innymi słowy, graf musi być *drzewem* (rys. 1). Liczba wejść do labiryntu nie ma znaczenia – ta interpretacja byłaby prawdziwa również, gdybyśmy mieli jedno, czy też więcej niż dwa wejścia.



Rys. 1: Chcemy znaleźć takie drzewo, które połączy wszystkie pola i usunie z żywopłotu jak najmniej krawędzi z cisem.

Zatem musimy znaleźć drzewo łączące wszystkie wierzchołki o najmniejszym koszcie. Jeżeli w ostatecznym drzewie między sąsiednimi wierzchołkami jest krawędź, to nie posadzimy tam żywopłotu. W zadaniu chcemy mieć jak najwięcej kawałków żywopłotu, które są cisami. Oznacza to, że w drzewie chcemy użyć jak najmniej cisów, czyli koszt krawędzi odpowiadającej cisowi to 1, a krawędzi odpowiadającej tui to 0.

Zadanie sprowadza się więc do klasycznego problemu znajdowania minimalnego drzewa rozpinającego (ang. *MST – minimum spanning tree*). By je rozwiązać, możemy użyć znanych algorytmów znajdujących MST, np. algorytmu Kruskala [6] (`zyw2.cpp`, `zyw5.cpp`) czy też Prima [6] (`zyw3.cpp`). Dla grafu o zbiorze wierzchołków V i zbiorze krawędzi E algorytmy te działają w czasie $O((|V|+|E|)\cdot\log|V|)$. W naszym przypadku $|V| = nm$ i $|E| \leqslant 2nm$, więc ta złożoność to $O(nm\log(nm))$. Umiejętne użycie jednego z tych algorytmów pozwalało na zdobycie 100 punktów.

Nie są to optymalne rozwiązania pod względem złożoności czasowej. By je ulepszyć, należy wykorzystać strukturę grafu, a dokładnie to, że krawędzie mają dwa możliwe koszty: 0 lub 1.

Algorytm Prima zaczyna od jednowierzchołkowego drzewa i zachłannie rozszerza je o nowe wierzchołki, które aktualnie są najtańsze do dodania. Wykorzystując to, że mamy tylko dwa możliwe koszty krawędzi, możemy w kubełkach segregować krawędzie o odpowiednich kosztach, zamiast utrzymywać kosztowną kolejkę priorytetową. Dzięki takiej optymalizacji algorytm działa w czasie $O(|V|+|E|)$, czyli w naszym przypadku $O(nm)$ (`zyw12.cpp`).

W poniższym pseudokodzie zakładamy, że wszystkie krawędzie są początkowo wybrane do żywopłotu, a następnie usuwamy te z nich, które wybieramy do MST. Końce krawędzi oznaczamy jako $v1$ oraz $v2$.

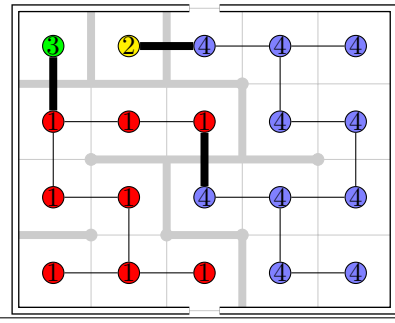
```
 1: procedure PRIM(v_0)
 2: begin
 3:   rozmiarDrzewa := 1;
 4:   dodany[v_0] := true;
 5:   krawedziePoCisach := {krawędzie z v_0 po cisach};
 6:   krawedziePoTujach := {krawędzie z v_0 po tujach};
 7:   while rozmiarDrzewa < n do begin
 8:     if not krawedziePoTujach.puste() then begin
 9:       najtanszaKrawedz := krawedziePoTujach.ostatnia();
10:       krawedziePoTujach.usunOstatnia();
11:     end else begin
12:       najtanszaKrawedz := krawedziePoCisach.ostatnia();
13:       krawedziePoCisach.usunOstatnia();
14:     end
15:     if dodany[najtanszaKrawedz.v1] then nowy := najtanszaKrawedz.v2
16:     else nowy := najtanszaKrawedz.v1;
17:     if not dodany[nowy] then begin
18:       usunZywoplot(najtanszaKrawedz);
19:       krawedziePoCisach += {krawędzie z nowego wierzchołka po cisach};
20:       krawedziePoTujach += {krawędzie z nowego wierzchołka po tujach};
21:       rozmiarDrzewa++;
22:       dodany[nowy] := true;
23:     end
24:   end
25: end
```

Algorytm Kruskala przedstawia się następująco: dopóki nie wszystko jest połączone w jedno drzewo, znajdź najtańszą krawędź, która łączy wierzchołki z niepołączonych jeszcze składowych, i dodaj ją do aktualnego minimalnego drzewa rozpinającego. Składowe są przechowywane za pomocą struktury danych do zbiorów rozłącznych, tzw. *Find-Union*: *Find* znajduje identyfikator składowej, do której należy dany wierzchołek, a *Union* łączy składowe zawierające podane dwa wierzchołki. Zamortyzowany koszt operacji *Find-Union* na zbiorze rozmiaru k to $O(\log^* k)$. Analogicznie jak w przypadku algorytmu Prima rozdzielając krawędzie według kosztów, jesteśmy w stanie sprawić, by algorytm Kruskala działał w czasie $O((|V| + |E|) \cdot \log^* |V|)$.

```
 1: procedure KRUSKAL
 2: begin
 3:   while not krawedzie.puste() do begin
 4:     najmniejsza := krawedzie.wezNajmniejsza();
 5:     krawedzie.usunNajmniejsza();
 6:     if Find(najmniejsza.v1) ≠ Find(najmniejsza.v2) then begin
 7:       usunZywoplot(najmniejsza);
 8:       Union(najmniejsza.v1, najmniejsza.v2);
 9:     end
10:   end
11: end
```

Okazuje się, że w przypadku naszego zadania także algorytm Kruskala możemy zmodyfikować, by działał w czasie liniowym (tj. $O(nm)$). Jako pierwsze w algorytmie Kruskala będą rozpatrzone krawędzie o koszcie 0. W takim razie chcemy podzielić wierzchołki grafu na takie grupy, że w obrębie jednej da się przejść między każdą parą wierzchołków wyłącznie po tujach. Można łatwo takie grupy znaleźć w czasie liniowym, np. za pomocą algorytmu DFS, startując z każdego nieodwiedzonego wierzchołka, odwiedzając wszystkie wierzchołki osiągalne po tujach i oznaczając je numerem grupy. Wierzchołki w obrębie danej grupy jesteśmy w stanie połączyć, nie usuwając z żywopłotu żadnego cisu. Natomiast by połączyć różne grupy nie mamy innego wyboru jak usunąć jakieś cisy na ich granicy (rys. 2). By połączyć k grup w drzewo, musimy postawić dokładnie $k-1$ krawędzi.

Znamy już odpowiedź, ile cisów zostanie w żywopłocie, ale teraz pytanie brzmi: jak szybko znaleźć te cisy, które zostaną usunięte? Możemy to osiągnąć, tworząc graf grup: iterujemy po każdym polu i jeżeli jego grupa to g_1, a grupa jego sąsiada to g_2 i $g_1 \neq g_2$, to g_1 i g_2 są sąsiadami w grafie grup, a krawędź między nimi odpowiada krawędzi między rozważanymi polami. W powstałym grafie znajdujemy dowolne drzewo rozpinające np. za pomocą algorytmu DFS. To rozwiązanie zostało zaimplementowane w pliku `zyw11.cpp`.



Rys. 2: Najpierw łączymy wierzchołki wewnątrz każdej z grup kosztem 0 (cienkie linie), a później łączymy grupy (grube linie).