CMPUT 396 – Sliding Tile Puzzle

Sliding Tile Puzzle

2x2 Sliding Tile States

```
observe: on 2x2, a slide is a rotation

goal state  a b
            c .

solvable

  a b   a .   . a   c a   c a   c .
  c .   c b   c b   . b   b .   b a

  . c   b c   b c   b .   . b   a b
  b a   . a   a .   a c   a c   . c

not solvable

  a c   a .   . a   b a   b a   b .
  b .   b c   b c   . c   c .   c a

  . b   c b   c b   c .   . c   a c
  c a   . a   a .   a b   a b   . b
```

Exactly half of the states are solvable, the other half are not. In the case of 2x2 puzzles, I can solve it if I start with a configuration where there is a cycle. If not, I can't.

Solvable?

We call a state **solvable** if it can be transformed into the row-by-row sorted state (with blank last). For a puzzle with at least two each rows and columns for any fixed final state, exactly half of the states can be transformed into the final state.

A **parity check** tells us whether an arbitrary state is solvable.

column number	solvability condition
odd	even number inversions
even	blank's row-from-bottom parity != inversions parity

1) Look at a puzzle as a line of numbers (row 1, row 2, row 3, … row n)
2) For each number in the line, count how many inversions there are (how many numbers before it are bigger than it)
3) If the grid width is odd, then the number of inversions in a solvable situation is even.

4) If the grid width is even, and the blank is on an even row counting from the bottom (second-last, fourth-last etc), then the number of inversions in a solvable situation is odd.
5) If the grid width is even, and the blank is on an odd row counting from the bottom (last, third-last, fifth-last etc) then the number of inversions in a solvable situation is even.

```
5 4 3   odd number cols, 4+3+2+1=10 inversions, solvable
2 1 0

7 6 5 0  even number cols, 6+5+4+3+2+1=21 inversions,
4 3 2 1    blank in row 2 from bottom, solvable

7 6 5 4  even number cols, 6+5+4+3+2+1=21 inversions,
3 2 1 0    blank in row 1 from bottom, unsolvable
```

Search sliding tile space

```
children ordered by blank-mv: U D L R

           235
           41*
         U     L
        /       \
       /         \
      /           \
  23*               235
  415               4*1
   L               U   L
   |              /     \
   |             /       \
  2*3          2*5       235
  415          431       *41
  D L          L R        U
  / \          / \        |
 /   \        /   \       |
213  *23    *25   25*    *35
4*5  415    431   431    241
...  ...    ...   ...    ...
```

Exhaustive Search

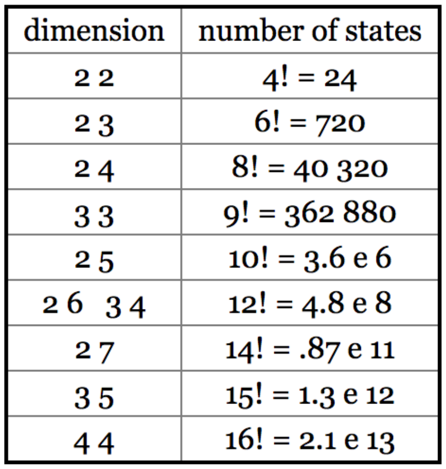
Random walk is so much slower than BFS and DFS that we will ignore it for this problem. Both BFS and DFS are exhaustive so they will solve the problem, however it may take too long.

Estimate state space: for any (r,c) problem, there are (r x c)! states.

State space adjacency graph has 2 components:

- Solvable states, (rc)!/2 nodes
- Unsolvable states, (rc)!/2 nodes

So starting from a fixed state, in the worst case, we examine (rc)!/2 nodes.



dimension	number of states
2 2	4! = 24
2 3	6! = 720
2 4	8! = 40 320
3 3	9! = 362 880
2 5	10! = 3.6 e 6
2 6 3 4	12! = 4.8 e 8
2 7	14! = .87 e 11
3 5	15! = 1.3 e 12
4 4	16! = 2.1 e 13

Solving Slide Tile with BFS

In maze traversal, we considered adjacency graphs of cells and used BFS to traverse this graph. The associated graph in a sliding tile puzzle is as follows:

- Each node in a graph is a sliding tile state
- Two nodes are adjacent if there is a single-slide between the states
- With this graph, we use BFS as before

To implement this in Python we can use a dictionary of the parents so each time we see a new state, we add it to the dictionary so that we know we have seen a state iff it is in the dictionary.

Runtime Analysis

For a 3x3 search space: $\frac{9!}{2} = 181440$

We divide by 2 here because half of the total states are solvable, half are not.

How can we use this to estimate a 2x5 runtime?

Let's say that we have 181440 iterations (of our algorithm's graph) in 2.2 sec at 82600 iterations/sec.

For BFS with a graph of N nodes and E edges, the runtime is O(N+E).

Average degree (number of neighbours) in a 3x3 search space?

- 4 corners with 2 neighbours
- 4 sides with 3 neighbours

- 1 middle with 4 neighbours

Average degree: $2 * \frac{4}{9} + 3 * \frac{4}{9} + 4 * \frac{1}{9} = \frac{24}{9} = 2\frac{2}{3} = 2.67$
So the number of edges in a 3x3 tile search space equals $9! * 2.67$.

Average degree for 2x5 search space?
- 6 non-corners with 3 neighbours
- 4 corners with 2 neighbours

Average degree: $3 * \frac{6}{10} + 2 * \frac{4}{10} = \frac{26}{10} = 2.6$

So we expect the worst case (no solution runtime for a 2x5 tile search) to take about $\frac{10*2.6}{2.67} = 9.75\ times\ as\ long.$

Therefore, 1814400 iterations in 21.5 seconds at 84400 iterations/sec (which is close to what was expected).

How about to solve a 4x4 puzzle?
To get a lower bound, we compare the sizes of the search spaces. A 4x4 search space is $16 * 15 * 14 * 13 * 12 * 11 = R\ times\ the\ size\ of\ 2x5\ search\ space.$ So we expect a 4x4 no-solution runtime at least $R * 21.5\ seconds = about\ 2.3\ years$

BFS takes too long to solve a 4x4 puzzle so we need a faster algorithm.

Why use BFS? To get the shortest solution.
DFS will ignore many moves at each stage.

How can we solve a 4x4 puzzle in a reasonable amount of time? Is there a way to tell which moves are more promising than other moves?

Knowledge: information about the particular problem, could be proved of heuristic.

Heuristic: suggestion, idea, not making provable claim about it
e.g. in early 2000s, computer Go programs were full of heuristics, but almost nothing was proved.

Special Purpose Algorithms: these are good for solving your problem, but you can't use it anywhere else. A general algorithm helps you solve more problems.

Special purpose algorithms do exist for the sliding tile puzzle. One such algorithm:

1. In sorted order (left to right, row by row) move next element into position while avoiding elements already placed.
2. Last 2 elements of each row need special technique
3. Last 2 rows need special technique
4. Final 2x2 grid (last 2 rows, last 2 columns) rotate into solution if and only if original state is solvable

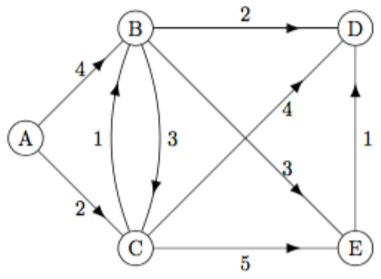
Heuristic search is a *guided* search. A **heuristic function** is used to decide which node of the search tree to explore next.

Dijkstra's Algorithm (single-source shortest path on weighted graphs): given a starting point, this algorithm will find all shortest paths from that starting point. At each node, we know the shortest path to get to that node so far.

Input: graph/digraph with non-negative edge/arc weights and a start node S

Output: for each node v, the shortest path from S to V

In a **weighted graph** each edge has a weight.



Algorithm:

Let the node we start at be S, the distance of node Y be the distance from S to Y.

1. Mark all nodes as unvisited. Create a set of all the unvisited nodes called the *unvisited set*.
2. Assign to every node a tentative distance value: zero for S and infinity for all other nodes. Set S as current.
3. For the current node, C, consider all of its unvisited neighbors and calculate their tentative distances through C. Compare the newly calculated tentative distance to the current assigned value and assign the smaller one.
    a. For example, if the current node A is marked with a distance of 6, and the edge connecting it with a neighbor B has length 2, then the distance to B through A will be 6 + 2 = 8. If B was previously marked with a distance greater than 8 then change it to 8. Otherwise, keep the current value.

4. When we are done considering all of the unvisited neighbors of C, mark the C as visited and remove it from the unvisited set. A visited node will never be checked again.
5. If the destination node has been marked visited (when planning a route between two specific nodes) or if the smallest tentative distance among the nodes in the unvisited set is infinity (when planning a complete traversal; occurs when there is no connection between the initial node and remaining unvisited nodes), then stop. The algorithm has finished.
6. Otherwise, select the unvisited node that is marked with the smallest tentative distance, set it as the new "current node", and go back to step 3.

Dijkstra's SSSP Algorithm:

```
dist[s] == 0    fringe = { s }
while not isempty(fringe):
  v = a fringe vertex with min dist
  remove v from fringe
  for each nbr w of v:
    add w to fringe (if not there already)
    newdist = dist[v] + wt(v,w)
    if newdist < dist[w]:
      dist[w] = newdist    parent[w] = v
```

Parent: in our final solution, we can look back at the parent nodes saved to "build" the shortest path we have found.

Fringe: set of nodes that have a neighbour among the nodes whose distances we know. A **greedy algorithm** is an algorithmic paradigm that follows the problem solving heuristic of making the locally optimal choice at each stage. This algorithm is greedy because at each step we remove the fringe node with the minimum distance so far. It is optimal on graphs (or acyclic digraphs) with non-negative edge weights: distance-so-far of fringe node with minimum distance-so-far in length of shortest path from start to that node.

A* Algorithm

A*: *uses heuristic* to estimate remaining distance to target. If the heuristic underestimates the cost (always less than or equal to the cost), then A* finds the shortest path and usually considers less nodes than Dijkstra's. In this algorithm, we have some extra information that allows us to process certain nodes before others.

Add start node, each node has parent/cost

- Cost: current minimum distance to that node (0 for the start node)

- Heuristic (target, v): a heuristic can be the Euclidean distance to the goal. If a heuristic must be good estimate, you cannot guarantee that you have the shortest/lowest cost path.

Why is it coded like “if next not in done”? You don’t want to process a node that has already been added to the path.

We have three sets of nodes:

- Done: finished processing, assigned path and weight
- Fringe: currently processing
- Nodes we haven’t seen (will be in neighbours of the nodes we are looking at)

Cost[current] + wt(current, next)

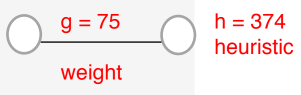
- Cost of current node + weight of an edge
- If next is not in cost (we don’t have the distance for it)
- Or if new_cost < cost[next]: don’t bother updating if it’s not better

Algorithm:

```
fringe = PQ()   PQ = priority queue
fringe.add(start, 0)
parent, cost, done = {}, {}, []
parent[start], cost[start] = None, 0
#cost[v] will be min dist-so-far from start to v
#if heuristic(target, v) is always less/equal than min dist(target,v),
#then final cost[v] will be min dist from start to v

while not fringe.empty():
  current = fringe.remove() # min priority   Get lowest cost node
  done.add(current)
  if current == target: break
  for next in nbrs(current):   Look at neighbours of current node
    if next not in done:
      new_cost = cost[current] + wt(current, next)
        if next not in cost or new_cost < cost[next]:
          cost[next] = new_cost       h
                       g
        f priority = new_cost + heuristic(target, next)
          fringe.add(next, priority)
          parent[next] = current
```

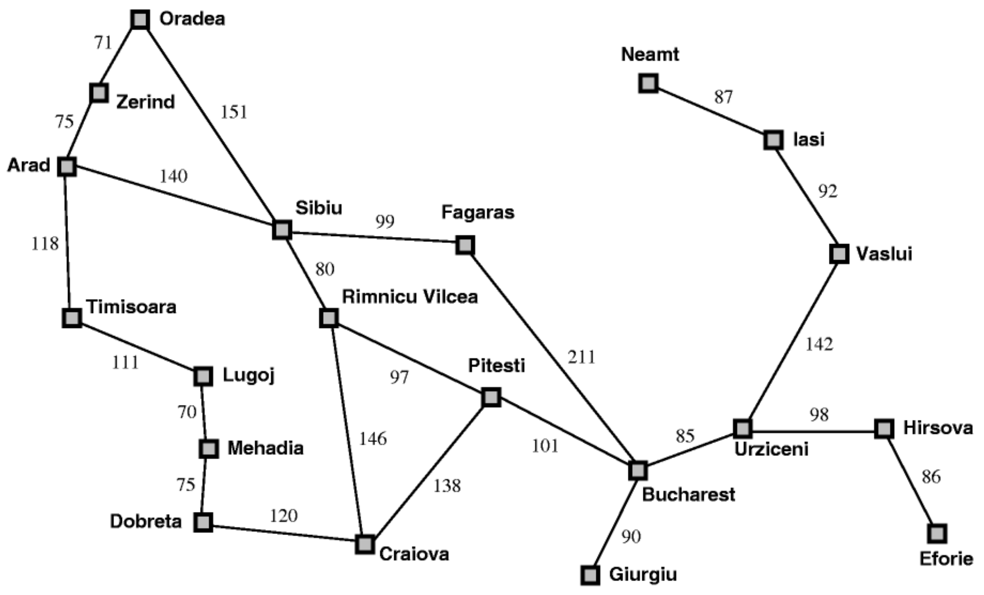
f = g + h
f = 75 + 374 = 449



Arad to Bucharest

Heuristic: straight line distance to B, Euclidean distance to the end node (this is easy to compute with latitude/longitude coordinates)

- This means that we’ve calculated the distance to Bucharest for each location and put that as a value for each node.



Straight−line distance to Bucharest

Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

```
      heuristic: straight line dist to B
      (easy to compute using latitute/longitude coordinates)
 A   C   D   F   L   M   O   P   R   S   T   Z
366 160 242 176 244 241 380  10 193 253 329 374

initialize:  (other costs initially infinite)
           A
cost       0
priority 366
           * current A cost 0
---------------------------------
A nbrs:
S newcost 0 + 140
T newcost 0 + 118
Z newcost 0 +  75

       S   T   Z
cost  140 118  75
heur  253 329 374
pri   393 447 449
       *          current S cost 140
--------------------------------
S nbrs:
A done
F newcost 140 +  99  239 update
O newcost 140 + 151  291 update
R newcost 140 +  80  220 update

       S   T   Z   F   O   R
cost  140 118  75 239 291 220
heur  253 329 374 176 380 193
pri   393 447 449 415 671 413
                           *  current R cost 220
--------------------------------
... (exercise: do the next step)
```

Initialize all nodes to infinity, start (Arad) to 0

- Put its priority (366, heuristic + weight)
- What’s the first node we process? Arad, the only one
- Arad has three neighbours so we update their costs (S, T, Z)
  - New cost = shortest path (heuristic) + edge weight
  - Now we see cost, heuristic, and priority (sum of those two prior values)
  - Pick the one with the lowest priority value, etc. (see trace below)

Now let's apply A* to a sliding tile puzzle. To do this, we need a heuristic. If we want to find the shortest solution, we need to make sure we use A* *with a heuristic that doesn't overestimate the true path*.

We can start with a usual state space adjacency graph:

- Node: sliding tile state (position)
- Edge: a pair of states, can single-slide from one to another
- Cost of a path: sum of number of edges from start (unit-cost weights)
- Choice of heuristic function:
    - Number of misplaced tiles
    - Sum, over all tiles, of Manhattan distance (taxicab distance) from current to final location
        - **Manhattan distance:** The sum of the horizontal and vertical distances between points on a grid

Each of these heuristic functions is always less/equal to the number of moves to solve so with A* each yields the shortest solution.

Example: 4 3 2, 1 5 0

- 4: 1
- 3: 1
- 2: 1
- 1: 1
- 5: 0
- Sum: 4

Is an underestimate? Yes
Are all tiles only going to move once? No

Humans solving sliding tile

Humans and computers often solve problems differently.

We can solve sliding tile puzzles by decomposition. We can solve a 2x3 sliding tile puzzle by reducing it to a 2x2 puzzle. Consider any 2x3 puzzle with tiles 1-5.

- Claim A: we can always get to position with numbers in left column correct (1, 4)
- Claim B: after getting to that position, original problem solvable if and only if solving remaining 2x2 problem (while leaving left column in place) solvable

Claim A Proof:

```
get to  1 * *  [ how ? ]
        * *

now where is 4? two cases

case 1:  1 * *   done :)
         4 *

case 2:  1 * *   1 4 *  1 * 4
         * 4     * *    * *

in each of these cases  1 * *
get to this             * 4

then ... *   *  ...  * * *  ...  1 * *
         1 * 4       1 4         4 *

end of proof :)
```

Claim B Proof:

- Each tile move preserves the solvability condition
    - E.g. assume number of rows is odd
        - Solvability condition: number of inversions is even
        - Each tile move preserves parity of number of inversions
            - Moving a tile left or right does not change number of inversions, and therefore doesn't change its parity of number of inversions
            - Moving a tile up or down does change number of inversions. The tile moves past an even number of other tiles (which is the width of the board minus 1). So, the number of inversions changed by $\pm 1, \pm 1, \pm 1, \ldots, \pm 1$, an even number of times. So it's still even, even if the *number* of inversions change, the *parity* did not.
- So original 2x3 position solvable if and only if position with 1,4 in place solvable
- Two cases:
    - Case 1: clockwise cyclic order of other three tiles is (2, 3, 5)
        - Subproblem solvable: in order, just need to be cycled into place
        - Original position is solvable: because the subproblem is solvable
        - Original position had even number of inversions: because odd number of columns so if it was solvable this has to be true
    - Case 2: clockwise cyclic order of other three tiles is (2, 5, 3)

- Sub problem unsolvable: out of order, can never switch inversion of 5 and 3
- Original position has odd number of inversions (so unsolvable): why?