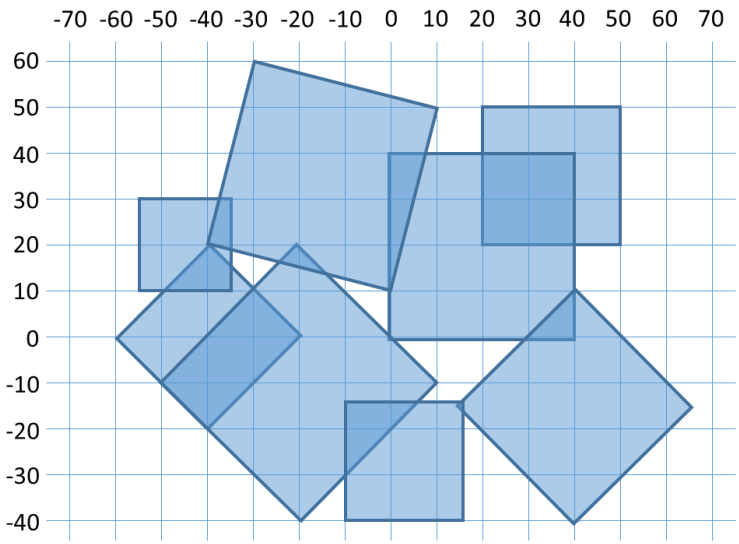
АНАЛИЗ НА РЕШЕНИЕТО НА ЗАДАЧА
КВАДРАТИ

Задачата изисква да се намери **лицето на обединение от квадрати в общо положение** или по-обобщено лице на обединение на многоъгълници в равнината.

Геометричният подход с намиране на обединението и изчисляване на неговото лице е доста тежък за реализация, защото при обединяването на квадрати се получават многоъгълници, които не са непременно изпъкнали, дори могат да съдържат затворени празни области в себе си(дупки).



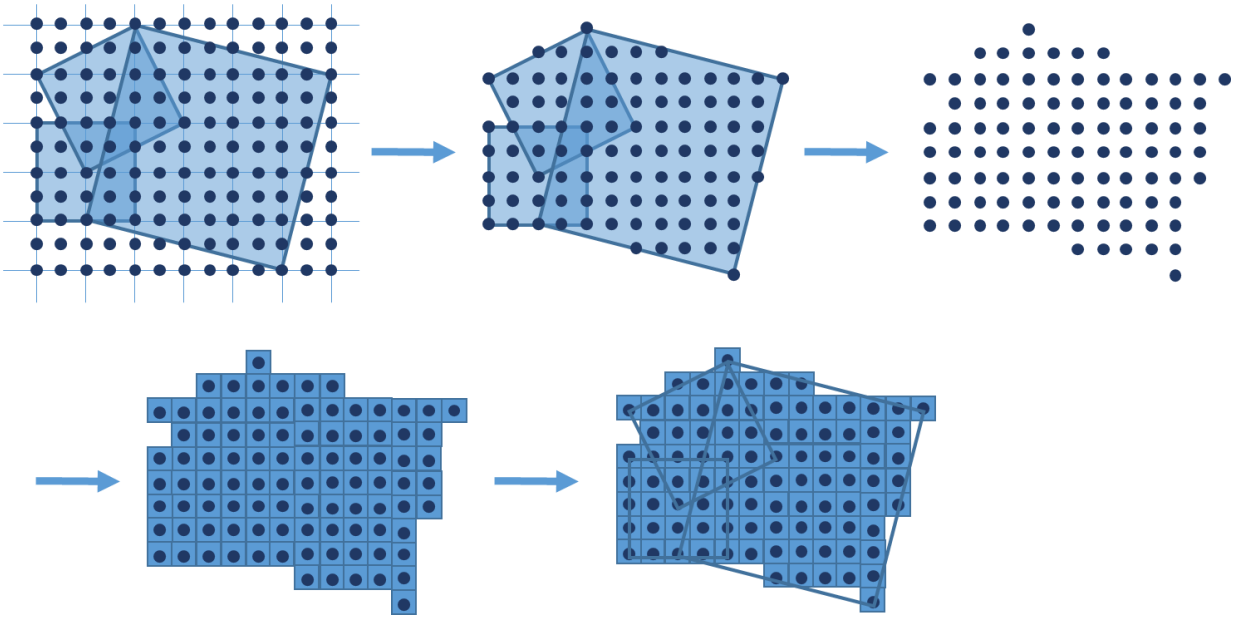
Следователно **намирането на чисто геометрично решение на тази задача ще е доста сложно** и не се очаква да се реализира в рамките на състезание по програмиране.

Като вземем предвид ограниченията за координатите на квадратите и изискването резултатът да се закръгли до най-близкото цяло число, можем да се насочим към **приближено решение (апроксимация)**.

Да разгледаме по какъв начин можем **да апроксимираме лицето** на обединението на квадрати. Можем ли да разбием първоначално зададените квадрати на по-малки фигури и да сумираме техните лица, когато по-малките фигури са вътре в някой от квадратите? Това би дало апроксимация на лицето. Да разгледаме няколко такива подхода.

Решение 1 – сканираща мрежа от точки

Сканираме равнината със стъпка **r = 0.0001** (по редове и колони) и за всяка **сканираща точка** проверяваме дали е вътре в някой от квадратите. Ако е вътре, добавяме към лицето на квадратите лицето r * r на **сканиращото мини-квадратче** около сканиращата точка. Изображението по-долу показва примерна мрежа от сканиращи точки и как те апроксимират обединението на квадратите чрез голям брой мини-квадратчета:



Стъпки от реализация на решението

- Прочитане на входните данни и изчисляване на координатите на четирите ъгъла за всеки квадрат.
- Проверка дали дадена **точка е вътре в някой от квадратите** (трябва да е в една и съща посока спрямо всяка от страните на квадрата).
- Имплементация на **сканираща точка с фиксирана стъпка** (напр. r = 0.1).

Решението е имплементирано във файла **`Squares-Scan-Points-Simple.cpp`**.

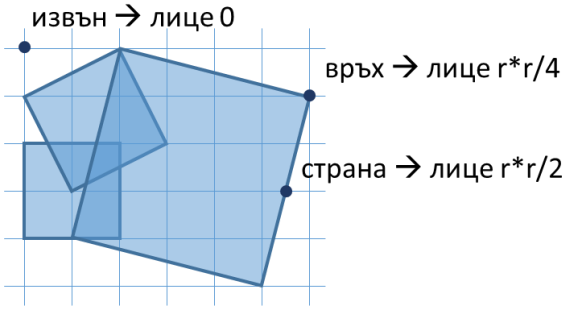
- При **r = 0.10** преминава **35%** от тестовете: ✗✗✗✗✓✓✓✓✓✓✗✗✗✗✓✗✗✗✗✗ 35 / 100
- При **r = 0.05** преминава **40%** от тестовете: ✗✗🕒🕒✓✓✓✓✓✓✗✓✗✗✗✗✗✗✓✗ 40 / 100
- При **r = 0.03** преминава **50%** от тестовете: ✓🕒🕒🕒✓✓✓✓✓✓✓✗✗✗🕒🕒✗✗✓ 50 / 100

Както се вижда при по-малки **r** се получават грешни резултати (липса на точност), а при по-големи се увеличава точността, но времето за изпълнение не достига.

Решение 2 – сканираща мрежа от точки с проверка за връх и стена

Върху алгоритъма със сканиращата мрежа от точки можем да направим следната **оптимизация**: да проверяваме къде се намира всяка от точките от сканиращата мрежа спрямо квадратите:

- **Вътре** в някой от квадратите→ добавяме лице **r * r**
- Върху **стена** от някой квадрат → добавяме лице **r * r / 2**
- Върху **връх** от някой квадрат → добавяме лице **r * r / 4**
- **Извън** всички квадрати → прескачаме точката (не добавяме нищо към лицето)



Решението е имплементирано във файла **`Square-Scan-Points-with-Side-Checks.cpp`**.

- При **r = 0.10** преминава **30%** от тестовете: ✗✗✗✗✓✓✓✓✓✓✗✗✗✗✗✗✗✗✗✗ 30 / 100
- При **r = 0.05** преминава **50%** от тестовете: ✗✗🕒🕒✓✓✓✓✓✓✓✓✗✗✓🕒🕒✗✓✗ 50 / 100

- При **r = 0.03** преминава **45%** от тестовете:  45 / 100

Ако се замислим, това решение би трябвало да работи по-точно, когато сканиращите точки уцелват по-често **ъглите** и **стените** на квадратите. Там често пъти има симетрични ситуации и грешките в изчисленията се унищожават вместо да се натрупват. Неточности ще има, когато един връх е общ за няколко квадрата и когато една сканираща точка лежи на повече от една стена на квадрат, както и когато сканиращите точки не уцелват стените на квадратите.

Можем да очакваме при сканираща стъпка, която е степен на 1/2, да уцелваме по-често ъгли и стени на квадратчета. Да тестваме със стъпки 1/4, 1/8 и 1/16:

- При **r = 0.25** (1/4) резултатът е **50%**:  50 / 100
- При **r = 0.125** (1/8) резултатът е **60%**:  60 / 100
- При **r = 0.0625** (1/16) резултатът е **50%**:  50 / 100

Вижда се, че има подобрение на резултатите но те са далеч от цялостното решение на задачата.

Решение 3 – сканираща мрежа от точки с проверка за връх и стена и минимална сканираща стъпка

Следващата **оптимизация** е да изчислим така **сканиращата стъпка `r`**, че да е възможно най-малка, но изчисленията да се събират в допустимото време за изпълнение на задачата. Можем да заложим някакъв фиксиран **максимален брой операции `MAX_OPERATIONS`**, които искаме алгоритъмът да извърши. Можем да сметнем броя операции за проверка на положението на точка спрямо зададените квадрати при стъпка **`r`** по формулата:

```
operations = n * (maxX - minX) / r * (maxY - minY) / r
```

Вече стана ясно, че стъпката трябва да е **степен на 1/2**, за да се падат сканиращите точки по-често по стените и в ъглите на квадратите. За да изчислим сканиращата стъпка **`r`**, можем да започнем от **`r`** = 1 и да делим последователно **`r`** на 2 докато броят операции не достигне **`MAX_OPERATIONS`**.

Стойността на **`MAX_OPERATIONS`** можем да изберем чрез няколко опита, съобразявайки допустимото време за изпълнение на задачата. Установяваме, една добра настройка е **`MAX_OPERATIONS`** = **`6 000 000`**.

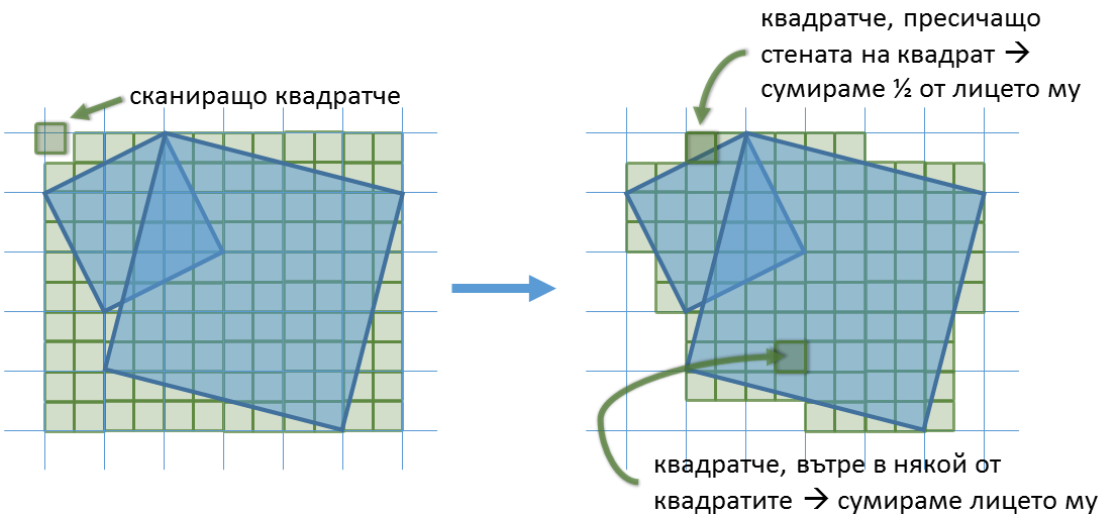
Решението е имплементирано във файла **`Squares-Scan-Points-Optimized.cpp`**. Подобреното решение преминава успешно **75%** от тестовете:  75 / 100

Макар и да е най-доброто за момента, това решение не е съвършено, така че продължаваме да търсим по-добро.

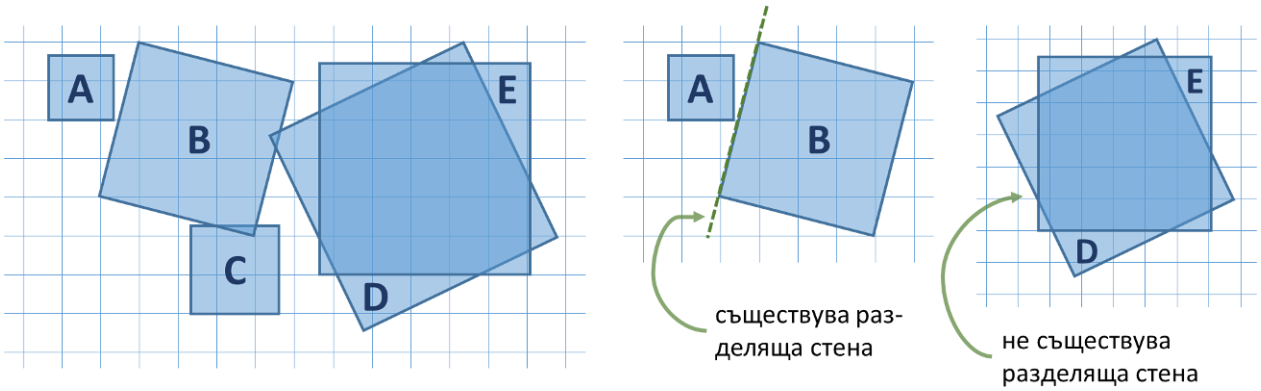
Решение 4 – сканиращи мини-квадратчета

Да разгледаме един малко по-различен подход. Можем ли вместо сканираща мрежа от точки да използваме **сканираща мрежа от мини-квадратчета**? Квадратчета, с размер **`r`** x **`r`**, които за наредени плътно едно до друго по хоризонтал и по вертикал. На фигурата по-долу е визуализирана мрежа от мини-квадратчета. Някои от тях са извън входните квадрати, други са изцяло в някой от тях, а трети са частично вътре, частично вън от входните квадрати (пресичат се с тях).
Ето как изглежда постъпково алгоритъмът:

- Разделяме полето на мини-квадратчета с фиксиран размер, примерно 0.1 x 0.1 (r = 0.1).
- Проверяваме всяко мини-квадратче дали се засича с някой от големите квадрати:
  - Ако мини-квадратче **вътре** в някой от квадратите → сумираме лицето му (r * r)
  - Ако мини-квадратче е **вън** от всички квадрати → добавяме 0
  - Иначе (имаме **пресичане** с някой квадрат) → добавяме ½ от лицето му (r * r / 2)
- Сканираме само областта от най-левия връх до най-десния и от най-горния до най-долния (minX … maxX, minY … maxY).



Това решение е по-сложно за имплементация, тъй изисква проверка дали два квадрата се пресичат. **Проверката за пресичане на квадрати** за прави по следния начин:



- Ако **стена** от някой квадрат **разделя равнината на две полуравнини**, така че двата квадрата попадат в различни полуравнини, то квадратите **не се пресичат** (разделени са).
- Ако не съществува **разделяща стена**, то квадратите **се пресичат**.

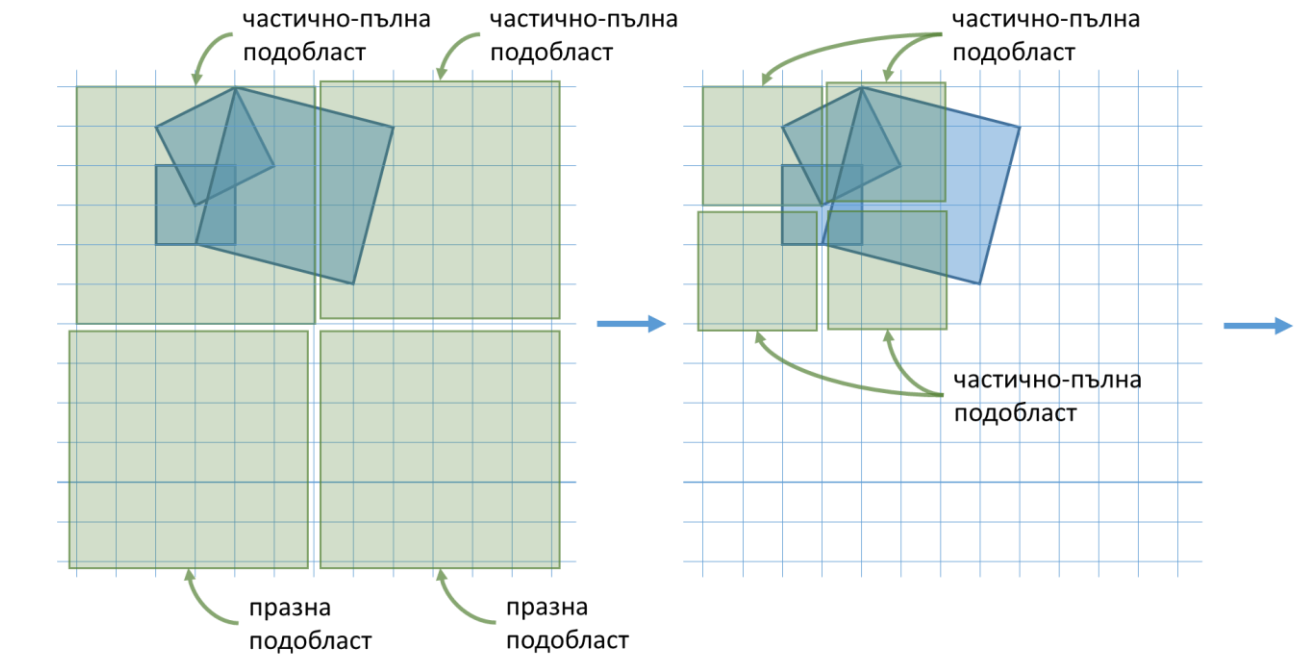
Решението е имплементирано във файла **`Squares-Scan-Mini-Squares.cpp`** и при **r=0.1** преминава успешно **45%** от тестовете: ✓✓✓✓✓✓✓✓✓✗✗✗✗✗✗✗✗✗✗ 45 / 100

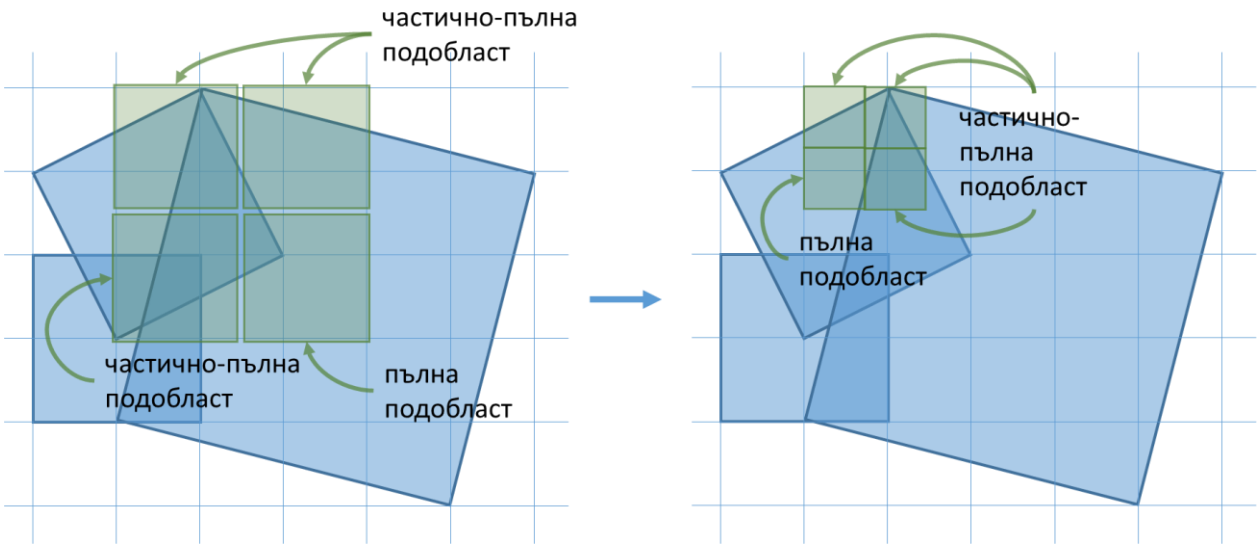
Изглежда идеята няма шанс да проработи, защото ако намалим **r**, ще имаме недостиг на време, а ако увеличим r, ще загубим точност. Продължаваме да търсим по-добро решение.

Решение 5 – разделяне на равнината на 4 подобласти

Можем ли да разширим и обобщим идеята за сканиращи мини-квадратчета? Например да **увеличаваме** размера на сканиращите мини-квадратчета когато сме вътре или извън някой от входните квадрати (за да сканираме по-наедро) и да **намаляваме** размерите на сканиращите мини-квадратчета около стените и ръбовете на входните квадрати (за да сканираме по-прецизно).

Можем да започнем с една голяма правоъгълна подобласт, побираща цялото поле с всички квадрати. Можем да я разделим през средата на **4 еднакви подобласти** и за всяка от тях да проверим дали е **празна**, **пълна** или **частично-пълна**. Процесът е илюстриран на фигурата по-долу:





Получава се разделяне на равнината като при структурата от данни **quadtree** – https://en.wikipedia.org/wiki/Quadtree.

Алгоритъм за изчисление на лицето на квадратите в дадена правоъгълна подобласт от равнината:

- Започваме с подобласт, определена от най-лявата, най-дясната, най-горната и най-долната точки, в които има връх на някой от входните квадрати: (minX … maxX, minY … maxY).
- Всяка достатъчно голяма подобласт **се разбива на 4 еднакви подобласти**.
- **Празните подобласти** (които са извън всички входни квадрати), не се разглеждат и тяхното лице не участва в търсената площ.
- **Пълните подобласти** (които са вътре в някой от входните квадрати), се включват в търсеното обединение и тяхното лице се добавя към търсената площ.
- **Частично-пълните подобласти** (които са частично вътре и частично вън спрямо някой от входните квадрати и не са изцяло вътре в някой друг от квадратите), се **разделят на 4 подобласти** и се разглеждат рекурсивно.
- Процесът на рекурсивно разбиване приключва, когато разглежданата подобласт **твърде малка** (под определен размер – константата `SPLIT_RESOLUTION`). Тогава към търсената площ се добавя **½ от нейното лице**.

Решението е имплементирано във файла **`Squares-4-Subareas.cpp`**. Както и при предходните решения, точността зависи от сканиращата дълбочина **r**. Трябва да изберем такава дълбочина, че да получим максимална точност, без да надвишаваме ограниченото време.

- При **r = 0.001** резултатът е **35%**: 
- При **r = 0.0001** резултатът е **70%**: 
- При **r = 0.00001** резултатът е **100%**: 
- При **r = 0.000001** резултатът е **85%**: 

Забелязваме, че при по-голяма сканираща дълбочина (по-малко **r**) се вдига точността на резултата, но не можем безкрайно да намаляваме **r**, защото решението става все по-бавно. Можем да пробваме със сканиращи дълбочини, степени на ½:

- При **r = 1/4096** резултатът е **60%**: 
- При **r = 1/32768** резултатът е **75%**: 
- При **r = 1/65536** резултатът е **95%**: 
- При **r = 1/131072** резултатът е **100%**: 
- При **r = 1/262144** резултатът е **100%**: 
- При **r = 1/524288** резултатът е **90%**: 

Извод: оптималната дълбочина на сканиране е **r = 1/262144** и при нея решението с разделяне на 4 подобласти работи с достатъчно голяма точност.

Автор: Светлин Наков