

# Minesweeper

Radosław Urbaś

radoslaw.urbas@gmail.com

<http://student.uci.agh.edu.pl/~urbas/minesweeper>

## Streszczenie

Tematem projektu jest program rozwiązujący problem (klasy  $NP - COM$ ) prawidłowego rozmieszczenia min na zadanej planszy (zgodnie z specyfikacją <sup>a</sup>).

Realizowany w ramach przedmiotu Teoria Obliczeń i Złożoności Obliczeniowej, trzeci rok studiów dziennych na kierunku Informatyka, KI EAIE AGH Kraków.

Semestr zimowy 2005/2006.

Implementacja w języku C++ (ANSI/ISO). Wersja 1.0.

---

<sup>a</sup><http://winnie.ics.agh.edu.pl/dydaktyka/toizo/minesweeper/>

## 1 Ogólny charakterystyka struktur danych i opis zaimplementowanych metod.

### 1.1 Klasa *Area*.

Plansza jest przechowywana jako obiekt klasy *Area*. Klasa ta opakowuje STL'owy kontener *vector*  $\langle Field \rangle$ .

#### 1.1.1 *Area()*.

Konstruktor klasy wczytuje dane z standardowego wejścia. Tworzy obiekty klasy *Field* i umieszcza je w kontenerze. Ustawiane są także wskaźniki na sąsiadów umieszczone w obiektach klasy *Field*, ilość min do rozłożenia, ilość zakrytych sąsiadów. Pola sąsiadujące z cyframi zero ustawiane są jako puste. Poza tym w drugim kontenerze *vector*  $\langle Field* \rangle$  przechowywane są wskaźniki do obiektów pól z cyframi, które trzeba zaspokoić (w celu zmniejszenia przeglądanej struktury z całej planszy do interesujących obiektów). Zlicza liczbę min które już są rozłożone na planszy wejściowej.

#### 1.1.2 *void display()*.

Wyświetla końcowy wynik w żądanej w specyfikacji formie. Pisze na standardowe wyjście.

## 1.2 Klasa *Field*.

Poszczególne pola planszy przechowywane są jako obiekty klasy *Field*. Klasa ta zawiera atrybuty informujące o typie pola: mina, pole z cyfrą, zakryte pole, puste pole. Wskaźniki na sąsiadów na planszy.

### 1.2.1 *Field(short)*.

Konstruktor, ustawia atrybuty na podstawie liczby podanej jako parametr. 0-8 liczba min, 9 pole zakryte, -1 mina.

### 1.2.2 *void notifyBomb()*.

Zmniejszenie licznika min do rozłożenia u sąsiadów. Wywoływana przy początkowej analizie planszy - w momencie wykrycia min ustawionej na początkowej planszy.

### 1.2.3 *void notifyCover()*.

Powiadomienie o tym, że pole jest zakryte (zwiększenie licznika zakrytych sąsiadów w polach sąsiednich). Wykorzystywana zarówno na etapie rozpoznania planszy, przygotowania jak i w głównym algorytmie.

### 1.2.4 *void notifyZero()*.

Oznaczenie sąsiadów jako pola puste, gdy na planszy pojawia się liczba zero.

### 1.2.5 *void notifyUncover()*.

Powiadomienie, że stan pola się zmienił - przestało być zakryte.

### 1.2.6 *void putBomb()*.

Położenie miny - oznaczenie pola jako mina, powiadomienie sąsiadów o tym, że pole przestało być zakryte.

### 1.2.7 *void notifyPotentialZero()*.

Powiadomienie o tym, że pole jest potencjalnie puste. Zmiana wartości atrybutów określających status pola, wywołanie metody *notifyUncover()*. Metoda z której korzysta się na etapie zasadniczego rozwiązywania.

### 1.2.8 *void unNotifyPotentialZero()*.

Powiadomienie o tym, że pole przestało być potencjalnie puste. Zmiana wartości atrybutów określających status pola, wywołanie metody *notifyCover()*. Wykorzystana analogicznie jak metoda ją uzupełniająca,

### 1.2.9 *void putPotentialBomb()*.

Położenie potencjalnej miny. Z wywołaniem *notifyPotentialZero()* jeśli licznik min do położenia spadł do zera.

### 1.2.10 *void unPutPotentialBomb()*.

Zdjęcia potencjalnej miny. Z wywołaniem *unNotifyPotentialZero()* jeśli licznik min został inkrementowany z wartości zero.

## 1.3 Klasa *Solver*.

Klasa *Solver* ma zaimplementowane metodę służące rozwiązaniu planszy, a także metody pomocnicze, przygotowujące planszę do działania głównego algorytmu. Obiekt tej klasy przechowuje wskaźnik na obiekt planszy, którą ma za zadanie rozwiązać.

### 1.3.1 *Solver(Area\*)*.

Konstruktor dokonuje kilku działań, po pierwsze wywołuje metodę *prepare()*, następnie *findArea()*. Dodatkowo monotonizuje podobszary planszy wg ich liczności, tzn przyporządkowuje numer zero najmniej licznemu obszarowi. Ma to na celu ograniczenie liczby powrotów rekurencji, poprzez rozpatrzenie obszarów z najmniejszą liczbą kombinacji rozłożenia min na początku.

### 1.3.2 *void prepare()*.

W pętli przegląda pola planszy. Jeśli natrafi na pole o ilości min do rozłożenia równej zero - wywołuje metodę *notifyZero()*. Następnie szuka takich, dla których liczba min do rozłożenia jest równa liczbie zakrytych sąsiadów. Dla każdego takiego pola wywoływana jest metoda *putBomb()*. Potem wywołuje metodę *putNoneImportantBombs()*. Warunkiem wyjścia z pętli jest nie rozłożenie nowej miny w danym kroku.

### 1.3.3 *void putNoneImportantBombs()*.

Rozkładane są miny na polach, w sytuacji gdy nie ma to wpływu na resztę planszy. Przykładowo jeśli pole o wartości 1 ma dwóch zakrytych sąsiadów, którzy nie mają żadnego innego sąsiada z cyfrą poza danym, to można położyć minę na dowolnym z nich, a drugie zaznaczyć jako pole puste.

### 1.3.4 *void findArea()*.

Metoda ta odpowiada za znalezienie obszarów pseudoniezależnych na planszy. Obszar taki, to zbiór pól z cyframi, który jest rozłączny z pozostałymi, tzn nie ma wspólnych sąsiadów na których można położyć mine z innymi obszarami. Jedynym czynnikiem łączącym obszary pseudoniezależne jest globalna liczba min od rozłożenia. Proces ten realizowany jest poprzez - znalezienie

pierwszego pola nieoznaczonego jako odwiedzone, oznaczeniu go, przypisaniu do odpowiedniego podobszaru - poprzez nadanie wartości atrybutowi obiektu *Field* odpowiadającemu za obszar kolejnego numeru (począwszy od zera), a następnie wywołaniu metody *findNeighbor(Field\*, short)* z parametrami - wskazanie na rozpatrywane pole i numer podobszaru. Szukanie kandydatów na elementy nowego podobszaru kończy się po przejściu przez całą planszę.

### 1.3.5 void *findNeighbor(Field\*, short)*.

Wyszukuje pola z cyframi, które są w oddalony od danego o nie więcej niż jedno pole zakryte, lub z nim bezpośrednio graniczą. Oznacza to, że mają one wspólnych sąsiadów, na których można położyć mine. Takie pole, jest oznaczone jako przynależące do tego samego obszaru, oznaczane jako odwiedzone i funkcja jest wywoływana rekurencyjnie - tym razem z parametrem w postaci wskazania na nowe pole należące do podobszaru i tym samym numerem. Gdy zakończą się wszystkie wywołania rekurencyjne tej funkcji - oznacza to, że zostały znalezione wszystkie pola zależne od początkowego i wszystkim polom należące logicznie od tego samego obszaru został nadany ten sam numer.

### 1.3.6 bool *putFreeBombs()*.

Metoda ta rozkłada miny, na polach wolnych, tak aby wykorzystać zadaną ilość min. Jeśli liczba min na wejściu jest niekreślona (zero), od razu zwraca true, jeśli liczba min pozostałych jeszcze do rozłożenia jest równa zero, także zwraca true. W innym przypadku liczy pola, na których można położyć miny. Jeśli ich liczba jest mniejsza od pozostałych min - zwraca false, w przeciwnym wypadku rozkłada miny na pierwszych nadających się do tego polach - do wyczerpania puli min.

### 1.3.7 bool *solve(int)*.

Jako parametr przyjmuje numer podobszaru jakim ma się zająć. Wartość zwracana oznacza, czy udało się rozłożyć żadaną liczbę min, czy nie. Szczegóły działania opisane są w punkcie mówiącym o algorytmie działania programu.

## 2 Algorytm.

Algorytm składa się z kilku etapów. Zostaną one omówione w odniesieniu do metod opisanych w punkcie dotyczącym struktur danych i zaimplementowanych metod.

### 2.1 Przygotowanie planszy.

Pierwszy etap realizowany jest na etapie konstruktora klasy *Area*. Następnie rozmieszczane są miny wokół pól z cyframi, dla których istnieje tylko jedna możliwość położenia - metoda *prepare()* klasy *Solver*. W kolejnym kroku umieszczane

są miny wokół pól, dla których położenie min nie ma wpływu na resztę planszy - metoda *putNoneImportantBombs()* w klasie *Solver*.

## 2.2 Podział na obszary pseudoniezależne.

Zbiór pól z cyframi dzielony jest na podzbiory, które nie mają ze sobą wspólnych sąsiadów, na których istnieje możliwość położenia miny. Punkt ten realizuje metoda *findArea()* w klasie *Solver*. Każdemu polu z cyfrą nadawany jest numer obszaru do którego należy. Obszary te nazwane zostały pseudoniezależnymi, gdyż nie zależą od siebie bezpośrednio, aczkolwiek zależą pośrednio, poprzez całkowitą liczbę min na planszy. Nie dotyczy to plansz dla których liczba min nie została określona - wtedy obszary te są od siebie całkowicie niezależne. Dodatkowo obszary te są monotonizowane według ich liczności, w celu rozpatrzenia najpierw najmniejszych (w konstruktorze klasy *Solver*).

## 2.3 Główny algorytm.

Wszystkie wcześniejsze działania mają na celu ułatwienie znalezienia rozwiązania głównemu algorytmowi. Jego implementacją jest metoda *solve(int)* w klasie *Solver*. Parametrem metody jest numer obszaru pseudoniezależnego. Metoda wywoływana jest z parametrem równym zero, na obiekcie klasy *Solver*, utworzonym w oparciu o obiekt klasy *Area* reprezentujący planszę do rozwiązania. Funkcja wywoływana jest rekurencyjnie. Zwraca false, jeśli krok nie prowadził do poprawnego rozwiązania, true jeśli znaleziono poprawne rozwiązanie. Na początku sprawdzane są warunki poprawności częściowego rozwiązania. Po pierwsze czy liczba rozłożonych min nie jest większa od zadanej liczby min (po uwzględnieniu min początkowo rozłożonych na zadanej planszy, nie dotyczy to przypadku gdy liczba min nie jest określona). Następnie dla wszystkich pól z cyframi (wskaźniki przechowywane na osobnej liście) sprawdzane jest, czy gdzieś nie zaszła sytuacja taka, że licznik min jest ujemny - wtedy dane pole sąsiaduje z większą ilością min niż powinno, czy liczba min pozostałych do rozłożenia nie jest większa od ilości dostępnych nieodkrytych jeszcze sąsiadów tego pola. Jeśli zaszedł któryś z warunków funkcja zwraca false. Jednocześnie liczona jest ilość nie zaspokojonych jeszcze pól. Jeśli licznik ten jest równy zero, to wywoływana jest funkcja dla obszaru o numerze o jeden wyższym niż bieżącym, o ile nie jest to ostatni obszar. W takim przypadku wywoływana jest funkcja *putFreeBombs()* obiektu klasy *Solver*, a jej wynik jest zwracany jako rezultat całej funkcji. Jeśli zwróci ona true, kończy się działania algorytmu, wszystkie miny zostały rozłożone, jeśli false, należy się cofnąć gdyż to rozwiązanie nie spełnia warunków. Analogicznie - przy wywołaniu funkcji dla kolejnego podobszaru - rezultat zwracany, jest zwracany przez funkcję wywołującą. Jeśli zwróciła false, należy się cofnąć i rozważyć inne rozwiązanie w tym obszarze (możliwość cofania się pomiędzy różnymi podobszarami występuje tylko w przypadku gdy liczba min jest ustalona, jeśli jest dowolna, to nie mają one żadnego wpływu na siebie). Jeśli nie zaszedł żaden z wyżej wymienionych warunków przystępuję się do rozmieszczania min w danym podobszarzu. Z listy pól z cyframi należących od danego podobszaru wybierane

jest pierwsze (lub ostatnie) pole z licznikiem min do rozłożenia większym od zera. W zależności od wartości tego licznika, rozkładana jest odpowiednia liczba min. W jednej próbie rozkładana jest nie jedna mina, a od razu cały komplet min dla tego pola. Podejmowana jest próba rozłożenia min wokół pola (wywołanie metody *putPotentialBomb()* dla pierwszej kombinacji sąsiadów, licznik położonych min jest odpowiednio zwiększany) i następuje rekurencyjne wywołanie funkcji. Jeśli zwróci ono true, zwracane jest true na danym poziomie rekurencji. Jeśli nie powiedzie się, cofane są zmiany w planszy (*unPutPotentialBomb()* dla tego samego zestawu pól oraz licznik min jest zmniejszany odpowiednio) i próbowana jest druga kombinacja sąsiednich pól w analogiczny sposób. Jeśli żadna z próbowanych kombinacji nie okaże się na danym etapie poprawna, z tego poziomu rekurencji zwracane jest false. Co skutkuje podjęciem próby dla innej kombinacji min w poziomie wyżej, lub całkowitym zakończeniem funkcji jeśli nastąpiło do na pierwszym poziomie - wtedy planszy nie da się rozwiązać. Tak więc wyjście z funkcji z pozytywnym rezultatem nastąpi gdy w ostatnim podobszarze nie ma już niezaspokojonych pól z cyframi, a wywołanie metody *putFreeBombs()* zakończy się sukcesem. Wynik negatywny wystąpi, gdy na pierwszym poziomie rekurencji algorytm wypróbuje wszystkie możliwe kombinacje dla pierwszego podobszaru.

### 3 Analiza złożoności.

#### 3.1 Złożoność czasowa.

Złożoność najgorszego przypadku jest wykładnicza, a dokładnie wynosi  $70^n$ , gdzie  $n$  jest liczbą pól z cyframi, wszystkie te pola mają wartość 4, i ośmiu możliwych do rozłożenia na nich miny sąsiadów oraz nie da się wyodrębnić żadnego obszaru niezależnego. Wtedy algorytm sprowadza się do klasycznej rekurencji z powrotami (z dokładnością do etapu wstępnego). Wartość 70 - to liczba kombinacji rozłożenia czterech min na ośmiu polach, maksymalna jaka może wystąpić w zadaniu dla pojedynczego pola.

#### 3.2 Złożoność pamięciowa.

Złożoność pamięciowa (jeśli chodzi o struktury danych) jest liniowa względem całkowitej ilości pól. Gdyż przy wywołaniach rekurencyjnych nie są kopiowane żadne wartości. Poza liniowość nie wychodzi także ilość pamięci zużyta na stosie systemowym, gdyż w najgorszym przypadku metoda *solve()* zostanie wywołana tyle razy ile istnieje kombinacji ułożenia min na planszy, ale jednocześnie aktywnych może być tylko tyle wywołań rekurencyjnych ile jest pól z cyframi. Na stosie odkładane są tylko adresy powrotu, oraz trzy zmienne w funkcji (dwa razy liczba całkowita i jeden wskaźnik).

#### 3.3 Rozmiar problemu.

Rozmiar problemu stanowi nie bezpośrednio rozmiar planszy ale ilość kombinacji rozłożenia min dla pól z cyframi. Duży wpływ na to ma sposób rozłożenia

tych pól na planszy, ich gęste i równomierne rozłożenie, szczególnie z średnimi wartościami, powiększa znacząco rozmiar problemu.

#### 4 Silne strony algorytmu.

Silną stroną algorytmu jest przygotowanie planszy przed uruchomieniem głównego algorytmu. Wyciągane są wszystkie oczywiste wnioski na temat rozkładu min z planszy wejściowej. Rozkładane są miny dla tych pól, gdzie nie ma to wpływu na resztę planszy. Dużym usprawnieniem jest także wyodrębnienie obszarów pseudoniezależnych. W jednym z testowych przypadków, wprowadzenie tego etapu, spowodowało spadek liczby wywołań rekurencyjnych z ponad pół miliona do dziewięciu tysięcy. Algorytm dobrze radzi sobie w przypadku plansz rzadkich, gdzie liczba pól z cyframi jest nieduża, i można wyodrębnić wiele obszarów niezależnych. Zadawalające wyniki osiągnęte są także dla plansz gdzie gęsto rozłożone są początkowe miny i wiele jest pól o wartości zero. Wtedy etap początkowy algorytmu rozwiązuje problem rozłożenia dużej ilości min. Także plansze z niekreśloną ilości min są znacznie łatwiejsze od tych, gdzie liczba min jest ściśle określona, gdyż nie występuje wtedy możliwość powrotu - po wejściu do metody *putFreeBombs()*, a wystarczy zaspokoić pola z cyframi, co w połączeniu dodatkowo z dużą ilością obszarów niezależnych daje dobre wyniki, nawet dla plansz o stosunkowo dużych rozmiarach.

Dla poprawy wydajności czasowej programu, wszystkie metody zostały zaimplementowane jako inline. Dodatkowo wiele elementów w kodzie zostało rozpisanych ręcznie dla ograniczenia ilości wywołań metod i wykonań pętli.

#### 5 Słabe strony algorytmu.

Najsłabszą stroną algorytmu jest jego złożoność. Jest to algorytm dokładny, więc złożoność najgorszego przypadku jest wykładnicza dla tego problemu należącego do klasy  $NP - COM$ . Poza tym podział na obszary pseudoniezależne nie zawsze daje oczekiwane zyski. W wielu przypadkach nie da się podzielić zbioru pól z cyframi w ogóle, lub jest to wyodrębnienie kilku bardzo małych obszarów, a pozostałe pola tworzą jeden duży. I tak wprowadza to przyspieszenie algorytmu, aczkolwiek nie jest ono równomierne dla wszystkich plansz. Trudne są dla algorytmu plansze, dla których nieduży postęp daje etap wstępny, oraz etap podziału na obszary niezależne. Bardzo trudną planszą jest taka, gdzie liczba min do rozłożenia ma duży wpływ na ilość rozwiązań. Gdy rozłożenie min zaspokajających pola z cyframi nie jest poprawne ze względu na globalną ilość min, następuję powrót z ostatniego zagłębienia rekurencji i trzeba wracać do poprzednich obszarów pseudoniezależnych, próbując kolejno możliwe zmiany w każdym z nich. W takim przypadku podział na obszary nie daje dużych zysków.

Dla zwiększenia przeciętnej wydajności algorytmu, na początku programu losowana jest wartość: 0 lub 1. Decyduje ona o tym, czy w przy rozkładaniu min

wokół pól z cyframi zaczynamy od pierwszego pola na liście, czy ostatniego. Stanowi to zarówno mocną jak i słabą stronę algorytmu. Słabą, gdyż wprowadza niedeterminizm. Mocną, ponieważ umożliwia różne podejście do problemu, a z nabytego w czasie realizacji projektu doświadczenia wynika, że kierunek przeglądania pól ma bardzo duże znaczenie w przypadku wielu plansz, może nastąpić redukcja ilości wywołań rekurencyjnych o kilka rzędów wielkości w przypadku zmiany tego kierunku. Także monotonizacja obszarów według ich liczności daje poprawę średniej efektywności algorytmu.

## 6 Możliwości udoskonalenia algorytmu.

Efektywność programu można nieco podnieść poprzez zoptymalizowanie dostępu do danych i przeimplementowaniu samych struktur danych. W samej logice algorytmu nie ma zbytnio elementów do udoskonalenia. Można jedynie podejść do problemu z zupełnie innego punktu widzenia.

Rozpatrywane, lecz niezrealizowane pomysły:

- Redukcja do innego problemu  $NP - COM$ . I implementacja dla niego algorytmów przybliżonych, które zostały już wcześniej opracowane.
- Próba wyliczania prawdopodobieństw położenia miny na danym polu. Z uwzględnieniem przyjętej strategii: rozmieszczania minimalnej liczby min (próba nasycenia jak największej ilości pól z cyframi przez jedną minę), maksymalnej ilości min (nasycenie jak najmniejszej ilości pól przez położenie danej miny), lub form pośrednich. Odpowiednią strategię należałoby przyjąć na podstawie analizy planszy. Wymagałoby to prób na dużej populacji plansz i doświadczalnym wyznaczeniem 'typów' plansz dla których dana strategia jest prawdopodobnie najlepsza.