Üvegvágás

Bobély Bernárd, Birszki Levente 2024. operációkutatás projekt

# A feladat

Adott egy input fájlban néhány téglalap alakú üveglap, rajta téglalap alakú hibákkal. A hibákat nyílt téglalapnak tekintjük. Adottak továbbá a kivágandó, téglalap alakú itemek listái, azaz stackjei. Minden stacken az itemeket sorban kell kivágni, úgy, ahogyan az üveglapokat is sorban kell felhasználni. A cél, hogy minél kevesebb szemét (waste) maradjon a végén. Az itemeket lehet forgatni, illetve az utolsó lapon az utolsó 1-cuttal kapott érintetlen rész nem számít szemétnek, ez maradék (residual).

## Vágások

Legfeljebb 4 mélységben vághatunk, egy vágás mindig végig vágja a teljes üveglapot. Először csinálhatunk bármennyi függőleges vágást, ezek neve 1-cut. Majd vízszintes vágásokat, 2-cutokat. Ezután ismét függőleges 3-cut, majd végül vízszintes 4-cut. A 4-cuttal kapott 2 üvegdarabból az egyik item, a másik szemét, vagy mindkettő item.

További korlátozások, hogy

* nem vághatunk ketté hibát,
* minden lapon muszáj legalább egyet vágnunk,
* nem lehet egymáshoz túl közel, és túl távol két 1-cut,
* nem lehet egymáshoz túl közel két 2-cut,
* a wastenek adott egy minimális mérete.

Az alábbi ábra bemutat egy üveglapot, melyen nincs maradék, de ha ez a feladat utolsó üveglapja, akkor a jobb oldali oszlop akár lehetne az is.

A képen szöveg, képernyőkép, Téglalap, tér látható

Automatikusan generált leírás

# A megoldás

## Programkód

A megoldásunk 4 python fájlból áll.

* classes.py
* input\_output.py
* solve.py
* main.py

A main.py fájlban található a keretprogram, amiben ki tudjuk választani, hogy a solverek közül melyiket szeretnénk használni, és melyik adathalmazon szeretnénk futtatni, továbbá, hogy az egész adathalmazon, vagy csak egy feladatot fusson. Itt történik az időmérés is.

Az input\_output.py fájlban olyan függvények vannak implementálva, melyek az adatok beolvasásáért, az eredmények kiíratásáért, továbbá a konzolra történő logolásért felelősek.

A classes.py fájlban a globális változók, és a feladat modellezését lehetővé tevő osztályok vannak megvalósítva, néhány belső művelettel együtt. Ezek a következők.

* Defect – hiba:
  + id: azonosító,
  + x: bal alsó sarok x koordinátája,
  + y: bal alsó sarok y koordinátája,
  + width: szélesség,
  + height: magasság.
* Bin – üveglap:
  + id: azonosító,
  + width: szélesség,
  + height: magasság,
  + defect: defektek listája.
* Item
  + id: azonosító,
  + width: szélesség,
  + lenght: hosszúság (magasság).
  + rotate(): megforgatja az itemet, azaz felcseréli a width és length értékeket.
* Stack – Itemek listája
  + id: azonosító,
  + sequence: itemek listája.
* Batch
  + stacks: stackek istája.
* Residual – Vágás után kapott maradék üveglap
  + x: bal alsó sarok x koordinátája,
  + y: bal alsó sarok y koordinátája,
  + widht: szélesség,
  + height: magasság,
  + defects: defektek lisája.
  + has\_defect\_in(x\_low, x\_high, y\_low, y\_high): Megadja, hogy van-e hiba a paraméterekkel határolt téglalapon belül (beleértve a határát).
  + defects\_in(x\_low, x\_high, y\_low, y\_high): Visszaadja, a hibákat a paraméterekkel határolt téglalapon belül (beleértve a határát).
  + find\_place(width, length, is\_vertical): Egy width\*length dimenziós téglalapnak talál helyet úgy, hogy ne legyen átfedésben hibával. A bal alsó sarokból indulva, ha az is\_vertical paraméter igaz, akkor felfele indul, és ha nem talált elérhető helyett, akkor ettől jobbra is egy teljes oszlopot vizsgál lentről felfelé. Ha az is\_vertical false, akkor balról jobbra sorokat vizsgál, a bal alsó sarokból indulva.
* A képen vázlat, diagram, Műszaki rajz, Tervrajz látható

  Automatikusan generált leírásNode: A feladat megoldását fákban tároljuk. Egy üveglaphoz egy fa tartozik. A gyökér csúcs az érintetlen üveglap, és a fa első szintje a 1-cuttal kapott lapok, a második szintje a 2-cuttal kapottak, sít. Az oldaalt látható ábra személteti a fa struktúráját.
  + plate\_id: üveglap azonosítója,
  + x: a fa csúcsának bal alsó sarkának x koordinátája (az eredeti üveglap bal alsó sarka a (0, 0)),
  + y: a bal alsó sarok y koordinátája,
  + width: szélesség,
  + height: magasság,
  + type: a csúcs típusa
    - >=0: Item id
    - -1: waste,
    - -2 branch (van gyereke),
    - -3 residual
  + cut: vágás típusa, amivel a csúcsot kaptuk. Ha egy a gyökér csúcs, akkor 0.
  + residual: a csúcsban a vágások után maradt aktuális maradék.
  + parent: Pointer a parent node-ra. Ha ez a gyökér, akkor önmagára mutat.
  + children: pointerek listája, amik a gyerekekre mutatnak, balról jobbra rendezetten (a balrább található gyerekek időben hamarabb jöttek létre).
  + get\_root(): Visszadja a fa gyökér csúcsát.
  + last\_descendant: Visszaadja a csúcs részfájában találhat utolsó leszármazottat (jobbra lefele sétál).

Végük a solve.py fájlban a megoldás lényegét tartalmazó algoritmusok, és egy-két segédfüggvény található.

* first\_fit\_solve(id): Egy feladat id-ját megadva megoldja azt (létrehozza a megoldást tartalmazó erdőt). Részletesebben az „Algoritmus” fejezetben.
* first\_fit\_with\_rotate(id): Egy feladat id-ját megadva megoldja azt (létrehozza a megoldást tartalmazó erdőt). Részletesebben az „Algoritmus” fejezetben.
* backtrack\_solve(id, max\_depth, extended\_weight\_calculation, cut\_all): Egy feladat id-ját megadva megoldja azt (létrehozza a megoldást tartalmazó erdőt). Részletesebben az „Algoritmus” fejezetben.
* place\_item(current\_item, current\_node): A megadott itemet a megadott node-ból kivágja, így építve tovább a fát. A bal alsó sarokba próbálja behelyezni, ha ez nem megy, akkor a vágástól függően vagy soronként, vagy oszloponként próbálkozik.
* start\_new\_bin(bins, trees): Új üveglapot, új fát kezd. A régi fának waste-é alakítja a maradék lapjait.
* vertical\_cut(current\_node, x): Függőlegesen vág az x koordináta mentén, ezzel új csúcsot hozva létre.
* horizontal\_cut(current\_node, x): Vízszíntesen vág az y koordináza mentén, ezzel új csúcsot hozva létre.
* make\_node(current\_node): A node-ban található maradék reziduálist új waste-nodeként hozzáfűzi a fához, és nullázza a reziduálist.
* trim(current\_node, current\_item): egy trimet (4-cutot) végez.
* find\_right\_to\_x(current\_node, cut\_place): A cut\_placetől, mint x koordinátától jobbra megkeresi az első helyet, amin már lehet vágni a feltételek szerint. Ha cut\_place-ben lehet vágni, akkor azt adja vissza.
* find\_up\_to\_y(current\_node, cut\_place): A cut\_placetől, mint x koordinátától jobbra megkeresi az első helyet, amin már lehet vágni a feltételek szerint. Ha cut\_place-ben lehet vágni, akkor azt adja vissza.
* sum\_waste\_area(root): Összeadja a részfában található waste típusú csúcsok területét.
* backtrack(): a backtack\_solve() segédfüggvénye, ez a tényleges rekurzív megoldó.

## Algoritmus

Három különböző megoldó algoritmust csináltunk a feladathoz, amik egymásra épülnek, és egyre komplexebbek.

### First\_fit\_solve()

Ez az algoritmus egyszerre csak egy itemmel dolgozik. Veszi az első stacket, azon belül is az első itemet, és azt forgatás nélkül kivágja. A fának minden szintjén pontosan egy „maradék” lapot tartunk fenn, amiből meg lehet próbálni kivágni. Egy teljes üveglapból indulva a következő történik. Megpróbálja elhelyezni a jobb alsó sarokba az itemet. Ha oda nem tudja, mert van defekt, akkor kicsit feljebb próbálkozik: megkeresi a legfelső defektet, amivel még átfedne a kivágandó lap, és afölé próbálja elhelyezni. Az alatta található részt waste node-á alakítja

Ha az oszlopban nem járt sikerrel, akkor kicsit jobbra próbálkozik: annyival, hogy legalább egy defekt a vágástól balra legyen. A balra található wate-node-á alakítja. Ha nem a gyökérben vagyunk, hanem egy lejjebbi szinten, akkor a vágás irányától függően előbb felfele (függőleges vágás), vagy előbb jobbra (vízszintes vágás) halad.

Ha így sehova nem sikerült elhelyeznie, akkor a teljes lapot szemétté alakítja, és a szőlőjében tárolt maradék lapból próbálja meg kivágni, hasonló módon. Ha a gyökérből se sikerül kivágni, akkor új üveglapot kezd.

### First\_fit\_with\_rotate()

Ez az algoritmus az előzőhöz nagyon hasonló, az egyetlen különbség, hogy mohón megnézi, hogy ha forgatás nélkül helyezi el az itemet, akkor mennyi szemetet csinálna, illetve, ha forgatással, akkor mennyit, és azt a lehetőséget választja, amelyik kevesebb szemet eredményez. Ha a két érték megegyezik, akkor nem forgat.

### Bakctrack\_solve()

Ennek az algoritmusnak több változata van, melyeket a paraméterekkel lehet változtatni.

* max\_depth: A rekurzív hívások mélysége. alapértelmezetten 1. Ha ez az érték -1, akkor dinamikusan határozza meg a mélységet úgy, hogy a futásidő nagyjából 5 perc legyen.
* extended\_waste\_calculation: Alapértelmezetten hamis, azaz a csúcs kivágásával kapott szemét alatt csak a ténylegesen szemétként létrehozott csúcsokat számolja bele, a 1-cuttal kapott csucs alatti maradékokat, amiből még lehet vágni, azokat nem. Ha ennek a paraméternek igaz az értéke, akkor ezzel azt érjük el, hogy a nagyobb szabad helyekre a nagyobb területű itemeket részesítjük előnyben.
* cut\_all: bool, ha az érték True, akkor az adott mélységig kiszámolt optimális item-sorozat minden elemét kivágja, ha False, akkor csak az első elemét.

Nézzük most az algoritmus működését. Megnézi az összes stack első itemét, azaz minden olyan itemet, amit ki szabad vágni. Ezeket egy „szinten” végig próbálja, hogy hogy mi történne, ha ezt az itemet vágnánk ki eredeti állásban, vagy fordítottan. Mindkét esetben ezután egy szinttel lejjebb megy, újra az összes elérhető itemet tekinti, és ezt ismétli rekurzívan, amíg van item, vagy amíg el nem éri a maximális mélységet. Ha valamelyik feltétel miatt megáll, akkor kiszámolja, hogy ez a vágási minta mennyi szemetet eredményezett (az extended\_waste\_calculation értékétől függően). Az algoritmus ezek közül a minimumot fogja választani, és végrehajtani a vágásokat (a cut\_all értéktől függően), majd ismét meghívja az algoritmust.

## Eredmények

Az alábbi ábrán az A adathalmazon futtatott algoritmusok Waste arányíiból készült boksz-plotok láthatók (minél kisebb annál jobb). Látható hogy az első 3 algoritmus egyre jobban teljesít, az azonban meglepő, hogy az 1 mélységű backtracket egyik se tudja megverni. Az ábrán az is látszik, hogy van olyan adathalmaz, amin a Backtrack(1, ext), illetve a Bakctrack(2) jobban teljesít, mint a backtrack(1).

2 mélység esetén a Backtrack(2) ugyanazt az eredményt adja, mint a Backtrack(2, ext), és nagyobb mélység esetén is elhanyagolható a különbség, ami csak elsőre meglepő, de ha belegondolunk mit is csinál ez a paraméter, akkor ez a várható eredmény.

A képen szöveg, diagram, Tervrajz, Műszaki rajz látható

Automatikusan generált leírás

Ebben, és a többi adathalmazban is volt olyan példa, amire a nagyobb mélységű backtrack sokáig futott. A teljesség igénye nélkül például:

* A2, 2 mélység: 7,2 perc
* A5, 3 mélység: 25 perc
* B7, 2 mélység: 7,6 óra
* X2, 2 mélység: 11,5 óra

Így a Backtrack(2, one) algoritmust nem futtattuk a B és X adathalmazokon, továbbá 3 mélységet is csak egy konkért A adathalmazbeli példára próbáltunk ki, az A5-re. Ezen a Backtrack(3, one) pontosan ugyanúgy teljesített, mint a 2 mélységű változata. Az alábbi ábrán a többi algoritmus futásidejét, és waste arányát láthatjuk az A5-ös példán.