Path Planning

CMPS 146, Fall 2013

Josh McCoy
Reading

- Path Planning: 197-255
- Dijkstra and graphs early.
Pathfinding

Reach Lovers' Peak

Visited: 1184, Time: 0.013 ms, Max Queue: 145, Poly Searches: 0/145 Search Data Req: 1346 kB (45 x 851 x 36) 35/45 PathFindingMeshes: 9276 vertices (x 12 B), 4596 polygons (x 6 B), 19764 edges (x 6 B) (total: 564 kB) (s: 309)

Max mesh values: 1258/1800 vertices, 546/850 polygons, 2608/3800 edges, 1460/4600 grid Reserved size: 45 x 58 KB = 2630 kB
Why search?

- Consider this problem – given a set of integers, we want to output them in sorted order, smallest number first
  - Say we want to sort $\{5, 2, 27, 12\}$ – the answer is $\{2, 5, 12, 27\}$

- First of all, do we have a quick way of checking an answer? Yes.

```plaintext
currentNumber := first element of SA (SA is a list containing the answer)
while (not at the end of SA) do
    nextNumber := the next element in SA
    if (currentNumber > nextNumber) signal no
    else currentNumber := nextNumber
signal yes
```
So here’s a way to sort

Is this a good way to sort numbers?
Intriguing facts about computation

- There exist problems which can be solved in a relatively small number of computations, in fact a polynomial number (P), like sort.

- There exist (many many interesting) problems which can not be solved in a polynomial number of computations – these are non-deterministic polynomial complete (NP-complete) problems.

- The NP-complete problems are the ones that can only be solved by search.
Formalizing search

- A search problem has five components: $Q$, $S$, $G$, $\text{succs}$, $\text{cost}$

- $Q$ is a finite set of states.
- $S \subseteq Q$ is a non-empty set of start states.
- $G \subseteq Q$ is a non-empty set of goal states.
- $\text{succs} : Q \rightarrow P(Q)$ is a function which takes a state as input and returns a set of states as output. $\text{succs}(s)$ means “the set of states you can reach from $s$ in one step”.
- $\text{cost} : Q, Q \rightarrow \text{Positive Number}$ is a function which takes two states, $s$ and $s'$, as input. It returns the one-step cost of traveling from $s$ to $s'$. The cost function is only defined when $s'$ is a successor state of $s$. 
Representing the search space

• Search space represented as a graph of state transitions

• Two typical representations
  • A graph of states, where arrows represent \texttt{succ} and nodes represent states – a state appears only once in the graph
  • A tree of states, where a path in the tree represents a sequence of search decisions – the same state can appear many times in the tree

Here’s our sort problem written out as a search graph
An abstract search problem

\[ Q = \{ \text{START, } a, b, c, d, e, f, h, p, q, r, \text{ GOAL} \} \]
\[ S = \{ S \} \]
\[ G = \{ G \} \]
\[ \text{succs}(b) = \{ a \} \]
\[ \text{succs}(e) = \{ h, r \} \]
\[ \text{succs}(a) = \text{NULL} \ldots \text{ etc.} \]
\[ \text{cost}(s, s') = 1 \text{ for all transitions} \]
Some example search problems
Aside: Why do we have to search?

- “We may regard the present state of the universe as the effect of its past and the cause of its future. An intellect which at any given moment knew all of the forces that animate nature and the mutual positions of the beings that compose it, if this intellect were vast enough to submit the data to analysis, could condense into a single formula the movement of the greatest bodies of the universe and that of the lightest atom; for such an intellect nothing could be uncertain and the future just like the past would be present before its eyes.”

-Marquis Pierre Simon de Laplace
Breadth-first search

- Label all states that are reachable from S in 1 step but aren’t reachable in less than 1 step.

- Then label all states that are reachable from S in 2 steps but aren’t reachable in less than 2 steps.

- Then label all states that are reachable from S in 3 steps but aren’t reachable in less than 3 steps.

- Etc… until Goal state reached.

- Try this out for our abstract search problem
Depth-first search

- An alternative to BFS. Always expand from the most recently-expanded node, if it has any untried successors. Else backup to the previous node on the current path.

- We use a data structure we’ll call a Path to represent the path from the START to the current state. E.G. Path P = <START, d, e, r >

- Along with each node on the path, we must remember which successors we still have available to expand. E.G. P = <START (expand=e,p) , d (expand = b,c) , e (expand = h), r (expand = f) >
DFS Algorithm

Let \( P = \langle \text{START} \ (\text{expand} = \text{succs}(	ext{START})) \rangle \)

While \((P \ \text{not empty and top}(P) \ \text{not a goal})\)
  
  if expand of top\((P)\) is empty then
    remove top\((P)\) ("pop the stack")
  else
    let \( s \) be a member of expand of top\((P)\)
    remove \( s \) from expand of top\((P)\)
    make a new item on the top of path \( P \): \( s \ (\text{expand} = \text{succs}(s)) \)

If \( P \) is empty then
  return FAILURE

Else
  return the path consisting of states in \( P \)
Uniform cost (Dijkstra) search

- A conceptually simple BFS approach when there are costs on transitions

- It uses a priority queue
  - PQ = Set of states that have been expanded or are awaiting expansion
  - Priority of state $s = g(s) =$ cost of getting to $s$ using path implied by backpointers.

- Main loop
  - Pop least cost state from PQ (the open list)
  - Add the successors
  - Stop when the goal is popped from the PQ (the open list)
DIY: Uniform Cost

Searching for a path from A to G
Uniform cost example

Closed List: A
Open List: B, C, D
Uniform cost example

A
- cost-so-far: 0
- connection: none

B
- Cost-so-far: 1.3
  - Connection: I
  - Cost: 1.3

C
- Cost-so-far: 1.6
  - Connection: II
  - Cost: 1.6

D
- Cost-so-far: 3.3
  - Connection: III

Closed List: A
Open List: B, C, D
Uniform cost example

Closed List: A
Open List: B, C, D
Uniform cost example

- A (cost-so-far: 0, connection: none)
  - Connection: I (Cost: 1.3)
  - Connection: II (Cost: 1.6)
  - Connection: III (Cost: 3.3)
- B (Cost-so-far: 1.3, Connection: I)
- C (Cost-so-far: 1.6, Connection: II)
- D (Cost-so-far: 3.3, Connection: III)

Closed List: A
Open List: B, C, D
Uniform cost example

Closed List: A
Open List: B, C, D
Uniform cost example

- **A**
  - Cost-so-far: 0
  - Connection: none
- **B**
  - Cost-so-far: 1.3
  - Connection: I
- **C**
  - Cost-so-far: 1.6
  - Connection: II
- **D**
  - Cost-so-far: 3.3
  - Connection: III

Closed List: A
Open List: B, C, D
Data structure for the open list

- We need a data structure for the open list that makes the following operations inexpensive:
  - Adding an entry to the list
  - Removing an entry from the list
  - Finding the smallest element
  - Finding an entry corresponding to a particular node

- Let’s analyze how well a linked list works
Priority heap

http://nova.umuc.edu/~jarc/idsv/lesson2.html
Heuristic search

- Suppose in addition to the standard search specification we also have a *heuristic*.

- A *heuristic function maps a state onto an estimate of the cost to the goal from that state*.

- What’s an example heuristic for path planning?

- Denote the heuristic by a function \( h(s) \) from states to a cost value.
Best first “greedy” search

insert-PriQueue(PQ, START, h(START))
while (PQ is not empty and PQ does not contain a goal state)
    (s, h) := pop-least(PQ)
    foreach s’ in succs(s)
        if s’ is not already in PQ and s’ never previously been visited
            insert-PriQueue(PQ, s’, h(s’))

- Greedy ignores the actual costs

- An improvement to this algorithm becomes A*!
A* search

- Uniform-cost (Dijkstra): on expanding node $n$, take each successor $n'$ and place it on priority queue with priority $g(n')$ (cost of getting to $n'$)

- Best-first greedy: on expanding node $n$, take each successor $n'$ and place it on priority queue with priority $h(n')$

- A*: When you expand node $n$, place each successor $n'$ on priority queue with priority $f(n') = g(n') + h(n')$
When should A* terminate?

Should it terminate as soon as we generate the goal state?
Correct A* termination rule

A* terminates when a goal state is popped from the priority queue (open list)
What if A* revisits a state that was already expanded and discovers a shorter path?

In this example, a state that was expanded gets re-expanded.
What if A* revisits a state that was already expanded and discovers a shorter path?

In this example, a state that was expanded gets re-expanded.
A* algorithm

- Priority queues Open and Closed begin empty. Both queues contain records of the form (state, f, g, backpointer).
- Put S into Open with priority \( f(s) = g(s) + h(s) \)
- Is Open empty?
  - Yes? No solution.
  - No. Remove node with lowest \( f(n) \). Call it \( n \).
  - If \( n \) is a goal, stop and report success (for pathfinding, return the path).
  - Otherwise, expand \( n \). For each \( n' \) in successors(\( n \))
    - Let \( f' = g(n') + h(n') = g(n) + \text{cost}(n, n') + h(n') \)
    - If \( n' \) not seen before or \( n' \) previously expanded with \( f(n') > f' \) or \( n' \) currently in Open with \( f(n') > f' \)
    - Then place or promote \( n' \) on Open with priority \( f' \)
    - Else ignore \( n' \)
    - Place record for \( n \) in Closed
Is A* guaranteed to find the optimal path?

- Nope. Not if the heuristic overestimates the cost.
Admissible heuristics

- \( h^*(n) = \) the true minimal cost to goal from \( n \)
- A heuristic is admissible if \( h(n) \leq h^*(n) \) for all states \( n \)
- An admissible heuristic is guaranteed to never overestimate the cost to the goal
- An admissible heuristic is optimistic

- What’s a common admissible heuristic for pathplanning?
Euclidian distance

- Guaranteed to be underestimating
- Can provide fast pathplanning on outdoor levels with few constraints
- May create too much fill on complex indoor levels
Cluster heuristic

- Can be good for indoor levels
- Pre-compute accurate heuristic between clusters
- Use pre-computed values for points in two clusters, Euclidean (or something else simple) within clusters