# Planning with A*

Benned Hedegaard

Last Revised June 11th, 2020

## What's the Point?

This workshop will detail how we can get robots to plan their actions in advance. How can we formalize the planning problem, apply it to specific robot platforms, and account for obstacles in the robot's environment?

## 1 Planning by Search

We first consider the **planning problem** using an abstract formulation from the field of artificial intelligence. By precisely defining what components a planning problem needs to have, we can create general **algorithms** (i.e. problem-solving techniques) that will work regardless of the underlying task at hand.

**General Problem Domain** - Requires these five components:

1. A finite set of **states** $S$ which defines all ways the world could be.
2. A finite set of **actions** $A$ which defines all actions the agent can make.
3. An **applicability function** $Actions(s)$ which returns the set of all possible actions the agent can take from any state $s$.
4. A **transition model** which is a function $Result(s, a)$. This returns the state $s'$ where the agent ends up if they are in state $s$ and take action $a$.
5. A **cost function** $Cost(s, a, s')$ which defines the cost for the agent to take action $a$ in state $s$ and end up in state $s'$.

To create a specific instance of the planning problem, we need an **initial state** $s_0$ where the agent starts and a set of **goal states** $G \subseteq S$ where the agent must end up. Any **solution** to the problem will be a sequence of actions that takes the agent from the start state $s_0$ to any goal state $g \in G$.

To find such a solution, we can **search** over all possible actions and resulting states starting from $s_0$. The exact order we consider these possible states and actions is the important part of searching; we'll discuss this further in Section

3. This general problem formulation was presented in *Artificial Intelligence: A Modern Approach*, an excellent textbook on the field of AI [2].

Finally, a note about **abstraction**. The power of this formulation is that it abstracts away all other details from any problem we're trying to solve. If we have some new problem that can be formatted as specified above, we can apply the same solutions we'll discuss today to the new problem.

## 2 Planning on a Grid

Let's connect this abstractness with some actual robot planning. Consider a robot that operates in the 2D plane. The robot has some location $(x, y)$ and some heading $\theta$. We somehow need to turn this continuous **configuration space** (the space of all possible robot poses) into a set of discrete states. One solution is to cover the world in a grid of points. An interactive illustration can be found [here]. The intersections in this grid define our states and our actions move from one intersection to another.

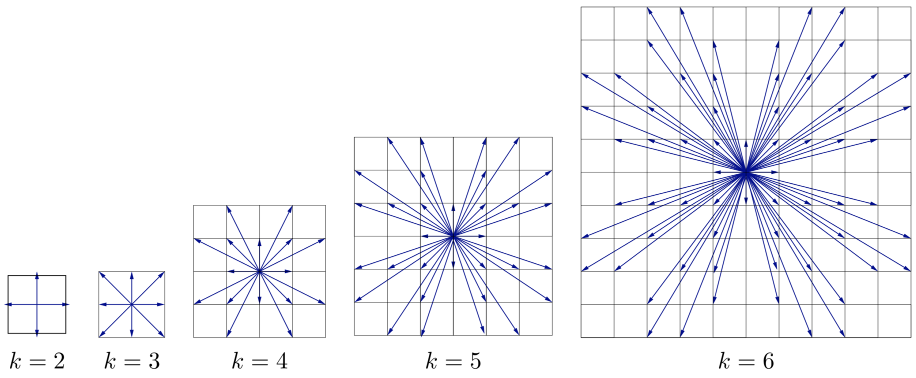

Figure 1 - Connectivity options for grid-discretized planning from [1]. Here $k$ says "allow actions to the $2^k$ closest neighbors of the current state"

The final step of defining this grid representation as a planning problem is deciding which actions are allowed; which neighbors can we move to in one action? This choice of $2^k$ **connectivity** can be seen in Fig. 1 as well as [here], where I represent each state as a node in a graph connected by arcs representing valid actions. I'll use the words "state" and "node" interchangeably in the following sections due to this correspondence, as each node contains one state.

## 3 The A* Algorithm

We're finally ready to plan. We can set the initial state $s_0$ to the grid point nearest to the robot and specify wherever we'd like as some final destination.

**A*** (pronounced "A star") is an **informed search strategy**, which means it uses information beyond the state-action problem definition presented above.

This extra information comes from a **heuristic function** $h(s)$ that returns how far a given state is from a goal. A* will use this to prioritize its search.

To run the algorithm, we'll create a Node **data structure** to organize our data. Each Node will represent one possible path to the Node's state. A Node $n$ will store the current state $s$, the total cost $g$ of the path to reach the current state, and the sum $f = g + h(s)$. This $f$ value estimates of the cost of the entire path through the Node to the closest goal. By organizing these values into one object, we can write functions that operate on the object instead of each piece of information separately. This structure is typical of **object-oriented programming**, the most popular paradigm for software development today.

We'll now look at the **pseudocode** and then reconsider what's going on. If you're unfamiliar with pseudocode, it's a simplified way to write out how an algorithm works. One could then write the algorithm in any programming language they'd like. For example, during the development of this workshop I ended up implementing the below pseudocode in both C++ and Python.

**Algorithm 1:** A* Search

```
   Input: $s_0$, $G$, $Neighbors(s)$, $Cost(s, s')$, $h(s)$
   Output: Path $\{s_0, s_1, \ldots, g\}$
1  $O \longleftarrow \{s_0\}$
2  $C \longleftarrow \emptyset$
3  while $O$ not empty do
4      $curr \longleftarrow O.Pop()$
5      if $curr \in G$ then
6          return $Backtrack(curr)$
7      $C.Push(curr)$
8      for $next \in Neighbors(curr)$ do
9          if $next \in C$ then
10             continue
11         $next.g \longleftarrow curr.g + Cost(curr, next)$
12         $next.f \longleftarrow next.g + h(next)$
13         $next.prev \longleftarrow curr$
14         $O.Push(next)$
15 return FAILED
```

There's a lot going on, so let's take it line-by-line. The core data structures of A* are the **open list** $O$ and **closed list** $C$. The open list is a **priority queue** of future nodes to explore and the closed list stores nodes we've already explored. A priority queue stores a list of items sorted by some value, in this case the $f$ value of each node.

We'll start exploring from $s_0$ (Line 1). The closed list starts empty, as we haven't explored any nodes when we start (Line 2). Each step of our search (Line 3), we pop an item from the open list (Line 4). This gives us the "best" unexplored node in $O$ according to its $f$ value. If this node is a goal state, we're done and can return the corresponding path by backtracking through the previous nodes we've visited (Lines 5-6).

Otherwise we add the node to the closed list (Line 7) and expand the node by considering all of its neighbors (Line 8). The $Neighbors(s)$ function combines the $Actions(s)$ and $Result(s, a)$ functions together; it returns all valid states resulting from actions taken in state $s$. Reformatting the functions in this way simplifies the A* code and makes it easier to ensure the validity of new states.

If the new node has already been explored, we skip it (Lines 9-10). Otherwise we compute the total cost to get there through the current node, which we call $g$ (Line 11). We then compute the new node's $f$ value as $g + h$; the cost so far plus an estimate of the cost to a goal state (Line 12). We also store how we got to the new node and then put it into the open list (Lines 13-14). If the open list is ever empty (Line 3), the while loop will exit. The search has failed; we've exhausted every possible option without finding a goal state (Line 15).

As long as our heuristic $h(s)$ is **admissible**, meaning that it never over-estimates the cost to a goal, the solution returned by A* is guaranteed to be optimal. An example will hopefully solidify how all of this works; see slides.

## 4 A* in the Joint Space

All of the above discussion, at least on the surface, was concerned with a mobile robot on a grid. This was no accident. One of the fundamental lessons I hope to get across with this workshop is the power of the abstraction we've set up: A* can be used to plan on a map between cities, between points on a plane, or even the joint space for our robotic arm. As long as we can define states, actions and their costs, and a transition model, we can run A* as it's been presented above.

In the case of our robot arm, we have a continuous joint space. How can we define each aspect of the planning problem for our arm? **Think this through before continuing**. One answer is to discretize the joint angles to form a finite set of states. Changing the angle of one servo by a certain amount could be our action set. Defining a heuristic function is left to the reader. Hint: What space should our heuristic function act on? What space are our goal states in? The forward kinematics of the previous workshop should allow you to plan smooth motions for your robotic arm. □

## References

[1] Carlos Hernández, Nicolás Hormazábal, and Jorge Baier. "The $2^k$ Neighborhoods for Grid Path Planning". In: *Journal of Artificial Intelligence Research* 67 (Jan. 2020), pp. 81–113. DOI: 10.1613/jair.1.11383.

[2] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach.* 3rd ed. Prentice Hall, 2010.