1.)
In uniform-cost search, the formula to calculate the cost to move between two nodes is $f_{OS}(n) = g(n)$.
In A* search, the formula to calculate the cost to move between two nodes is $f_{A*}(n) = g(n) + h(n)$.
If for all nodes n $h(n) = 0$, then $f_{A*}(n) = g(n) + h(n) = g(n) + 0 = g(n) = f_{OS}(n)$.
Therefore uniform-cost search is a special case of A* where the function h(n) always equals 0.

2.)
Current node, g(n), h(n), f(n)
(frontier node, g(n), h(n), f(n))

F, 0, 227, 227
(S, 99, 173, 272)
(B, 211, 274, 485)
-> expand Sibiu

S, 99, 173, 272
(R, 179, 145, 324)
(A, 239, 202, 441)
(B, 211, 274, 485)
(O, 250, 273, 523)
-> expand R

R, 179, 145, 324
(C, 325, 96, 421)
(A, 239, 202, 441)
(P, 276, 177, 453)
(B, 211, 274, 485)
(O, 250, 273, 523)
-> expand C

C, 325, 96, 421
(A, 239, 202, 441)
(D, 445, 0, 445) goal state
(P, 276, 177, 453) <- (P, 463, 177, 640) calculated, but not chosen since it is worse
(B, 211, 274, 485)
(O, 250, 273, 523)

3.)
a.
Each iteration, the algorithm will expand one node and choose the one best successor from that. This is the same as what best-first search does.

b.

This is similar to uniform-cost search. Every time the algorithm expands a node, all of its successors are remembered, and the best ever successor is expanded next.

c.
This is similar to first-choice hill climbing. Successors of the current node are randomly selected, and then evaluated according to the value function. This node is then only selected if it has a higher value than the current node. Since T is always equal to 0, it has no effect on the choosing of successors since $e^{-\Delta E/T}$ is always 0.

d.
If T = ∞, then $e^{-\Delta E/T} = e^{0} = 1$. In each iteration of the loop, if the randomly generated successor is better than its parent, it will always be chosen. However, since the probability that a worse successor node, $e^{-\Delta E/T}$, is always 1, then every worse successor will also be chosen. This means the algorithm will randomly choose successors until the goal state is reached. This is similar to the random walk algorithm.

4.)
Heuristic h1 was simple to implement. For each position 1-8 in the puzzle, 1 is added to the estimated solution cost if the tile at index i does not equal i.

To implement h2, I calculate the manhattan distance between a tile and its position in the solved puzzle. For the x coordinate of a tile I use the index modulo 3 since the x coordinate of every third tile is 0. To calculate the y coordinate of a tile I use the index divided by 3 since the y coordinate increments every third tile. Since the result of integer division is truncated, this works.

Both functions take a node as an argument instead of a state because it is implemented that way in the book. Looking forward to the A* implementation though, this will save me time because I can simply pass each node to the heuristic functions when calculating the cost of an edge, rather than having to first get the state of the node.

I use a wrapper function heuristic(Node node, String heuristic) to simplify the command code. I implemented the heuristic function in the Node class because Node objects need to be able to access the function in order to be compared to each other.

I test the heuristic functions on a solved and very scrambled puzzle, and I also test their estimates compared to BFS and A* solves. Both heuristics correctly estimated a cost of 0 to achieve the solved puzzle. When the heuristic estimates were compared to an actual BFS solve on a barely scrambled puzzle, they both estimated the actual cost of the solution. The heuristics' cost estimate on a puzzle that was scrambled 100 times revealed more information about their accuracy, with h1 estimating 7 and h2 estimating 14. Since h1 only checks if each tile is out of position, it has a maximum estimated cost of 8, meaning h1 gets less accurate the more a puzzle is scrambled. Heuristic h2 provides a much better estimate of the solution's cost by taking into account tile distance, although it is optimistic, so it also becomes less accurate with more scrambled puzzles, as can be seen with the final A* solve I do.

5.)
I used Java's Set API to implement repeated state checking. I also refactored so that my code matches the pseudocode. A large part of this was adding the method expand(EightPuzzle puzzle) to my Node class that produces a list of all the children of the Node object, so that they can be iterated over and added to the queue. I also added another command, move <direction> -q, that allows classes outside of EightPuzzle to attempt movements without flooding the command line. In the future, I would like to make a queue-based sort object factory that allows me to create sorting algorithm objects, so I don't have mostly duplicated code between my different algorithms.

6.)
I once again copied the code I have for BFS and DFS for this algorithm, but I used a priority queue data structure for the frontier. For the priority queue, I implemented comparable in my Node class by using the heuristic assigned to a node to compare it to another. The priority queue also considers move direction if two nodes have equal heuristic values. Adding the heuristic function to my Node class made me realize that I might want to move other functions to the class such as move. I will think about this and make changes for the next assignment.

7.)
As mentioned previously, I added tests for the heuristic functions and A*. Tests are explained in the comments. There is not a test in the file to prove repeated state checking works since working examples fill the command line. However if you want to check, you can run DFS with state 1 2 3 4 5 6 7 8 0 and maxnodes=1000000.