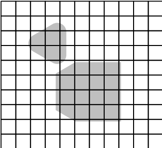
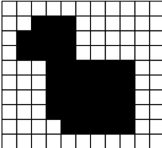
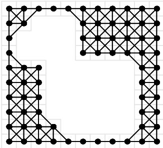
Discrete State Path Planning

Nathan Sprague

Discrete-State Path Planning







Generic Planning Algorithm

```
def search(problem):
    """ Generic graph search algorithm.
    Args:
        problem: a problem instance that provides three methods:
              problem.start() -       returns the start state
              problem.goal()  -       returns the goal state
              problem.successors(s) - returns the states that are
                                      adjacent to s
    Returns:
        True if there exists a sequence of states leading from
        problem.start() to problem.goal(), or False if no such
        path exists
    """

    frontier = Collection()  # Queue for BFS, Stack for DFS
    closed = set()
    frontier.add(problem.start())

    while not frontier.is_empty():
        cur_state = frontier.pop()
        closed.add(cur_state)

        if cur_state == problem.goal():  # Success!
            return True

        else:
            # Add the neighbors of the selected state to the frontier...
            for next_state in problem.successors(cur_state):
                if (next_state not in closed and
                        next_state not in frontier):
                    frontier.add(next_state)

    return False   # No path was found!
```

Generic Planning - Returning the Path

```
class Node:
    """The Node class stores backward references from each state
       to the state that preceded it.
    """
    def __init__(self, state, parent_node):
        self.state = state
        self.parent = parent_node


def search(problem):
    """ Returns: A sequence of states leading from problem.start() to
        problem.goal(), or None if no path exists
    """
    frontier = Collection()
    closed = set()

    frontier.add(Node(problem.start(), None))

    while not frontier.is_empty():
        cur_node = frontier.pop()
        cur_state = cur_node.state
        closed.add(cur_state)

        if cur_state == problem.goal():
            return construct_path(cur_node) # path ending at this node

        else:
            for next_state in problem.successors(cur_state):
                next_node = Node(next_state, cur_node)
                if (next_state not in closed and
                        next_node not in frontier): # <- Why??
                    frontier.add(next_node)

    return None  # Search ended with no path found.
```

Dijkstra's Algorithm - Minimum Cost First

```
def dijkstra(problem):
    frontier = PriorityQueue()  # <-- Priority Queue for Frontier!
    closed = set()

    start_node = Node(problem.start(), None, 0.0)
    frontier.add(start_node, 0.0) # <-- Priority for the start node is 0.

    while not frontier.is_empty():
        cur_node = frontier.pop() # <-- Lowest priority/smallest cost from start
        cur_state = cur_node.state
        closed.add(cur_state)

        if cur_state == problem.goal():
            return construct_path(cur_node)

        else:
            for next_state in problem.successors(cur_state):
                cost = problem.cost(cur_state, next_state)
                next_node = Node(next_state, cur_node, cost)

                if next_state not in closed:
                    # Priority (path_cost) is total cost to reach this state...
                    frontier.add(next_node, next_node.path_cost)
    return None
```

A* Search

- Key idea – reduce the number of expansions by taking advantage of a *heuristic function*.
- heuristic function – $h(s)$ maps from states to an estimate of the cost to reach the goal from that state.
- A* is exactly the same as Dijkstra's algorithm, except the frontier is ordered by:
  - $f(s) = c(s) + h(s)$
    - $c(s)$ – Actual cost to reach s
    - $h(s)$ – Estimated cost from s to goal.
    - $f(s)$ – Estimated total cost of the shortest path through s

Heuristic Functions

- A* is guaranteed to find the optimal path as long as the heuristic function is:
  - *admissible* – Never overestimates the cost to goal
  - *consistent* – $h(s) \leq h(s') + cost(s, s')$ for all states s and s'
- We want a heuristic function that is:
  - Efficient to compute
  - As close as possible to the true cost