# Priority Queues and Heaps

Thursday, November 21, 2024 4:20 PM

- Implementing a priority queue using a heap
  - Standard approach is to use an away
  - Starts at position 1 not 0
  - Root node is always in array
  - Insert
    - Add value so heap still has the order and shape properties
    - Add new value to end (adding to rightmost leaf)
    - Compare new value to parent value
    - If parent is smaller, swap
    - loop-continue swap until order property holds or we get to the root
  - Remove
    - Largest value is always of the root
    - Always returns the root value
    - How do you replace the root node so heap still has order and shape
    - Replace the value in the root with value at the end of the array
    - Work down through tree, swapping to restore the order property

- Heapq algorithm
  - Heap queue
  - Also known as a priority queue algorithm
  - Smallest element is always root heap[0]
  - Pop method returns smallest item not largest (min heap)
- warmup 1
  - Priority represents path length - shorter paths get higher priority

  paths get higher priority)
  - Fed as tuples - priority, value
  - Same children (nodes) - the rode that joined the queue earlier should always be popped off before the other node
- Breadth first search (BFS)
  - Returns path of nodes from a given start node to a given end node
  - List
  - If start and goal are some return []
  - graph [node]
  - Graph.neighbors(node)
  - Traverse through one level of children nodes, then grandchildren nodes
  - Use an evaluation function for each node - estimate of desirability
  - Spreads out in waves of uniform depth
- uniform-cost search or dijkstra algorithm
  - best-first search
  - Evaluation function is the cost of the path from the root to the current node
  - Ai calls this uniform cost
  - Search spreads out in waves of uniform path-cost
  - Can be implemented using best-first-search with a path-cost as the evaluation function
  - Always looking to expand the least expensive route
  - Algorithms test for goals only wen it expands a node, not when it generates a node
  - uniform-cost search can explore large trees of actions with low cost that is at least as low as the cost of any other node in the frontier
  - Never gets caught down a single infinite path
  - evaluation function f(n)
- Informed (heuristic) search strategy
  - Domain specific hints about specific goals
  - More efficient than uninformed search

- Heuristic h(n).
- H(n) - estimated cost of the cheapest path from the state at node n to a goal state

- A* search
    - F(n) = g(n) + h(n)
    - G(n) is the path cost from the initial state node n
    - H(n) is the estimated cost of the shortest path from n to a goal state
    - F(n) is the estimated cost of the best path that continues from n to a goal
    - Admissible heuristic is one that never overestimates the cost to reach a goal - optimistic
- Bidirectional
    - Simultaneously searches forward from the initial state and backwards from the goal states - hoping that the two searches will meet
    - Need to keep track of two frontiers and two tables of reached states
    - Need to be able to reason backwards
    - need to be able to travel forwards and backwards - link of successors
    - Bidirectional best-first search
    - Two separate frontiers - the node to be expanded next is always one with a minimum value of the evaluation function across either frontier
    - Two frontiers and two tables of reached states
    - When a path in one frontier reaches a state that was also reached in the other half of the search, the two paths are joined to form solution
    - First solution is tubes
    - Reached data structure supports a query asking whether a given state is a member

    and the frontier data structure (a pq) dose not sole check for collision using reached
  - Can handle multiple goal state by loading the node for each goal state info the backward frontier and backwards reached table
  - Cost of the optimal path is C - then no node with cost > c/2 will be expanded
  - When the evaluation function is the path cost - we know that the first solution found will be an optimal solution but with different evaluation functions that is not necessarily true
  - Therefore we keep track of the best solution found so far and will continue to search for the best solution possible
  - bi-directional stopping condition
    - UCS
    - Search from the start and the goal towards each other until frontiers interact
    - Frontier interaction does not guarantee the best path
    - Exhaustively search all possible paths that could be the shortest paths in the vicinity of the frontier interaction
    - When an element is in the explored set we found the shortest path to the element
    - When performing searches from either side, we can wait until the explored sets intersect
      - When Hey do, we would have found the shortest path to the intersecting point from both the start and goal nodes
      - Combining the two should give overall shortest path
      - Expand the frontiers until the

- Expand the frontiers until the path cost on one side-is more than the upper limit or the frontiers became empty
- or: stopping condition is a node being expanded is found in the explored set of the opposing search
  - Set of values to explore: intersection of the explored set of the current search with ice union of the frontier and explored set of the opposite search

- Policy on expanding frontiers
  - Alternate expanding backwards and forewords frontiers - a chance that searches might cross - one search might over shoot a goal before the searches cross
  - Or expand the frontier with the minimum cost element on top - explores more nodes than alternating