2024 機械情報夏学期
ソフトウェア第二

担当：岡田 慧 (k-okada-soft2@jsk.t.u-tokyo.ac.jp)

8. アルゴリズムとデータ構造２（知識と探索）

1　問題解決と木・グラフ

計算機による知的処理の一つとして問題解決 (Problem Solving) がある．ここでいう問題解決とは，与えられた状態 (Given State / 初期状態：Initial State) から目標 (State / 目標状態: Goal State) に到達する方法の解を見つけることである．

解を見つけるためには木 (Tree) やグラフ (Graph) を用いた探索 (Search) を行う．すなわち，木やグラフのノードで状態 (state) を表し，エッジで状態間の遷移を引き起こす行為 (Action) を表し，初期状態であるノードからゴール状態を示すノードまでをたどる行為列，すなわちエッジの系列を見つけ，その中で最良なものを選らぶことを解の探索 (search) と呼ぶ．

この時，状態に対して適用可能な行為とその時の遷移先の状態のペアを返す関数を用いながら行為列を見つけるが，このペアを返す関数を後続関数 (successor function) とよぶ．

また，初期状態からたどることが出来る全ての状態を状態空間 (state space) と呼ぶ．

さらに，ある状態から別の状態までの行為列にかかるコストを経路コスト (path cost) と呼び，各行為に対するコストをステップコスト (step cost) と呼ぶ．また探索は終了判定 (goal test) の結果により停止する．

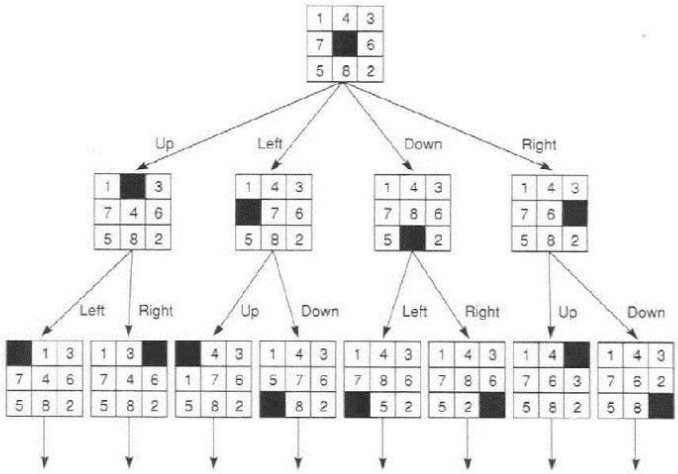
1.1　状態空間 (State Space)

状態空間における探索を用いた問題解決は非常に強力な手法であり，様々な面で利用されている．いくつか例を見ていこう．

8 パズル[1] は 8 つのタイルのそれぞれが 9 個の区画のどの位置にあるか決まるものであり，ここでは図に示すような初期状態からゴール状態までをたどる経路を見つる問題を考える．このときの状態は 8 個のタイルの位置で決まり，行為は空白を上下左右に動かすことに相当する．この問題における状態空間は 9!/2=181,440 になる[2]．15 パズル (4x4) では 1.3 兆状態あり,24 パズル (5x5) は 10^{25} 状態となる．

[1] 8 パズルは slideing-block puzzles と呼ばれる問題の一種で，この問題は NP 完全問題であることが知られているため探索アルゴリズムのテストによく使われている．

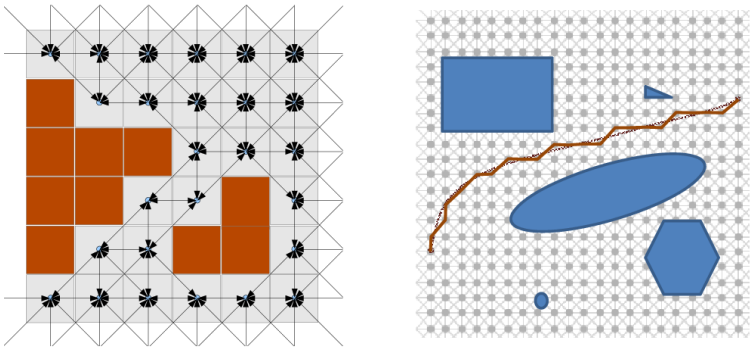
[2] 2 で割っているのはスライドパズルの偶奇性という性質で，9!で計算した状態の半分の状態は現在状態からたどり着くことの出来ない状態である．



8 パズルにおける状態空間の例
(https://cis.temple.edu/~pwang/3203-AI/Lecture/Search-1.htm)

またロボットの場合の例も見てみよう.

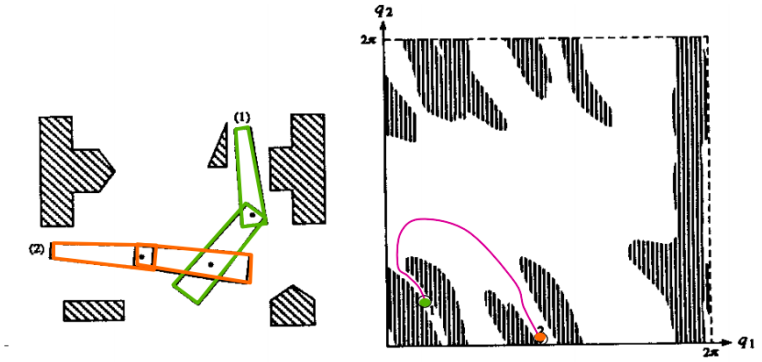
以下の図の様にロボットの移動の経路生成の問題は，地図上の位置 (x, y) を離散化し状態空間とし，現在位置から目標位置までの経路を見つける問題として定式化できる.



(a) A graph created by uniform square parti- (b) A trajectory in the continuous configuration

ロボットの移動経路生成における状態空間の例 (https://www.semanticscholar.org/paper/Topological-and-geometric-techniques-in-graph-robot-Bhattacharya/1509bb7430bddde891a19fe1a0019babbbe48adf)

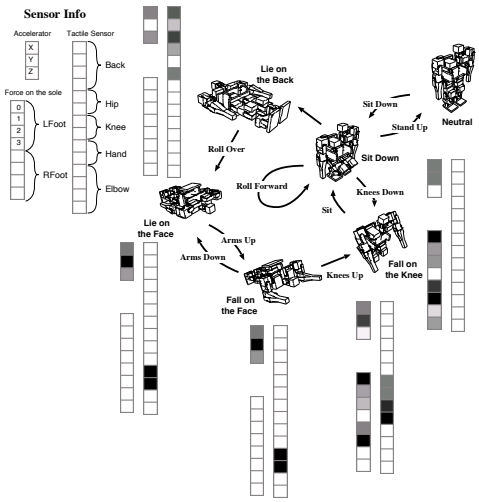
一方，以下の図のようにロボットのアームの経路生成の問題では，アームの各軸を (q_1, q_2) とし，これを離散化したものを状態空間として現在姿勢を示す (q_1^s, q_2^s) から (q_1^g, q_2^g) へと遷移する経路を見つける問題として定式化できる.



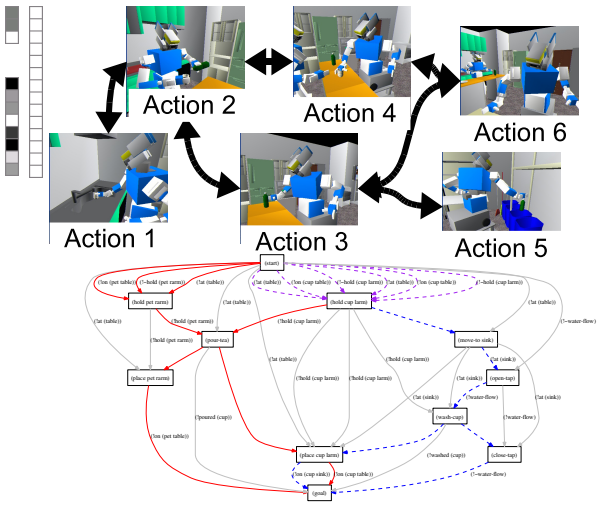
ロボットのアーム経路生成における状態空間の例
(http://gamma.cs.unc.edu/courses/planning-f13/SLIDES/Lecture3.pdf)

更に，より上位の行動も状態空間における探索問題に定式化できる．

例えば，以下の左図の様に，ロボットの姿勢と重力加速度センサの値をベクトル化したものを状態とすることで，例えば転んだロボットが起き上がる問題は，寝そべった姿勢から２脚で立つ姿勢までの経路を見つける問題として定式化できる．また，以下の右図の様に，家事作業を行うヒューマノイドロボットの場合，コップを洗う問題は，テーブルの前にいる，右手にコップを掴んでいる，シンクの前にいる，コップが洗われている，のような状態を構成し，テーブルの前に移動する，コップを掴む，コップを洗う，といった行為を定義することで，テーブルの前に入る状態から，コップが洗われている，という状態への経路を見つける問題として定式化できる．



ロボットの基本動作を記述したステートネット
(金広, et al. 2002)



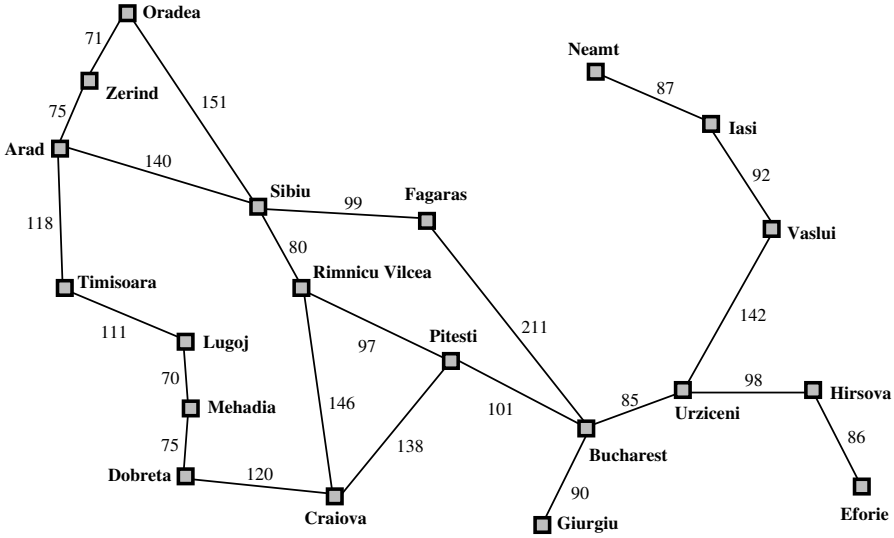
タスク記述の例 (Okada, et al. 2008)

1.2 探索木 (Search Tree) による解の探索

問題は探索木 (search tree) を利用して解くことが出来る．

1.2.1 ルーマニアを旅する問題

例えば以下のルーマニアの都市[3] を考える．ここでグラフのノードが都市であり，エッジはノード（都市）間の道を表している．



ルーマニアの都市を旅する問題

ここで問題 (Problem) を Arad から Bucharest までたどり着くための問題を考える．

まず初期状態 (initial state) は現在地を指し $In(Arad)$ と表現する．次に，ここで適用可能な行為 (action) 考える．ここでは $< action, successor >$ のペアを返す SUCCESSOR-FN(x) という後続関数 (successor function) を考える．$In(Arad)$ に対するこの関数の返り値は

$$\{< Go(Sibiu), In(Sibiu) >, < Go(Timisoara), In(Timisoara) >, < Go(Zerind), In(Zerind) >\}$$

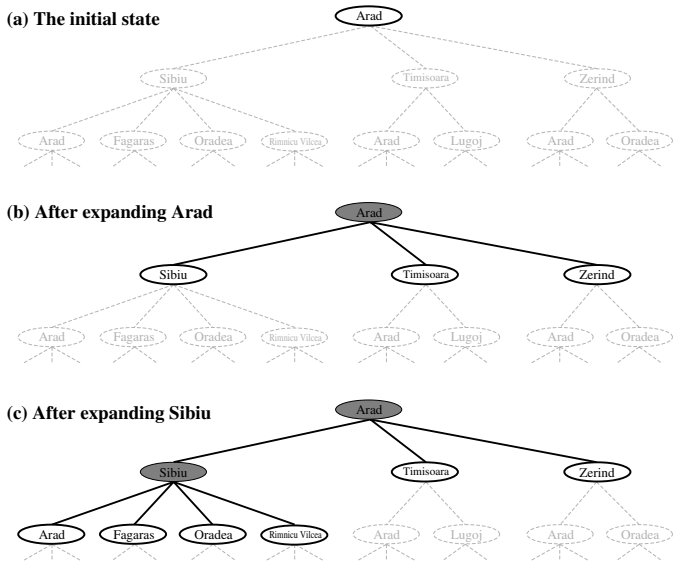
となる．また終了判定は与えられた状態がゴールかどうかを判定するものでありこの場合は，現在状態が{In(Bucharest)}になっているか否かで判定する．またこの問題のステップコストはそれぞれの状態間の距離と考える．

1.2.2 探索木 (Search Tree) による解の探索

定式化された問題は探索木 (search tree) を利用して解くことが出来る．探索木は初期状態と後続関数を用いて構築する．以下に示した図はルーマニア問題における探索木である．

ここで，ルーマニアの都市とその間の道を表すグラフと，ある都市からある都市までを移動するための探索木は異なるものである，という点に注意して欲しい．

[3]Arad :アラド，産業中心地でありルーマニア正教会の主教座がある．Bucharest： ブカレスト，文化，産業，金融の中心地．Craiova： クラヨーヴァ，ブカレスト西方にあり昔からの政治的中心地．Dobreta：ドロベタ，ドナウ川左岸，鉄門の下流に位置する都市．Eforie：エフォリエ，黒海沿いのリゾート地．Fagaras：ファガラッシュ，ルーマニア中央部に属する都市．Giurgiu：ジュルジュ，ドナウ川を挟んで対岸にはブルガリアの都市ルセがある．Hirsova：フルショバ，ドナウ川沿いの小さな港町．Iasi：ヤシ，東部にあるルーマニア第二の都市，かつてのモルダヴィア公国の首都．Lugoj：ルゴジ，かつての要塞都市．ティミッシュ川両岸にあり川が市を２つの区に分けている．Mehadia：メハディア，鉱物資源の産出値．Neamt：ネアムツ，ルーマニア・モルダヴィア地方の県．Oradea：オラデア，ハンガリーとの国境に位置し，温泉で有名．Pitesti： ピテシュティ，貿易工業中心地でピテシュティ大学とコンスタンティン・ブランコヴェアヌ大学が本拠を置く．RimnicuVilcea：ルムニク・ヴルチャ，中南部トランシルヴァニアアルプス山脈のふもとの都市．Sibiu：シビウ，１７世紀には西洋社会の東の端の街として捉えられていた．Timisoara： ティミショアラ．西部の工業都市，ヨーロッパで最初に電気による街路灯が導入された．Urziceni：ウルジチェニ，ワラキア平原の東部、バラガン平原のステップ地帯に位置に属している．Vaslui：ヴァスルイ，ルーマニア東部のヴァスルイ県の県都．Zerind：ゼリンド，アラド県の都市．



ルーマニア問題における探索木の構築の例

探索木の根 (root) は初期状態，すなわち In(Arad) を表す．まず，初期状態 Arad から開始し，それがゴール状態であるかテストする．そうでなければ，他の状態を考えるために，現在の状態に後続関数を適用し新しい状態の集合を生成する．これを状態の展開 (expand) と呼ぶ．ここでは，Sibiu, Timosoara, Zerind という 3 つの新しい状態が生成 (generate) される．次に，この 3 つから一つを選び，そこを訪れる (訪問:visit) し，処理はゴール状態のテストの段階に戻る．

生成はされているが展開されていないノード，あるいは子が存在しないノードを木の葉ノード (leaf node) とよび，この集合は fringe と呼ばれるリストに入っている．fringe は一般的にキュー (queue) で実装されることが多い．fringe から展開するノードを一つ選ぶ処理の違いによって異なる探索アルゴリズムを実現できる．

キューに対する操作としては

- MAKE-QUEUE(elements,...)：与えられた elements を使って queue を作る
- EMPTY?(queue)：queue が空か否かを判定する
- FIRST(queue)：queue の先頭の要素を返す
- REMOVE-FIRST(queue)：queue の先頭の要素を返し，その要素を queue から取り出す
- INSERT(element,queue)：element を queue に入れ，queue を返す
- INSERT-ALL(elements,queue)：elements を queue に入れ，queue を返す

が定義できる．これを使った探索アルゴリズムの一般解を以下に示す．

```
function TREE-SEARCH( problem, fringe) returns a solution, or failure
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST(problem, STATE(node)) then return node
        fringe ← INSERTALL(EXPAND(node, problem), fringe)
```

```
function EXPAND( node, problem) returns a set of nodes
    successors ← the empty set
    for each action, result in SUCCESSOR-FN(problem, STATE[node]) do
        s ← a new NODE
        PARENT-NODE[s] ← node;  ACTION[s] ← action;  STATE[s] ← result
        PATH-COST[s] ← PATH-COST[node] + STEP-COST(STATE[node], action, result)
        DEPTH[s] ← DEPTH[node] + 1
        add s to successors
    return successors
```

1.2.3 ループ対応について

探索木を展開していく中で同じノードを２回以上訪れて展開する場合がある．この対応として，探索中に訪れたノードをすべて覚えておいて，展開する際に，そのノードは既に訪れていたかをチェックすればよい．このとき訪れたノードを保持しているリストを open list と呼ぶことがある．また訪れていないノードの集合を保持しているリストは close list と呼ばれる．このようにして TREE-SEARCH アルゴリズムを発展させた GRAPH-SEARCH アルゴリズムを以下に示す．このアルゴリズムでは訪れたすべてのノードを保持するため，深さ優先探索等において空間計算量のメリットを相殺してしまう点に注意する必要がある．

```
function GRAPH-SEARCH( problem, fringe) returns a solution, or failure

    closed ← an empty set
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST(problem, STATE[node]) then return node
        if STATE[node] is not in closed then
            add STATE[node] to closed
            fringe ← INSERTALL(EXPAND(node, problem), fringe)
    end
```

1.3 Python による実装

まずは問題を定義しよう．`problem` は以下のようにグラフとして表現する．これにより，前出のグラフ探索（木の探索）プログラムを用いて探索することができる．また，ステップコスト（都市

間の距離）は二次元の Dictionary として表した[4].

```
problem = {
    'Arad': ['Zerind', 'Timisoara', 'Sibiu'],
    'Bucharest': ['Urziceni', 'Giurgiu', 'Pitesti', 'Fagras'],
    'Craiova': ['Dobreta', 'Pitesti', 'Rimnicu_Vilcea'],
    'Dobreta': ['Mehadia', 'Craiova'],
    'Eforie': ['Hirsova'],
    'Fagras': ['Sibiu', 'Bucharest'],
    'Giurgiu': ['Bucharest'],
    'Hirsova': ['Eforie', 'Urziceni'],
    'Iasi': ['Neamt', 'Vaslui'],
    'Lugoj': ['Mehadia', 'Timisoara'],
    'Mehadia': ['Lugoj', 'Dobreta'],
    'Neamt': ['Iasi'],
    'Oradea': ['Zerind', 'Sibiu'],
    'Pitesti': ['Rimnicu_Vilcea', 'Bucharest', 'Craiova'],
    'Rimnicu_Vilcea': ['Sibiu', 'Pitesti', 'Craiova'],
    'Sibiu': ['Rimnicu_Vilcea', 'Fagras', 'Oradea', 'Arad'],
    'Timisoara': ['Lugoj', 'Arad'],
    'Urziceni': ['Bucharest', 'Hirsova', 'Vaslui'],
    'Vaslui': ['Iasi', 'Urziceni'],
    'Zerind': ['Oradea', 'Arad'],
}

step_cost = {
    'Arad': {'Zerind': 75, 'Timisoara': 188, 'Sibiu': 140},
    'Bucharest': {'Urziceni': 85, 'Giurgiu': 90, 'Pitesti': 101, 'Fagras': 221},
    'Craiova': {'Dobreta': 120, 'Pitesti': 138, 'Rimnicu_Vilcea': 146},
    'Dobreta': {'Mehadia': 75, 'Craiova': 120},
    'Eforie': {'Hirsova': 86},
    'Fagras': {'Sibiu': 99, 'Bucharest': 211},
    'Giurgiu': {'Bucharest': 90},
    'Hirsova': {'Eforie': 86, 'Urziceni': 98},
    'Iasi': {'Neamt': 87, 'Vaslui': 92},
    'Lugoj': {'Mehadia': 70, 'Timisoara': 111},
    'Mehadia': {'Lugoj': 70, 'Dobreta': 75},
    'Neamt': {'Iasi': 87},
    'Oradea': {'Zerind': 71, 'Sibiu': 151},
    'Pitesti': {'Rimnicu_Vilcea': 97, 'Bucharest': 101, 'Craiova': 138},
    'Rimnicu_Vilcea': {'Sibiu': 80, 'Pitesti': 97, 'Craiova': 146},
    'Sibiu': {'Rimnicu_Vilcea': 80, 'Fagras': 99, 'Oradea': 140, 'Arad': 151},
    'Timisoara': {'Lugoj': 111, 'Arad': 118},
    'Urziceni': {'Bucharest': 85, 'Hirsova': 98, 'Vaslui': 142},
    'Vaslui': {'Iasi': 92, 'Urziceni': 142},
    'Zerind': {'Oradea': 71, 'Arad': 75},
}
```

記述した problem の確認のために視覚化してみよう．グラフを表現する記述言語として DOT 言語（https://ja.wikipedia.org/wiki/DOT%E8%A8%80%E8%AA%9E, https://graphviz.org/doc/info/lang.html）がある．

このように記述言語は，一つの抽象化の壁，として捉えられる．つまり，この記述を前提として，この記述書き出すプログラムや，また，この記述を視覚化するビューワプログラムが開発されている．例えば，Ubuntu 上で利用できるビューワの一つに xdot ががある．

[4] それぞれ，定義としては冗長であるが，プログラムのシンプルさを優先した．

そこで，以下のようなプログラムで problem から DOT 言語を出力し，xdot problem.dot として表示することができる．

```
def problem_to_dot(problem):
    with open("problem.dot", mode='w') as f:
        f.write("digraph problem {\n")
        f.write("  // node\n")
        for node in problem.keys():
            f.write("  {} [shape = box];\n".format(node))
        f.write("  // edge\n")
        for node1 in problem.keys():
            for node2 in problem[node1]:
                f.write("  {} -> {} [label = {}];\n".format(
                    node1, node2, step_cost[node1][node2]))
        f.write("}\n")
```

例えば，前述の幅優先プログラムを用いることで bfs(problem, 'Arad', 'Bucharest') としてグラフの探索が可能になる．しかしながら，この探索は本日扱う探索木に基づく問題解決ではない．

探索木を構成するためには，状態に対して適用可能な行為とその時の遷移先の状態のペアを返す関数である，後継関数 (successor function) を定義する．

ここでは，遷移先は都市であり，行為はその都市に移動する，となることから，行為については特に返すことなく，ある都市から，移動可能な遷移先の都市を以下のように返す関数として定義する．

```
def successor(problem, node):
    return problem[node]
```

探索木におけるノードは，都市ではなく，その都市までどの様にしてたどってきたか，という情報が必要になる．ここででは以下のように定義する．1.2.2 節の EXTEND 関数の中の処理に対応している．

```
class Node:
    def __init__(self, parent, state):
        self.parent = parent
        self.state = state
        if parent:
            self.path_cost = parent.path_cost + step_cost[parent.state][state]
            self.depth = parent.depth + 1
        else:
            self.path_cost = 0
            self.depth = 0
```

以上の準備に基づき探索木による問題解決のプログラムは以下のように書くことができる．

```
def remove_front(fringe):
    ret = fringe[0]
    del(fringe[0]) # remove first element
    return ret

def tree_search(problem, start, goal_test):
    fringe = [start]
    while fringe:
        node = remove_front(fringe)
        # https://stackoverflow.com/questions/52374104/typeerror-unsupported-format-string-passed-to-list-i
        print("traverse {0: <16}  cost {1: >5}, depth {2}".format(
            "{}".format(node.state), node.path_cost, node.depth))
        if goal_test(node) == True:
            print("found target {}".format(node.state))
            n = node
            result = []
            while n:
                result.append(n.state)
                n = n.parent
            result.reverse()
            print(result)
            return node
        for n in successor(problem, node.state):
            fringe.append(Node(node, n))
    return None
```

`node = remove_front(fringe)` の部分は，本来以下のように書くべきところであるが，次節以降で利用するプログラムとの関係からこの様に書く．`del` はリストから要素を削除する関数である．Python のリストが参照私であることを利用し破壊的操作を行う関数としている．

```
        node = fringe[0]
        fringe = fringe[1:]  # pop
```

定義した `tree_search` を利用するためには，初期ノードを `Node(None, 'Arad')` として定義し，またゴールの判定は `lambda x: x.state == 'Bucharest'))` として以下のように呼び出す．

```
print(tree_search(problem, Node(None, 'Arad'), lambda x: x.state == 'Bucharest'))
```

結果だけでなく，ノードを訪れる順番も幅優先探索と同じになる．

2　知識あり探索アルゴリズム

知識あり探索 (Informed Search) とはゴールでない状態（ノード）について，それが別のものよりも見込みがある（ゴールへ近い）ことを知ることが出来る場合の探索であり，発見的探索 (Heuristic Search) とも呼ばれる．ヒューリスティック (Heuristic) とは発見的，経験的という意味であり[5]，ヒューリステッィクス (Heuristics) とは発見的・経験的知識のことである．知識あり探索においては，ノードに対してゴールへの近さを得る関数が必要になるが，これは問題に依存し，固有の知識であり経験的に得る必要があるため，ヒューリスティック関数と呼ぶ．

[5]語源となる Heuristik はアルキメデスがお風呂に入っていてアルキメデスの原理を発見したときに裸のまま飛び出して「発見した（heureka:エウレカ）！」と叫んだという故事から由来した言葉

知識あり探索の一般的なアルゴリズムは最良優先探索 (best-first search)[6]と呼ばれ，TREE-SEARCH あるいは GRAPH-SEARCH アルゴリズムの中で次に展開するノードを，評価関数 ($f()$: evaluation function) に基づいて選択する方法である．一般には，評価関数は現在のノードとゴールのノードの距離を返すため，評価関数が最も低い値を返したノードを次に展開する．したがって，TREE-SEARCH あるいは GRAPH-SEARCH アルゴリズムにおける fringe キューについて，各ノードの $f()$ の値の小さい順に並べておけば最良優先探索を実装できる．

ノード n とゴールノード間の最短コストの推定値を計算する関数をヒューリステック関数 $h(n)$ とする．ゴールノードにおける関数の返り値は 0 になる．前回示したルーマニアの都市を旅する問題では，例えば各都市とゴール (Bucharest) の直線距離を使うことが出来る．

2.1 欲張り最良優先探索 (greedy best-first search)

欲張り最良優先探索 (greedy best-first search) [7]はゴールに最も近いノードを展開するものであり，以下のように各ノードの評価をヒューリスティック関数のみで行えばよい．

$$f(n) = h(n)$$

ルーマニア問題の例で考えると，ここでは以下に示すような h_{SLD} と呼ぶノードとゴールの距離を返すヒューリステック関数を考える．これは問題の定義には含まれていない知識である．

Arad	366	Bucharest	0	Craiova	160
Dobreta	242	Eforie	161	Fagaras	176
Giurgiu	77	Hirsova	151	Iasi	226
Lugoj	244	Mehadia	241	Neamt	234
Oradea	380	Pitesti	100	RimnicuVilcea	193
Sibiu	253	Timisoara	329	Urziceni	80
Vaslui	199	Zerind	374		

h_{SLD} の値（ゴールノードまでの直線距離）

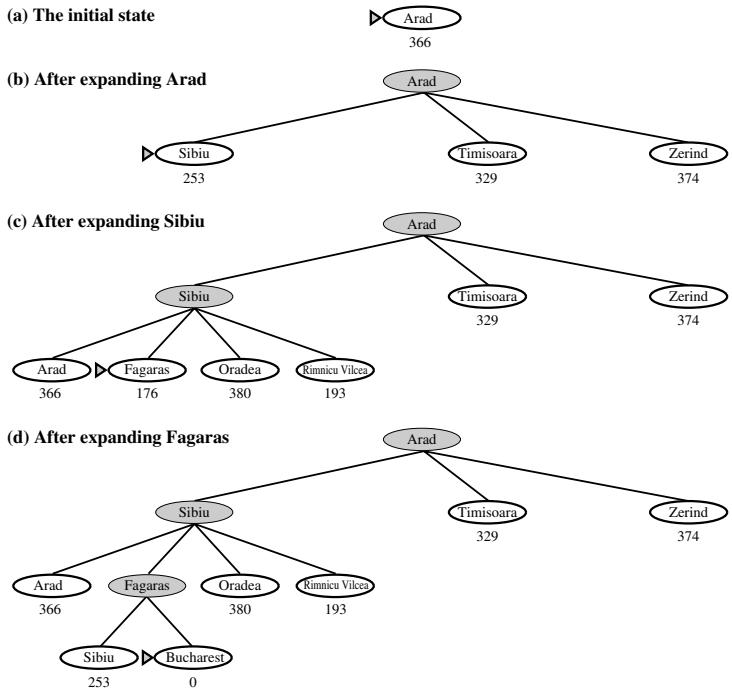
以下の図は欲張り最良優先探索 (greedy best-first search) の状況を示している．(a) の初期状態から 3 つの子ノードを展開した状態が (b) である．このとき，各ノードの評価関数を調べると Sibiu が最小となるためこれを次に訪問し (c) の状態になる．この次に展開される 4 つのノードで最小の評価関数を持つ Fagaras を訪問し，これを展開し Bucharest に到着する．

一方，別の状況として Iasi から Fagaras へと旅する問題を考える．このときヒューリスティック関数は先ず Neamt に行こうとするはずであり，そこが行き止まりであることから，次に Vaslui に戻ってくる．この場合は同じノードに 2 度以上訪問しないようなアルゴリズムにしないと Neamt と Iasi 間で無限ループに陥ってしまう．

欲張り最良優先探索の振る舞いは，ゴールまで一つの経路を深く深くたどっていき，行き止まりになった点で戻っていく点で深さ優先探索の振る舞いに類似している．このことからわかるように，欲張り最良優先探索もまた深さ優先探索と同様，解を見つける保障がない（完全性を満たさない），最小コストの経路からなる解を最初に見つけない（最適性がない）という欠点を持つ．一方，時間計算量は深さ優先探索と同様に $O(b^m)$ となり，空間計算量もまた同様に $O(bm)$ となる．

[6] 一般に完璧な評価関数を得ることはないので（得られれば探索アルゴリズムは必要ない），正確には最良優先ではなく，最良に見えるものを優先している．

[7] greedy search, best-first search 等とも呼ばれる



欲張り最良優先探索の様子．ノードの下に書いてある数値が評価関数の帰り値である．

2.1.1 Python による実装

ヒューリスティクスとしてゴール (Bucharest) までの距離を用いる．これは，以下のように定義する．

```
heuristic = {
    'Arad': 366,     'Bucharest': 0,  'Craiova': 160,
    'Dobreta': 242,  'Eforie': 161,   'Fagras': 176,
    'Giurgiu': 77,   'Hirsova': 161,  'Iasi': 226,
    'Lugoj': 244,    'Mehadia': 241,  'Neamt': 234,
    'Oradea': 380,   'Pitesti': 100,  'Rimnicu_Vilcea': 193,
    'Sibiu': 253,    'Timisoara': 329,'Urziceni': 80,
    'Vaslui': 199,    'Zerind': 374
}
```

`tree_search` で利用している `remove_front` において，以下のように `fringe` に入っているノードから，ゴールまで最も近いノードを展開し探索を進めることで最良優先探索を実現できる．

```
def remove_front(fringe):
    fringe.sort(key = lambda x: heuristic[x.state])
    # print(["{} {}".format(x.state, heuristic[x.state]) for x in fringe])
    ret = fringe[0]
    del(fringe[0]) # remove first element
    return ret
```

探索自体は `tree_search(problem, Node(None, 'Arad'), lambda x: x.state == 'Bucharest')` で実行できる．

2.2 A*探索 (A* search)

A*探索は初期状態から現在のノードまでのコストと，現在のノードからゴールノードまでのコストの和を評価関数とする最良優先探索であり，以下のように表すことが出来る．

$$f(n) = g(n) + h(n)$$

$g(n)$ は初期ノードからノード n までのステップコストの累積として計算することができ，$h(n)$ はノード n とゴールノード間の最短コストの推定値を計算する関数であることから $f(n)$ は n を通過する解（経路）の最短コストの推定値ということができる[8]．

A*では $h(n)$ が実際のコストより大きくならない許容的ヒューリステック (admissible heuristic) という条件を満たせば最適性，完全性を持つ．前節で示したヒューリスティック関数 h_{SLD} は，ノード (n) とゴールの直線距離からなり必ず実際の経路のコストより小さいため許容的である[9]．

以下の A*探索の様子では各ノードに評価値 $f(n) = g(n) + h(n)$ が記入してある．$g(n)$ の計算に必要なステップコストは前回の資料にあるルーマニア国内の経路を示すグラフの各エッジに記入してある．また，$h(n)$ は h_{SLD} を用いる．注目すべき点はゴールノードである Bucharest は (e) の時点で展開され fring キューに入っているにもかかわらず，ここでは訪問しない．Pitesti が先に訪問される理由は，Bucharest ノードの評価値は 450 であるが，fringe キューの中にはこれより小さい 417 のコストを持つ Pitesti が存在するためである．別の見方をすると，Pitesti を経由して Bucharest に行くコストは $417 + \alpha$ であり，Faguras 経由で Bucharest に到達するコスト 450 よりも小さい可能性があるから，Pitesti が先に訪問されると言える．

$h(n)$ が許容的であれば TREE-SEARCH が最適性 (最小コスト経路解を最初に見つける) を持つ理由は次のように証明できる．

もし最適でないゴールノード G_2 が fringe キューに現れ，最適解のコストが C^* としたとき，G_2 が最適でないゴールノードであり，$h(G_2) = 0$ であることから，

$$f(G_2) = g(G_2) + h(G_2) = g(G_2) > C^*$$

という関係が成り立つ．次に最適解の経路上にあるノード n[10]を考え，$h(n)$ が実際のコストよりも大きくないとすると，次の式が成り立つ．

$$f(n) = g(n) + h(n) \leq C^*$$

このことから，$f(n) \leq C^* < f(G_2)$ が成り立ち，G_2 が展開されることはなく，A*探索はかならず最適解を導出することができる．

しかしながら GRAPH-SEARCH アルゴリズムを用いる場合は，もし最適経路に属するノードが最初に生成されていない場合は，そのノードはループ対策のために fringe に入らず最適解を導出しなくなる．これへの対応としては２つある．一つは同じノードが２つの経路から見つかった場合に，コストの高いほうを取り除くという方法である．もう一つは最適経路に属するノードを必ず最初に展開するような $h(n)$ を考えることである．このような特性を単調性 (monotonicity, 整合性 consistency) と呼ぶ．ヒューリスティック関数 $h(n)$ は，n に対して行動 a により展開される子 n'

[8] $h(n)$ はノード n とゴールノード間の最短コストの推定値であった．

[9] $h(n)$ が実際のコストより大きくなる可能性もあるヒューリステックの場合は A 探索と呼ばれる．A*探索は求まる経路がスタートからゴールまでの最短経路であることが保証されている $h(n)$ が許容的ヒューリスティック (admissible heuristic) の場合を指している．

[10] 先程の例だと Pitesti ノードに相当する

に対してそれぞれのステップコスト $c(n, a, n')$ を考えたとき，以下の式が成り立てば単調であるという．

$$h(n) \leq c(n, a, n') + h(n')$$

また，ここから単調性を持つヒューリスティック関数は許容的であるのは自明であろう．

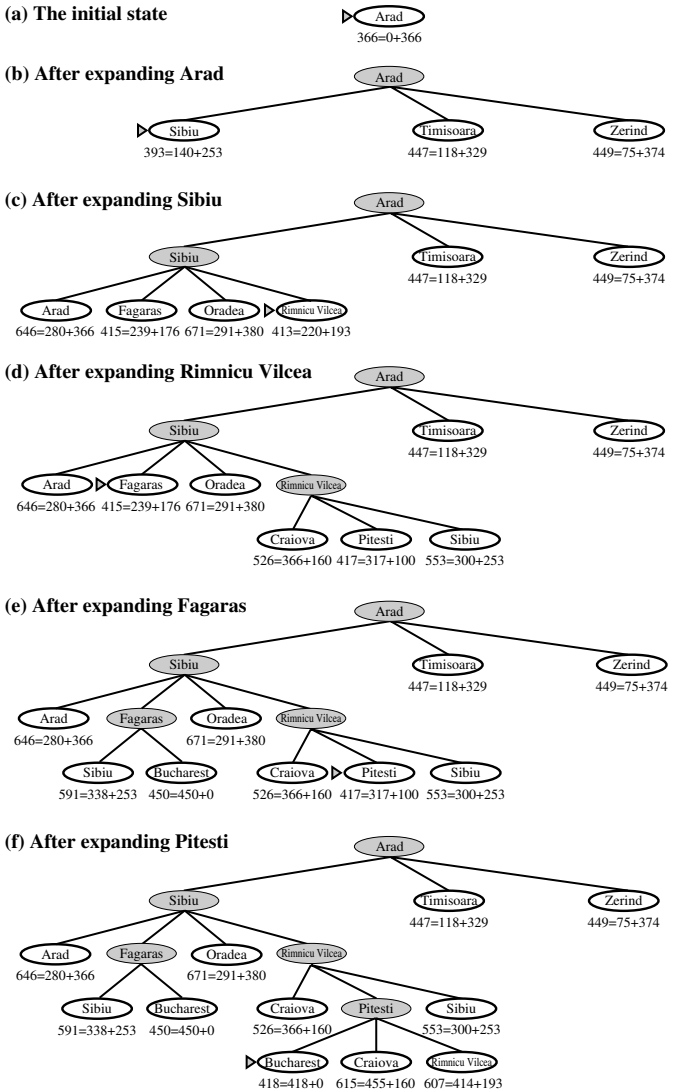
ここまでに示してきた h_{SLD} は，三角形の 2 辺の長さの和は残りの 1 辺の長さよりも大きいという三角不等式から単調性を有することがわかる．

また，単調性を持てば探索木に沿ったすべての経路で f のコストが非減少になる，という関係がある．これは，$g(n') = g(n) + c(n, a, n')$ の関係から，

$$f(n') = g(n') + h(n') = g(n) + c(n, a, n') + h(n') \geq g(n) + h(n) = f(n)$$

と計算できる．

A*探索では時間計算量，空間計算量ともにヒューリスティックの精度に依存する．一般には，すべてのノードをメモリに記録するので，時間切れになるまえにメモリ空間が不足する．



A*探索の様子．ノードの下に書いてある数値が評価値．

2.2.1 Python による実装

A*探索を行うためには，remove_front において，以下のように fringe に入っているノードから，初期状態から現在のノードまでのコストと，現在のノードからゴールノードまでのコストの予想値の和がもっと小さいノードを展開し探索を進めること実現できる．

```
def remove_front(fringe):
    fringe.sort(key = lambda x: x.path_cost + heuristic[x.state])
    # print(["{} {} + {}".format(x.state, x.path_cost, heuristic[x.state]) for x in fringe])
    ret = fringe[0]
    del(fringe[0]) # remove first element
    return ret
```

探索自体は tree_search(problem, Node(None, 'Arad'), lambda x: x.state == 'Bucharest') で実行できる．

3 より高度な話題

3.1 メモリ限定探索

A*探索では生成したすべてのノードを記憶するため空間計算量に難があった．これを解決するための方法として深さではなく，評価値 ($f = g + h$) を用いて制限を決める反復深化 A*(IDA*:Iterative-Deeping A*) や，次善の評価値 (f) を常に記憶しておいて，現在辿っている経路がこの値を超えたら，再帰的に経路を戻り，別の枝の探索を勧める再帰的最良優先探索 (RBFS:Recursive best-first search)，長さに制限のあるキューを使って最悪の評価値 (f) を持つノードは捨てていく SMA*(簡潔メモリ制限 A*) 等がある．

3.2 ヒューリステック関数

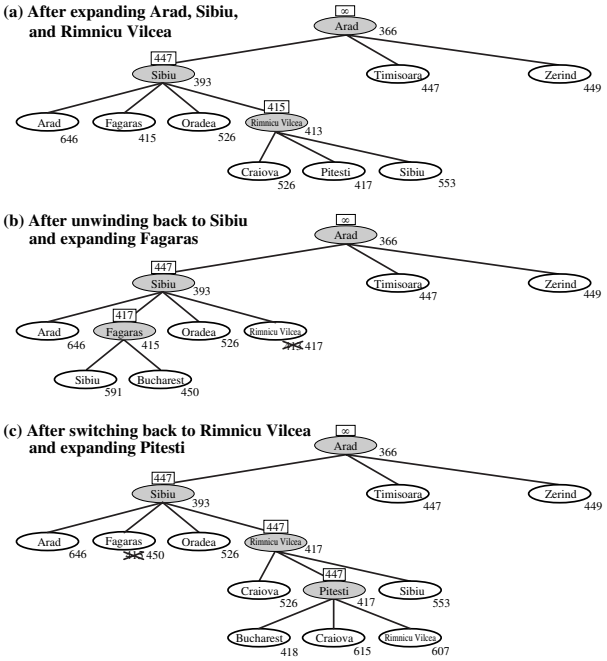
下に示した 8 パズルの例においてヒューリステック関数を考みると，(1) 間違った場所におかれているタイルの数 (h_1) と，(2) 各タイルのゴールまでのマンハッタン距離の合計からなる (h_2) が考えられる．図の例では h_1 は 8 であり，$h_2 = 3+1+2+2+2+3+3+2 = 18$ となる．それぞれを用いた探索における平均的な展開ノードの数を示した h_1 に h_2 の方が効率がよいことがわかる．ここで，よいヒューリスティックスを見つけるために制約緩和問題 (relaxed problem) という考え方を紹介する．relaxed problem は問題における行動 (action) に関して制限を緩めたものであり，relaxed problem に対する最適解のコストは，元の問題の許容的かつ単調性をもつヒューリステックスとなる．realxed problem を作るには問題を自然言語で書けばよい．8 パズルの場合は

A tile can move from square A to square B if
A is horizontally or vertically adjacent to B and B is blank

となり，この条件部分を分解すると，

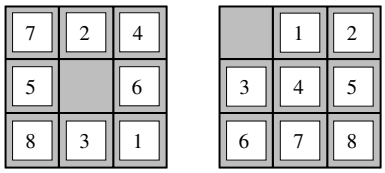
(a) A tile can move from square A to square B if A is adjacent to B.
(b) A tile can move from square A to square B if B is blank.
(c) A tile can move from square A to square B

となる．(a) からヒューリステックス h_2 が得られ，(c) から h_1 を得ることが出来る．



RBFS*探索の様子．ノードの上の値が評価値の制限．

	Search Cost		
d	IDS	A*(h_1)	A*(h_2)
2	10	6	6
4	112	13	12
6	680	20	18
8	6384	39	25
10	47127	93	39
12	364404	227	73
14	3473941	539	113
16	-	1301	211
18	-	3056	363
20	-	7276	676
22	-	18094	1219
24	-	39135	1641



8パズルにおける初期状態と目標状態の例．

4　おわりに

今回の講義は人工知能の教科書 Artificial Intelligence: A Modern Approach の第4章の導入部分を紹介した．本の大部分は `http://aima.cs.berkeley.edu/` で読めるので興味があれば見てほしいまた本資料の図版は特に注がない限り，ここのホームページで公開されているものである．

探索アルゴリズムは完全性，最適性，時間計算量，空間計算量の基準で判断できいる．代表的なアルゴリズムとして以下のものを紹介した．幅優先探索は，探索木が浅いノード順に展開し，完全かつ最適であるが，時間空間計算量が膨大であった．深さ優先探索は探査木の最も深いノードを最初に展開する．これは完全でも最適でもなく，時間計算量も大きいが，空間計算量は小さい．ちなみに，幅優先探索を横型探索，深さ優先探索を縦型探索と呼ぶ．深さ制限探索は深さ優先探索の深さ制限をつけたものであり，これを深さ制限を徐々に大きくとりながら繰り返し計算する反復深化探索は完全かつ最適で，かつ空間計算量も少なく，幅優先探索と深さ優先探索の長所を併せ持ったアルゴリズムである．

また，知識を用いることで効率的に探索問題を解くことができる．欲張り最良優先探索では最も低い評価関数を与えるノードを展開していくアルゴリズムであった．最適ではないが一般にはよい結果をあたることが多い．また，A*では各ノードにおいてこれまでの経路のコストと，ゴールまでのコストの推定値の和を評価関数としたものであり，木の探索では評価関数が実際のコスト以下であり許容的であれば，完全かつ最適である．グラフ探索では単調性が必要であることを示した．

今回紹介した探索アルゴリズムは完全性，最適性を満たすものだった．このようなアルゴリズムでは大規模な問題を解くことが出来ずに実世界の問題には対応できないという批判が長くあった．最近はかならずしも完全，最適ではないが，たいていの場合においてうまく働くような探索方法が研究されてきており，大規模問題も解けるようになっている．

ある問題があった場合に状態空間を構成することができ，効率的なヒューリステックスを見つけることが出来れば探索アルゴリズムを用いてその問題を解くことができる．どのような状態空間にするか，どのようなヒューリスティックスを選択するかについては試行錯誤を繰り返すしかないだろう．

宿題

提出先：ITC-LMS を用いて提出すること

提出内容：以下の問題の実行結果の画面をキャプチャしファイル名は「問題番号.png」とし，また講義中にでてきたキーワードについて知らなかったもの, 興味のあるものを調べ「学籍番号.txt」としててアップロードすること．テキストファイルはワードファイルなどだと確認出来ないことがあるため，emacs/vi 等のテキストエディタを使って書こう．プログラムが長くなりキャプチャ画面に入り切らなくなってきたらプログラムファイルと実行結果を「問題番号.txt」にまとめてアップロードしてよい．
画像で提出する場合は，各自のマシンの Mac アドレスが分かるようにすること．例えば画面中に ifconfig というコマンドを打ち込んだターミナルを表示すればよい．
ITC-LMS にアップロードする際には講義・宿題の感想を必ずコメントに記すこと．また授業中に質問した者はその旨を記すこと．質問は成績評価時の加点対象となる．

キーワード：木と探索木, ヒューリスティック関数，許容的ヒューリステック (admissible heuristic),

プログラムは `https://github.com/jsk-lecture/software2/tree/main/08` にあるので，打ち込むのが大変な者，バグがとり切れない者は参照すること．

1. ルーマニアを旅する問題について，深さ優先探索を行うと何が起こるか，プログラムを実行し確認せよ．

2. ルーマニアを旅する問題について，最良優先探索と A*探索のプログラムを実装し，結果を比較してみよ．

3. 発展問題：8 パズルの問題を解くプログラムを実装せよ．
   ヒント 1：まず，一般的な問題解決プログラムを記述してみる．`problem` として問題を扱うクラスを作成し，問題別に `successor`，`step_cost`，`heuristic` を作成すれば良いようにする．`Node`，`tree_search` もこれに合わせて変更してある．

```
class Problem:
    def __init__(self, start, goal):
        self.start = start
        self.goal = goal
        return None

    def remove_front(self, fringe):
        fringe.sort(key = lambda x: x.path_cost + self.heuristic(x.state))
        ret = fringe[0]
        del(fringe[0]) # remove first element
        return ret

    def goal_test(self, state):
        return state == self.goal
```

```
    def string(self, state):
        return state

class Romania(Problem):
    def successor(self, state):
        return problem[state]

    def step_cost(self, parent_state, state):
        return step_cost[parent_state][state]

    def heuristic(self, state):
        return heuristic[state]

class Node:
    def __init__(self, problem, parent, state):
        self.parent = parent
        self.state = state
        if parent:
            self.path_cost = parent.path_cost + problem.step_cost(parent.state, state)
            self.depth = parent.depth + 1
        else:
            self.path_cost = 0
            self.depth = 0

def tree_search(problem):
    fringe = [Node(problem, None, problem.start)]
    while fringe:
        node = problem.remove_front(fringe) # pop
        # https://stackoverflow.com/questions/52374104/typeerror-unsupported-format-string-passed-to-li
        print("traverse {0: <16}  g {1: >5} + h {2: >5}, depth {3}".format(
            "{}".format(node.state), node.path_cost, problem.heuristic(node.state), node.depth))
        if problem.goal_test(node.state) == True:
            print("found target {}".format(node.state))
            n = node
            result = []
            while n:
                result.append(n.state)
                n = n.parent
            result.reverse()
            for r in result:
                print(problem.string(r))
            return node
        for n in problem.successor(node.state):
            fringe.append(Node(problem, node, n))
    return None
```

これにより，以下のように初期状態と目標状態を指定して問題のインスタンスを作成し，
tree_search を呼び出す．

```
romania = Romania('Arad', 'Bucharest')
tree_search(romania)
```

ヒント２：８パズルの問題は，例えば以下のように定義することができる．

```
import copy
class Puzzle(Problem):
    def find_index(self, state, number):
```

```
        for i in range(0, 3):
            for j in range(0, 3):
                if state[i][j] == number:
                    x = j
                    y = i
        return (x, y)

    def successor(self, state):
        # find 0 (blank)
        (x, y) = self.find_index(state, 0)
        # get sucessor state
        ret = []
        if x > 0:
            r = copy.deepcopy(state)
            r[y][x-1], r[y][x] = 0, r[y][x-1]
            ret.append(r)
        if x < 2:
            r = copy.deepcopy(state)
            r[y][x], r[y][x+1] = r[y][x+1], 0
            ret.append(r)
        if y > 0:
            r = copy.deepcopy(state)
            r[y-1][x], r[y][x] = 0, r[y-1][x]
            ret.append(r)
        if y < 2:
            r = copy.deepcopy(state)
            r[y][x], r[y+1][x] = r[y+1][x], 0
            ret.append(r)

        return ret

    def step_cost(self, parent_state, state):
        return 1

    def heuristic(self, state):
        return 1

    def string(self, state):
        return '{}\n{}\n{}\n'.format(state[0], state[1], state[2])
```

以下のようにすると，幅優先探索で問題を解くことができる．答えがでるまで時間がかかるが，この問題の深さは 8 である．より効率の良いヒューリスティック関数に変更してほしい．

```
puzzle = Puzzle([[1,2,5],[4,0,8],[3,6,7]], [[0,1,2],[3,4,5],[6,7,8]]) # depth 8
tree_search(puzzle)
```