D* Lite
Koenig & Likhachev, 2002 AAAI-02 Proceedings. (www.aaai.org)

robot navigation in unknown terrain, including goal-directed navigation in unknown terrain and mapping of unknown terrain.
…fast replanning methods in artificial intelligence and robotics.

DynamicSWSF-FP (Ramalingam & Reps 1996), are currently not much used in artificial intelligence. They reuse information from previous searches to find solutions to series of similar search tasks much faster than is possible by solving each search task from scratch. An overview is given in (Frigioni, Marchetti- Spaccamela, & Nanni 2000). Heuristic search methods, such as A* (Nilsson 1971), on the other hand, use heuristic knowledge in form of approximations of the goal distances to focus the search and solve search problems much faster than uninformed search methods.
…
We recently introduced LPA* (Lifelong Planning A*), that generalizes both DynamicSWSF-FP and A* and thus uses two different techniques to reduce its planning time (Koenig & Likhachev 2001). In this paper, we apply LPA* to robot navigation in unknown terrain. The robot could use conventional graph-search methods when replanning its paths after discovering previously unknown obstacles.
…
we therefore present D* Lite, a novel replanning method that implements the same navigation strategy as D* but is algorithmically different. D* Lite is substantially shorter than D*
…
We also present an experimental evaluation of the benefits of ***combining incremental and heuristic search*** across different navigation tasks in unknown terrain, including goal-directed navigation and mapping.

runtime complexity at best, that is, no terrain changes, would be similar to A* which is dependent upon having a good heuristic.

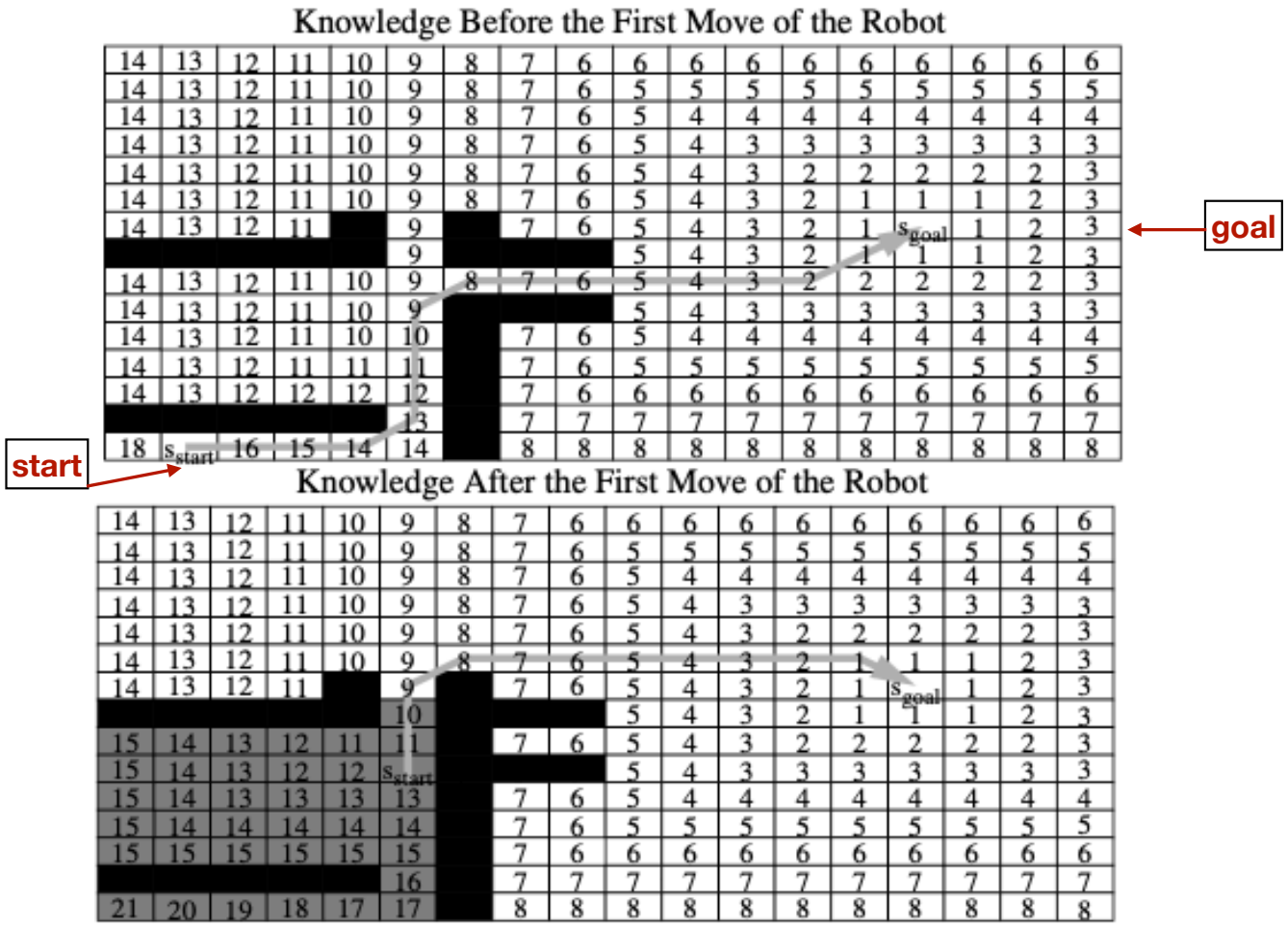

Figure 1: Simple Example.

Cells whose goal distances have changed are shaded gray.

one can efficiently recalculate a shortest path from its current cell to the goal cell by recalculating only those goal distances that have changed (or have not been calculated before) and are relevant for recalculating the shortest path. This is what D* Lite does. The challenge is to identify these cells efficiently.

D* Lite (optimized, search from goal to start)

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

procedure Initialize()
{02"} $U = \emptyset$;
{03"} $k_m = 0$;
{04"} for all $s \in S$ $rhs(s) = g(s) = \infty$;
{05"} $rhs(s_{goal}) = 0$;
{06"} U.Insert$(s_{goal}, [h(s_{start}, s_{goal}); 0])$;

procedure UpdateVertex(u)
{07"} if $(g(u) \neq rhs(u)$ AND $u \in U)$ U.Update$(u, CalculateKey(u))$;
{08"} else if $(g(u) \neq rhs(u)$ AND $u \notin U)$ U.Insert$(u, CalculateKey(u))$;
{09"} else if $(g(u) = rhs(u)$ AND $u \in U)$ U.Remove(u);

procedure ComputeShortestPath()
{10"} while (U.TopKey() $<$ CalculateKey(s_{start}) OR $rhs(s_{start}) > g(s_{start}))$
{11"} u = U.Top();
{12"} k_{old} = U.TopKey();
{13"} k_{new} = CalculateKey(u));
{14"} if$(k_{old} < k_{new})$
{15"} U.Update(u, k_{new});
{16"} else if $(g(u) > rhs(u))$
{17"} $g(u) = rhs(u)$;
{18"} U.Remove(u);
{19"} for all $s \in Pred(u)$
{20"} if $(s \neq s_{goal})$ $rhs(s) = \min(rhs(s), c(s, u) + g(u))$;
{21"} UpdateVertex(s);
{22"} else
{23"} $g_{old} = g(u)$;
{24"} $g(u) = \infty$;
{25"} for all $s \in Pred(u) \cup \{u\}$
{26"} if $(rhs(s) = c(s, u) + g_{old})$
{27"} if $(s \neq s_{goal})$ $rhs(s) = \min_{s' \in Succ(s)}(c(s, s') + g(s'))$;
{28"} UpdateVertex(s);

LPA* uses Succ() here.
D*Lite searches ***Pred()*** due to opposite direction

procedure Main()
{29"} $s_{last} = s_{start}$;
{30"} Initialize();
{31"} ComputeShortestPath();
{32"} while $(s_{start} \neq s_{goal})$
{33"} /* if $(g(s_{start}) = \infty)$ then there is no known path */
{34"} $s_{start} = \arg\min_{s' \in Succ(s_{start})}(c(s_{start}, s') + g(s'))$;
{35"} Move to s_{start};
{36"} Scan graph for changed edge costs;
{37"} if any edge costs changed
{38"} $k_m = k_m + h(s_{last}, s_{start})$;
{39"} $s_{last} = s_{start}$;
{40"} for all directed edges (u, v) with changed edge costs
{41"} $c_{old} = c(u, v)$;
{42"} Update the edge cost $c(u, v)$;
{43"} if $(c_{old} > c(u, v))$
{44"} if $(u \neq s_{goal})$ $rhs(u) = \min(rhs(u), c(u, v) + g(v))$;
{45"} else if $(rhs(u) = c_{old} + g(v))$
{46"} if $(u \neq s_{goal})$ $rhs(u) = \min_{s' \in Succ(u)}(c(u, s') + g(s'))$;
{47"} UpdateVertex(u);
{48"} ComputeShortestPath();

D* Lite (optimized, search from goal to start)

variables

u = min priority queue, aka open list.
key is from calculateKey().
NOTE: I need to specialize my Fibonacci. Heap.java to use a node that has a comparator which uses the 2 keys.
The node also needs to retain a reference to kM which will be the same for all nodes and is held and altered by the class owning the heap instance.

double[] rhs = new double[nV];
rhs(s)=0 if s=s_start,
else = min(g(s')+c(s',s)) where s' is Succ(s) <== check this

double[] g = new double[nV]; // dist to goal; a node state is inconsistent when g != rhs

TIntObject<TIntSet>() pred;// predecessor map. can be made from adjMap by reversing the edges.

TIntObject<TIntSet>() succ;// successor map. is given as adjMap. for a grid, one can construct it by the allowed 8 neighboring cells of each vertex.

input: graph as adjacency Map, supplies Succ(s) and code will calculate Pred() as reverse map of Succ()

input: cost for each edge.

input: h(s, s_goal) approximates the goal distances of the vertices s.
h obeys: h(s, s_goal) ≤ c(s, s′) + h(s′, s_goal) where s' is Succ(s). h can be all 0s.

Additional considerations: want the code to respond to updates in topology (presumably, cost and maybe adjacency).
algorithm quote is "Scan graph for changed edge costs". This implies polling for changes rather than an event-driven paradigm
that responds to changes asynchronously.
for a simple model of the algorithm, without asynchronous threads for an event driven model, cost changes to d*lite would be present
in the cost data structure owned by the class which creates the instance of d*Lite and that class can modify cost. D*Lite can maintain a
a copy of the costs at the beginning of the while loop in Main and scan for changes between the copy and the original and act upon them.

test: example in https://www.cs.cmu.edu/~motionplanning/lecture/AppH-astar-dstar_howie.pdf
Their example uses h composed of all 0s.