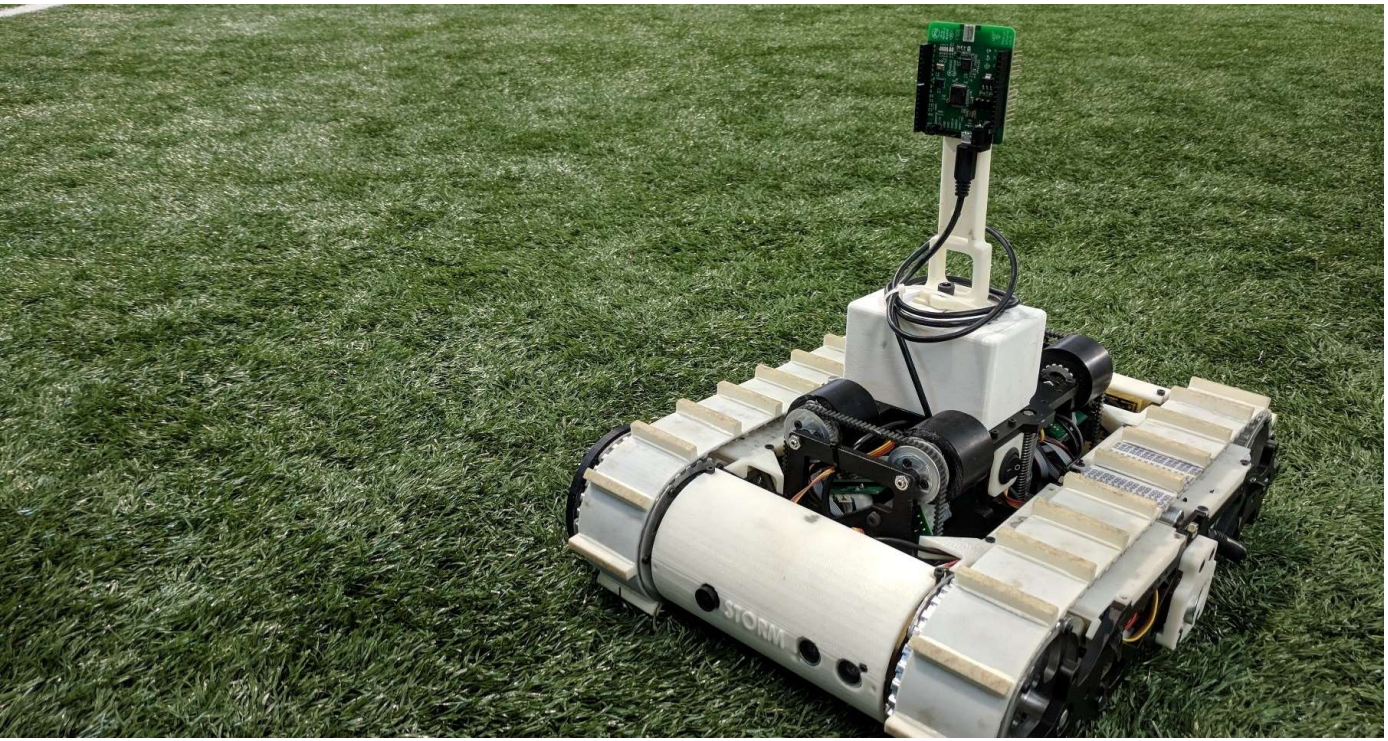


ED 5215
INTRODUCTION TO MOTION PLANNING

Instructor: Bijo Sebastian

MOTIVATION

In Module 1 we looked at cases where we have:

- Point robot that can move in any direction
- Known environment with stationary obstacles
- Perfect sensing
- Perfect control

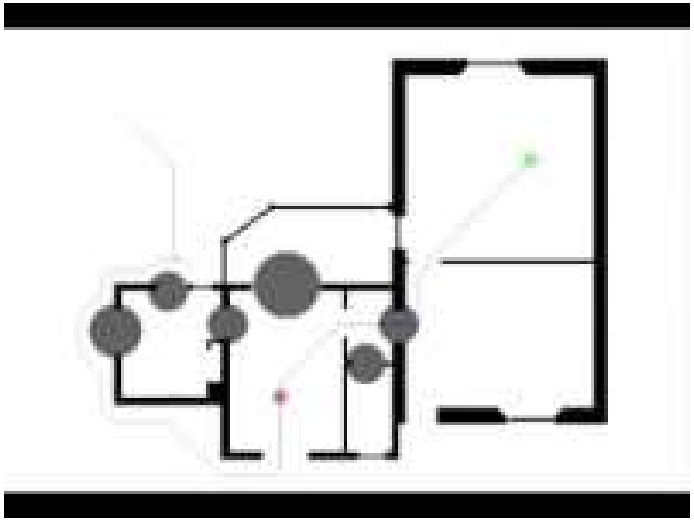
We looked into the following topics to handle above scenarios:

- *Bug Algorithms*
- *Artificial potential fields*
- *Graph search: Depth-first, Breadth-first, Uniform Cost Search, Dijkstra, and A**

MOTIVATION

In Module 2 we will look into cases where we have:

- Point robot that can move in any direction
- Unknown/changing environment and/or moving obstacles
- Perfect sensing
- Perfect control



We will look into the following topics to handle above scenarios:

- *Weighted A**
- *Anytime A*, LPA*, D* Lite*
- *Dynamic programming*

https://www.youtube.com/watch?v=X5a149nSE9s

EFFICIENCY

Efficiency of Search

An algorithm is efficient if it finds the solution with least possible expansions (for all inputs)

For instance:

- Uniform Cost Search and A* are equally efficient if no heuristic is available
- A* is more efficient than Uniform Cost Search with any consistent and admissible heuristic

OPTIMALITY VS EFFICIENCY

In real world situations with unknown/changing environment and/or moving obstacles, finding a “good-enough” solution as fast as possible becomes a higher priority

Question: Can we sacrifice optimality to improve efficiency of search?

OPTIMALITY VS EFFICIENCY

In real world situations with unknown/changing environment and/or moving obstacles, finding a "good-enough" solution as fast as possible becomes a higher priority

Question: Can we sacrifice optimality to improve efficiency of search?

→ Weighted A*

Strategy:

Expand the node in fringe with lowest $g + \epsilon h$, where ϵ >= 1

WEIGHTED A*

Strategy:

Expand the node in fringe with lowest $g + \epsilon h$, where $\epsilon >= 1$

In practice, faster than A*

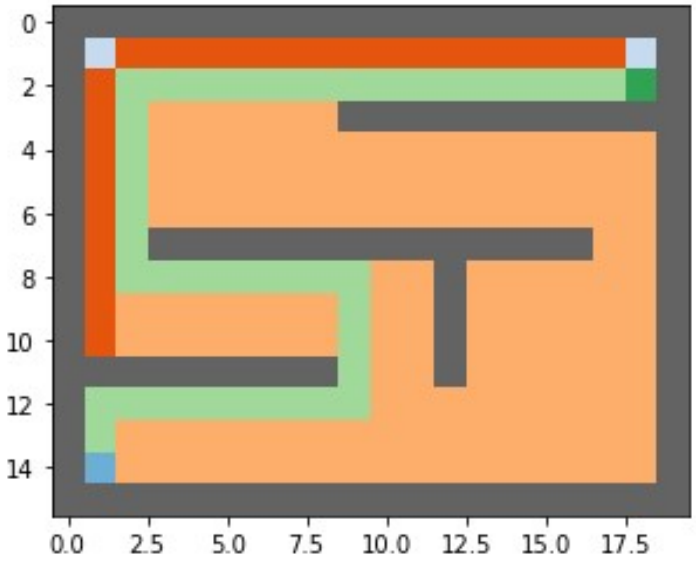
Higher we set ϵ to be, the faster we find the solution

Maze 4

$\epsilon = 1$

Number of nodes expanded 190

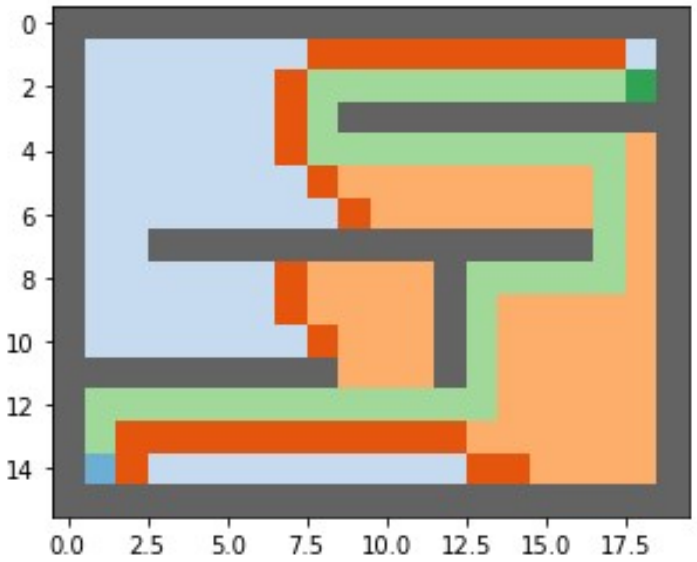
Found a path of 43 moves



$\epsilon = 5$

Number of nodes expanded 119

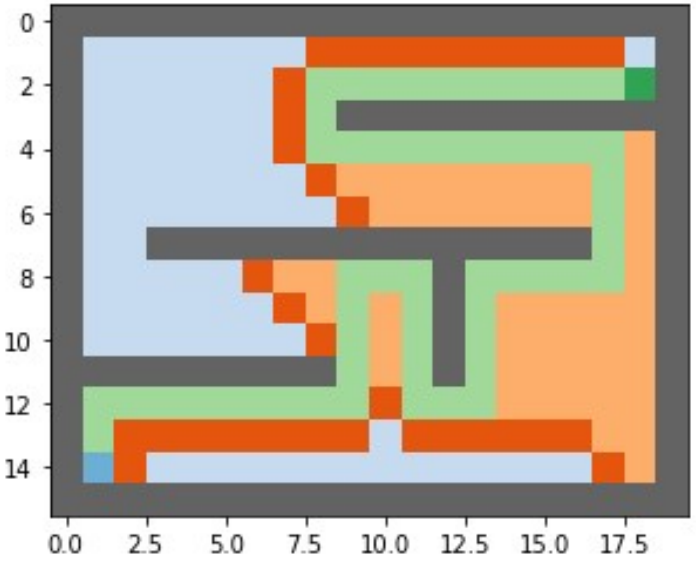
Found a path of 47 moves



$\epsilon = 10$

Number of nodes expanded 106

Found a path of 55 moves



ANYTIME ALGORITHM

Anytime Algorithm: Class of planning algorithms that find a feasible solution as fast as possible and then improves the solution iteratively

Advantages:

- If we stop the algorithm at any point in time, we should be able to return a feasible solution
- More time we give to the algorithm, the closer to optimal the returned solution should be

Question: How to make an anytime algorithm using weighted A*?

ANYTIME A*

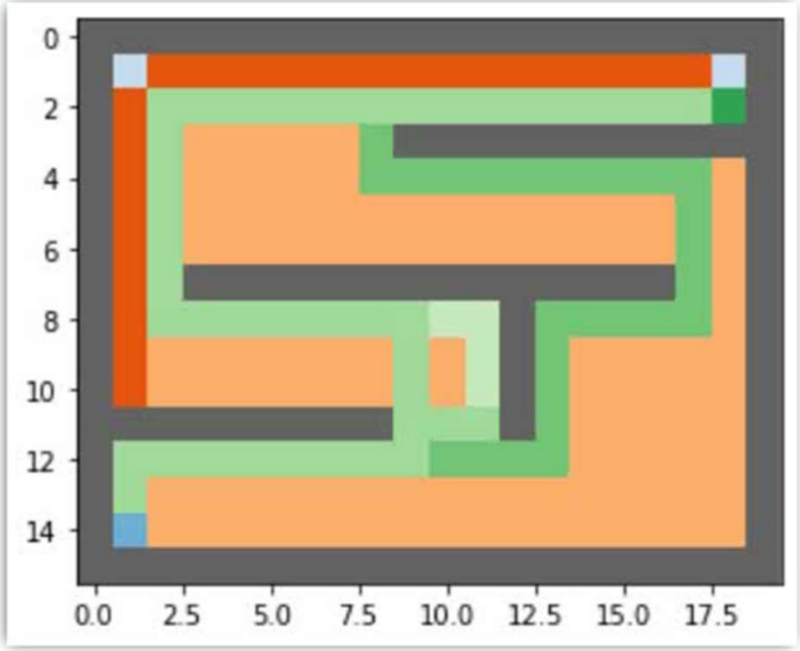
Question: How to make an anytime algorithm using weighted A*?

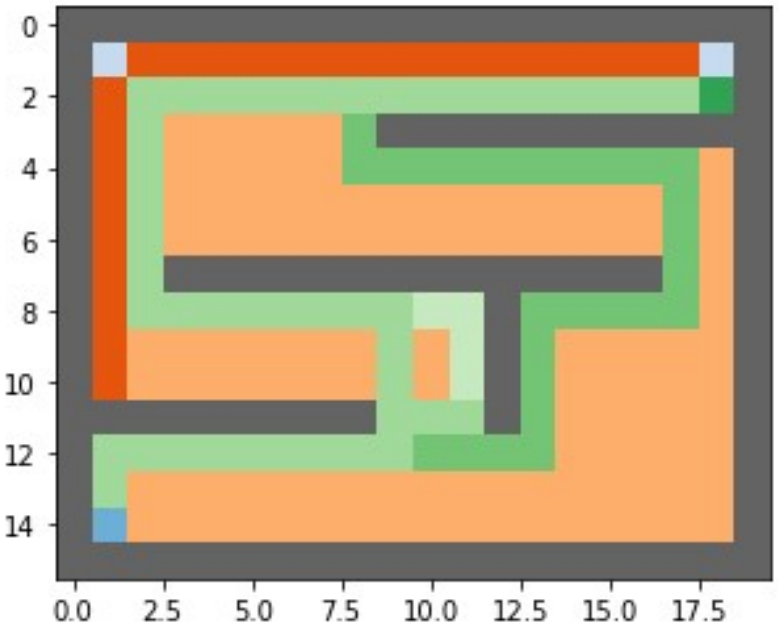
Algorithm:

set ϵ = very high number (choice depends on application)
while $\epsilon > 1$:
find path using weighted A*
set $\epsilon = \epsilon/2$ (or any similar update rule that over time converges to $\epsilon = 1$)

→ Anytime A*

Maze 4





ANYTIME A*

Disadvantage of Anytime A*:

At every iteration, we recompute the path from scratch

We could speed up the algorithm if we could find a way to re-use computations done from previous iterations, assuming the environment did not change

LIFELONG PLANNING

In real world situations, it is rare that the environment remains static for a planning problem

Often there are changes:

- Effect of a door closing for an indoor robot
- Effect of change in weather on outdoor robot

We need to continuously plan to account for changes in the environment

→ Lifelong Planning algorithms

Solution (naïve) : Replan from scratch, every time something changes

LIFELONG PLANNING

Disadvantage of naïve Lifelong Planning:

Often the changes to the environment are minimal or locally contained, they do not affect the entire map

In such situations it is wasteful to recompute the path from scratch

We could be more efficient by finding a way to re-use computations done from previous iterations, for parts of the map/environment that did not change

→ Lifelong Planning A*

LIFELONG PLANNING A* (LPA*)

Under the assumption that the robot has not started moving, LPA* allows for recomputing path from start to goal taking into account the environmental changes and reusing as much information as possible from previous iterations

In other words, LPA* is used in scenarios where Start and Goal nodes remain the same, but edge costs change over time

DIJKSTRA (RECAP)

Pseudo code:

function **Dijkstra**(problem) returns an optimal path

create an empty closed set
create open set (fringe) and add all nodes of the graph to it
set the cost of every node in open list, $g(x_i) = \infty$, except start node
set start node cost, $g(x_0) = 0$
while open set is not empty do:
sort open set and pop the node with least cost, x_j
add popped node to closed set
expand the popped node by performing all possible actions
for each successor (child node), x_i:
compute new cost = current (popped) node cost + cost of action to get to successor
update the cost $g(x_i)$ of the successor node if new cost < existing cost in the open set
except:
if successor already exists in closed set → do not add it to open set

LPA*

Note that $g(x_i)$ is an estimate of the optimal cost to get x_i from the start node (cost-to-come)

We start with $g(x_i) = \infty$ and once the open list becomes empty, we would have estimated optimal cost for each node

Question:

- If x_i is in closed set, then $g(x_i)$ __ Optimal cost
- If x_i is in open set, then $g(x_i)$ __ Optimal cost

LPA*

Note that $g(x_i)$ is an estimate of the optimal cost to get x_i from the start node (cost-to-come)

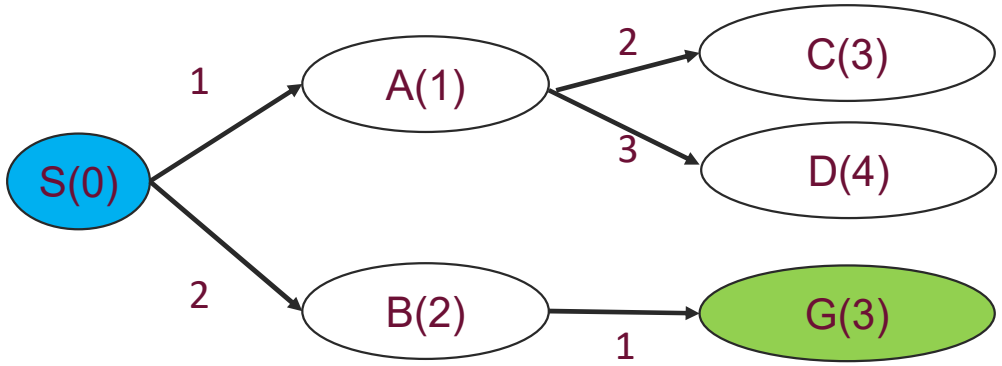
We start with $g(x_i) = \infty$ and once the open list becomes empty, we would have estimated optimal cost for each node

Question:

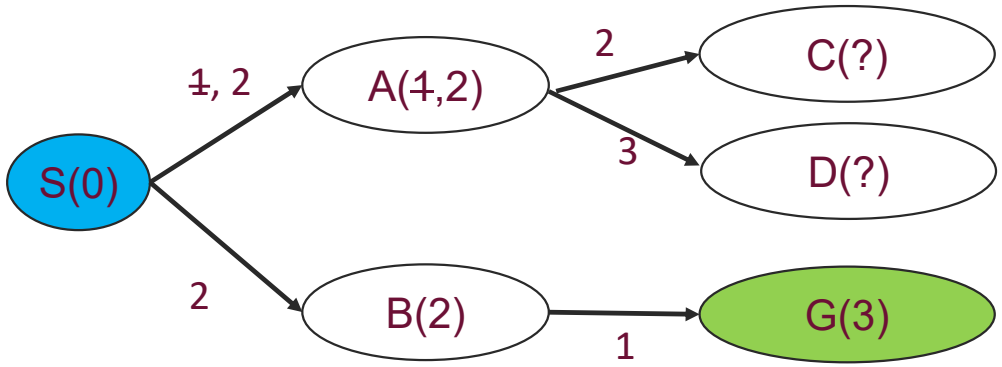
- If x_i is in closed set, then $g(x_i)$ = Optimal cost
- If x_i is in open set, then $g(x_i)$ >= Optimal cost

LPA*

Consider a problem where we have already solved for the shortest path from start to goal



What happened to the optimal cost of a node when an edge cost changes?



LPA*

When an edge cost changes:

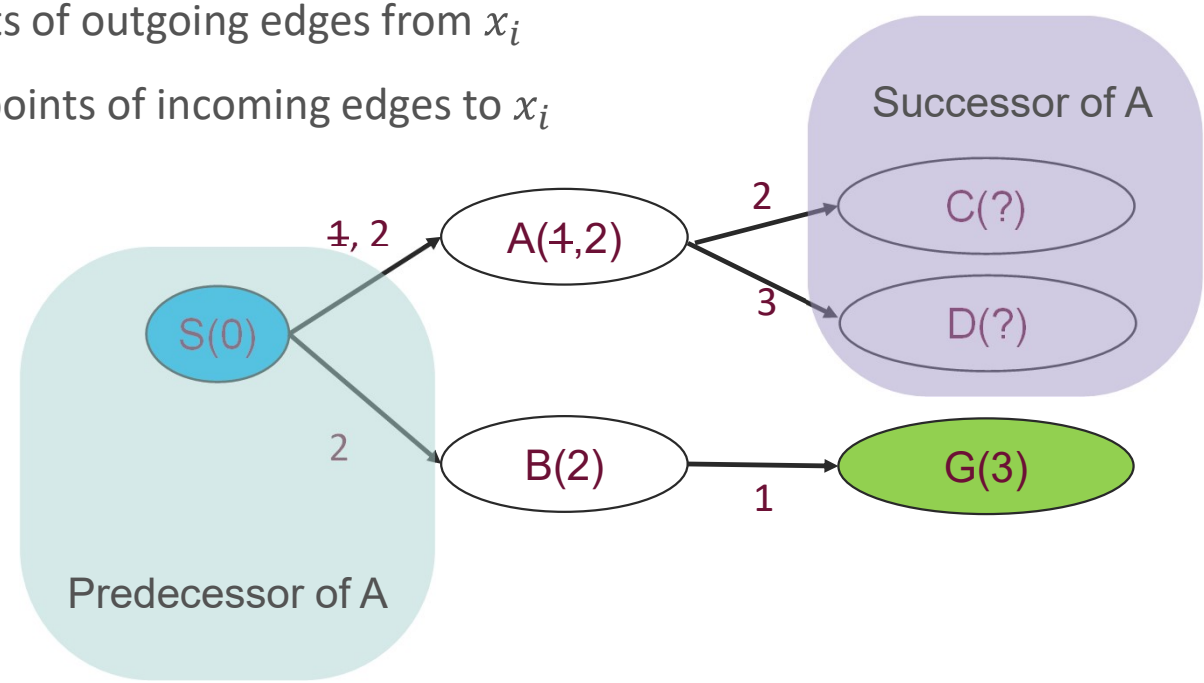
- The optimal cost of a successor node may change
- This means the optimal cost of its successor's node may change …….. (Cascade effect)
- All of this may or may not affect the shortest path from start to goal
- Assuming we are solving single query planning problems (shortest path from one start node to one goal node) and not from all nodes to a single goal node, there is no need to compute the new optimal cost for every node

Question: How do we do this efficiently ?

LPA*

To keep track of changes in node cost due to edge cost change, we define the following:

- Successors of x_i → Endpoints of outgoing edges from x_i
- Predecessors of x_i → Start points of incoming edges to x_i



- $rhs(x_i) = \min\{C(x_i, x_j) + g(x_j)\}$, for all x_j in predecessor of x_i

For instance, in above case $g(A) = 1, rhs(A) = 2$

LPA*

If $g(x_i) = rhs(x_i)$ we will call x_i as locally consistent

Whenever an incoming edge cost changes for x_i , it is possible that $g(x_i) \neq rhs(x_i)$

→ x_i becomes locally inconsistent

- x_i is locally over consistent if $g(x_i) > rhs(x_i)$
- x_i is locally under consistent if $g(x_i) < rhs(x_i)$

Both cases could lead to changes in the optimal path from start to goal

LPA*

An efficient way to evaluate the changes to the optimal path from start to goal based on changes to one or more edge costs in the graph is:

- Evaluate rhs for all immediate successors of the edge/s with new cost
- Add all locally inconsistent nodes back into the open set
- This will trigger a cascading effect but keep the re-computation contained to only the nodes that matter in the shortest path from start to goal

Question: What should be the strategy to pop newly added nodes from open set (fringe)?

Should we use $g(x_i)$ or $rhs(x_i)$?

LPA*

Question: What should be the strategy to pop newly added nodes from open set (fringe)?

Should we use $g(x_i)$ or $rhs(x_i)$?

We define a new term, key:

$$k(x_i) = \min\{ g(x_i), rhs(x_i) \}$$

The strategy for LPA* is to pop the node with minimum key value from the open set/ fringe

LPA*

initialize()

forever
  x_i = vertex in the open list with the lowest key
  while $k(x_i) < k(x_G)$ OR x_G is inconsistent

    if x_i is overconsistent
      $g(x_i) = rhs(x_i)$ // *make x_i consistent*
      remove x_i from open list
    else
      $g(x_i) = \infty$; // *make x_i overconsistent*
      update-vertex (x_i)

    for each x_j in $Succ(x_i)$
      update-vertex (x_j)

  wait for changes in the graph

  for each edge (x_i, x_j) with changed cost
    update edge cost $C(x_i, x_j)$
    update-vertex (x_j)

end forever

initialize ()
  for each x_i in Graph
    $g(x_i) = rhs(x_i) = \infty$;
  $rhs(x_O) = 0$
  add x_O to open list

update-vertex (x_j)
  if x_j is not x_O
    update $rhs(x_j)$ & $k(x_j)$
  if x_j is locally inconsistent
    add x_j to the open list

key $k(x_i)$
  // assuming zero heuristic
  $k(x_i) = \min \{ g(x_i), rhs(x_i) \}$

LPA* (FORMAL)

procedure CalculateKey(s)
{01} return $[\min(g(s), rhs(s)) + h(s, s_{goal}); \min(g(s), rhs(s))]$;

procedure Initialize()
{02} $U = \emptyset$;
{03} for all $s \in S$ $rhs(s) = g(s) = \infty$;
{04} $rhs(s_{start}) = 0$;
{05} U.Insert(s_{start}, CalculateKey(s_{start}));

procedure UpdateVertex(u)
{06} if $(u \neq s_{start})$ $rhs(u) = \min_{s' \in Pred(u)}(g(s') + c(s', u))$;
{07} if $(u \in U)$ U.Remove(u);
{08} if $(g(u) \neq rhs(u))$ U.Insert$(u,$ CalculateKey$(u))$;

procedure ComputeShortestPath()
{09} while (U.TopKey() $<$ CalculateKey(s_{goal}) OR $rhs(s_{goal}) \neq g(s_{goal})$)
{10} $u =$ U.Pop();
{11} if $(g(u) > rhs(u))$
{12} $g(u) = rhs(u)$;
{13} for all $s \in Succ(u)$ UpdateVertex(s);
{14} else
{15} $g(u) = \infty$;
{16} for all $s \in Succ(u) \cup \{u\}$ UpdateVertex(s);

procedure Main()
{17} Initialize();
{18} forever
{19} ComputeShortestPath();
{20} Wait for changes in edge costs;
{21} for all directed edges (u, v) with changed edge costs
{22} Update the edge cost $c(u, v)$;
{23} UpdateVertex(v);

Taken from:

Lifelong Planning A*: Sven Koenig, Maxim Likhachev, and David Furcy

(link)

LIFELONG PLANNING

In real world situations, it is quite rare that the changes in environment can be observed by a robot remaining stationary at start point

Often, we start with an empty map and fill it in as the robot explores the world while moving towards the goal

→ Edge costs change as the robot starts moving towards the goal, and senses the surroundings

We need lifelong planning algorithms that can be used in scenarios where goal node/s remain fixed, but the start node and edge costs change over time as the robot moves and gains more information about the world

→ D* and D* Lite

D* & D* LITE

D* and D* Lite allow for computing the path from current robot position to goal while reusing as much information as possible from previous iterations

See: D* Lite: Sven Koenig, Maxim Likhachev (link)

In fact, there exists many variants:

- Anytime D*
- Field D*
- Theta* …

It is recommended that you implement LPA* and then modify the implementation to get to D* Lite

DYNAMIC PROGRAMMING

Consider a stochastic robot tasked to go from A to B

It is possible that the robot ends up in C for reasons as simple as imperfect sensing, controls, actuation or a combination of these

In such situations, you need an algorithm that allows you to compute not only the shortest path from start to goal, but the shortest path from all vertices to goal

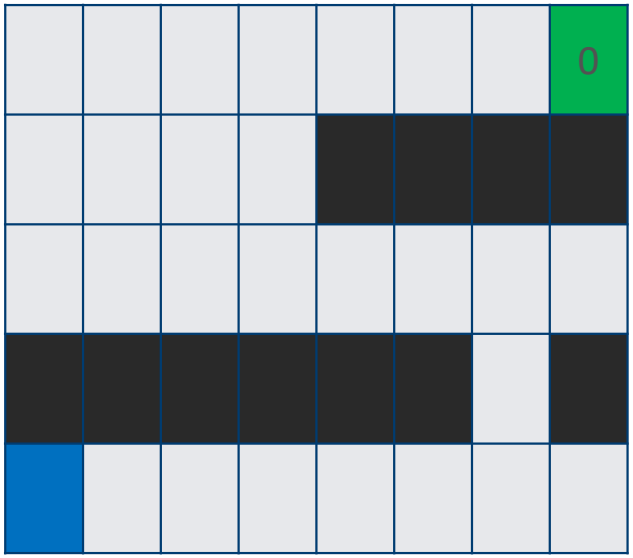
→ Dynamic Programming (DP)

DYNAMIC PROGRAMMING

Compute the optimum value ("cost-to-go" to the goal) for each cell

Start by setting the goal node to 0 and recursively filling the neighboring nodes

Value $(x_i) = \min\{C(x_i, x_j) + value(x_j)\}$, for all x_j in successor of x_i and $C(x_i, x_j)$ is cost of going from x_i to x_j



7	6	5	4	3	2	1	0
8	7	6	5				
9	8	7	6	7	8	9	10
						10	
17	16	15	14	13	12	11	12

DYNAMIC PROGRAMMING

Pseudo code:

```
function Dynamic programming ( Grid) returns value function
  set value of every grid cell to ∞
  set goal cell value to be 0
  set a Boolean flag change to True

  while change is True:
    Set change to False
    for every cell in the grid:

      if cell is not an obstacle:
        for all successors of current cell:
          new value = min{cost-to-go from current cell to successor + cost of successor}
          if new value < existing value of current cell:
            set cell to new value
            set change to True
```

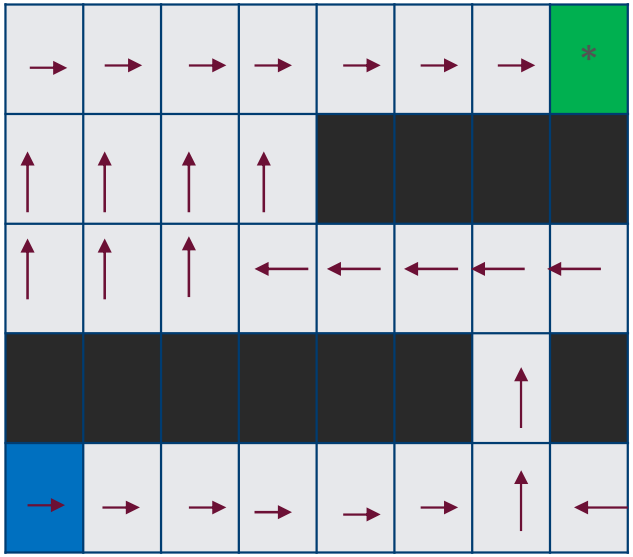
DYNAMIC PROGRAMMING

If you know the value for each cell, you can compute the optimal path directly through steepest gradient descent

→ At any state x choose to go to a neighbor that has the lowest cost-to-go

The result is a policy for each state of the system that will get you to goal state via the optimal path

7	6	5	4	3	2	1	0
8	7	6	5				
9	8	7	6	7	8	9	10
						10	
17	16	15	14	13	12	11	12



ADDITIONAL READING

Additional reading:

- Chapter 2 of “Planning Algorithms” by S. M. LaValle
- Chapter 3 of “Principles of Robot Motion, Theory, Algorithms, and Implementation” by Howie Choset, Kevin Lynch, Seth Hutchinson, George Kantor, Wolfram Burgard, Lydia Kavraki, and Sebastian Thrun
- Lifelong Planning A*: Sven Koenig, Maxim Likhachev, and David Furcy (link)
- D* Lite: Sven Koenig, Maxim Likhachev (link)