These are heuristics:

1. Heuristic: pick configuration with greatest number of correctly placed tiles.
2. Heuristic: pick configuration that requires the least number of moves.
3. Heuristic: pick configuration that requires the least number of moves from their.

Example: for Puzzles, from all configurations in Sequence Structure, we can explore to "find" the graph search.

In many applications, we have some domain-specific information.
How do we implement the breadth-first search in our approach?

Stated: This heuristic is good only when you are close to the goal.

Breadth-first search: explore nodes generation by generation.

## Generation 0: {A}

## Generation 1: {B, H}

## Generation 2: {C, I}

## Generation 3: {D, J}

## Generation 4: {G}

## Generation 5: {F}

The Breadth-first Search generates A, B, C, D, G, J, I, H, F.

For priority queues, the lowest priority should always be returned. In some cases, the lowest priority is the highest.

Priority Queue

Depth-first search given some order of completeness.

Depth-first search: explore nodes generation by generation.

Two most popular strategies:

1. Depth-first search
2. Breadth-first search

Sometimes, you may not have heuristics to guide your graph search.

Optional Graph Search

Optional Graph Search
0, 1, 2...: these are generation numbers for configurations.

In the context of queues:

- Circuit simulations: data are simulated in time order.
- Transportation etc., where service discipline is FIFO.
- Simulations of systems like bank tellers, machines, public transport, etc.

Queues show up in many applications where the service discipline is

**First-in-first-out (FIFO)**.

**Queues** are faster than priority queues.

Queues are simpler to implement than priority queues.

**Breadth-first search**.

Sequence structure: queue

Queue: First-in-first-out Sequence Structure

Queue: get returns item that was put earliest.

First-in-first-out discipline

Another approach: use a sequence structure that obeys

- property 1: queue returns lowest priority entity
- property 2: queue returns highest priority number

One approach: use priority queues

How do we implement sequence structure for BFS?
V stack is a LIFO sequence structure.

This is a graph search problem.

Why use heuristic graph search then?

If we use Bernoulli graph search, then we can solve another node in the graph. However, in Bernoulli graph search, we take the shortest path from a node to another node in the graph. Remember: in general, there may be many paths from a node to another.

Configuration: (smallest number of moves.) From configuration to sorted.

It is easy to show that BFS can determine the shortest path.

Why use a DFS?


In the context of stacks, push and pop is known as pop.

get is known as get.

Another alternative to graph search strategies: use a stack.
Sequence Structure

However, we will start with arrays to get a feel for search and

structures grow and shrink.

Arrays are not very appropriate for our purpose because our data

Another idea:

Implementing Sequence Structures

Use Sorted Arrays

Bridge Classes: decouple abstractions from implementations
Use the array as a "circular buffer."

Implementing Queues using Arrays:

- **size**: number of elements in Q
- **head**: element that arrived earliest
- **tail**: empty slot for enqueue
- **size**: number of elements in Q

See class StackArray at end of handout.
1. Calculate the size of the array.

2. Check if the expression works for empty queue and full queue.

Exercise: Can you compute size from values of head and tail?

4. Implement a queue using linked lists.

5. Define a sentinel node for the queue.

6. Use a circular buffer for efficient memory usage.

Implementation notes using linked lists:

- Node structure:
  ```
  struct Node {
    int value;
    struct Node* next;
  }
  ```

- Enqueue operation:
  ```
  void enqueue(int value) {
    // Allocate new node
    struct Node* newNode = malloc(sizeof(struct Node));
    newNode->value = value;
    newNode->next = NULL;

    // Find last node in the queue
    struct Node* last = head;
    while (last->next != NULL) {
      last = last->next;
    }

    // Append new node to the end of the queue
    last->next = newNode;
  }
  ```

- Dequeue operation:
  ```
  int dequeue() {
    // Check if the queue is empty
    if (head == NULL) {
      return -1; // Queue is empty
    }

    // Extract the first element
    int value = head->value;
    struct Node* temp = head;
    head = head->next;

    // Free the last node
    free(temp);
    return value;
  }
  ```

- Size calculation:
  ```
  int size() {
    int count = 0;
    struct Node* current = head;
    while (current != NULL) {
      count++;
      current = current->next;
    }
    return count;
  }
  ```
1. Use an array of priority queues (one for each priority level).

2. Implement priority queues into Queue for the appropriate priority.

3. Implement efficient search for non-empty queues with highest priority.

Priority is determined and deduced from that queue.

\( O(1) \) time: where \( d \) is number of priority levels.

Cool implementation of priority queues for this case.

Example: best case search with \# of one-of-place items = 0.9

Important special case of priority queues:

Best priority queue implementation: heap.
Heap:

1. Each node is less than or equal to its children.
2. The root node (top of the heap) is the largest element in the heap.

Examples of heaps: arrays or people in family tree.

`Heap of Priority Queue Elements`

"Don of the living dead"

"Joe Schmoe"

"Don Pingali"

"Don Perignon"

"Don Juan"

Parent is always older than children, but you can have an uncle who is younger than you.

Demos of heaps: ages of people in family tree.

max heap example of heap: crime family

Heap:

1. Each node is less than or equal to its children.
2. The root node (top of the heap) is the largest element in the heap.

Examples of heaps: arrays or people in family tree.

`Heap of Priority Queue Elements`

"Don of the living dead"

"Joe Schmoe"

"Don Pingali"

"Don Perignon"

"Don Juan"

Parent is always older than children, but you can have an uncle who is younger than you.

Demos of heaps: ages of people in family tree.

max heap example of heap: crime family

Heap:

1. Each node is less than or equal to its children.
2. The root node (top of the heap) is the largest element in the heap.

Examples of heaps: arrays or people in family tree.

`Heap of Priority Queue Elements`

"Don of the living dead"

"Joe Schmoe"

"Don Pingali"

"Don Perignon"

"Don Juan"

Parent is always older than children, but you can have an uncle who is younger than you.

Demos of heaps: ages of people in family tree.

max heap example of heap: crime family
Heapifying after removing root element.

Question: How do we get and print Can we implement a property العام العربية؟

Problem: This creates a hole which "Don Pingali" used to be.

Solution: We should fill that hole. Remove that element leave a "hole" in the tree.

Let us look at & get first.

Can we insert another element into this structure and preserve heap property

Get: extract the element with largest property

Put: insert element into data structure and preserve heap property

Remove empty root

Keep moving elements up till you move a leaf element up (Don of the living dead) and move him into Don Pingali's slot

Don DeLouise or Red Don which is more ruthless.

* in our example, "Don Pingali" is moved into root.

Put into a heap.
where advanced design of heap is a complete binary tree.

Advantages of heap:

We can implement priority queues using either linked lists or heaps.

Get Operation That Maintains Complete Tree-ness

Problem with our heap design: there is no guarantee that we will lose completeness.

- Get algorithm that ensures completeness:

Our implementation will not be any more efficient than the list, but we end up with (and shrinking the heap list) so in the worst case, problem with our heap design: there is no guarantee that we will gain completeness at any time.

Advantages of heap:

- In list, it is not large and shrinks each time and get:

- Clever way to fill root: promote element from "last" filled node to root, and keep going recursively down the tree. However, this:

counterexample 1:

- In old algorithm for get, we would promote max of nodes 2 and 3:

counterexample 2:

- But with elements in 2 and 3, and exchanging root with largest of its children etc.

counterexample 3:

Property is maintained after operation?

- In example, size is 12.

Put will make size = 13 and 15 to 101, so push to new level in RB.

- If insert new element into "root", then "root node" in example node 13:}
See class Heap for code.

Heapify is an in-place down the tree $O(n \log n)$ time.

- Put: insert into "leaf +1" element of complete binary tree, $O(1)$ time.
- Binary representation of integer size $O(\log n)$ time.
- Access: Retrieve heap property: Access to the complete binary tree at the root position and walk down the complete tree to the root of the path that the "leaf" element of
- Get: return the root element. Promote the "last" element of
- Keep: heap of size $O(1)$

Summary of priority queue implementation using heaps.