Adatszerkezetek és Algoritmusok
Pót (nagy) házifeladat
Leadási határidő: december 10. 23:59 CET

Bevezető

A pótháziban az eddig tanultak alapján kell írni egy adatszerkezetet és egy hozzá tartozó algoritmust. Egy tologatós játékot kell megvalósítani, majd ehhez egy megoldót kell írni.

Tologatós játék

Egy $N * N$-es négyzet mezőin $N^2 - 1$-ig találhatóak a számok, egy mező pedig üresen marad. Ezek a mezők mozgatható csempék, az üres helyre a szomszédos csempék közül bármelyiket oda lehet tolni. A cél az, hogy sorbarendezzük a csempéket, mégpedig a megadott minta szerint, az a feladat, hogy megállapítható legyen, hogy legkevesebb hány mozgatással érhető el ez az állapot. A mi esetünkben $N = 3$, tehát 8 darab csempén lesz szám, egy lesz üres.
Ahhoz hogy meg lehessen oldani ezt a játékot, egy fabejárást kell végrehajtani, ahol az optimális ágat kell megkeresni, ami elvezet a kezdőtől a végállapotig.

Optimális ág megkeresése

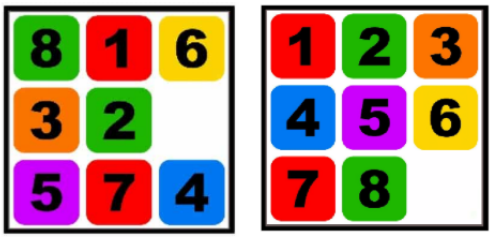
Ahhoz, hogy megtalálható legyen az optimális ág *heurisztikus keresést* kell alkalmazni. A *heurisztikus keresés* egy olyan keresés, amely heurisztikus érték alapján optimalizálja a keresést. A keresés közben minden lépésnél figyelembe veszi a heurisztikus értéket és azt az ágat fogja követni, amely a legoptimálisabb.

Heurisztikus érték(cost function)

A heurisztikus érték f.

$f = h + g$

Ahol h, megadja, hogy milyen messze van a végső állapot, míg g, hogy milyen mélyen van a csúcs a fában.



Kezdő és végső állapota a játéknak

Implementáció

State

Ahhoz, hogy reprezentálni lehessen a játékot, szükség van egy osztályra, ez lesz a **State**. Ez az osztály a mezők egy egyedi kombinációját tárolja. Ezek a statek, amelyek egy-egy node-ot fognak reprezentálni a fában.

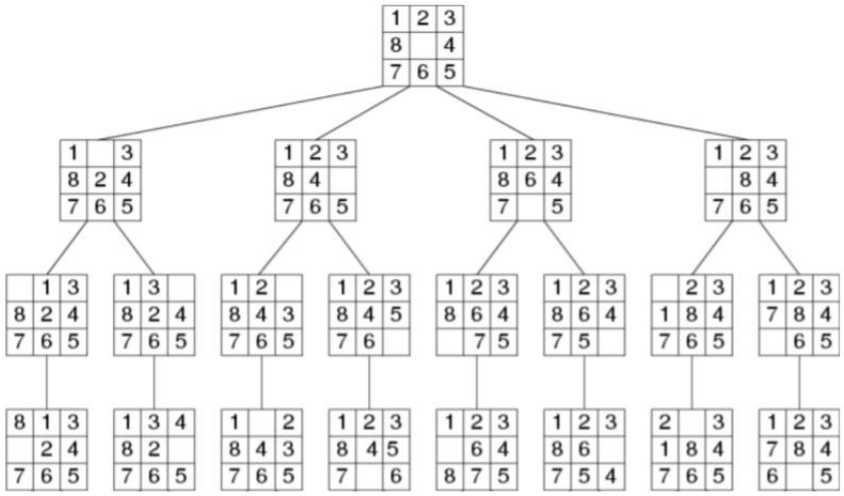
A mezők értékét egy array-ban tárolja az osztály. Az array indexei mutatják, hogy azon a csempén milyen értékű mező van.

State - API

```
class State {
    /*TODO*/
public:
    //Konstruktor és copy konstruktor
    State (int, const std::vector<int>&);
    State(const State&);
    //Assignment operator
    State& operator=(const State&);
    //==,!= operator overload
    bool operator==(const State&);
    bool operator!=(const State&);
    //Alapműveletek
    void swap_values(int, int); //Kicseréli az értékeket a megadott indexeken
    [[nodiscard]] const std::vector<int>&get_array() const;
    [[nodiscard]] int get_number_of_rows() const;
    };
```

Node/State Tree

A Stateket faként fogjuk tárolni, a gyökere a kiinduló állapot lesz, a gyerekek, pedig azok az állapotok, hogy merre lehet tovább menni. A Node a mostani esetben ne a gyerekeire, hanem a szülőre mutasson. A Node ezen kívül tárolja, hogy melyik szinten van és, hogy melyik **State**-t reprezentálja. Fontos, hogy nem szükséges ezt a fát felépíteni, ez az osztály 1-1 Node-ot fog reprezentálni.



State tree/Node

Node - API

```
class Node{
    /*TODO*/
public:
    Node(const State&, Node*, int);
    [[nodiscard]] const State& get_state() const;
    [[nodiscard]] const int get_depth() const;
    };
```

Functor

A functor egy C++ osztály, amely úgy viselkedik mint egy függvény. Ahhoz, hogy functort lehessen írni túl kell terhelni a () operatort. A heurisztikus érték h változója a get_cost_1, get_cost_2 összege lesz. Ez a kiadott kódban implementálva van.

Solver

Ez a class fogja megoldani a játékot. A változói:

- openlist: ez fogja tárolni azokat a Node*-okat, amelyeket be kell járnunk, ez automatikusan generálódik a már bejárt szomszédaiból. A solver az optimális node-ot fogja visszaadni, ami a minimális lépésű bejáráshoz kell. Amelyik node meg lett látogatva, azt el kell távolítani az openlistből.
- closedlist: ebben a listben azok a node-ok lesznek, amelyek már látogatva voltak. Ez a list segít abban, hogy ugyan azt a node ne járja be az algoritmus többször.
- Start és Goal State
- bool, hogy megoldotta-e a solver a feladatot
- edges, egy map, ahol a kulcs a csempék indexe, a hozzájuk tartozó érték pedig egy tömb, amelyben a szomszédos csempék indexei találhatóak. Ehhez a neighbours.gif ad egy segítséget.

A solver függvény működése:

- Ha nincs megoldás, -1-et ad vissza, ha van, akkor a minimális lépésszámot.
- Addig iterál, amíg el nem éri a végállapotot
  - Az openlistből az optimális node-ot kiválasztja, azt átrakja a closedlistbe. Az optimális az, melynek minimális a heurisztikus értéke.
  - A kiválasztott node-ból elérhető állapotokkal feltölti az open list-et.
- Miután elérte a végállapotot, onnan megállapítja a minimális lépésszámot, majd a solution vektort feltölti (végállapot->parent...->kezdő állapot)
- get_next_node() függvény: Az openlistben megkeresi az optimális node-ot, majd kitörli azt az és belerakja a closedlistbe
- expand_node(Node*) függvény: Az argumentumként kapott node-ból elérhető node-okat létrehozza és hozzáadja őket az openlisthez. Ehhez az üres csempe szomszédait használja fel.

Solver - API

```
class Solver{
private:
    [[nodiscard]] bool is_solved();
    //Segédfüggvény, hogy a closed_listben megtalálható-e az adott állapot
    bool is_in_closed_list(const State&);
    Node* get_next_node();
    void expand_node(Node*);
    void create_graph(); //Feltölti az edges mapet
    [[nodiscard]] bool is_solved();
public:
    Solver(const State&, const State&);
    int solve();
    };
```

API - egyben

```
class State {
    /*TODO*/
public:
    //Konstruktor és copy konstruktor
    State (int, const std::vector<int>&);
    State(const State&);
    //Assignment operator
    State& operator=(const State&);
    //==,!= operator overload
    bool operator==(const State&);
    bool operator!=(const State&);
    //Alapműveletek
    void swap_values(int, int); //Kicseréli az értékeket a megadott indexeken
    [[nodiscard]] const std::vector<int>&get_array() const;
    [[nodiscard]] int get_number_of_rows() const;
    };

class Node{
    /*TODO*/
public:
    Node(const State&, Node*, int);
    [[nodiscard]] const State& get_state() const;
    [[nodiscard]] const int get_depth() const;
    };

class Solver{
private:
    [[nodiscard]] bool is_solved();
    //Segédfüggvény, hogy a closed_listben megtalálható-e az adott állapot
    bool is_in_closed_list(const State&);
    Node* get_next_node();
    void expand_node(Node*);
    void create_graph(); //Feltölti az edges mapet
    [[nodiscard]] bool is_solved();
public:
    Solver(const State&, const State&);
    int solve();
    };
```