Анализ на задача bipartite

Задачата ни пита за даден граф G да определим дали е двуделен и да при зададени допълнителни ребра да намерим първото, чието добавяне чупи двуделността.

В случая, когато нямаме добавяне на ребра ($q = 0$), тази задача е учебникарска и може да се реши по множество начини – dfs, bfs, дори и с union-find / disjoint-set union.

Установяване на двуделност с dfs / bfs

Удобното при задачата за двуделност на граф е, че двата цвята вършат една и съща работа, тоест Независимо дали сложа конкретен връх u да е черен или бял, ако има решение, алгоритъмът все трябва да може да го намери.

В случая с dfs, нека просто го пуснем от произволен връх с произволен цвят. При слизането на обхождането ще разменяме цвета, с който посещаваме върха. Проблеми могат да създадат back-edge-ове[1] в dfs дървото. За тях просто трябва да проверим, че цветът, който сме поставили на предишния връх, се различава от цвета, който поставяме на сегашния връх.

Подобна стратегия ще използваме и в алгоритъм, който ползва bfs за задачата. Всеки слой образуван от bfs-а ще се състои от върхове с един и същи цвят. Тук проблемни ребра ще бъдат тези, които свързват два върха от един и същи слой – ще трябва да проверяваме за тяхното съществуване.

Установяване на двуделност с union-find

Идеята на проверяването на двуделност с union-find е малко по-сложна от предходната, но може директно да ни доведе до решение на задачата за 100 точки. Все пак по-долу ще разгледаме и други решения.

Ще разширим графа G по следния начин: всеки връх u ще има две версии – u_{white} и u_{black}, които съответно представляват идеята връх u да се оцвети в бял или черен цвят. Добавянето на ребро $u - v$ ще има значението на свързване на $u_{white} - v_{black}$ и $u_{black} - v_{white}$. Можем интуитивно да си мислим, че съществуването на ребро $u_{white} - v_{black}$ е еквивалентно на твърдението "Ако u е бял, v е черен" и обратното. Тук проблемите възникват, когато, следвайки тези твърдения, получим твърдение от типа "Ако u е бял, u е черен" – това е невъзможно, противоречие.

В крайна сметка, алгоритъмът ни е: добавяме всички ребра, по начина на горния абзац, и проверяваме накрая, че не сме свързали u_{white} и u_{black} за всяко u.

Всъщност причината да нямаме решение на цялата задача още сега е, че въпреки че си строим графа бързо, проверяваме абсолютно всеки връх. Това се разрешава със следното наблюдение: Когато добавим ребро $u - v$, което разваля двуделността на оригиналния графа, това е защото сме създали цикъл с нечетна дължина – в допълнение за всеки връх от този нечетен цикъл е вярно, че w_{white} и w_{black} са свързани в една компонента. Възползвайки се от това, след добавянето на реброто $u - v$ можем да проверим само двойките (u_{white}, u_{black}) и (v_{white}, v_{black}). Сега вече имаме едно решение за 100 точки със сложност $O((n+m+q)logn)$ или $O((n+m+q)\alpha(n))$ в зависимост от приложените оптимизации при имплементацията на union-find.

[1] Това са ребра, които свързват връх с друг, вече посетен от dfs обхождането.

Други две решения за 100 точки

Съществуват други два погледа над задачата, които също ще ни доведат до пълни решения.

"Граф G е двуделен" е монотонно твърдение. До някакъв момент в задачата, графът наистина е двуделен, но след това спира да бъде такъв. Можем да намерим този момент с двоично търсене върху отговора, а за определяне на двуделността на G ще ползваме някой от горните алгоритми. Сложността на такова решение е $O((n+m+q)logn)$, но с висока константна (идва от многото ползване на командата push_back, докато си построяваме графа), съответно бяха заделени около 66 точки, които гарантирано да бъдат взети с така. Може да се добута до 100 точки, но си е играчка.

От тук почва анализът на "оригиналното" решение – това покрай което задачата беше създадена.

В началото имаме набор от n върха без каквито ребра – спокойно можем да кажем, че в момента всички върхове са бели. При добавяне на нови ребра разглеждаме два случая:

- двата върха са разноцветни – хубавият случай, просто добавяме реброто.
- двата върха са едноцветни – тук вече може малко да се оплашим от идеята, че една от двете компонентни ще трябва да се преоцвети. Ако не внимаваме, лесно може да избием на $O(n^2)$ операции на преоцветяване.

Една евристична идея, която можем да изпробваме (и с право) е да преоцветяваме по-малката от двете компоненти. Доказуемо е, че такава стратегия ще работи в $O(nlogn)$ време. Нека разгледаме колко пъти един връх сменя цвета си в този алгоритъм – връх u трябва да е бил част от компонента (с размер a), която се свързва с друга (с размер b), по-голяма от нея ($a < b$ е в сила). Новообразувата компонента ще има размер $a+b > 2 \times a$ или с други думи, компонентата на връх u ще удвои размера си. "Колко пъти се удвоява размера на компонентата?" е въпрос, еквивалентен на "Колко пъти връх u сменя цвета си?", нека се опитаме да отговорим на първия. Компонентата на връх u определено ще има размер $\leq n$, а в началото е имала размер 1. Значи се интересуваме колко пъти можем да умножим 1 по две, без то да надвиши n – това е точно смисъла на функцията $logn$.

Сходни идеи се наричат small to large merging и се използват нерядко по задачи. Операциите по добавяне на ребрата ще се обработват удобно с union-find. Това решение вече минава за 100 точки.

Съществува обаче оптимизация, макар и трудно различима само по тайм лимита на системата, която за пълнота на анализа ще разгледаме.

За да работи, union-find строи дървета, които да представляват всяка компонента - свързването на две компоненти означава свързване на корените им. Нека когато свързваме две компоненти, наместо веднага да преоцветяваме по-малката, да запишем в корена на съответното дърво, че тази компонента трябва да се преоцвети (lazily записваме някаква информация). Когато ни интересува реалният цвят на даден връх, можем да проверим колко пъти са преоцветявани върховете между него и текущият корен на дървото му. Още повече, можем да правим path compression с тази информация. Именно тази оптимизация ще свали сложността ни до $O((n+m+q)\alpha(n))$.

Релевантни материали:

- Small-to-large merging
- Disjoint set union

Анализ: Иван Лупов