Introduction to Pathfinding III

Greedy Best-First and A* Search

Ryan Toner

Intelligent Graph Pathfinding - Learning Objective

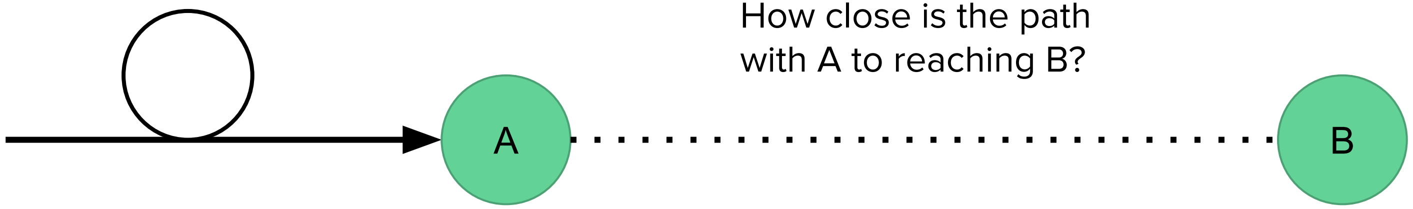
- We are now familiar with Dijkstra’s Algorithm
  - Informed search for optimal pathfinding in weighted graphs
- What about large graphs and time-constrained systems? Can we do better?
  - Dijkstra may be too slow
- What if there were a way to tell our algorithm in real-time what nodes are closer to the goal node?
  - Creating intelligent searches with intelligent search strategies
- The objective of this lecture is
  - Introduce the notion of heuristics
  - Gradually refine Dijkstra’s algorithm to incorporate heuristics
  - Demonstrate heuristic search with Greedy Best-First Search
  - Heuristic integration with A* Search for optimality and performance

Rethinking Dijkstra’s Algorithm

- Useful in weighted graphs
- Solves *single-source shortest path* problem
    - Computes shortest path to all (reachable) nodes in a graph
    - By consequence of this, Dijkstra can also find shortest path between two nodes and terminate
- Selects paths with the lowest weights from a Priority Queue data structure to ensure minimum path property
    - Mini-proof that Dijkstra returns the shortest path:
        i. Assume by contradiction that Dijkstra does not return the shortest path
        ii. This means that a longer path had higher priority than the optimal path and was selected first
        iii. This violates the min-heap property (the lowest value must be selected first)
        iv. By contradiction, it is shown that Dijkstra must find a minimum path

Being “Smarter” about selecting paths

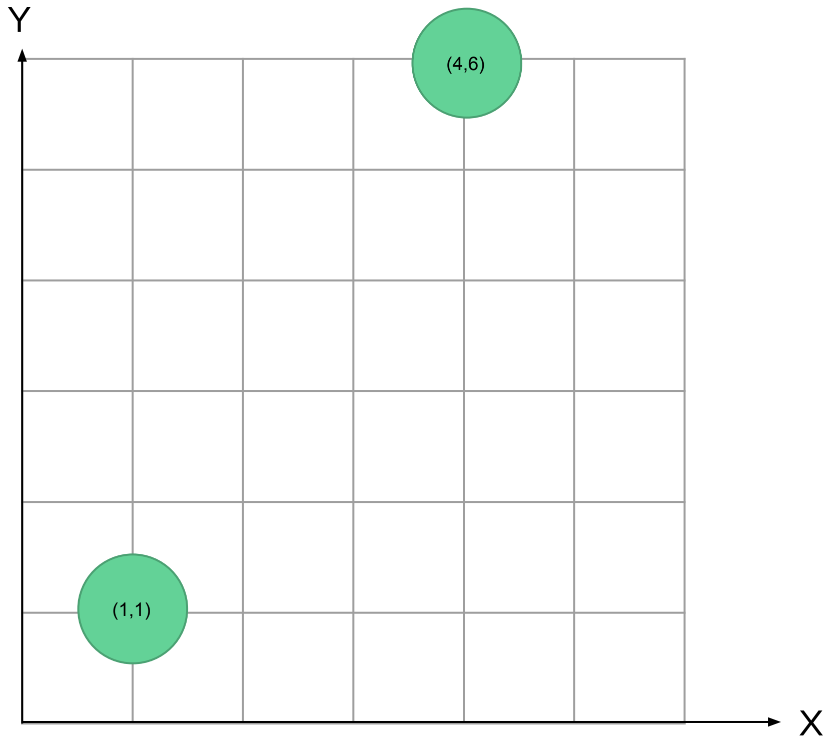
- Dijkstra is an informed search because it constrains itself by its path length property e.g. it selects min-length paths
- Suppose we had another selection metric for paths other than path length
- Let’s call our function that estimates the “quality” of a path $\hat{h}(\bar{x})$
  - In the case of pathfinding between arbitrary node A and terminus B, we can define the “quality” of a path as how close it is at getting us to B.
  - This function will only analyze where the path is currently (its head node)
- $\hat{h}(\bar{x})$ is a heuristic function



Heuristic Function Explained

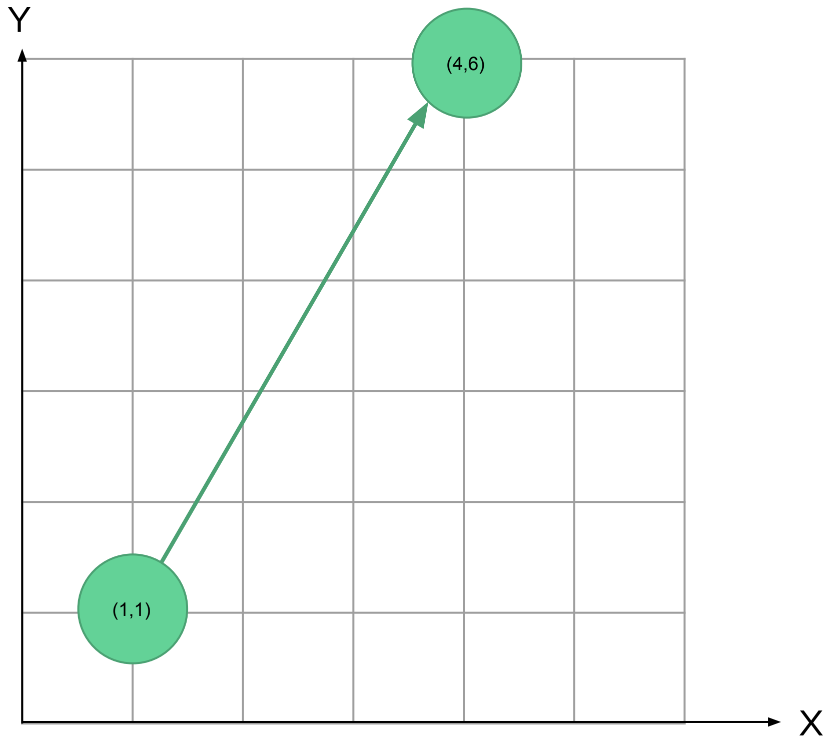
- A heuristic function is a function that estimates how good something is
- In the case of our path example, $\hat{h}(\bar{x})$ estimates how close how close we are to the destination node B
- The domain of this function is nodes
- The codomain (and range) of this function is the set of real numbers including zero, $\{\mathbb{R}^{+} \cup \{0\}\}$
- A good heuristic:
  - $\bar{x} = B \rightarrow \hat{h}(\bar{x}) = 0.$
  - $\bar{x} \neq B, \rightarrow \hat{h}(\bar{x}) > 0.$

Example: Lattice (grid)



- Here is an example square lattice.
- At each intersection, there is a node, and the connection lines indicate the neighbors of that node.
- If we wanted to find the best path between the node at (1,1) and the node at (4,6), we could easily use BFS and it would work as expected.
- However, we can do better by using a heuristic function to find the path faster!

Example: Lattice (grid)



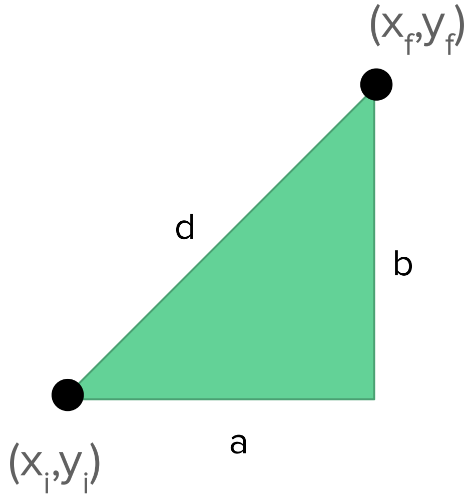
- We want a path from (1,1) to (4,6)
- For any given node on the lattice, can we conjure up a formula to estimate how close it is to the end goal?
- Of course! A possible answer is to use euclidean distance.
- For our random node on the lattice n = (x,y)
- Define $d(n) = [(x-4)^2 + (y-6)^2]^{½}$
- This is simply the euclidean distance between our node n and the destination node.
- Notice in the case where n = (4,6), the distance is clearly 0!
- An “admissible” heuristic function is actually euclidean distance:

$$\hat{h}(\bar{x}) = \lceil d(n) \rceil$$

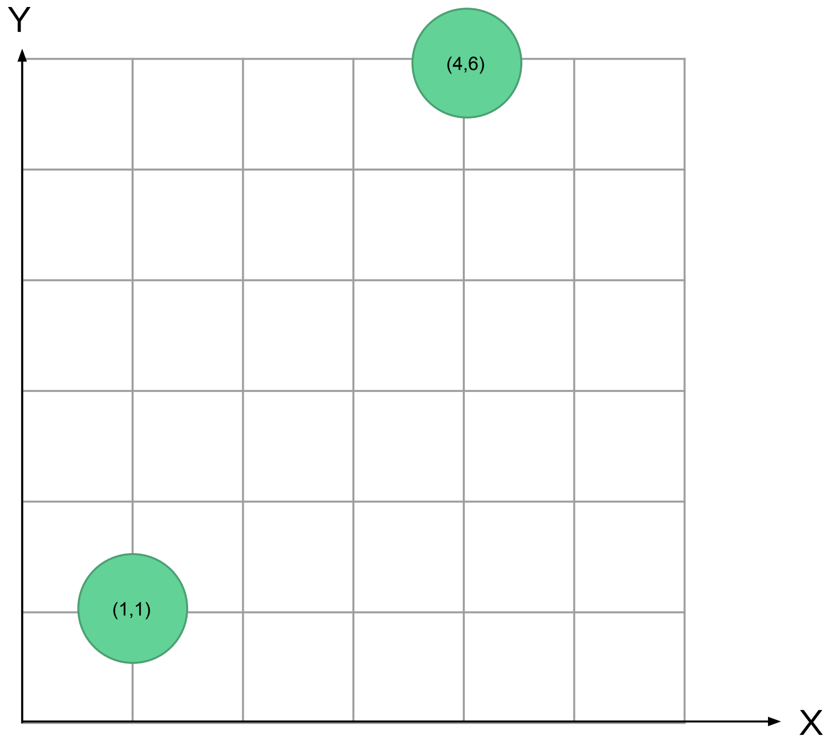
The ceiling function of d.

Euclidean Distance

- Distance from node n = (x_i,y_i) to node f = (x_f,y_f)
- Derived from the pythagorean formula:
  - $a^2 + b^2 = d^2$
  - a is the x distance: $|x_i-x_f|$
  - b is the y distance: $|y_i-y_f|$
  - $(x_i-x_f)^2 + (y_i-y_f)^2 = d^2$
  - $d = [(x_i-x_f)^2 + (y_i-y_f)^2]^{1/2}$
- Useful for graphs with nodes represented in continuous space
- But graphs are discrete structures, which is why we took the ceiling function to round up



Example: Lattice (grid)



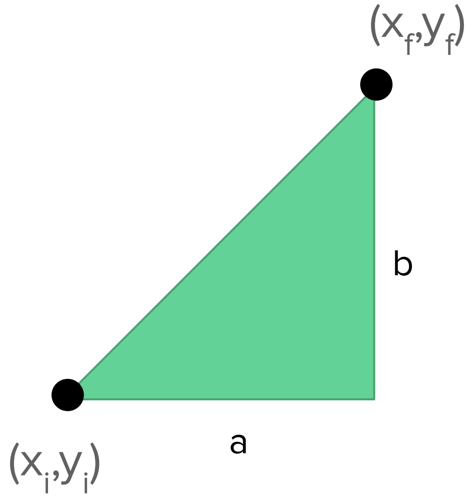
- Back to our lattice example
- Let us present another heuristic that will better indicate our proximity to the end node (4,6).
- Euclidean distance is actually not a tightly-bounded heuristic on a lattice, since we cannot actually traverse the diagonal shortest path — notice that all neighbors are strictly vertical or horizontal adjacent to a node.
- What if we simply sum the distances in each direction?
- For node n = (x,y), let $d_2 = |x-4| + |y-6|$

$$\hat{h}(\bar{x}) = d_2$$

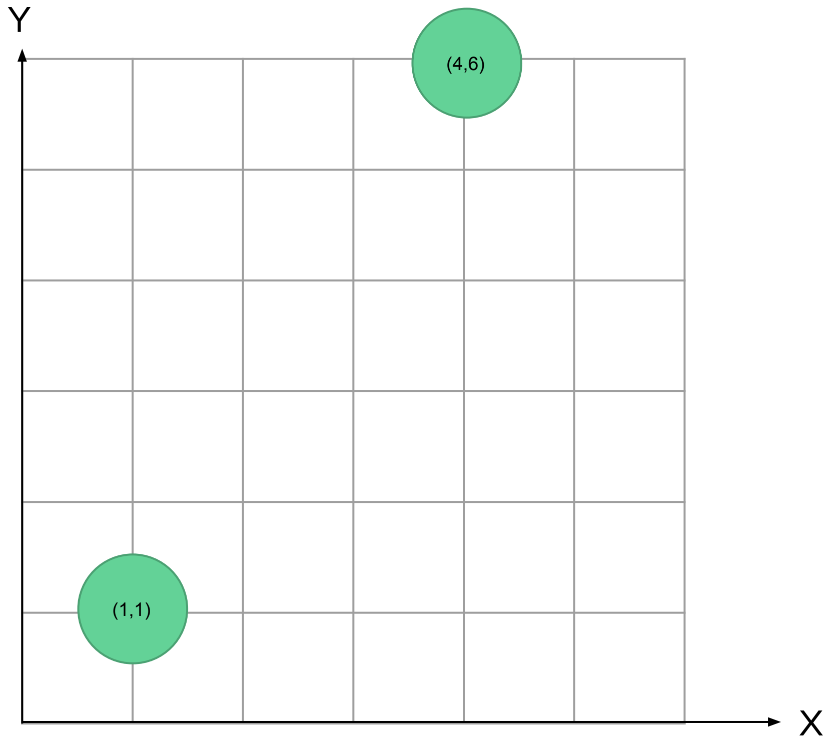
The manhattan distance.

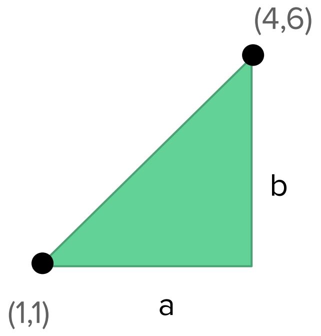
Manhattan Distance

- Tends to work better in grids, lattices, and boards
    - Think about NYC Grid layout (a car must use the roads and can't cut through buildings ⇒
- Much simpler computation with clear integer output (no need for ceil)
    - $d = a + b$
    - $a = |x_f - x_i|$
    - $b = |y_f - y_i|$



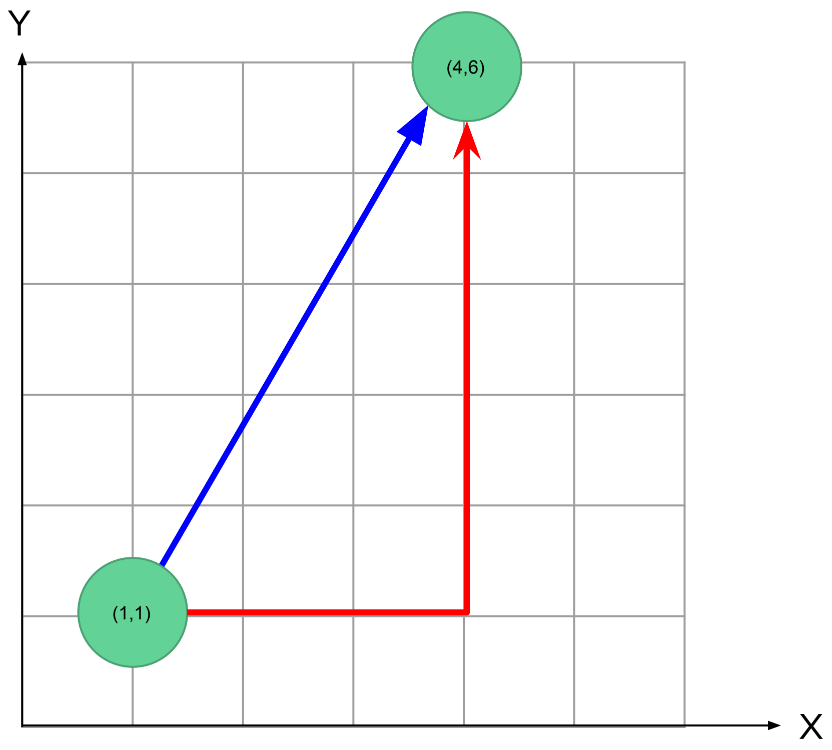
Example: Lattice (grid)





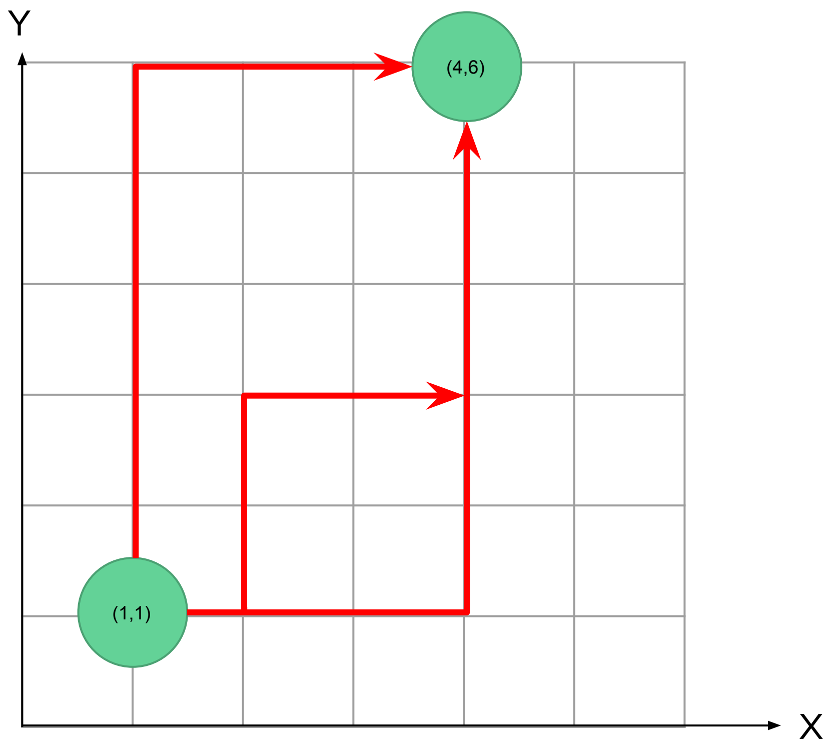
- The distance a = |4-1| = 3
- The distance b = |6-1| = 5
- The euclidean distance is
  - $(3^2 + 5^2)^{½} = (34)^{½} \approxeq 5.8$
- ⇒ the heuristic will take ceil(5.8) = 6
- The manhattan distance is
  - 3 + 5 = 8

Example: Lattice (grid)



- It should be clear that the blue distance is the cartesian (euclidean) distance, and the red distance is the manhattan distance.
- The blue line underestimates how far away these two nodes are, since it is impossible to traverse along a diagonal in a square lattice.
- The manhattan distance is the true minimum distance, and is a better heuristic function for this graph
  - The closer you can estimate without going over!
  - The price is right?
- What does this difference really make?

Example: Lattice (grid)



- Another advantage of manhattan distance is there is in scenarios like grids where there are many paths to the solution — all with the minimal distance
- This flexibility of path independence means this estimation will work even if there are obstacles in the graph, as the actual distance will only ever be greater than the heuristic estimation
- This means manhattan distance may be flexible for a search in maze scenarios

Admissible heuristics

- We have seen two heuristic functions
    a. Euclidean Distance
    b. Manhattan Distance
- Both are commonly used in graphs, but there are many more
- “Admissible” heuristic functions **never** overestimate how far the distance is
- Good heuristic functions <u>should</u>
    a. Evaluate the solution node as having 0 distance
    b. Evaluate all other states with a distance > 0
    c. Tend to bound their estimation within a minimized margin of error
    d. Consequently, it is okay to underestimate, but not too much for good performance in a pathfinding algorithm
- Try to pick a heuristic that bounds below or equal to the actual cost

An aside on Philosophy — Teleology of Heuristics

- It seems somewhat paradoxical to think that a heuristic can estimate how close we are to the solution without having the solution first!
- This is what makes developing heuristics so hard…
  - Sometimes you need the answer prior to the premise!
- **Teleology** is the act of describing something in terms of its end goal
- Heuristics themselves are a teleological construct, since they estimate the end goal without knowing how to get there
  - Heuristics allow us to consider the solution through estimation
- We may work towards the solution in a teleological manner via heuristic searching

Using a heuristic function for pathfinding

- Let’s modify Dijkstra’s algorithm a bit.
- Suppose that instead of assigning the path cost as the **priority** value for our priority queue, we instead assign our heuristic function’s output as the priority

High-Level Dijkstra’s Algorithm:

1. Push root node onto Priority Queue
2. Dequeue the top node (node with the lowest path cost total)
    a. If the top node is marked, ignore it and repeat step 2
3. Mark the top node
4. Enqueue all unmarked neighbors and calculate a new path cost for each neighbor
5. Repeat step 2-5, until queue is empty or terminus point is found

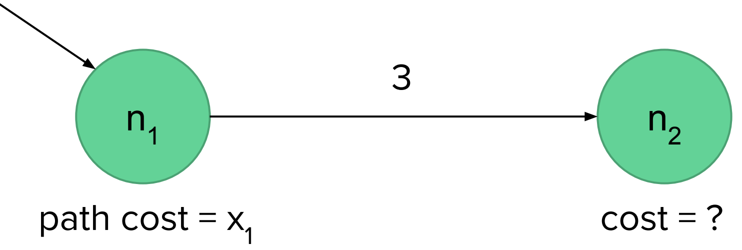
High-Level Dijkstra's Algorithm:

1. Push root node onto Priority Queue (initialize path total cost to 0)
2. Dequeue the top node (node with the lowest path total cost)
   a. If the top node is marked, ignore it and repeat step 2
3. Mark the top node
4. Enqueue all unmarked neighbors and calculate a new path cost for each neighbor
5. Repeat step 2-5, until queue is empty or terminus point is found

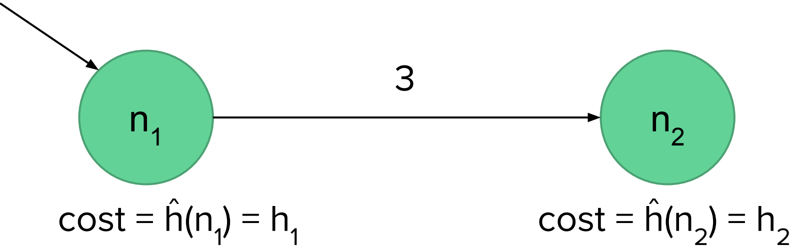
Replacing path cost with heuristic cost:

1. Push root node onto Priority Queue **with cost 0**
2. Dequeue the top node (**node with lowest heuristic cost**)
   a. If the top node is marked, ignore it and repeat step 2
3. Mark the top node
4. Enqueue all unmarked neighbors and **assign heuristic value $\hat{h}$(neighbor) for each neighbor**
5. Repeat step 2-5, until queue is empty or terminus point is found

Path preference with a heuristic?



- Dijkstra's Algorithm computes the path cost of getting to n_2 from n_1 as the total running cost x_1 + the weight of the edge from (n_1,n_2).
  - $x_2 = x_1 + 3$



- This new algorithm is setting the cost of $n_1 = h_1$, and the cost of $n_2 = h_2$. Notice it does not care about the running total path cost or edge weight!

What is this new algorithm?

- This algorithm is called **greedy** best-first search.
- It is greedy since it will simply pick to dequeue the node with the lowest heuristic cost. It does not account for the path length!
- By consequence, greedy best-first search does **not** find optimal solutions!
- However, greedy best-first search tends to find solutions very quickly
    - It will prefer solutions that bring it closer to the end goal.
- Dijkstra’s algorithm simplest searches for the shortest path, and ends up reaching the shortest path to a goal state
- Greedy best-first search actually tries to evaluate the proximity of these nodes to the solution state at each step!

Greedy Best-First Search - Overview

- Does not account for edge weights or path length
    - Operates by calculating heuristic function for each node found
- Useful when you simply want to know if a path to the solution state exists
- Great at quickly making gains towards solution by leveraging heuristic function
- Like DFS, may produce strange and efficient paths
- Efficient in large graphs

From greedy to smart!

- You might imagine that we can be more intelligent about the way we use our heuristic function.
- A smarter algorithm could leverage our heuristic function to find perfect solutions faster!
- What if we combined the heuristic notion from Greedy Best-First Search with the optimal pathfinding of Dijkstra’s Algorithm?

A* Algorithm

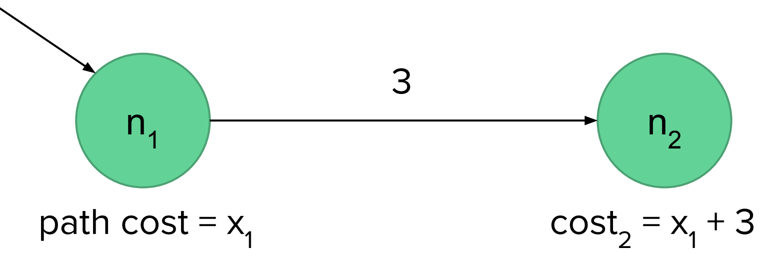
A heuristic modification of Dijkstra’s Algorithm:

1. Push root node onto Priority Queue with path cost 0 and path weight 0
2. Dequeue the top node (node with lowest path cost)
    a. If the top node is marked, ignore it and repeat step 2
3. Mark the top node
4. For each unmarked neighbors of the top node, calculate the new path cost, increment path weight, and enqueue
5. Repeat step 2-5, until queue is empty or goal state is found

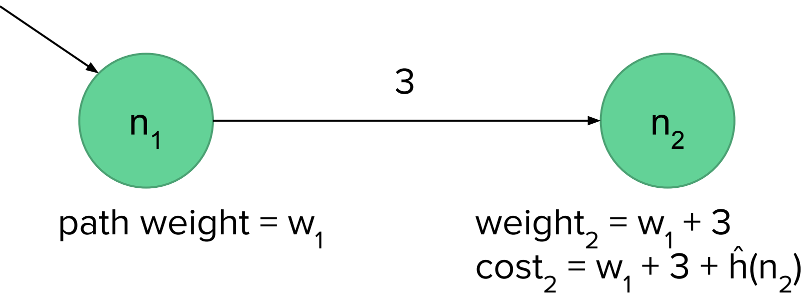
Path cost and path weight calculation:

1. $\text{PathWeight}_{\text{neighbor}} = \text{top.PathLength} + \text{EdgeWeight}(\text{top, neighbor})$
2. $\text{PathCost}_{\text{neighbor}} = \text{PathWeight} + \hat{h}(\text{neighbor})$

*Smarter [Path preference with a heuristic?]



- Dijkstra's Algorithm computes the path cost of getting to n_2 from n_1 as the total running cost x_1 + the weight of the edge from (n_1,n_2).
    - $x_2 = x_1 + 3$



- A* uses the same notion of running path length as Dijkstra's algorithm, but also adds the heuristic estimation for each neighbor node!
- This makes A* run much faster by punishing poor moves and promoting good ones by their estimated proximity to the goal!

Check your understanding: Path costs

When adding a neighbors of the **Top** node **(N)** to the priority queue, understand what each algorithm does to compute the priority of **N**:

1.	Dijkstra	➡	pathCost(**Top**) + E(**Top**, **N**)
2.	Best-First	➡	$\hat{h}$(**N**)
3.	A*	➡	pathWeight(**Top**) + E(**Top**, **N**) + $\hat{h}$(**N**)

   a. Important to recognize that A* requires both a path cost and path weight metric. Only the path cost is used for the priority value.
   b. PathCost in Dijkstra is just “PathWeightToHere”

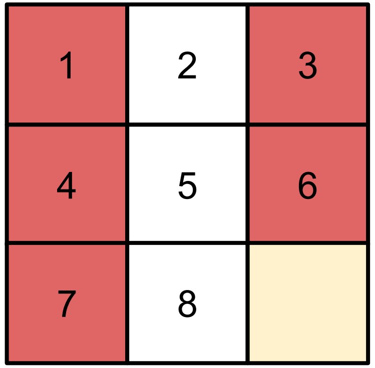
Dijkstra is a subset of A* search

- Dijkstra's algorithm can be thought of as a subset of A*
- By setting an equal heuristic value to all nodes, only path weight metric will affect the path priority
- In the worst-case scenario (assuming an admissible heuristic), A* is bounded in asymptotic runtime complexity by Dijkstra's runtime.
- Consequently, it is never bad to use A* unless no admissible heuristic exists

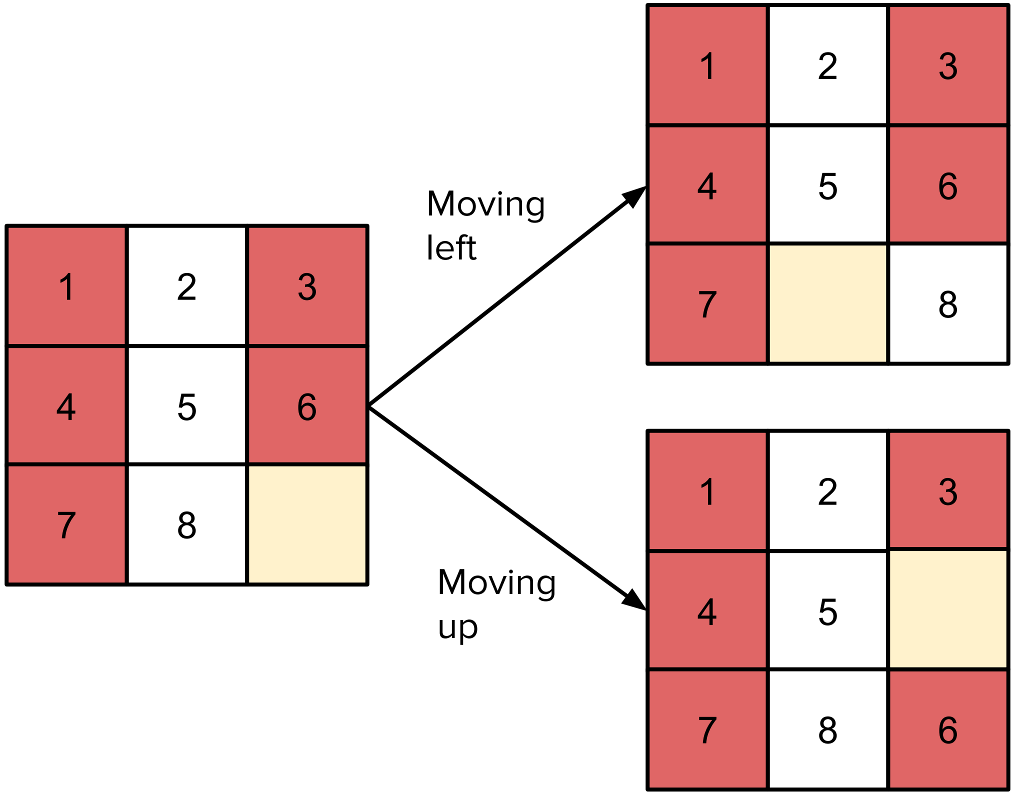
Search-ception: BFS is a subset of A* where the heuristic value is constant for all nodes and all edge weights are equal in the graph.

Using Heuristics for Puzzle Solving: n^2-1/ 8-Puzzle

- The n^2-1 Puzzle is a classic sliding puzzle game
- As the name indicates, the puzzle is a square with number 1 through n^2-1, and a blank tile
- The blank tile allows for a between 2-4 degrees of movement freedom by swapping only with neighboring tiles (depending on where the blank tile is)
- It is computationally difficult to solve due to large number of board configurations, possible paths, etc
- This class of problem is NP-complete, meaning there is no polynomial time algorithm for solving this puzzle, but we can verify solutions in polynomial (constant) time



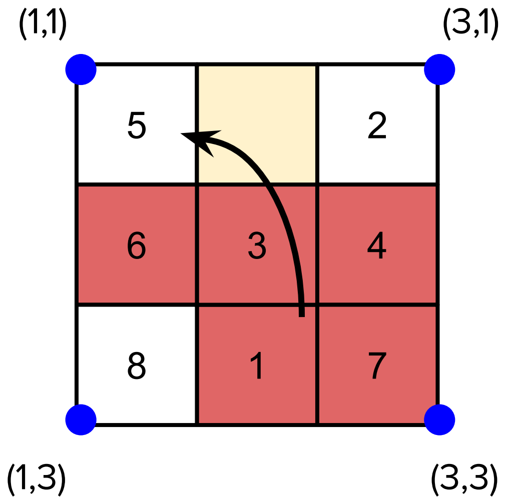
Valid Moves in n^2-1 Puzzle



The blank tile may be moved immediately up, down, left, and right, but not diagonally. In this case, there are only 2 degrees of freedom.

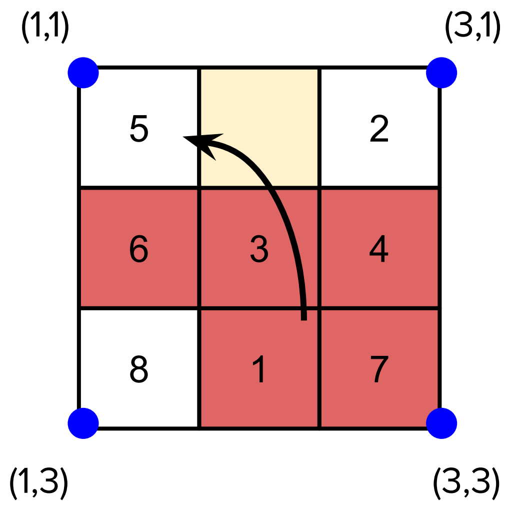
A Simple tile heuristic

- We can define the tiles of the puzzle with basic 2D coordinates
- Then, we can formulate how far away a tile is from its solution state using the manhattan distance!
- For example, tile 1 tile should be at (1,1), but it's currently at (2,3)
    - The manhattan distance is |3-1| + |2-1| = 3
- $\hat{h}_t$(tile) = manhattan distance from the tile to its solution location



Higher-order heuristic

- We can use the simple tile heuristic as a parameter to formulate an estimation of how far the entire board is from the solution state
- Take a simple sum of each non-blank tiles manhattan distance heuristic as described before
  - Exclude the blank tile since all other tiles in place implies the blank tile is also in place
- $\hat{h}_b(\text{board}) = \Sigma\, \hat{h}_t(t_i)$, for tiles 1 to n^2-1



t	1	2	3	4	5	6	7	8
$\hat{h}(t)$	3	1	2	2	2	2	2	1

$\hat{h}_b(\text{board}) = 3 + 1 + 2 + 2 + 2 + 2 + 2 + 1 = 15$

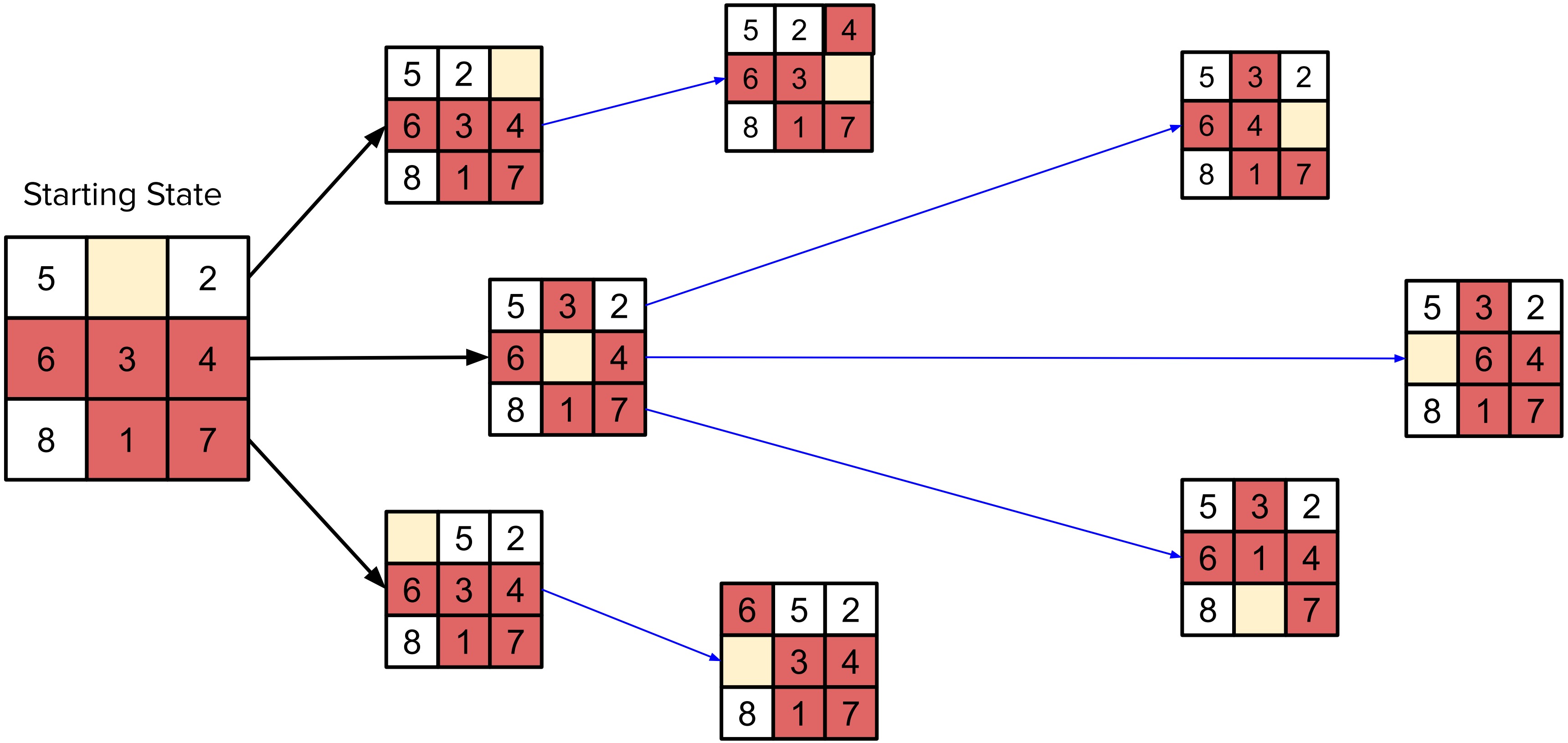
What is this heuristic doing?

- For a given board, we can estimate the number of moves away it is from being solved
- Is it an admissible heuristic?
  - Since this tile heuristic assumes you can easily move each tile directly to its solution space, it will underestimate the number of moves needed to get that tile in place
  - For the entire board, it should underestimate the number of moves needed to solve the puzzle
  - Only in the solution state will $\hat{h}$(board) = 0
- Yes, our $\hat{h}_b$(board) should generally underestimate the number of moves needed, so it is an admissible heuristic!
- Remember that bounding below or equal to the actual solution state is required for a heuristic to be admissible. Therefore, we conclude that this heuristic will permit A* to return optimal solution paths

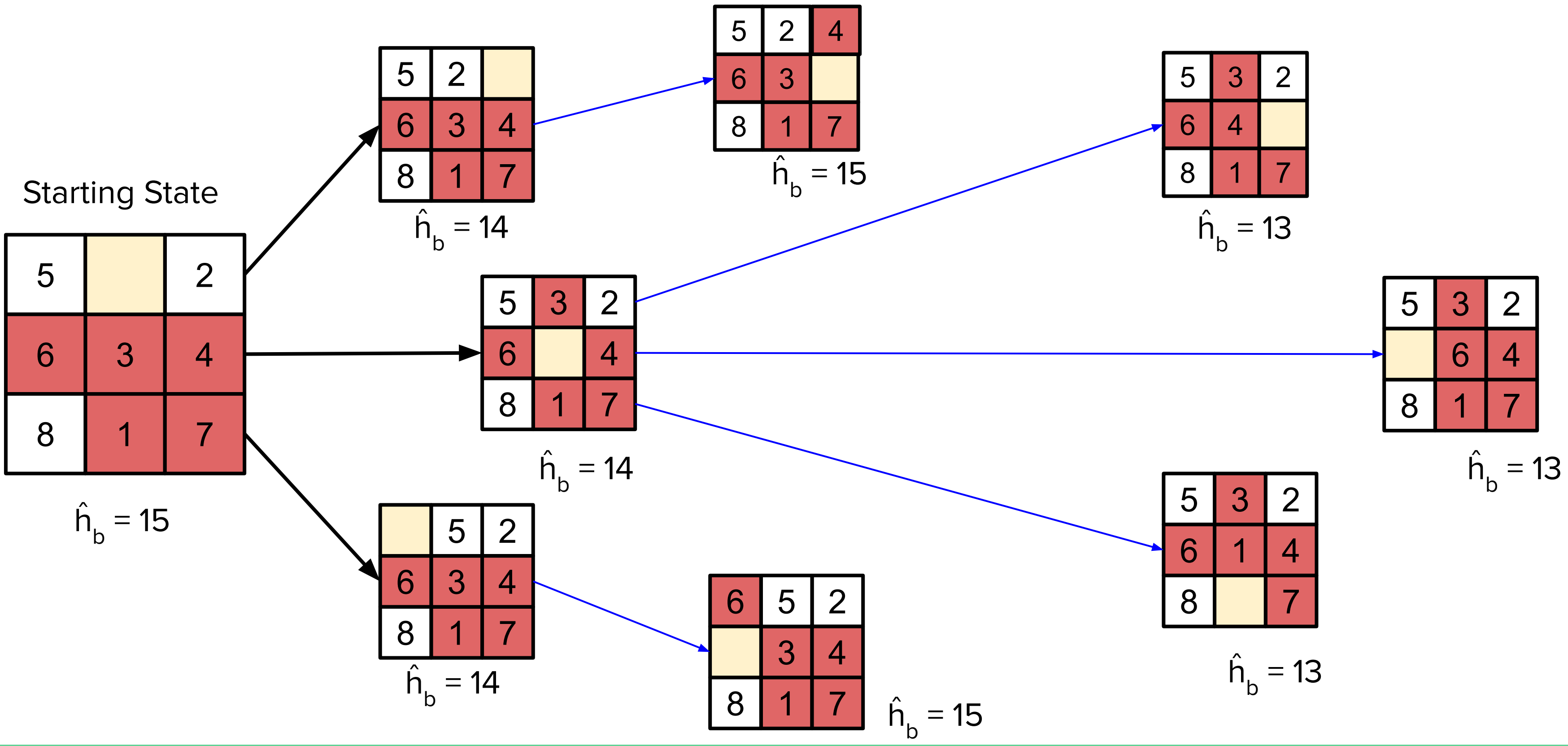
A graph-theoretic approach

- We can consider our board as being a node
- Each move will represent a new board, or a new node being generated
- We can construct edges from boards to their child boards
  - Child boards are the boards generated by making any permissible move in their parent
  - An edge weight of 1 is sensible to represent 1 move
- In this sense, we can use a heuristic search algorithm with a dynamically created graph structure, since we must dynamically unfold each parent's moves.
- Create a directed acyclic graph, or **tree**, that represents making valid moves (without being able to revert to previous moves).
- Run a heuristic-search on the nodes of this tree to solve the puzzle
  - A path through this tree from start to finish will tell us how to solve the puzzle

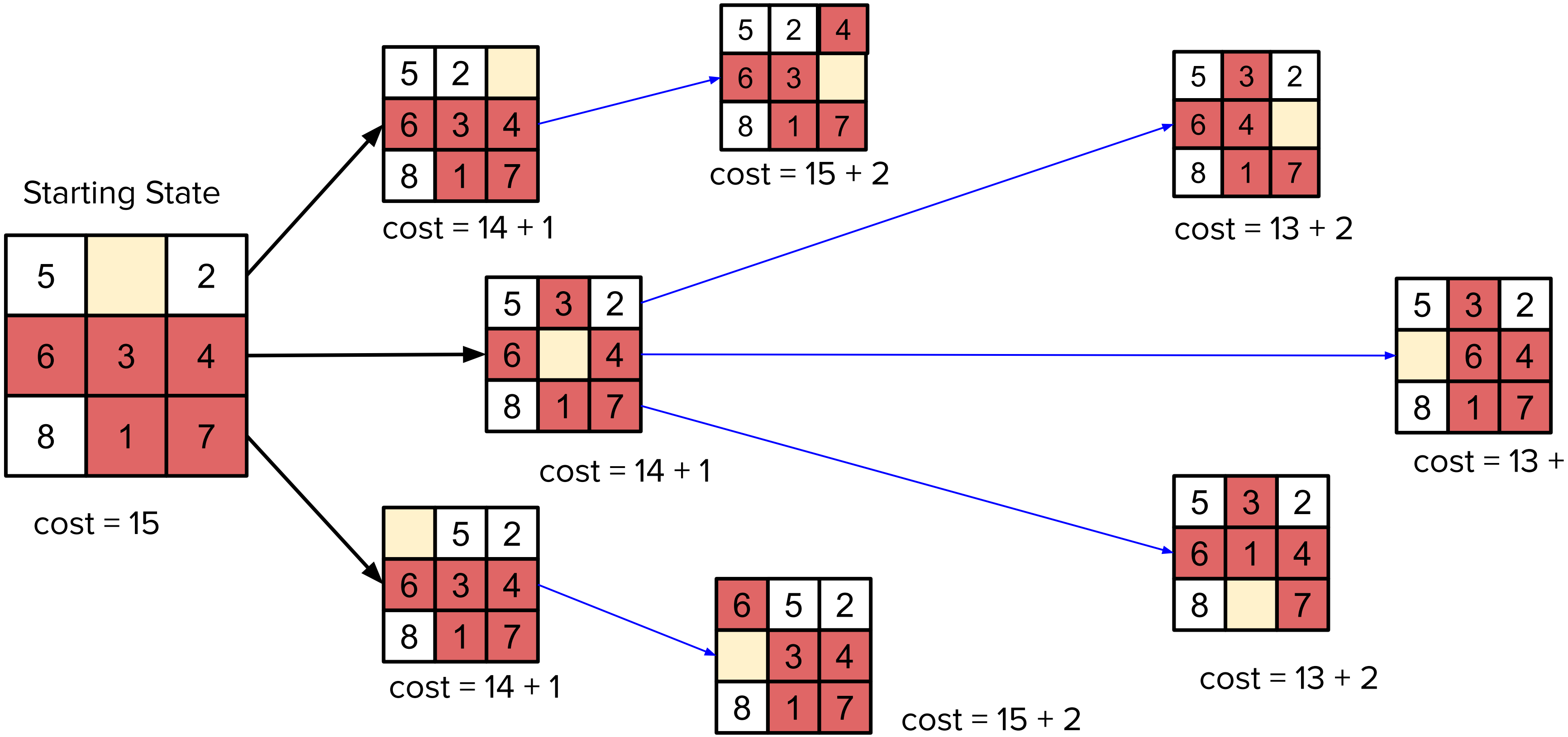
Dynamic tree generation



Dynamic tree generation with board heuristic



Dynamic tree generation with board heuristic, path length



How does this work?

- The root node of the tree is the starting state (a scrambled, valid board)
- We use a priority queue data structure and a hash set to designate discovered/ marked board configurations
    - You should implement a custom hash function for fast performance
- We use our priority queue, assigning priority based on the heuristic search
    - Greedy: $\hat{h}_b(x)$ for board x
    - A*: $\hat{h}_b(x) + g(x)$ where $g(x)$ is the path length to reach board x
- Upon the dequeue operation, we get a board with the higher priority
    - Then, we can add this board configuration to the hashset (mark it)
    - We generate the neighbors of the board by making all possible valid moves that find new board configurations
    - Add each of these neighbors with their computed priority to the queue

A Hash Function for the 8-Puzzle

- An unsigned-integer works for a hash value.
- This gives us 32-bits to store our puzzle in
- A tile can have a value from 0-8, (let the blank tile = 0)
- Storing 000-111 takes 3 bits
- 9 tiles * 3 bits per tile = 27 bits needed
- We can make a fast, unique, deterministic, and efficient hash!
  - This is an injective (one-to-one) hash function
- Iterate through the tiles from top-to-bottom and left-to-right:

```
hashValue = 0
For int row = 1 to 3
    For int col = 1 to 3
        hashValue = (hashValue  << 3) + (uint)tile[row,col].value
```

High-Level Algorithm using A*

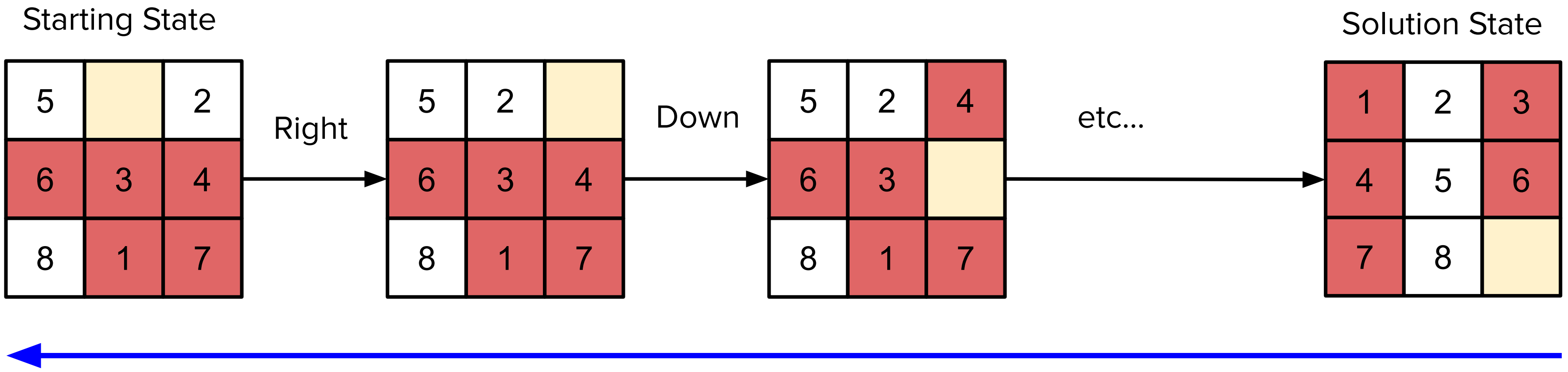
1. Push root node (scramble state) onto Priority Queue with path cost 0 and path weight 0
2. Dequeue the top node (node with lowest path cost)
   a. If the top node is marked, ignore it and repeat step 2
3. Mark the top node by adding the board configuration to a hashset
4. Generate the neighbors of the top board by making all possible valid moves that find a new board configuration not in our hashset, increment path weight, calculate the new path cost, and enqueue
5. Repeat step 2-5, until queue is empty or goal state (solved board) is found

Path cost and path weight calculation:

1. $\text{PathWeight}_{\text{neighbor}}$ = top.PathLength + EdgeWeight(top, neighbor) ← Edge weight always = 1
2. $\text{PathCost}_{\text{neighbor}}$ = PathWeight + $\hat{h}_b$(neighbor)

*We call the path weight function g(x). If you only use PathCost = $\hat{h}_b$(neighbor), you are using Greedy-Best

Backtracking for the solution



- A subset of our final tree structure will contain our solution.
- We can work backwards from the solution state by calling each unique parent to trace back our path through the tree!
- Recall that this technique is called backtracking. It can be easily accomplished with a stack.
- With the backtracked solution, we can iterate forward and know each move to take to reach the solution and solve our puzzle!

Overview

- Use heuristics-based searching whenever applicable
- Ask yourself the following:
  - Am I trying to get to a single solution state?
  - If yes, can this problem be modelled by a heuristic?
  - If yes, can you develop an admissible heuristic?
  - If yes, try A*, otherwise try greedy best-first
- Greedy search does not account for path length
- A* combines [$g(x)$ = path length to x] + [$\hat{h}(x)$ = heuristic estimation of node x]
- If you care about optimal solutions (note there may be several optimal solutions for a problem e.g. paths of equal cost), use A*
- Think about the structure of a dynamically generated tree for problems where neighbors must be computed upon a dequeue operation