EE 355 Unit 14
A-Star Algorithm &
Heaps/Priority Queues

Mark Redekopp
A* Search Algorithm

ALGORITHM HIGHLIGHT
Search Methods

• Many systems require searching for goal states
  – Path Planning
    • Roomba Vacuum
    • Mapquest/Google Maps
    • Games!!
  – Optimization Problems
    • Find the optimal solution to a problem with many constraints
Search Applied to 8-Tile Game

• 8-Tile Puzzle
  – 3x3 grid with one blank space
  – With a series of moves, get the tiles in sequential order
  – Goal state:

HW6 Goal State

Goal State for these slides
Search Methods

- **Brute-Force Search**: When you don’t know where the answer is, just search all possibilities until you find it.

- **Heuristic Search**: A heuristic is a “rule of thumb”. An example is in a chess game, to decide which move to make, count the values of the pieces left for your opponent. Use that value to “score” the possible moves you can make.
  
  - Heuristics are not perfect measures, they are quick computations to give an approximation (e.g. may not take into account “delayed gratification” or “setting up an opponent”)


Brute Force Search

- Brute Force Search Tree
  - Generate all possible moves
  - Explore each move despite its proximity to the goal node
Heuristics

- Heuristics are “scores” of how close a state is to the goal (usually, lower = better)
- These scores must be easy to compute (i.e. simpler than solving the problem)
- Heuristics can usually be developed by simplifying the constraints on a problem
- Heuristics for 8-tile puzzle
  - # of tiles out of place
    - Simplified problem: If we could just pick a tile up and put it in its correct place
  - Total x-, y- distance of each tile from its correct location (Manhattan distance)
    - Simplified problem if tiles could stack on top of each other / hop over each other

<table>
<thead>
<tr>
<th>1</th>
<th>8</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td></td>
</tr>
</tbody>
</table>

# of Tiles out of Place = 3

<table>
<thead>
<tr>
<th>1</th>
<th>8</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td></td>
</tr>
</tbody>
</table>

Total x-/y- distance = 6
Heuristic Search

• Heuristic Search Tree
  – Use total x-/y-distance (Manhattan distance) heuristic
  – Explore the lowest scored states
Caution About Heuristics

• Heuristics are just estimates and thus could be wrong
• Sometimes pursuing lowest heuristic score leads to a less-than optimal solution or even no solution
• Solution
  – Take # of moves from start (depth) into account
A-star Algorithm

• Use a new metric to decide which state to explore/expand

• Define
  – $h$ = heuristic score (same as always)
  – $g$ = number of moves from start it took to get to current state
  – $f = g + h$

• As we explore states and their successors, assign each state its f-score and always explore the state with lowest f-score.
A-Star Algorithm

- Maintain 2 lists
  - Open list = Nodes to be explored (chosen from)
  - Closed list = Nodes already explored (already chosen)

- Pseudocode

```python
open_list.push(Start State)
while(open_list is not empty)
    1. s ← remove min. f-value state from open_list
       (if tie in f-values, select one w/ larger g-value)
    2. Add s to closed list
    3a. if s = goal node then trace path back to start; STOP!
    3b. Generate successors/neighbors of s, compute their f values, and add them to open_list if they are not in the closed_list (so we don’t re-explore), or if they are already in the open list, update them if
```
Data Structures & Ordering

• We’ve seen some abstract data structures
  – Queue (first-in, first-out access order)
    • Deque works nicely

• There are others...
  – Stack (last-in, first-out access order)
    • Vector push_back / pop_back
  – Trees

• A new one...priority queues (heaps)
  – Order of access depends on value
  – Items arrive in some arbitrary order
  – When removing an item, we always want the minimum or maximum valued item
  – To implement efficiently use heap data structure
Path-Planning w/ A* Algorithm

• Find optimal path from S to G using A*
  – Use heuristic of Manhattan (x-/y-) distance

open_list.push(Start State)
while(open_list is not empty)
  1. s ← remove min. f-value state from open_list (if tie in f-values, select one w/larger g-value)
  2. Add s to closed list
  3a. if s = goal node then
      trace path back to start; STOP!
  3b. else
      Generate successors/neighbors of s, compute their f-values, and add them to open_list if they are not in the closed_list (so we don’t re-explore), or if they are already in the open_list, update them if they have a smaller f value

**If implementing this for a programming assignment, please see the slide at the end about alternate closed-list implementation**
Path-Planning w/ A* Algorithm

• Find optimal path from S to G using A*
  – Use heuristic of Manhattan (x-/y-) distance

```
open_list.push(Start State)
while(open_list is not empty)
  1. s ← remove min. f-value state from open_list (if tie in f-values, select one w/ larger g-value)
  2. Add s to closed list
  3a. if s = goal node then trace path back to start; STOP!
  3b. else
     Generate successors/neighbors of s, compute their f-values, and add them to open_list if they are not in the closed_list (so we don’t re-explore), or if they are already in the open list, update them if they have a smaller f value

**If implementing this for a programming assignment, please see the slide at the end about alternate closed-list implementation**
```
Path-Planning w/ A* Algorithm

• Find optimal path from S to G using A*
  -- Use heuristic of Manhattan (x-/y-) distance

open_list.push(Start State)
while(open_list is not empty)
  1. s ← remove min. f-value state from
     open_list (if tie in f-values, select one w/ larger g-value)
  2. Add s to closed list
  3a. if s = goal node then
      trace path back to start; STOP!
  3b. else
      Generate successors/neighbors of s,
      compute their f-values, and add them to
      open_list if they are not in the closed_list
      (so we don’t re-explore), or if they are
      already in the open list, update them if
      they have a smaller f value
Path-Planning w/ A* Algorithm

- Find optimal path from S to G using A*
  - Use heuristic of Manhattan (x-/y-) distance

```python
open_list.push(Start State)
while(open_list is not empty)
  1. s ← remove min. f-value state from open_list (if tie in f-values, select one w/ larger g-value)
  2. Add s to closed list
  3a. if s = goal node then
      trace path back to start; STOP!
  3b. else
      Generate successors/neighbors of s, compute their f-values, and add them to open_list if they are not in the closed_list (so we don’t re-explore), or if they are already in the open list, update them if they have a smaller f value
```
Path-Planning w/ A* Algorithm

• Find optimal path from S to G using A*
  – Use heuristic of Manhattan (x-/y-) distance

open_list.push(Start State)
while(open_list is not empty)
  1. s ← remove min. f-value state from
     open_list (if tie in f-values, select one w/ larger g-value)
  2. Add s to closed list
  3a. if s = goal node then
      trace path back to start; STOP!
  3b. else
      Generate successors/neighbors of s, compute their f-values, and add them to open_list if they are not in the closed_list (so we don’t re-explore), or if they are already in the open list, update them if they have a smaller f value
Path-Planning w/ A* Algorithm

- Find optimal path from S to G using A*
  - Use heuristic of Manhattan (x-/y-) distance

```python
open_list.push(Start State)
while(open_list is not empty)
    1. s ← remove min. f-value state from open_list (if tie in f-values, select one w/ larger g-value)
    2. Add s to closed list
    3a. if s = goal node then
        trace path back to start; STOP!
    3b. else
        Generate successors/neighbors of s, compute their f-values, and add them to open_list if they are not in the closed_list (so we don’t re-explore), or if they are already in the open list, update them if they have a smaller f value
```
Path-Planning w/ A* Algorithm

• Find optimal path from S to G using A*
  – Use heuristic of Manhattan (x-/y-) distance

```
open_list.push(Start State)
while(open_list is not empty)
  1. s ← remove min. f-value state from open_list (if tie in f-values, select one w/ larger g-value)
  2. Add s to closed list
  3a. if s = goal node then
      trace path back to start; STOP!
  3b. else
      Generate successors/neighbors of s, compute their f-values, and add them to open_list if they are not in the closed_list (so we don’t re-explore), or if they are already in the open list, update them if they have a smaller f value
```
Path-Planning w/ A* Algorithm

• Find optimal path from S to G using A*
  – Use heuristic of Manhattan (x-/y-) distance

open_list.push(Start State)
while(open_list is not empty)
  1. s ← remove min. f-value state from
     open_list (if tie in f-values, select one w/
     larger g-value)
  2. Add s to closed list
  3a. if s = goal node then
     trace path back to start; STOP!
  3b. else
     Generate successors/neighbors of s,
     compute their f-values, and add them to
     open_list if they are not in the closed_list
     (so we don’t re-explore), or if they are
     already in the open list, update them if
     they have a smaller f value
Path-Planning w/ A* Algorithm

- Find optimal path from S to G using A*
  - Use heuristic of Manhattan (x-/y-) distance

open_list.push(Start State)
while(open_list is not empty)
  1. $s \leftarrow$ remove min. f-value state from open_list (if tie in f-values, select one w/ larger g-value)
  2. Add s to closed list
  3a. if s = goal node then
       trace path back to start; STOP!
  3b. else
       Generate successors/neighbors of s, compute their f-values, and add them to open_list if they are not in the closed_list (so we don’t re-explore), or if they are already in the open list, update them if they have a smaller f value.
# Path-Planning w/ A* Algorithm

- Find optimal path from S to G using A*
  – Use heuristic of Manhattan \((x-/y-)\) distance

open_list.push(Start State)
while(open_list is not empty)

1. \(s \leftarrow \) remove min. \(f\)-value state from open_list (if tie in \(f\)-values, select one w/ larger \(g\)-value)
2. Add \(s\) to closed list
3a. if \(s\) = goal node then 
   - trace path back to start; STOP!
3b. else
   - Generate successors/neighbors of \(s\), compute their \(f\)-values, and add them to open_list if they are not in the closed_list (so we don’t re-explore), or if they are already in the open list, update them if they have a smaller \(f\) value
Path-Planning w/ A* Algorithm

- Find optimal path from S to G using A*
  - Use heuristic of Manhattan (x-/y-) distance

open_list.push(Start State)
while(open_list is not empty)
  1. s ← remove min. f-value state from open_list (if tie in f-values, select one w/ larger g-value)
  2. Add s to closed list
  3a. if s = goal node then
      trace path back to start; STOP!
  3b. else
      Generate successors/neighbors of s, compute their f-values, and add them to open_list if they are not in the closed_list (so we don’t re-explore), or if they are already in the open list, update them if they have a smaller f value
Path-Planning w/ A* Algorithm

• Find optimal path from S to G using A*
  – Use heuristic of Manhattan (x-/y-) distance

**Algorithm Steps**

1. `open_list.push(Start State)`
2. While `open_list` is not empty:
   1. `s ← remove min. f-value state from open_list (if tie in f-values, select one w/ larger g-value)`
   2. Add `s` to closed list
   3a. If `s` = goal node then
      - Trace path back to start; STOP!
   3b. Else
      - Generate successors/neighbors of `s`, compute their f-values, and add them to open_list if they are not in the closed_list (so we don’t re-explore), or if they are already in the open list, update them if they have a smaller f value
Path-Planning w/ A* Algorithm

• Find optimal path from S to G using A*
  – Use heuristic of Manhattan (x-/y-) distance

open_list.push(Start State)
while(open_list is not empty)
  1. s ← remove min. f-value state from
     open_list (if tie in f-values, select one w/
     larger g-value)
  2. Add s to closed list
  3a. if s = goal node then
      trace path back to start; STOP!
  3b. else
      Generate successors/neighbors of s,
      compute their f-values, and add them to
      open_list if they are not in the closed_list
      (so we don’t re-explore), or if they are
      already in the open list, update them if
      they have a smaller f value
Path-Planning w/ A* Algorithm

- Find optimal path from S to G using A*
  - Use heuristic of Manhattan (x-/y-) distance

```plaintext
open_list.push(Start State)
while(open_list is not empty)

1. s ← remove min. f-value state from open_list (if tie in f-values, select one w/larger g-value)
2. Add s to closed list
3a. if s = goal node then
    trace path back to start; STOP!
3b. else
    Generate successors/neighbors of s,
    compute their f-values, and add them to open_list if they are not in the closed_list
    (so we don’t re-explore), or if they are already in the open list, update them if they have a smaller f value
```
A* and BFS

• BFS explores all nodes at a shorter distance from the start (i.e. g value)
A* and BFS

• BFS explores all nodes at a shorter distance from the start (i.e. g value)
A* and BFS

- BFS is A* using just the g value to choose which item to select and expand
Implementation Note

- When the distance to a node/state/successor (i.e. g value) is uniform, we can greedily add a state to the closed-list at the same time as we add it to the open-list.

```
open_list.push(Start State)
while(open_list is not empty)
    1. s ← remove min. f-value state from open_list
       (if tie in f-values, select one w/ larger g-value)
    2. Add s to closed list
    3a. if s = goal node then trace path back to start; STOP!
    3b. Generate successors/neighbors of s, compute their f
         values, and add them to open_list if they are
         not in the closed_list (so we don’t re-explore), or
         if they are already in the open list, update them if
         they have a smaller f value

open_list.push(Start State)
Closed_list.push(Start State)
while(open_list is not empty)
    1. s ← remove min. f-value state from open_list
       (if tie in f-values, select one w/ larger g-value)
    3a. if s = goal node then trace path back to start; STOP!
    3b. Generate successors/neighbors of s, compute their f
         values, and add them to open_list and closed_list
         if they are not in the closed_list
```

Non-uniform g-values

Uniform g-values

The first occurrence of a board has to be on the shortest path to the solution
Heaps

DATA STRUCTURE HIGHLIGHT
Heap Data Structure

- Can think of heap as a full binary tree with the property that every parent is less-than (if min-heap) or greater-than (if max-heap) both children
  - But no ordering property between children
- Minimum/Maximum value is always the top element

Min-Heap Diagram:
Heap Operations

• Push: Add a new item to the heap and modify heap as necessary
• Pop: Remove min/max item and modify heap as necessary
• Top: Returns min/max
• To create a heap from an unordered array/vector takes $O(n \times \log_2 n)$ time while push/pop takes $O(\log_2 n)$
Push Heap

- Add item to first free location at bottom of tree
- Recursively promote it up until a parent is less than the current item
Pop Heap

• Takes last (greatest) node puts it in the top location and then recursively swaps it for the smallest child until it is in its right place
Practice

Push(11)

Push(23)

Pop()
Array/Vector Storage for Heap

- Binary tree that is full (i.e. only the lowest-level contains empty locations and items added left to right) can be modeled as an array (let’s say it starts at index 1) where:
  - Parent\( (i) = \frac{i}{2} \)
  - Left_child\( (p) = 2 \times p \)
  - Right_child\( (p) = 2 \times p + 1 \)

parent(5) = 5/2 = 2
Left_child(5) = 2*5 = 10
Right_child(5) = 2*5 + 1 = 11
STL Priority Queue

- Implements a max-heap by default
- Operations:
  - push(new_item)
  - pop(): removes but does not return top item
  - top() return top item (item at back/end of the container)
  - size()
  - empty()

```cpp
#include <iostream>
#include <queue>
using namespace std;

int main ()
{
    priority_queue<int> mypq;
    mypq.push(30);
    mypq.push(100);
    mypq.push(25);
    mypq.push(40);
    cout << "Popping out elements...";
    while (!mypq.empty()) {
        cout << " " << mypq.top();
        mypq.pop();
    }
    cout << endl;
    return 0;
}
```

Code here will print
100 40 30 25
STL Priority Queue Template

- Template that allows type of element, container class, and comparison operation for ordering to be provided
- First template parameter should be type of element stored
- Second template parameter should be vector\(<\text{type\_of\_elem}\>\)
- Third template parameters should be comparison function object/class that will define the order from first to last in the container

```cpp
// priority_queue::push/pop
#include <iostream>
#include <queue>

using namespace std;

int main ()
{
    priority_queue<int, vector<int>, greater<int>> mypq;
    mypq.push(30); mypq.push(100); mypq.push(25);
    cout << "Popping out elements...";
    while (!mypq.empty()) {
        cout << " \n" << mypq.top();
        mypq.pop();
    }
}
```

'greater' will yield a min-heap
'less' will yield a max-heap

Code here will print 25, 30, 100
STL Priority Queue Template

- For user defined classes, must implement operator<() for max-heap or operator>() for min-heap
- Code here will pop in order:
  - Jane
  - Charlie
  - Bill

```cpp
#include <iostream>
#include <queue>
#include <string>
using namespace std;

class Item {
public:
  int score;
  string name;
  Item(int s, string n) { score = s; name = n;}
  bool operator>=(const Item &lhs, const Item &rhs) const
  { if(lhs.score > rhs.score) return true;
    else return false;
  }
};

int main ()
{
  priority_queue<Item, vector<Item>, greater<Item> > mypq;
  Item i1(25,"Bill"); mypq.push(i1);
  Item i2(5,"Jane"); mypq.push(i2);
  Item i3(10,"Charlie"); mypq.push(i3);
  cout<< "Popping out elements...";
  while (!mypq.empty()) {
    cout<< " " << mypq.top().name;
    mypq.pop();
  }
}
```