دانشگاه صنعتی امیرکبیر

(پلی تکنیک تهران)

دانشکده برق

حل پازل با الگوریتم های BFS و DLS

نگارش

سید علی قاضی عسگر

9623705

https://github.com/ali-ghazi78/AP-midterm-project

استاد درس

دکتر جهانشاهی

۱۱ دی ۹۹

Contents

Node :

ابتدا ما حالت های مختلف پازل را به صورت نود در می آوریم . هر نود ، دارای ۴ نود کناری میتواند باشد ، یکی نودی که در آن فضای خالی بالا میرود و بقیه هم به پایین ، چپ و راست . همچنین هر نود یک نود پدر نیز دارد . حالت هر نود و اعداد داخل هر حالت نیز درون یک وکتور ذخیره میشود . از پوینتر استفاده شده است که به صورت دینامیک مقدار دهی انجام شود ، البته لزومی نداشت و صرفا برای تمرین بیشتر با shared_ptr صورت گرفته .

```
class Node
{

public:
    static size_t Node_no;
    Node(const std::vector<int> & initial_state);
    ~Node();
    std::shared_ptr<std::vector<int>> val;
    Node* up;
    Node* down;
    Node* right;
    Node* left;
    Node* parent;
};
```

BFS :

این کلاس در واقع الگوریتم BFS را پیاده سازی میکند.

```
std::vector<int >desire_final_state;
BFS(std::vector<int> val);
BFS()=default;
~BFS();
std::queue<Node *> search_queue;
std::set<std::string> all_record;
std::vector<Node *> all_address_to_del;
std::vector<std::vector<int>> final_val;
Node *head;
Node *current_node;
bool randome_or_costume ;
void make_adjacent_nodes(const std::vector<int> &current_node);
void move_zero(std::vector<int> &vec1, int x, int y, int loc);
bool check_if_is_answer(const std::vector<int> &v);
bool search_for_answer(Node*  cu_node);
bool is_solvable(const std::vector<int> &v);
void disp_in_menu(const std::vector<int> &v,const std::vector<int> &v2);
std::string make_str(const std::vector<int> &v);
size_t show_path(BFS::Node *n);
int loop();
```

search_que

از انجا که الگوریتم BFS نیاز مند یک que میباشد ، تا حالت ها را بررسی کند . یک صف به نام search_que درست شده است، و در مواقع مورد نیاز به صف اضافه یا از آن کم میشود .

Desire_final_state

درصورتی که کاربر بخواهد به حالت نهایی دلخواهی برسد ، حالت نهایی در این متغر ذخیره میشود .

all_address_to_delete

ما نود ها را به صورت داینامیک در حین برنامه اضافه میکنیم و برای اینکار از new استفاده میشود ، به همین دلیل نیاز داریم آدرس حافظه ها را در جایی نگه داریم تا در انتهای برنامه حافظه ها را به وسیله ی delete آزاد کنیم. از متغیر all_address_to_delete به همین منظور استفاده شده است .

Final_value

در صورتی که پازل حل شود ، تمام مراحل حالات مختلف به منظور حل پازل در این متغیر ذخیره میشوند .

Head,current Node

این اشاره گر ها یکی برای اشاره به نود اولیه و دیگری برای سهولت در جابجایی بین نود ها انتخاب شده است.

Make_adjacent nodes

وقتی ما یک حالت پازل داریم ، میتوانیم ۴ حالت مختلف برای پازل های بعدی متصور باشیم ، این پازل ، حالت ها ممکن بعدی را پیدا میکند و به نود فعلی میگوید که نود بالا و پایین و مثلا چپ و راست تولید شده است.

همانطور که مشاهده میفرمایید ابتدا ما بدنبال مکان صفر،

در پازل فعلی میگردیم و این مکان را پیدا میکنیم سپس بررسی میکنیم که آیا امکان حرکت به جهت های مختلف را دارد یا خیر و در صورتی که نداشت ، متغیر های x_down را به عنوان مثال ، برابر منفی یک قرار میدهیم . حال که حالت های قابل پیاده سازی بررسی شد ، برای حالت های ممکن یک نود جدید درست میکنیم و حافظه به آن اختصاص میدهیم توجه باید شود که وقتی داریم نود جدید اختصاص میدهیم ، حواسمان هست که این حالت قبلا تکرار نشده باشد به همین دلیل با استفاده از تابع count ، بررسی میکنیم که ایا این حالت قبلا تکرار شده است یا خیر از انجا که all_record از نوع set میباشد ، سرعت جستجو در آن بالااست و مشکلی برای برنامه پیش نمی آید . پس از اینکه مطمین شدیم این نود قبلا ساخته نشده است ، حافظه اختصاص میدهیم و روابط این نود با نود پدر خود را برقرار میکنیم و ادرس حافظه ای که اختصاص داده شده است را نیز ذخیره میکنیم تا در انتها بتوانیم فضای استفاده شده را حذف کنیم .

cpp > bfs.cpp > BFS::make_adjacent_nodes(const std::vector<int> &)

```cpp
}
void BFS::make_adjacent_nodes(const std::vector<int> &current_node)
{
    auto p = std::find(current_node.begin(), current_node.end(), 0);
    int loc = p - current_node.begin();
    int x = p - current_node.begin();
    int y = x / 3;
    x = x % 3;

    int y_down = (y + 1 <= 2) ? (y + 1) : (-1);
    int x_down = x;
    int x_right = (x + 1 <= 2) ? (x + 1) : (-1);
    int y_right = y;
    int y_up = (y - 1 >= 0) ? (y - 1) : (-1);
    int x_up = x;
    int x_left = (x - 1 >= 0) ? (x - 1) : (-1);
    int y_left = y;

    if (y_down != -1)
    {
        std::vector<int> down = current_node;
        move_zero(down, x_down, y_down, loc);
        if (all_record.count(make_str(down)) == 0)
        {
            this->current_node->down = new Node(move(down));
            all_address_to_del.push_back(this->current_node->down);
            if (this->current_node->down != nullptr)
                this->current_node->down->parent = this->current_node;
        }
    }
    if (y_up != -1)
    {
        std::vector<int> up = current_node;
        move_zero(up, x_up, y_up, loc);
        if (all_record.count(make_str(up)) == 0)
        {
            this->current_node->up = new Node(move(up));
            all_address_to_del.push_back(this->current_node->up);
            if (this->current_node->up != nullptr)
                this->current_node->up->parent = this->current_node;
        }
    }
    if (x_right != -1)
    {
        std::vector<int> right = current_node;
        move_zero(right, x_right, y_right, loc);
        if (all_record.count(make_str(right)) == 0)
        {
            this->current_node->right = new Node(move(right));
            all_address_to_del.push_back(this->current_node->right);
            if (this->current_node->right != nullptr)
                this->current_node->right->parent = this->current_node;
        }
    }
    if (x_left != -1)
    {
        std::vector<int> left = current_node;
        move_zero(left, x_left, y_left, loc);
        if (all_record.count(make_str(left)) == 0)
        {
            this->current_node->left = new Node(move(left));
            all_address_to_del.push_back(this->current_node->left);
            if (this->current_node->left != nullptr)
                this->current_node->left->parent = this->current_node;
        }
    }
}
```

check_if_is_answer

این تابع بررسی میکند که آیا نودی که ارسال کردیم جواب ما هست یا خیر ، جواب ما میتواند همان ترتیب معمولی باشد یا توسط کاربر انتخاب شده باشد.

```
127  bool BFS::check_if_is_answer(const std::vector<int> &v)
128  {
129
130      if (randome_or_costume) //true means random
131          return std::is_sorted(v.begin(), v.end() - 1) && v[v.size() - 1] == 0;
132      else
133      {
134          for (int i = 0; i < 9; i++)
135          {
136              if (desire_final_state[i] != v[i])
137                  return 0;
138          }
139          return 1;
140      }
141  }
```

is_solvable

بنابر اینورژن های مختلف پازل ، این پازل به دو گراف مستقل تقسیم میشود و از یک پازل نمیتوان به دیگری رسید ، با این تابع این امکان را بررسی میکنیم که ایا از حالت فعلی میتوانیم به حالت دلخواه برسیم یا خیر.

```
bool BFS::is_solvable(const std::vector<int> &v)
{
    size_t inver = 0;
    for (int i = 0; i < v.size(); i++)
    {
        for (size_t j = i + 1; j < v.size(); j++)
        {
            if (v.at(i) > v.at(j) && v[i] && v[j])
            {
                inver++;
            }
        }
    }
    if (int(sqrt(v.size())) % 2 == 1)
        return (inver % 2 + 1) % 2;
    return inver % 2;
}
```

search_for_answer

این تابع در واقع الگوریتم اصلی برنامه را اجرا میکند ، به این صورت که ابتدا برای نود فعلی نود های همسایه را تولید میکنیم ، سپس نود های موجود همسایه را به صف خود اضافه میکنیم ، در صورتی که جواب باشند هم از حلقه خارج میشویم ، در صورتی هم که جواب نباشند در all_record ذخیره میکنیم تا بدانیم قبلا این نود را گشته ایم و جواب ما نبوده است. در انتهای کار هم یکی از صف کم میکنیم ، چون نود فعلی را گشته ایم ، و اخرین نود از صف را برای لوپ دوباره آماده میکنیم .

```cpp
bool BFS::search_for_answer(Node *cu_node)
{

    current_node = cu_node;
    make_adjacent_nodes(*(cu_node->val));
    if (cu_node != nullptr)
    {
        if (check_if_is_answer(*cu_node->val))
        {
            current_node = cu_node;
            show_path(current_node);
            return 1;
        }
    }
    else
    {
        std::cerr << "current node = 0 " << std::endl;
        return 1;
    }

    if (cu_node->up != nullptr)
    {
        search_queue.push(cu_node->up);
        all_record.insert(this->make_str(*cu_node->up->val));
        if (check_if_is_answer(*cu_node->up->val))
        {
            current_node = current_node->up;
            show_path(current_node);
            return 1;
        }
    }

    if (cu_node->down != nullptr)
    {
        search_queue.push(cu_node->down);
        all_record.insert(this->make_str(*cu_node->down->val));

        if (check_if_is_answer(*cu_node->down->val))
        {
            current_node = current_node->down;

            show_path(current_node);
            return 1;
        }
    }
    if (cu_node->right != nullptr)
    {
        search_queue.push(cu_node->right);
        all_record.insert(this->make_str(*cu_node->right->val));
        if (check_if_is_answer(*cu_node->right->val))
        {
            current_node = current_node->right;

            show_path(current_node);

            return 1;
        }
    }

    if (cu_node->left != nullptr)
    {

        search_queue.push(cu_node->left);
        all_record.insert(this->make_str(*cu_node->left->val));
        if (check_if_is_answer(*cu_node->left->val))
        {
            current_node = cu_node->left;
            show_path(current_node);
            return 1;
        }
    }
    if (search_queue.size() >= 1)
        search_queue.pop();

    if (search_queue.size() >= 1)
        cu_node = search_queue.front();
    else
    {
        std::cerr << "no Node reamained" << BFS::Node::Node_no << std::endl;
        return 1;
    }
    current_node = cu_node;
```

Show_path

وقتی که جواب را پیدا کردیم باید بتوانیم به سمت نود پدر حرکت کنیم و یک مراحل مختلف رسیدن به جواب را بیابیم ، از این تابع به این منظور استفاده شده است . این تابع نود مقدار نود فعلی را ذخیره میکند و تا بالاترین نود پدرش ، بالا میرود و مقادیر پدرها را ذخیره میکند . سپس این وکتور را برعکس میکنیم ، تا مراحل تقدم و تاخر مناسب داشته باشند و در نهایت نمایش میدهیم .

```
size_t BFS::show_path(BFS::Node *n)
{
    size_t steps = 0;
    while (n != nullptr)
    {
        this->final_val.push_back(*n->val);
        steps++;
        n = n->parent;
    }
    std::reverse(final_val.begin(), final_val.end());
    std::cerr << "\n\033[32;4msteps : " << steps << "\033[0m" << std::endl;
    return steps - 1;
}
```

Loop :

در این قسمت هم توابع search_for_answer را صدا میزنیم دلیل اینکه از while استفاده شده است و تابع را به صورت بازگشتی ننوشتیم ، این است که stack overflow رخ میداد .

```
int BFS::loop()
{
    bool done = false;
    while (!done)
    {
        done = search_for_answer(current_node);
    }
    return final_val.size();
}
```

BFS destructor :

در این جا مقادیر از حافظه که گرفته شده بود ، ازاد میشود .

```
}
BFS::~BFS()
{
    for (int i = 0; i < all_address_to_del.size(); i++)
    {
        delete all_address_to_del[i];
        all_address_to_del[i] = nullptr;
    }
}
```

DLS :

قسمت بزرگی مانند BFS میباشد به همین دلیل فقط تفاوت ها ذکر میشود .

```
public:
    class Node
    {

    public:
        static size_t Node_no;
        Node(const std::vector<int> &initial_state);
        ~Node();
        std::shared_ptr<std::vector<int>> val;
        Node *up;
        Node *down;
        Node *right;
        Node *left;
        Node *parent;
        size_t number_of_parent;

        void set_parent_number();
    };
    std::vector<int> desire_final_state;
    void show_progress_bar();
    DLS(std::vector<int> val);
    DLS() = default;
    typedef unsigned long long size_t
    size_t max_depth;
    std::stack<Node *> search_queue;
    std::set<std::string> all_record;
    std::vector<int> all_grand_child;
    std::map<std::string,int> all_map;
    std::vector<Node *> all_address_to_del;
    std::vector<std::vector<int>> final_val;
    Node *head;
    Node *current_node;
    bool randome_or_costume;
    bool my_find(const std::string &my_str,bool edit=false);
    void make_adjacent_nodes(const std::vector<int> &current_node);
    void move_zero(std::vector<int> &vec1, int x, int y, int loc);
    bool check_if_is_answer(const std::vector<int> &v);
    int search_for_answer(Node *cu_node);
    bool is_solvable(const std::vector<int> &v, int &inver);
    void disp_in_menu(const std::vector<int> &v, const std::vector<int> &v2);
    std::string make_str(Node *n, const std::vector<int> &vec = std::vector<int>(9, -1));
    size_t show_path(DLS::Node *n);
    bool loop();
};
```

Max_depth

توسط این متغیر ماکزیمم عمق ذخیره میشود .

search_que

این متغیر بدلیل نو ع پیاده سازی dls از نوع stack میباشد و برای ذخیره کردن ترتیب نود ها برای بررسی مورد استفاده قرار میگیرد.

All_grand_chid

یک متغیر کمکی است برای ذخیره تعداد پدر های یک نود در واقع عمق را نشان میدهد .

All_record

تمام نود هایی که قبلا جستجو شده اند در این متغیر قرار میگیرند .

All_map

در واقع یک مپ یا دیکشنری است که ستون اول ، مقدار پازل و ستون دوم نشان دهنده ی عمق هر نود است.

```cpp
void DLS::make_adjacent_nodes(const std::vector<int> &current_node)
{
    this->current_node->up = nullptr;
    this->current_node->down = nullptr;
    this->current_node->right = nullptr;
    this->current_node->left = nullptr;

    auto p = std::find(current_node.begin(), current_node.end(), 0);
    int loc = p - current_node.begin();
    int x = p - current_node.begin();
    int y = x / 3;
    x = x % 3;

    int y_down = (y + 1 <= 2) ? (y + 1) : (-1);
    int x_down = x;
    int x_right = (x + 1 <= 2) ? (x + 1) : (-1);
    int y_right = y;
    int y_up = (y - 1 >= 0) ? (y - 1) : (-1);
    int x_up = x;
    int x_left = (x - 1 >= 0) ? (x - 1) : (-1);
    int y_left = y;
    bool resume = false;
    std::vector<int> up = current_node;
    std::vector<int> down = current_node;
    std::vector<int> right = current_node;
    std::vector<int> left = current_node;

    if (y_up != -1)
    {
        move_zero(up, x_up, y_up, loc);
    }
    if (y_down != -1)
    {
        move_zero(down, x_down, y_down, loc);
    }
    if (y_right != -1)
    {
        move_zero(right, x_right, y_right, loc);
    }
    if (y_left != -1)
    {
        move_zero(left, x_left, y_left, loc);
    }

    if (y_up != -1 && my_find(make_str(this->current_node, up)) == 0)
    {

        this->current_node->up = new Node(move(up));
        all_address_to_del.push_back(this->current_node->up);
        if (this->current_node->up != nullptr)
        {
            this->current_node->up->parent = this->current_node;
            this->current_node->up->set_parent_number();
        }
    }
    else if (y_down != -1 && my_find(make_str(this->current_node, down)) == 0)
    {
        this->current_node->down = new Node(move(down));
        all_address_to_del.push_back(this->current_node->down);
        if (this->current_node->down != nullptr)
        {
            this->current_node->down->parent = this->current_node;
            this->current_node->down->set_parent_number();
        }
    }
    else if (x_right != -1 && (my_find(make_str(this->current_node, right)) == 0))
    {
        this->current_node->right = new Node(move(right));
        all_address_to_del.push_back(this->current_node->right);
        if (this->current_node->right != nullptr)
        {
            this->current_node->right->parent = this->current_node;
            this->current_node->right->set_parent_number();
        }
    }
    else if (x_left != -1 && (my_find(make_str(this->current_node, left)) == 0))
    {

        this->current_node->left = new Node(move(left));
        all_address_to_del.push_back(this->current_node->left);
        if (this->current_node->left != nullptr)
        {
            this->current_node->left->parent = this->current_node;
```

Make_adjacent_node:

وقتی ما یک حالت پازل داریم ، میتوانیم ۴ حالت مختلف برای پازل های بعدی متصور باشیم ، این پازل ، حالت ها ممکن بعدی را پیدا میکند و به نود فعلی میگوید که نود بالا و پایین و مثلا چپ و راست تولید شده است.

همانطور که مشاهده میفرمایید ابتدا ما بدنبال مکان صفر،

در پازل فعلی میگردیم و این مکان را پیدا میکنیم سپس بررسی میکنیم که آیا امکان حرکت به جهت های مختلف را دارد یا خیر و در صورتی که نداشت ، متغیر های x_down را به عنوان مثال ، برابر منفی یک قرار میدهیم . حال که حالت های قابل پیاده سازی بررسی شد ، برای حالت های ممکن یک نود جدید درست میکنیم و حافظه به آن اختصاص میدهیم توجه باید شود که وقتی داریم نود جدید اختصاص میدهیم ، حواسمان هست که این حالت قبلا تکرار نشده باشد به همین دلیل با استفاده از تابع my_find ، بررسی میکنیم که ایا این حالت قبلا تکرار شده است یا خیر از انجا که all_record از نوع set میباشد ، سرعت جستجو در آن بالااست و مشکلی برای برنامه پیش نمی آید . پس از اینکه مطمین شدیم این نود قبلا ساخته نشده است ، حافظه اختصاص میدهیم و روابط این نود با نود پدر خود را برقرار میکنیم و ادرس حافظه ای که اختصاص داده شده است را نیز ذخیره میکنیم تا در انتها بتوانیم فضای استفاده شده را حذف کنیم .

تفاوتی که در اینجا با الگوریتم BFS دارد این است که در صورتی که یک همسایگی تولید شد، دیگر سایر همسایه ها را تولید نمیکنیم بدلیل ساختار این الگورتیم .

تفاوت دیگر این است که ممکن است ، یک نود قبلا سرچ شده باشد ولی در عمق بیشتر و عمق فعلی کمتر از عمق قبلی باشد ، در اینجا ما دوباره نود را بررسی میکنیم زیرا شانس رسیدن به حالتی که میخواهیم بیشتر است.

```cpp
s.cpp > DLS::search_for_answer(Node *)
: DLS::search_for_answer(Node *cu_node)

current_node = cu_node;
make_adjacent_nodes(*(cu_node->val));
bool checked = false;
if (cu_node->up != nullptr)
{
    if (my_find(make_str(cu_node->up), true) == 0)
    {
        search_queue.push(cu_node->up);
        all_grand_child.push_back(cu_node->number_of_parent + 1);
        all_record.insert(this->make_str(cu_node->up));
        all_map.insert(std::pair<std::string, int>(this->make_str(cu_node->up), all_grand_child.back()));
        checked = true;
    }
    if (check_if_is_answer(*cu_node->up->val))
    {
        current_node = current_node->up;
        show_path(current_node);
        return 1;
    }
}
else if (cu_node->down != nullptr)
{
    if (my_find(make_str(cu_node->down), true) == 0)
    {
        search_queue.push(cu_node->down);
        all_grand_child.push_back(cu_node->number_of_parent + 1);
        all_record.insert(this->make_str(cu_node->down));
        all_map.insert(std::pair<std::string, int>(this->make_str(cu_node->down), all_grand_child.back()));
        checked = true;
    }
    if (check_if_is_answer(*cu_node->down->val))
    {
        current_node = current_node->down;
        show_path(current_node);
        return 1;
    }
}
else if (cu_node->right != nullptr)
{
    if (my_find(make_str(cu_node->right), true) == 0)
    {
        search_queue.push(cu_node->right);
        all_grand_child.push_back(cu_node->number_of_parent + 1);
        all_record.insert(this->make_str(cu_node->right));
        all_map.insert(std::pair<std::string, int>(this->make_str(cu_node->right), all_grand_child.back()));
        checked = true;
    }
    if (check_if_is_answer(*cu_node->right->val))
    {
        current_node = current_node->right;
        show_path(current_node);
        return 1;
    }
}
else if (cu_node->left != nullptr)
{
    if (my_find(make_str(cu_node->left), true) == 0)
    {
        search_queue.push(cu_node->left);
        all_grand_child.push_back(cu_node->number_of_parent + 1);
        all_record.insert(this->make_str(cu_node->left));
        all_map.insert(std::pair<std::string, int>(this->make_str(cu_node->left), all_grand_child.back()));
        checked = true;
    }
    if (check_if_is_answer(*cu_node->left->val))
    {
        current_node = cu_node->left;
        show_path(current_node);
        return 1;
    }
}
if ((search_queue.size() >= 1 && checked == false) || search_queue.size() >= max_depth)
{
    search_queue.pop();
}
if (search_queue.size() >= 1)
    cu_node = search_queue.top();
else
{
    std::cerr << "a problem is occured maybe depth is not enough" << DLS::Node::Node_no << std::endl;
    return 2;
}
```

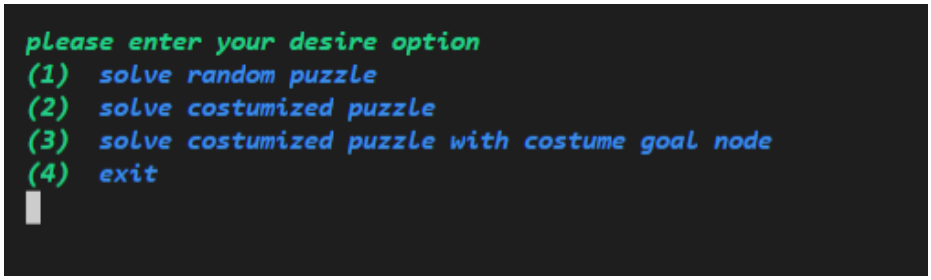
Search_for_answer :

در اینجا توسط تابعی که در بالا ذکر شد ابتدا همسایگی های مورد نظر ایجاد میشوند. سپس بررسی میکنیم که همسایگی مورد نظر ایا جواب ما هست یا خیر ، و در صورت بررسی این نود را به stack خود اضافه میکنیم همچنین در all_record ثبت میکنیم سپس در نهایت وقتی بررسی به اتمام رسید در صورتی که مقدار stack ما از عمقی که برای آن تعیین کردیم بیشتر شده باشد ، اخرین نود stack را حذف و نود بعدی را به عنوان نود فعلی برمیگزینیم ، شیوه تعیین عمق به این صورت است که تعداد نود های پدر هر نود در یه متغیر نود ذخیره میشود .

App :

Main menu :

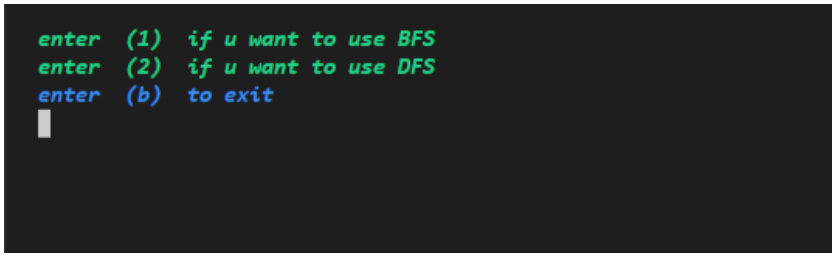
همانطور که در شکل مشاهده میکنید ، میتوان پازل به صورت رندوم انتخاب کرد یا به صورت دلخواه یا حالت نهایی و اولیه به صورت دلخواه که با وارد کردن شماره مناسب ، وارد برنامه میشویم .



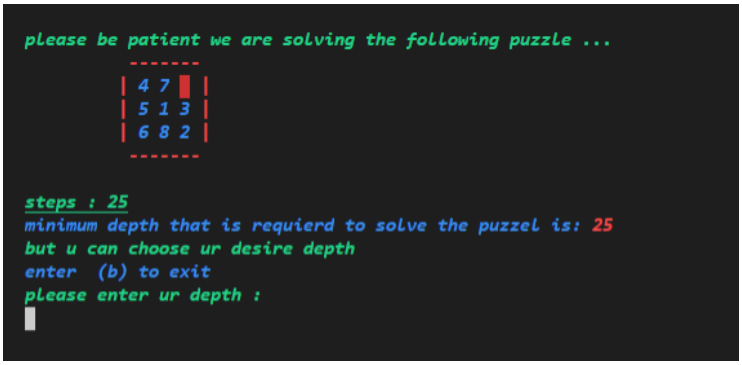
Random puzzle :

ابتدا از ما پرسیده میشود که با چند حرکت پازل به هم بریزد .

سپس نوع الگوریتم از ما سوال میشود

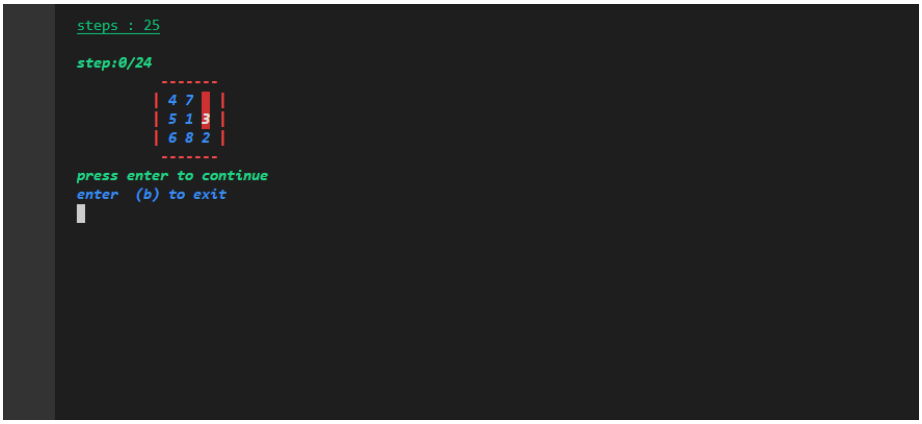


برای مثال با انتخاب الگورتیم DFS وارد صفحه زیر میشویم

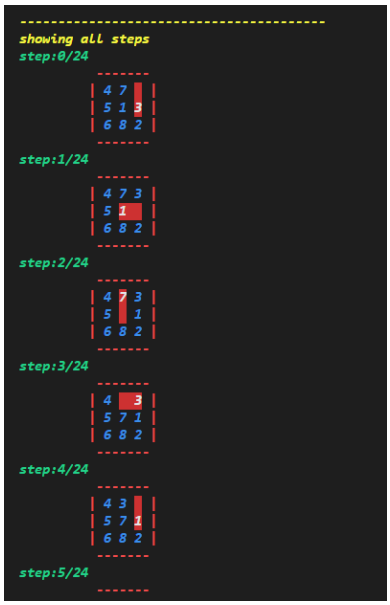


حال باید عمق را انتخاب کنیم ، برنامه میگوید که حداقل عمق مورد نیاز ۲۵ تا است این حداقل با bfs حساب شده است .

با انتخاب عمق پازل به صورت زیر نمایان میشود و با زدن اینتر ، حرکت تغییر میکند

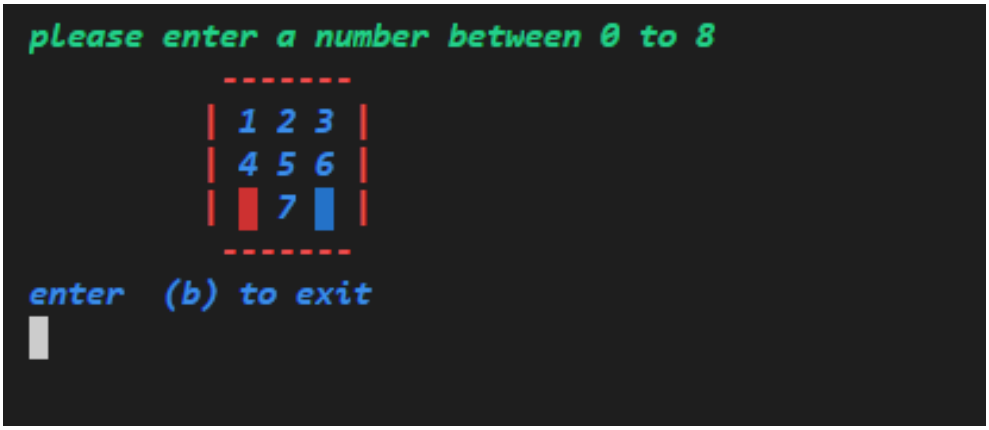


در نهایت نیز تمام مراحل به صورت پشت سر هم نمایش داده میشود .



Desire initial puzzle :

در این منو اعداد ۰ تا ۸ را وارد میکنیم در صورتی که پازل قابل حل باشد ، حل میکند در غیر اینصورت تقاضای دوباره وارد کردن میدهد ، خانه قرمز نشاندهنده ی جای خالی است .



Desire initial with desire final puzzle :

در این منو نیز دو پازل از کاربر گرفته میشود و در صورتی که از پازل دوم بتوان به پازل اولی رسید پازل را حل میکند.

بدلیل بهینه بودن و سرعت بیشتر از الگوریتم bfs برای حل این پازل استفاده شده است.