GIPP: General Interval Path Planning for Dynamic Environments

Author: Yu Hou

Last edit: 04/03/2020

Email: yhou0015@student.monash.edu

Contents

I. INTRODUCTION

This work is based on the SIPP (Safe Interval Path Planning) method, in which each location on the map is split into a sequence of safe/collision intervals along the timeline.

A safe interval is a continuous period of time for a location, during which there is no collision and it is in collision one timestep prior and after the period; whereas a collision interval is the opposite of a safe interval. Therefore, according to this definition, a safe interval should be bounded by collision intervals or infinity, so it is impossible for a robot to wait in place from one safe interval to another, because by doing so it will inevitably cross at least one collision interval.

The GIPP drops this limitation, by allowing a robot to cross collision intervals, with additional costs. It also introduces different costs (both temporal and spatial) for a robot to move between locations, which is set to 1 per grid for all time in the SIPP method.

Therefore, the GIPP method can be considered as a ‘superset’ to SIPP. According to the description, there is no need to know the location of obstacles for GIPP, the input will be a set of ‘arrows’, which represents the direction and value of costs.

II. ALGORITHM

Arrow

An Arrow is the representation of costs. For instance, an obstacle locates at A at time *0* is to be moved to B at time *dt*, as shown in Figure 1. Then, arrows can be constructed as shown in the timeline.

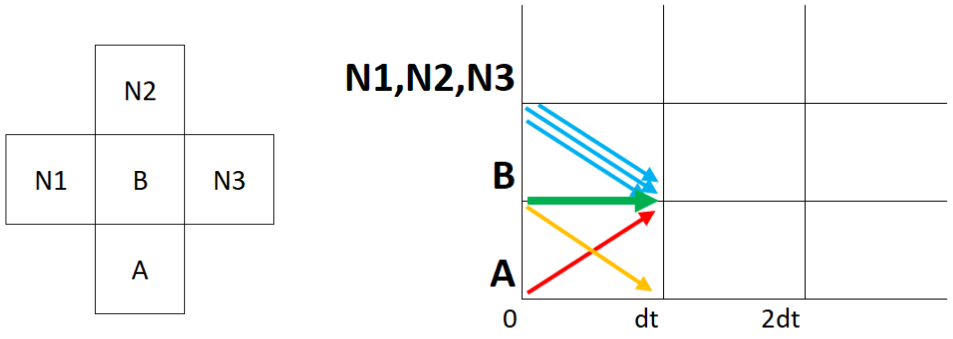


Figure 1 An example map and timeline, with an obstacle moving from A to B, (N1, N2 and N3 are neighbors of B)

The red arrow indicates the cost to move along the obstacle from A at time *0* to B at *dt*; The green arrow is the cost for the robot to wait in place at B from *0* to *dt*; The yellow arrow is the cost to move from B at *0* to A at *dt*, because by doing so the robot will collide with the obstacle en route; and finally, the 3 blue arrows are costs to move from any of the 3 neighbors at time *0* to B at *dt*, since the obstacle will arrive at B at time *dt*.

Therefore, for each step of an obstacle's path, there will be at most 6 arrows, they can all have different values. The green arrow can be termed as ***wait cost***, whereas the rest are ***transition costs***. The values of the costs are all relative to 1, which is the normal cost without the arrow.

Treat wait cost as vertex cost

To save the number of arrows, if the costs of green and blue arrows are equal, we can treat wait cost as a vertex cost, as show in Figure 2. Cost will be incurred for any path that passes location B at time *dt*. Consequently, the cost of red arrow should now be relative to the vertex cost, as opposed to the cost before.

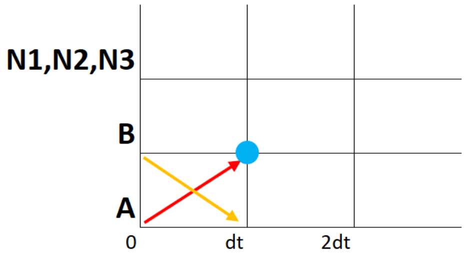


Figure 2 Treat wait cost as vertex cost

Store arrows

For transition costs, an arrow is specified by "arrives at location *B* at time *t*, from location *A*, at a cost of *c*". Therefore, the red arrow is specified as (B,dt,A,cost), the yellow arrow is (A,dt,B,cost) etc. I implement transition cost in Python as a 3-level depth dictionary (an unordered lookup table with best case time complexity of O(1) in insertion and lookup, similar to *std::unordered_map* in c++), i.e. (B,dt,A,cost)≡{B:{dt:{A:cost}}}, where {key:value} is a dictionary in Python.

For wait costs, an arrow is specified by "wait at location *B*, from time *t1* to *t2*, at a cost of *c*", so the two green arrows at the left of Figure 3 are (B,[dt,dt],cost) and (B,[2dt,2dt],cost), and the longer arrow at the right is (B,[dt,2dt],cost). They are also expressed as a dictionary in Python: (B,[dt,2dt],cost) ≡{B:{(dt,2dt):cost}}

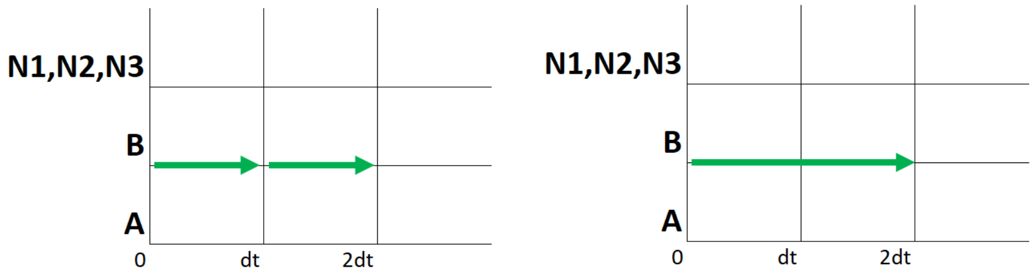


Figure 3 Wait cost arrows (left) can be combined into one arrow (right), if costs are same

As shown in Figure 3, consecutive wait cost arrows with equal costs at the same location can be combined into one arrow.

It is important to note that only the end time of an arrow is recorded, since by doing so, wait cost will be same whether it is treated as vertex cost or edge cost; and it will be easier to convert arrows into timeline intervals.

On the implementation side, however, we require 2 type of wait costs: the first one being the right side of Figure 3, i.e. $\{\mathrm{B}: \{(\mathrm{dt}, 2\mathrm{dt}): \mathrm{cost}\}\}$, used to construct timeline; the second one being the combination of the left and right side of Figure 3, i.e. for each timestep in the interval: $\{\mathrm{B}: \{(\mathrm{dt}, \mathrm{dt}): \mathrm{cost}, (2\mathrm{dt}, 2\mathrm{dt}): \mathrm{cost}, (\mathrm{dt}, 2\mathrm{dt}): \mathrm{cost}\}\}$, used to calculate the cost between States.

Space requirement will be higher, but not by too much in my opinion, since for wait cost arrows to combine into one, they need to be at the same location, consecutive in time, and equal in cost.

Create Timeline (convert arrows into intervals as in SIPP)

Step 1: for a location, its interval is initialized as from 0 to infinity

Step 2: split the interval for each wait cost on this location

Step 3: split the intervals further based on transition costs pointed to this location.

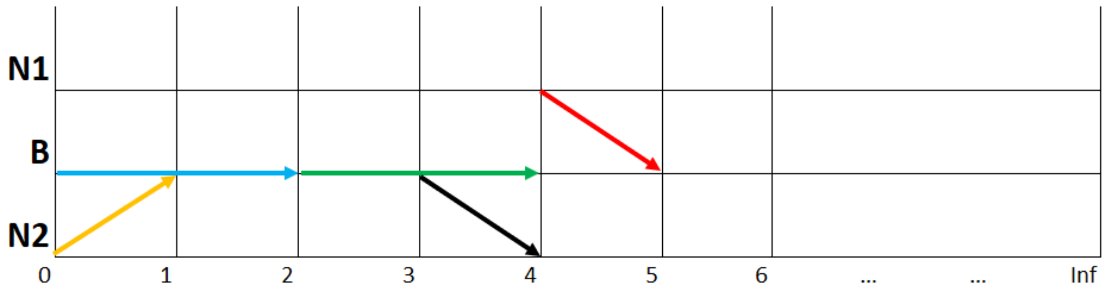


Figure 4 Example timeline. Different color means different costs.

As an example shown in Figure 4, to construct the timeline for location B:

Step 1: the interval of location B is initialized as

$$timeline(B) = [[0, Inf]]$$

Step 2: split the interval based on wait cost arrows:

$$timeline(B) = [[0,0],[1,2],[3,4],[5,Inf]]$$

Step 3 split the intervals based on transition costs pointed to the location:

$$timeline(B) = [[0,0],[1,1],[2,2],[3,4],[5,5],[6,Inf]]$$

The realization in code is a bit different, since the arrows are stored in a dictionary, which means they are unordered in time, so for each new interval, it has to find the correct interval in the list to split, meaning the time complexity will be $O(n^2)$, where n is the number of arrows.

To save time, we fist combine intervals stored in both wait cost and transition cost into one list for each stored location in wait and transition cost, in O(n) time. Then we sort the intervals based on the first then the second element of the interval in O(nlogn) time. Finally, for each interval, we only need to split the last one or two intervals in the list, in constant time, because we know the new one happens last, this final step can be competed in O(n) time.

Alternatively, at step 2, I directly build a heap using the list, in O(n) time, then for step 3 each popped element from the heap will happens last, therefore using constant amount of operations to split the interval. However, I forgot that pop elements from a heap costs

O(logn) time, so this final step actually costs O(nlogn) instead of O(n) as I previously thought.

As an example in Figure 4, time intervals of wait cost arrows in B are [1, 2] and [3, 4]; time intervals of transition cost arrows are [1, 1] and [5, 5], since the yellow arrow points to B at time 1 and the red arrow points to B at time 5. Comparison of the intervals is: $[1,1] < [1,2] < [3,4] < [5,5]$

Step 1: combine intervals at B into one list: [[1, 2], [3, 4], [1, 1], [5, 5]]

Step 2: built a heap from the list (or simply sort the list):

$$builtHeap([[1,2],[3,4],[1,1],[5,5]]) \rightarrow [[1,1],[3,4],[1,2],[5,5]]$$

Or,

$$sort([[1,2],[3,4],[1,1],[5,5]]) \rightarrow [[1,1],[1,2],[3,4],[5,5]]$$

Step 3: initialize $timeline(b) = [[0, Inf]]$ and split the intervals for each popped interval from the heap.

Successors

Similar to SIPP, we define a **State** by a location and an interval, and store time, cost, heuristics etc. as independent variables. The only major difference is that now intervals at the same location can be successors. As shown in Figure 5, green intervals including B2 and B3 are now all successors to the red interval. However, if wait cost at B2 is infinity, then all intervals after it at the same location will be ignored (i.e. B3); if the transition cost to an interval at a different location is infinity, then the interval will be ignored as well.

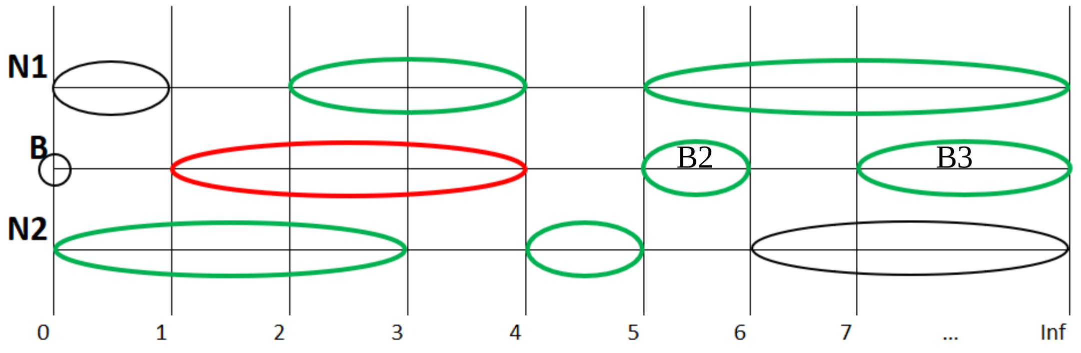


Figure 5 Green intervals are all successors to the red interval.

Cost between states

Similar to SIPP, we use $c(s, s')$ to represent the cost to execute from state s to state s'. It is calculated as:

$$c(s, s') = cost\ to\ transition + transition\ cost + target\ vertex\ cost$$

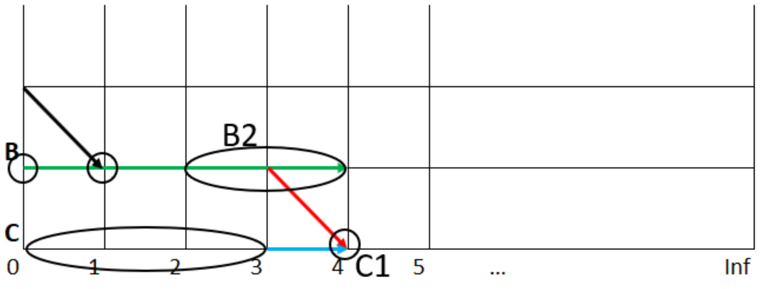


Figure 6 costs between states.

As shown in Figure 6, if we were to calculate the cost from state B2 to state C1, assuming the arrival time of B2 is at time 2, arrival time of C1 is 4, then the cost to transition is the wait cost arrow at B from time 2 to 3; transition cost is the value of the red arrow; and target vertex cost is the value of the blue arrow, if wait cost is to be treated as vertex cost, or 0 if wait cost is treated as edge cost.

Remember in the section **Store arrows** we create a new wait cost dictionary with every time step in the wait cost interval. This will be useful in calculating cost to transition. For example, in Figure 6, the original wait cost at B is [0, 4], but it is split by the black arrow arrived at time 1, so the interval [2, 4] will not be found at the original wait cost dictionary because the key to look up is [0, 4]. However, if we store the same cost for every time step in the wait cost interval, i.e. cost at time 0, 1, 2, 3, 4 into the dictionary, then we can read the wait cost at time 2, multiplied by the time difference to get the cost to transition.

If s and s' have the same location, i.e. wait in place, then the cost will be calculated as:

$$c(s, s') = cost\ to\ transition + wait\ cost\ of\ all\ states\ between\ s\ and\ s' \\ + target\ wait\ cost$$

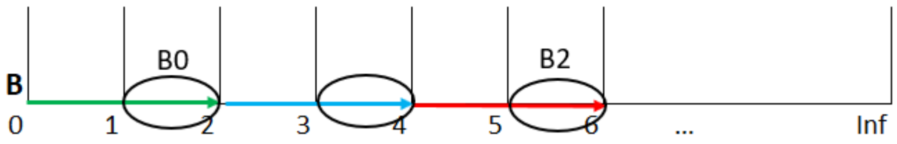


Figure 7 Example cost between states at the same location

As shown in Figure 7, if we were to calculate cost from state B0 to state B2, assuming the arrive time of B0 is 1, and arrival time of B2 is 5. Cost to transition is the wait cost from time 1 to 2, if wait cost is treated as vertex cost, then the wait cost arrow from 0 and 1 should also be included; wait cost of all states between B0 and B2 is the cost between time 3 and 4; finally the target wait cost is the wait cost from time 4 to 5.

A^* Termination criteria

In SIPP, the termination criteria is when the priority queue is empty. We can, however, terminate earlier in GIPP. As shown in Figure 8, the robot starts at location (1,1) at time *0* and want to find an optimal path to location (2,4). There are 4 intervals at the goal location, but the first two all ends before time (2,4) - (1,1) = 4, which is the earliest possible time the goal can be reached, so there are 2 possible intervals at which the robot may arrive. Therefore, we can initialize a counter before A^*'s main loop, and for each state popped from the priority queue, we add 1 to the counter if its location is the goal, we can terminate when the counter equals to the number of possible intervals at the goal. This is true only when the heuristic is consistent, without this the approach loses completeness.

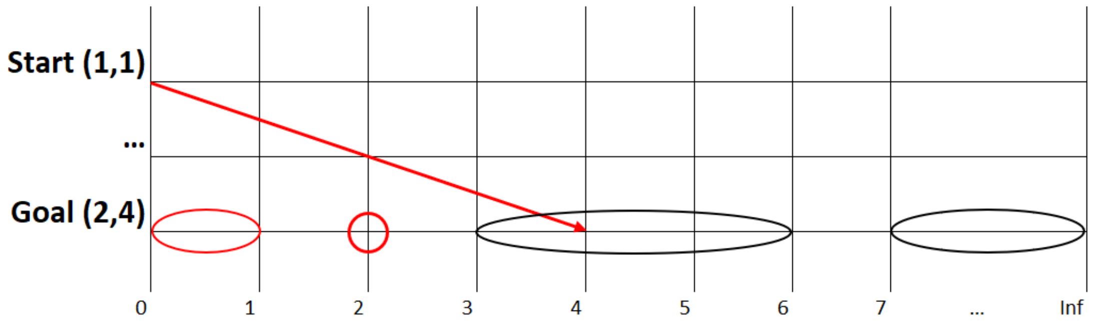


Figure 8 from (1,1) to (2,4). the red intervals can be omitted, since they end before the shortest amount of time the goal can be reached.

III. Code

Create timeline function

Input: wait cost, transition cost, start location, goal location

Function:

- Create timeline intervals for each relevant location on the map;
- Create a new wait cost dictionary, for each time step, as detailed in section **Store arrows**;
- Trim the intervals of the goal location to ignore first few intervals ends before it can be reached, as detailed in section $\boldsymbol{A}^*$ **Termination criteria**.

Output: timeline, new wait cost

```
def create_timeline(wait_cost, transition_cost, start, goal):

    # initialize new dictionary with location as key and combined
    # intervals as value
    loc_intervals = empty dictionary

    # initialize new wait cost
    new_wait_cost = empty dictionary

    # put every interval inside a list for each location
    for each location in wait_cost:
        loc_intervals[location] = empty list
        for each interval in wait_cost[location]:
            add the interval to the loc_intervals[location]
            add wait_cost[location][interval] to the new_wait_cost[location][interval]
            for each time step in interval:
                add wait_cost[location][interval] to new_wait_cost[location][timestep]

    for loc in transition_cost:
        retrieve the list at timeline[location] or initialize new one
        for each interval in transition_cost[loc]:
            add the interval to the loc_intervals[location]

    # reconstruct
    timeline = empty dictionary
    for each location in loc_intervals:
        timeline[location] = [[0, Inf]] # initialize with 0 to infinity
        retrieve the list at loc_interval[location]
        build heap from the list
        while the heap is not empty:
            pop the element from the heap
            split the interval at timeline[location]


    # trim: get rid of the first few intervals at the goal location, because they won't be reached.
    # calculate the minimum amount of time the goal can be reached
    min_time = start - goal
    if goal exists in timeline:
        retrieve timeline[goal]
        for each interval in timeline[goal]:
            if the end time of the interval >= min_time:
                break
        replace timeline[goal] with the rest intervals after the break point

    return timeline, new_wait_cost
```

Motion function

Input: state (stores location and interval number etc.)

Function: Similar to the original SIPP, the only difference is to add the current location into next available motion, if there are more intervals after the current state's interval.

Output: available motion

```
def M(State):
      # return possible neighbors for a given loc
      x, y = x and y coordinates of state location
      neighbours = [(x + 1, y), (x - 1, y), (x, y + 1), (x, y - 1)] # 4 neighbours
      for each neighbour, adds to the motion list if it is not the wall on the map
      add state location to motion list if there are more intervals after.
      return motion
```

Get successor function

Input: state

Function: find successor states, detailed in section **Successors**.

Output: successors to the input state

```
def getSuccessor(State):
      successors = empty list
      for m in M(State):
         if m equal to the input state location:
            intervals(m) = retrieve timeline intervals at m
            end_t = end time of the input state's interval
            for each interval in intervals(m):
               start_t = start time of each interval
               if start_t <= end_t:
                  continue
               if wait cost of the interval is infinity:
                  break
               s' = create new or retrieve state defined by location and interval
               insert s' into successors
         else, if m is not the input state location:
            same to SIPP, except ignore the state if transition cost to it is infinity
      return successors
```

Cost between states function

Input: two states (s and n)

Function: calculate the cost from state s to state n, as described in section **Cost between states**

Output: cost from state s to state n

```
def c(self, s, n):

      cost = 0

      if s and n have the same location:
         wait_time = arrival time of n – arrival time of s

         # cost to transition
         end_t = end time of state s interval
         time_diff = end_t – arrival time of s, or end_t – arrival time of s + 1 if treat wait cost as vertex cost
         wait_time += time_diff * wait cost of state s interval

         # wait cost of all states between s and n
         For each interval between s and n:
            start_t = start time of the interval
            end_t = end time of the interval
            wait_time += (end_t – start_t + 1) * wait cost of the interval

         # wait cost at n
         wait_time += wait cost of state n interval

         cost += wait_time

      else, if s and n have different locations:

         # cost to transition
         m_time = motion time
         transition_start_time = arrival time of n - m_time
         cost_to_transition = transition_start_time – arrival time of s
         cost_to_transition = wait cost of s interval * (transition_start_time – arrival time of s)

         # transition cost
         transition_cost = 1
         transition_cost += transition cost from s to n

         # target vertex cost
         target_vertex_cost = 0
         if treat wait cost as vertex cost:
            target_vertex_cost = wait cost at n

         cost += cost_to_transition + transition_cost + target_vertex_cost
      return cost
```

IV. EXAMPLE

Example 1

To show GIPP is a "superset" of SIPP, we first use an example that is covered in SIPP. As shown in Figure 9, two robots locate at (7,5) and (5,9) are moving leftwards and upwards respectively. The robot starts at (5,4) and the goal is (5,8).

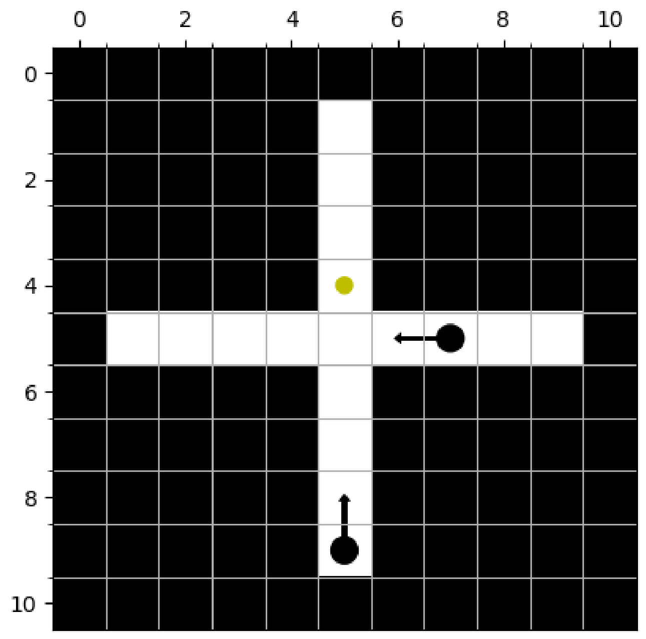
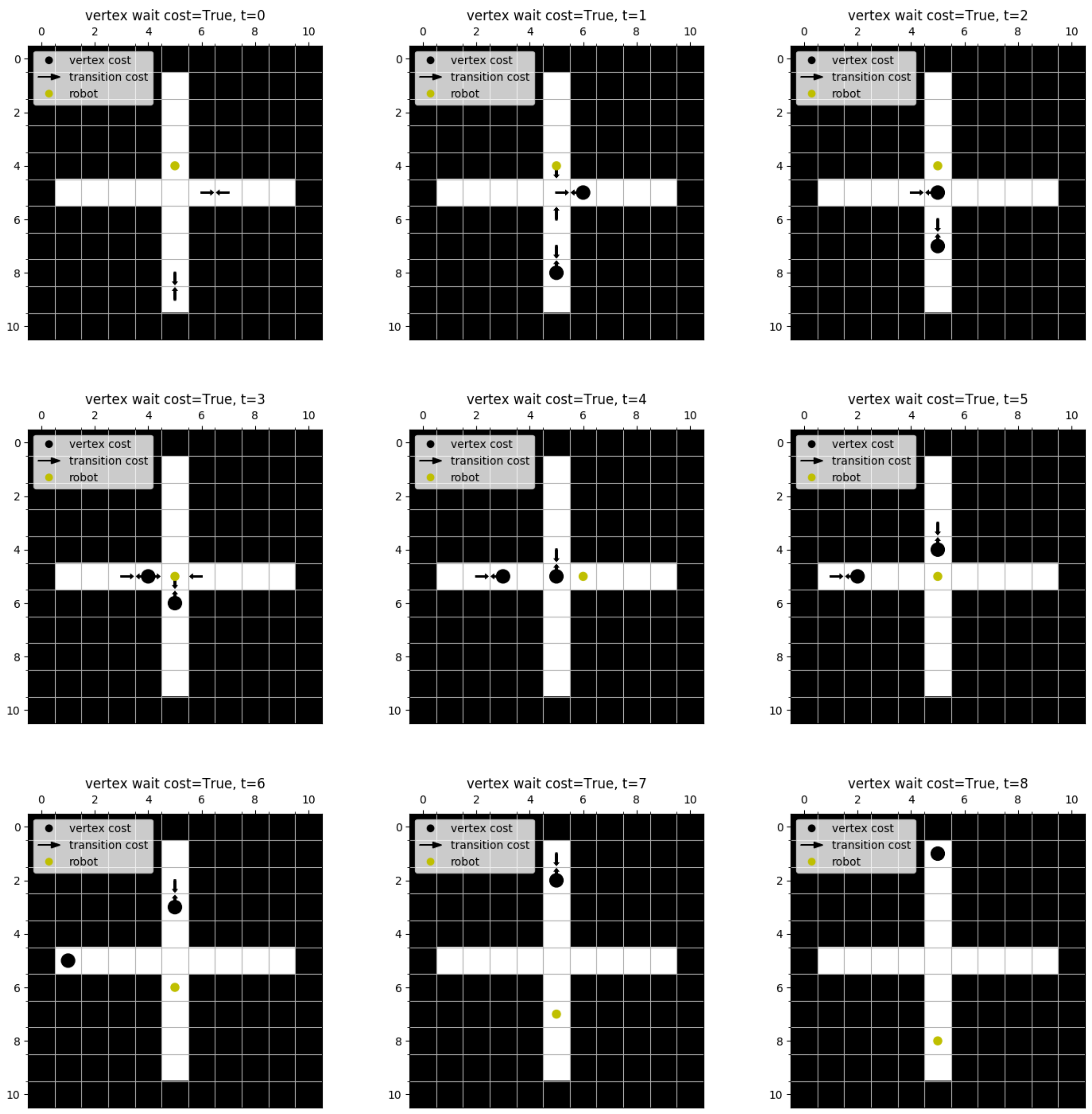


Figure 9 An example map with the robot locates at (5,4) and goal is (5,8).

Treat wait cost as vertex cost:

Since costs are all infinity, we can treat wait cost as vertex cost, to save the number of arrows. The solution is shown in Figure 10:



vertex wait cost=True, t=0
vertex wait cost=True, t=1
vertex wait cost=True, t=2
vertex wait cost=True, t=3
vertex wait cost=True, t=4
vertex wait cost=True, t=5
vertex wait cost=True, t=6
vertex wait cost=True, t=7
vertex wait cost=True, t=8
vertex cost
transition cost
robot

Figure 10 Example solution. Treat wait cost as vertex cost

Please be noted that the solution to this problem is not unique, there are multiple route with same costs.

The black circles are vertices, as the wait costs are all set to infinity, robot can never reside on it. A black arrow points from location A to B at time t means the cost will be infinite if the robot moves from A at time t to location B at time t+dt. For example, at time 1, there is a black arrow at the robot's location pointing downwards, this means the robot should not move downwards at time 1.

Treat wait cost as edge costs

Alternatively, we can treat wait cost as edge costs, the solution is same, except the presentation of the solution is a bit different:

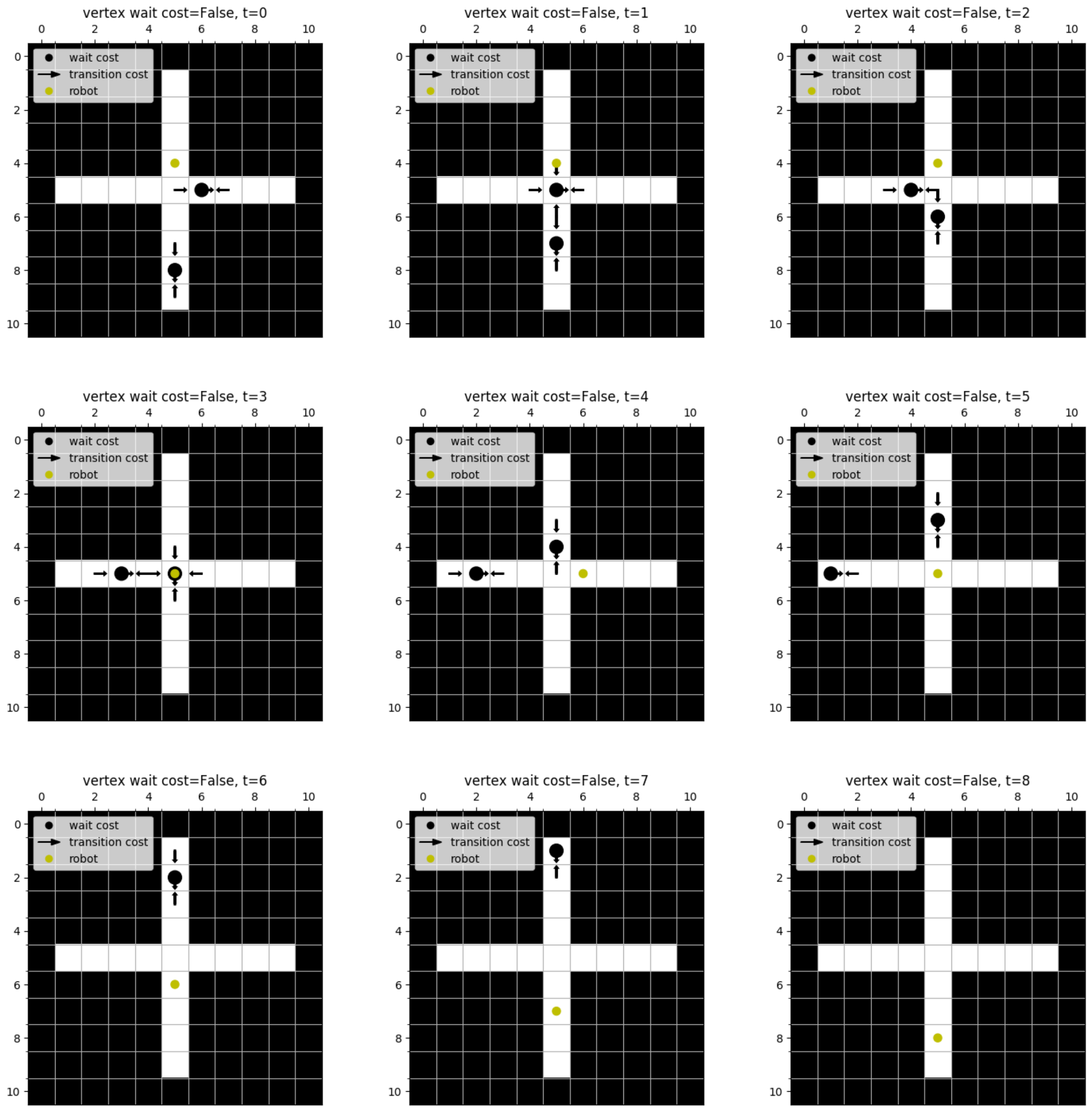

vertex wait cost=False, t=0
vertex wait cost=False, t=1
vertex wait cost=False, t=2
vertex wait cost=False, t=3
vertex wait cost=False, t=4
vertex wait cost=False, t=5
vertex wait cost=False, t=6
vertex wait cost=False, t=7
vertex wait cost=False, t=8
wait cost
transition cost
robot

Figure 11 Example solution. Treat wait cost as edge cost

As shown in Figure 11, now there are more arrows. And now black circles are edge wait costs, they can be thought of arrows pointing into the paper (or points along the time axis, if visualized in 3D). For example, at time 3, the robot resides on the black circle, it means if the robot chooses to wait in place, there will be an infinite cost, and based on the arrows around it, the robot can only choose to move rightwards or upwards.

Example 2

A robot plans to move from location (1,1) to location (1,3), with different cost as shown in Figure 12. Here the wait cost is treated as edge cost for generalizability. By common sense, the blue line is the optimal path. The total cost is 207. The solution is also shown in Figure 13, in which arrow color indicates cost.

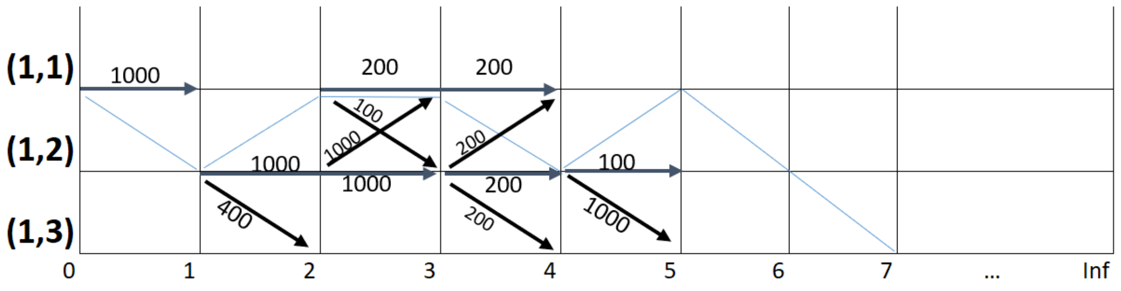


Figure 12 Example 2 timeline. Blue line is the optimal path.

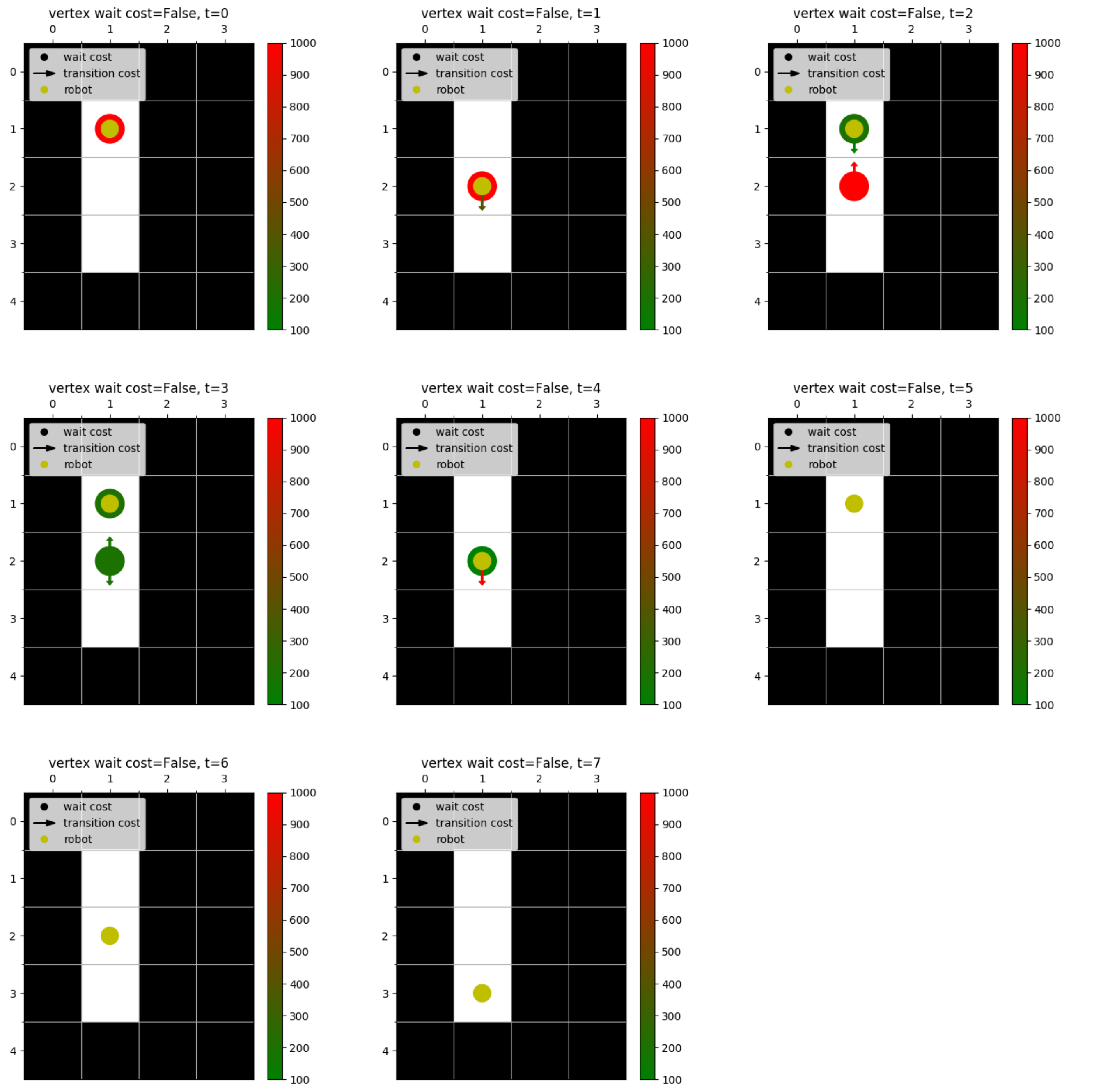


Figure 13 Solution to example 2. color indicates value