State-Space Search Revisited

- Earlier, we considered three algorithms for state-space search:
  - breadth-first search (BFS)
  - depth-first search (DFS)
  - iterative-deepening search (IDS)
- These are all *uninformed* search algorithms.
  - always consider the states in a certain order
  - do not consider how close a given state is to the goal
- 8 Puzzle example:

  ![8 Puzzle Diagram](image_url)

  - initial state
  - its successors
  - one step away from the goal, but the uninformed algorithms won’t necessarily consider it next
Informed State-Space Search

- *Informed* search algorithms attempt to consider more promising states first.
- These algorithms associate a *priority* with each successor state that is generated.
  - base priority on an estimate of nearness to a goal state
  - when choosing the next state to consider, select the one with the highest priority
- Use a *priority queue* to store the yet-to-be-considered search nodes. Key operations:
  - insert: add an item to the priority queue, ordering it according to its priority
  - remove: remove the highest priority item
- How can we efficiently implement a priority queue?
  - use a type of binary tree known as a *heap*

Complete Binary Trees

- A binary tree of height $h$ is *complete* if:
  - levels 0 through $h - 1$ are fully occupied
  - there are no “gaps” to the left of a node in level $h$
- Complete:

- Not complete (* = missing node):
Representing a Complete Binary Tree

- A complete binary tree has a simple array representation.
- The nodes of the tree are stored in the array in the order in which they would be visited by a level-order traversal (i.e., top to bottom, left to right).

Examples:

- The root node is in `a[0]`
- Given the node in `a[i]`:
  - its left child is in `a[2*i + 1]`
  - its right child is in `a[2*i + 2]`
  - its parent is in `a[(i - 1)/2]` (using integer division)

Examples:

- the left child of the node in `a[1]` is in `a[2*1 + 1] = a[3]`
- the right child of the node in `a[3]` is in `a[2*3 + 2] = a[8]`
- the parent of the node in `a[4]` is in `a[(4 - 1)/2] = a[1]`
- the parent of the node in `a[7]` is in `a[(7 - 1)/2] = a[3]`
Heaps

- Heap: a complete binary tree in which each interior node is greater than or equal to its children

- Examples:

```
      28
     /   \
    16    20
   /     / \
  12    8   5
```

- The largest value is always at the root of the tree.
- The smallest value can be in any leaf node – there’s no guarantee about which one it will be.
- Strictly speaking, the heaps that we will use are max-at-top heaps. You can also define a min-at-top heap, in which every interior node is less than or equal to its children.

A Class for Items in a Heap

```java
public class HeapItem {
    private Object data;
    private double priority;

    public int compareTo(HeapItem other) {
        double diff = priority - other.priority;
        if (diff > 1e-6)
            return 1;
        else if (diff < -1e-6)
            return -1;
        else
            return 0;
    }
}
```

- HeapItem objects group together a data item and its priority.
A Class for Items in a Heap (cont.)

```java
public int compareTo(HeapItem other) {
    // error-checking goes here...
    double diff = priority - other.priority;
    if (diff > 1e-6)
        return 1;
    else if (diff < -1e-6)
        return -1;
    else
        return 0;
}
```

- The `compareTo` method returns:
  - -1 if the calling object has a lower priority than the other object
  - 1 if the calling object has a higher priority than the other object
  - 0 if they have the same priority

- numeric comparison comparison using `compareTo`
  - item1 < item2 item1.compareTo(item2) < 0
  - item1 > item2 item1.compareTo(item2) > 0
  - item1 == item2 item1.compareTo(item2) == 0

Heap Implementation (~cscie119/examples/heaps/Heap.java)

```java
public class Heap {
    private HeapItem[] contents;
    private int numItems;
    public Heap(int maxSize) {
        contents = new HeapItem[maxSize];
        numItems = 0;
    }
    ...
}
```

![Heap Diagram]

Note: we're just showing the priorities of the items, and we're showing them as integers.
Removing the Largest Item from a Heap

- Remove and return the item in the root node.
- In addition, we need to move the largest remaining item to the root, while maintaining a complete tree with each node $\geq$ children.

Algorithm:
1. Make a copy of the largest item.
2. Move the last item in the heap to the root (see diagram at right).
3. “Sift down” the new root item until it is $\geq$ its children (or it's a leaf).
4. Return the largest item.

Sifting Down an Item

- To sift down item $x$ (i.e., the item whose key is $x$):
  1. Compare $x$ with the larger of the item’s children, $y$.
  2. If $x < y$, swap $x$ and $y$ and repeat.

Other examples:
- Sift down the 10:
- Sift down the 7:
**siftDown() Method**

```java
private void siftDown(int i) {
    HeapItem toSift = contents[i];
    int parent = i;
    int child = 2 * parent + 1;
    while (child < numItems) {
        // If the right child is bigger, compare with it.
        if (child < numItems - 1 && contents[child].compareTo(contents[child + 1]) < 0)
            child = child + 1;
        if (toSift.compareTo(contents[child]) >= 0)
            break;  // we're done
        // Move child up and move down one level in the tree.
        contents[parent] = contents[child];
        parent = child;
        child = 2 * parent + 1;
    }
    contents[parent] = toSift;
}
```

- We don't actually swap items. We wait until the end to put the sifted item in place.

**remove() Method**

```java
public HeapItem remove() {
    HeapItem toRemove = contents[0];
    contents[0] = contents[numItems - 1];
    numItems--;
    siftDown(0);
    return toRemove;
}
```

numItems: 6
toRemove: 28

numItems: 5
toRemove: 28

numItems: 5
toRemove: 28
Inserting an Item in a Heap

- Algorithm:
  1. put the item in the next available slot (grow array if needed)
  2. "sift up" the new item until it is <= its parent (or it becomes the root item)

- Example: insert 35
  put it in place:
  
<table>
<thead>
<tr>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
</tr>
<tr>
<td>12</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>8</td>
</tr>
</tbody>
</table>

  sift it up:
  
<table>
<thead>
<tr>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
</tr>
<tr>
<td>12</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>35</td>
</tr>
</tbody>
</table>

insert() Method

```java
public void insert(HeapItem item) {
    if (numItems == contents.length) {
        // code to grow the array goes here...
    }
    contents[numItems] = item;
    siftUp(numItems);
    numItems++;
}
```

<table>
<thead>
<tr>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
</tr>
<tr>
<td>12</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>35</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
</tr>
<tr>
<td>12</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>8</td>
</tr>
</tbody>
</table>
Converting an Arbitrary Array to a Heap

- Algorithm to convert an array with n items to a heap:
  1. start with the parent of the last element:
     \[ \text{contents}[i], \text{where } i = ((n - 1) - 1)/2 = (n - 2)/2 \]
  2. sift down \text{contents}[i] and all elements to its left

- Example:
  
  Last element's parent = \text{contents}[(7 - 2)/2] = \text{contents}[2]. Sift it down:

  

Converting an Array to a Heap (cont.)

- Next, sift down \text{contents}[1]:

  

- Finally, sift down \text{contents}[0]:

  

Creating a Heap from an Array

```java
public class Heap {
    private HeapItem[] contents;
    private int numItems;

    public Heap(HeapItem[] arr) {
        // Note that we don't copy the array!
        contents = arr;
        numItems = arr.length;
        makeHeap();
    }

    private void makeHeap() {
        int last = contents.length - 1;
        int parentOfLast = (last - 1)/2;
        for (int i = parentOfLast; i >= 0; i--)
            siftDown(i);
    }
}
```

Time Complexity of a Heap

- A heap containing n items has a height <= log₂n.
- Thus, removal and insertion are both O(log n).
  - remove: go down at most log₂n levels when sifting down from the root, and do a constant number of operations per level
  - insert: go up at most log₂n levels when sifting up to the root, and do a constant number of operations per level
- This means we can use a heap for a O(log n)-time priority queue.
- Time complexity of creating a heap from an array?
Using a Heap to Sort an Array

• Recall selection sort: it repeatedly finds the smallest remaining element and swaps it into place:

<table>
<thead>
<tr>
<th>5</th>
<th>16</th>
<th>8</th>
<th>14</th>
<th>20</th>
<th>1</th>
<th>26</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>16</td>
<td>8</td>
<td>14</td>
<td>20</td>
<td>5</td>
<td>26</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>8</td>
<td>14</td>
<td>20</td>
<td>16</td>
<td>26</td>
</tr>
</tbody>
</table>

• It isn’t efficient \(O(n^2)\), because it performs a linear scan to find the smallest remaining element \(O(n)\) steps per scan.

• Heapsort is a sorting algorithm that repeatedly finds the largest remaining element and puts it in place.

• It is efficient \(O(n\log n)\), because it turns the array into a heap, which means that it can find and remove the largest remaining element in \(O(\log n)\) steps.

Heapsort (~cscie119/examples/heaps/HeapSort.java)

```java
public class HeapSort {
    public static void heapSort(HeapItem[] arr) {
        // Turn the array into a max-at-top heap.
        // The heap object will hold a reference to the
        // original array, with the elements rearranged.
        Heap heap = new Heap(arr);
        int endUnsorted = arr.length - 1;
        while (endUnsorted > 0) {
            // Get the largest remaining element and put it
            // where it belongs -- at the end of the portion
            // of the array that is still unsorted.
            HeapItem largestRemaining = heap.remove();
            arr[endUnsorted] = largestRemaining;
            endUnsorted--;
        }
    }
}
```
Heapsort Example

• Sort the following array:

\[
\begin{array}{ccccccc}
13 & 6 & 45 & 10 & 3 & 22 & 5
\end{array}
\]

• Here’s the corresponding complete tree:

```
13
  6
  45
10  3  22  5
```

• Begin by converting it to a heap:

```
13
  6
  45
10  3  22  5
```

no change, because 
45 >= its children

```
13
  6
  45
10  3  22  5
```

sift down

```
13
  6
  45
10  3  22  5
```

6

sift down

```
13
  6
  45
10  3  22  5
```

13

sift down

```
13
  45
10  6  3  22  5
```

Heapsort Example (cont.)

• Here’s the heap in both tree and array forms:

```
45
  10
  22
  6  3
```

```
0  1  2  3  4  5  6
45 10 22 6 3 13 5
```

endUnsorted: 6

toRemove: 45

heapSort()

```
45
  10
  22
  6  3  5
```

endUnsorted: 5

largestRemaining: 45

remove() copies 45; moves 5 to root

```
5
  10
  22
  6  3  13
```

remove() sifts down 5; returns 45

```
5
  10
  22
  6  3  13
```

heapSort() puts 45 in place; decrements endUnsorted

```
10
  13
  6  5
```

```endUnsorted: 6```

endUnsorted: 6

largestRemaining: 45

remove() copies 45; moves 5 to root

```
6
  10
  13
  5
```

remove() sifts down 5; returns 45

```
6
  10
  13
  5
```

heapSort() puts 45 in place; decrements endUnsorted

```
10
  13
  6  5
```

```
10
  13
  6  5
```

endUnsorted: 5

largestRemaining: 45
Heapsort Example (cont.)

copy 22; move 5 to root

sift down 5; return 22

put 22 in place

put 22 in place

toRemove: 22

endUnsorted: 5
largestRemaining: 22

Heapsort Example (cont.)

copy 13; move 3 to root

sift down 3; return 13

put 13 in place

put 13 in place

toRemove: 13

endUnsorted: 4
largestRemaining: 13

Heapsort Example (cont.)

copy 10; move 3 to root

sift down 3; return 10

put 10 in place

put 10 in place

toRemove: 10

endUnsorted: 3
largestRemaining: 10

Heapsort Example (cont.)

copy 6; move 5 to root

sift down 5; return 6

put 6 in place

put 6 in place

toRemove: 6

endUnsorted: 2
largestRemaining: 6
Heapsort Example (cont.)

To Remove: 5

End Unsorted: 1

Largest Remaining: 5

How Does Heapsort Compare?

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Best Case</th>
<th>Avg Case</th>
<th>Worst Case</th>
<th>Extra Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection Sort</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Insertion Sort</td>
<td>$O(n)$</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Shell Sort</td>
<td>$O(n \log n)$</td>
<td>$O(n^{1.5})$</td>
<td>$O(n^{1.5})$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Bubble Sort</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Quicksort</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n^2)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Mergesort</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Heapsort</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>

- Heapsort matches mergesort for the best worst-case time complexity, but it has better space complexity.
- Insertion sort is still best for arrays that are almost sorted.
  - Heapsort will scramble an almost sorted array before sorting it
  - Quicksort is still typically fastest in the average case.
State-Space Search: Estimating the Remaining Cost

- As mentioned earlier, informed search algorithms associate a priority with each successor state that is generated.
- The priority is based in some way on the remaining cost – i.e., the cost of getting from the state to the closest goal state.
  - for the 8 puzzle, remaining cost = # of steps to closest goal
- For most problems, we can’t determine the exact remaining cost.
  - if we could, we wouldn’t need to search!
- Instead, we estimate the remaining cost using a **heuristic function** \( h(x) \) that takes a state \( x \) and computes a cost estimate for it.
  - heuristic = rule of thumb
- To find optimal solutions, we need an **admissible** heuristic – one that never overestimates the remaining cost.

Heuristic Function for the Eight Puzzle

- Manhattan distance = horizontal distance + vertical distance
  - example: For the board at right, the Manhattan distance of the 3 tile from its position in the goal state
    \[ = 1 \text{ column} + 1 \text{ row} = 2 \]
  - Use \( h(x) = \text{sum of the Manhattan distances of the tiles in } x \) from their positions in the goal state
  - for our example:
    \[
    h(x) = 1 + 1 + 2 + 2 + 1 + 0 + 1 + 1 = 9
    \]
- This heuristic is admissible because each of the operators (move blank up, move blank down, etc.) moves a single tile a distance of 1, so it will take at least \( h(x) \) steps to reach the goal.
Greedy Search

- Priority of state \( x \), \( p(x) = -1 \cdot h(x) \)
  - mult. by -1 so states closer to the goal have higher priorities

\[
\begin{array}{c|c|c|c}
\text{h = 3} & \text{h = 3} & \text{h = 1} & \text{h = 3} \\
\text{p = -3} & \text{p = -3} & \text{p = -1} & \text{p = -3} \\
\end{array}
\]

- Greedy search would consider the highlighted successor before the other successors, because it has the highest priority.
- Greedy search is:
  - incomplete: it may not find a solution
    - it could end up going down an infinite path
  - not optimal: the solution it finds may not have the lowest cost
    - it fails to consider the cost of getting to the current state

A* Search

- Priority of state \( x \), \( p(x) = -1 \cdot (h(x) + g(x)) \)
  - where \( g(x) = \text{the cost of getting from the initial state to } x \)

\[
\begin{array}{c|c|c|c}
\text{h = 3} & \text{h = 3} & \text{h = 1} & \text{h = 3} \\
p = -(3 + 1) & p = -(3 + 1) & p = -(1 + 1) & p = -(3 + 1) \\
\end{array}
\]

- Incorporating \( g(x) \) allows A* to find an optimal solution – one with the minimal total cost.
Characteristics of A*

- It is complete and optimal.
  - provided that \( h(x) \) is admissible, and that \( g(x) \) increases or stays the same as the depth increases

- Time and space complexity are still typically exponential in the solution depth, \( d \) – i.e., the complexity is \( O(b^d) \) for some value \( b \).

- However, A* typically visits far fewer states than other optimal state-space search algorithms.

<table>
<thead>
<tr>
<th>solution depth</th>
<th>iterative deepening</th>
<th>A* w/ Manhattan dist. heuristic</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>112</td>
<td>12</td>
</tr>
<tr>
<td>8</td>
<td>6384</td>
<td>25</td>
</tr>
<tr>
<td>12</td>
<td>364404</td>
<td>73</td>
</tr>
<tr>
<td>16</td>
<td>did not complete</td>
<td>211</td>
</tr>
<tr>
<td>20</td>
<td>did not complete</td>
<td>676</td>
</tr>
</tbody>
</table>

- Memory usage can be a problem, but it's possible to address it.


The numbers shown are the average number of search nodes visited in 100 randomly generated problems for each solution depth.

The searches do not appear to have excluded previously seen states.

Implementing Informed Search (~cscie119/examples/search)

- Add new subclasses of the abstract `Searcher` class.

- For example:
  ```java
  public class GreedySearcher extends Searcher {
    private Heap nodePQueue;

    public void addNode(SearchNode node) {
      nodePQueue.insert(
        new HeapItem(node, -1 * node.getCostToGoal()));
    }
  }
  ...
  ```