CMSC 425: Lecture 18
Motion Planning: Computing Shortest Paths

Reading: Some of the material from today's lecture comes from the book "Artificial Intelligence for Games" by I. Millington and J. Funge. The material on A* search is derived from Prof. Dana Nau's lecture notes.

Recap: In the previous lecture, we demonstrated how, through the use of waypoints and roadmaps, geometric path finding problems can be reduced to path finding in graphs. Today, we will discuss a number of efficient methods for computing shortest paths in graphs and applications to other types of path-finding problems.

Computing Shortest Paths: The problem of computing shortest paths in graphs is very well studied. Recall that a *directed graph* (or *digraph*) $G = (V, E)$ is a finite set of *nodes* (or *vertices*) V and a set of ordered pairs of nodes, called *edges* E (see Fig. 1(a)). If (u, v) is an edge, we say that v is *adjacent* to u (or alternately, that v is a *neighbor* of u). In most geometric settings, graphs are *undirected*, since if you get from u to v, you can get from v to u. It is often convenient to use a directed graph representation, however, since it allows you to model the fact that travel in one direction (say up hill) may be more expensive than travel in the reversed direction.

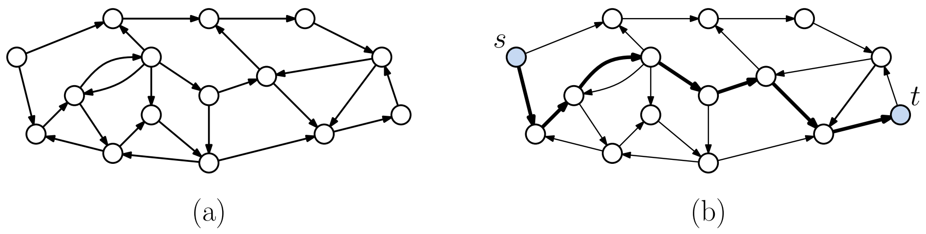

Fig. 1: A directed graph.

In the context of shortest paths, we assume that each edge (u, v) is associated with a numeric *weight*, $w(u, v)$. A *path* in G is any sequence of nodes $\langle u_0, \ldots, u_k \rangle$ such that (u_{i-1}, u_i) is an edge of G. The *cost* of a path is the sum of the weights of these edges $\sum_{i=1}^{k} w(u_{i-1}, u_i)$. The *shortest path problem* is, given a directed graph with weighted edges, and given a *start node* s and *destination node* t, compute a path of minimum cost from s to t (see Fig. 1(b)). Let us denote the shortest path cost from s to t in G by $\delta(s, t)$.

In earlier courses, you have no doubt seen examples of algorithms for computing shortest paths. Here are some of the better known algorithms.

Breadth-First Search (BFS): This algorithm is among the fastest algorithms for computing shortest paths, but it works under the restrictive assumption that all the edges have equal weight (which, without loss of generality, we may assume to be 1). The search starts at s, and then visits all the nodes that are connected to s by a single edge. It labels all of these nodes as being at distance 1 from s. It then visits each of these nodes one by one and visits all of their neighbors, provided that they have not already been visited. It labels each of these as being at distance 2 from s. Once all the nodes at distance 1 have been visited, it then processes all the nodes at distance 2, and so on. The nodes that are waiting to be visited are placed in a *first-in, first-out queue*. If G has n nodes and m edges, then BFS runs in time $O(n + m)$.

Dijkstra's Algorithm: Because BFS operates under the assumption that the edges weights are all equal, it cannot be applied to general weighted digraphs. Dijkstra's algorithm is such an algorithm. It makes the (not unreasonable) assumption that all the edge weights are nonnegative.[1] We will discuss Dijkstra's algorithm below, but intuitively, it operates in a greedy manner by propagating distance estimates starting from the source node to the other nodes of the graph, through an incremental process called *relaxation*. A straightforward implementation of Dijkstra's algorithm runs in $O(m \log n)$ time (and in theory even faster algorithms exist, but they are fairly complicated).

Bellman-Ford Algorithm: Since Dijkstra's algorithm fails if the graph has negative edge weights, there may be a need for a more general algorithm. The Bellman-Ford algorithm generalizes Dijkstra's algorithm by being able to handle graphs with negative edge weights, assuming there are no negative-cost cycles, that is, there is no cycle such that the sum of edge weights along the cycle is strictly smaller than zero. It runs in time $O(nm)$. (Note that the assumption that there are no negative-cost cycles is very reasonable. If such a cycle exists, the path cost could be made arbitrarily small by looping through this cycle an arbitrary number of times. Therefore, no shortest path exists.)

Floyd-Warshall Algorithm: All the algorithms mentioned above actually can be used to solve a more general problem, namely the *single-source shortest path problem*. The reason is that if you run each algorithm until every node in the graph has been visited, it computes the shortest path from s to every other node. It is often useful in games to compute the shortest paths between *all pairs* of nodes, and store them in a table for efficient access. While it would be possible to do this by invoking Dijkstra's algorithm for every possible source node, an even simpler algorithm is the Floyd-Warshall algorithm. It runs in time $O(n^3)$.

Other Issues: There are a number of other issues that arise in the context of computing shortest paths.

Storing Paths: How are shortest paths represented efficiently? The simplest way is through the use of a *predecessor pointer*. In particular, each node (other than the source) stores a pointer to the node that lies immediately before it on the shortest path from s. For example, if the sequence $\langle s, u_1, \ldots, u_k \rangle$ is a shortest path, then $\text{pred}(u_k) = u_{k-1}$, $\text{pred}(u_{k-1}) = u_{k-2}$, and so on (see Fig. 2(a)). By following the predecessor pointer back to s, we can construct the shortest path, but in reverse (see Fig. 2(b)). Since this involves only a constant amount of information per node, this representation is quite efficient.

By the way, in the context of the all-pairs problem (Floyd-Warshall, for example) for each pair of nodes u and v, we maintain a two-dimensional array $P[u, v]$, which stores

[1]Negative edge weights do not typically arise in geometric contexts, and so we will not worry about them. They can arise in other applications. For example, in financial applications, an edge may model a transaction where money can be made or lost. In such contexts, weights may be positive or negative. When computing shortest paths, however, it is essential that the graph have no cycles whose total cost is negative, for otherwise the shortest path is undefined.

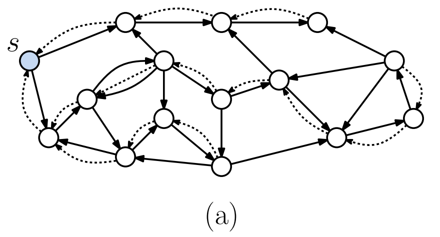

Fig. 2: Storing/reconstructing the shortest path.

either null (meaning that the shortest path from u to v is just the edge (u, v) itself), or a pointer to *any* node along the shortest path from u to v. For example, if $P[u, v] = x$, then to chart the path from u to v, we (recursively) compute the path from u to x, and the path from x to v, and then we concatenate these two paths.

Single Destination: In some contexts, it is desirable to compute an *escape route*, that is, the shortest path from every node to some common destination. This can easily be achieved by reversing all the edges of the graph, and then running a shortest path algorithm. (This has the nice feature that the predecessor links provide the escape route.)

Closest Facility: Suppose that you have a set of locations, called *facilities*, $\{f_1, \ldots, f_k\}$. For example, these might represent safe zones, where an agent can go to in the event of danger. When an alarm is sounded, every agent needs to move to its closest facility. We can view this as a generalization of the single destination problem, but now there are multiple destinations, and we want to compute a path to the closest one.

How would we solve this? Well, you could apply any algorithm for the single-destination problem repeatedly for each of your facilities. If the number of facilities is large, this can take some time. A more clever strategy is to reduce the problem to a *single instance* of an equivalent single destination problem. In particular, create a new node, called the *super destination*. Connect all your facilities to the super destination by edges of cost zero (see Fig. 3(a)). Then apply the single destination algorithm to this instance. It is easy to see that the resulting predecessor links will point in the direction of the closest facility (see Fig. 3(b)). Note that this only requires *one* invocation of a shortest path algorithm, not k.

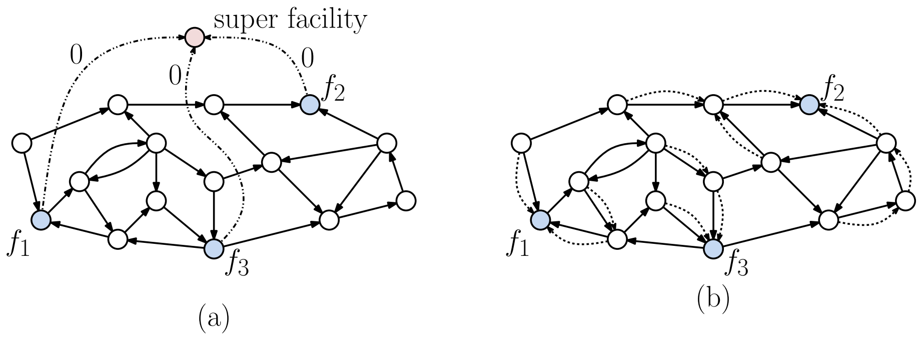

Fig. 3: Using shortest paths to compute the closest facility.

Of course, this idea can be applied to the case of multiple source points, where the goal is to find the shortest path from any of these sources.

Informed Search: BFS and Dijkstra have the property that nodes are processed in increasing order of distance from the source. This implies that if we are interested in computing just the shortest path from s to t, we can terminate either algorithm as soon as t has been visited. Of course, in the worst case, t might be the last node to be visited. Often, shortest paths are computed to destinations that are relatively near the source. In such cases, it is useful to terminate the search as soon as possible. If we are solving a single-source, single-destination problem, then it is in our interest to visit as few nodes as possible. Can we do better than BFS and Dijkstra? The answer is yes, and the approach is to use an algorithm based on informed search.

To understand why we might expect to do better, imagine that you are writing a program to compute shortest paths on campus. Suppose that a request comes to compute the shortest path from the Computer Science Building to the Art Building. The shortest path to the Art Building is 700 meters long. If you were to run an algorithm like Dijkstra, it would visit every node of your campus road map that lies within distance 700 meters of Computer Science before visiting the Art Building (see Fig. 4(a)). But, you know that the Art Building lies roughly to the west of Computer Science. Why waste time visiting a location that is 695 meters to east, since it is very unlikely to help you get to the Art Building. Dijkstra's algorithm is said to be an *uninformed* algorithm, but it makes use of no external information, such as the fact that the shortest path to a building to the west is more likely to travel towards the west, than the east. So, how can we exploit this information?

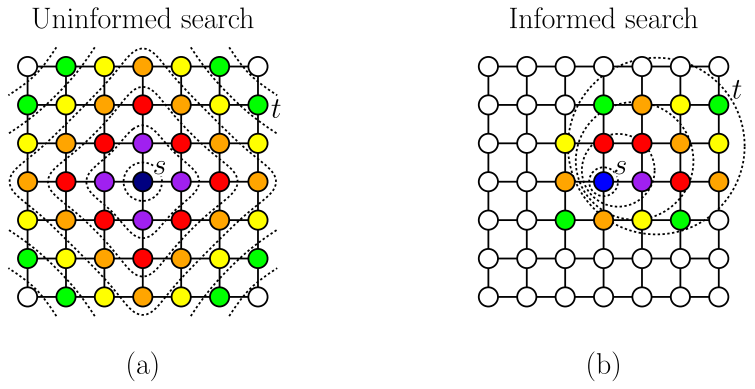

Fig. 4: Search algorithms where colors indicate the order in which nodes are visited by the algorithms: (a) uninformed search (such as Dijkstra) and (b) informed search (such as A*).

Information of the type described above is sometimes called a *heuristic*. It can be thought of as an "educated guess." An *informed search algorithm* is one that employs heuristic information to speed up the search. An example in our case would be using geometric information to direct the search to the destination (see Fig. 4(b)). If your heuristics are good, then they can be of significant benefit. Ideally, of course, even if your heuristics are bad, you still want a correct answer (although it might take longer to compute it).

Informed and Uninformed Search Algorithms: To develop this idea further, lets begin by presenting a simple implementation of Dijkstra's algorithm. (See the code block below.) Vertices are in one of three possible states: *undiscovered* (not yet seen), *discovered* (seen but not yet processed), and *finished* (processed). When a node u has been processed, its associated d-value should equal the actual cost of the shortest path from s to u, that is $d[u] = \delta(s, u)$. Thus, when t has been reached, $d[t]$ is the desired cost. (For simplicity, we ignore storing predecessor links, but that is an easy addition.)

Dijkstra's Algorithm

```
Dijkstra(G, s, t) {
  foreach (node u) {                          // initialize
    d[u] = +infinity;  mark u undiscovered
  }
  d[s] = 0;  mark s discovered                // distance to source is 0
  repeat forever {                            // go until finding t
    let u be the discovered node that minimizes d[u]
    if (u == t) return d[t]                   // arrived at the destination
    else {
      for (each unfinished node v adjacent to u) {
        d[v] = min(d[v], d[u] + w(u,v))       // update d[v]
        mark v discovered
      }
      mark u finished                         // we're done with u
  }
}
```

Best-First Search: What sort of heuristic information could we make use of to better inform the choice of which vertex u to process next? We want to visit the vertex that we think will most likely lead us to t quickly. Assuming that we know the spatial coordinates of all the nodes of our graph, one idea for a heuristic is the Euclidean distance from the node u to the destination t. Given two nodes u and v, let $\text{dist}(u, v)$ denote the Euclidean (straight-line) distance between u and v. Euclidean distance disregards obstacles, but intuitively, if a node is closer to the destination in Euclidean distance, it is likely to be closer in graph distance. Define the *heuristic function* $h(u) = \text{dist}(u, t)$. Greedily selecting the node that minimizes the heuristic function is called *best-first search*. Do *not* confuse this with breadth-first search, even though they share the same three-letter acronym. (See the code block below, as an example.)

Unfortunately, when obstacles are present it is easy to come up with examples where best-first search can return an incorrect answer. By using the Euclidean distance, it can be deceived into wandering into dead-ends, which it must eventually backtrack out of. (Note that once the algorithm visits a vertex, its d-value is fixed and never changes.)

A* Search: Since best-first search does not work, is there some way to use heuristic information to produce a correct search algorithm? The answer is yes, but the trick is to be more clever in how we use the heuristic function. Rather than just using the heuristic function $h(u) = \text{dist}(u, t)$ alone to select the next node to process, let us use both $d[u]$ and $h(u)$. In particular, $d[u]$ represents an estimate on the cost of getting from s to u, and $h(u)$ represents an estimate on

Best-First Search

```
BestFirst(G, s, t) {
  foreach (node u) {                          // initialize
    d[u] = +infinity;  mark u undiscovered
  }
  d[s] = 0;  mark s discovered                // distance to source is 0
  repeat forever {                            // go until finding t
    let u be the discovered node that minimizes dist(u,t)
    if (u == t) return d[t]                   // arrived at the destination
    else {
      for (each unfinished node v adjacent to u) {
        d[v] = min(d[v], d[u] + w(u,v))  // update d[v]
        mark v discovered
      }
      mark u finished                         // we're done with u
  }
}
```

the cost of getting from u to t. So, how about if we take their sum? Define

$$f(u) \;=\; d[u] + h(u) \;=\; d[u] + \text{dist}(u,t).$$

We will select nodes to be processed based on the value of $f(u)$. This leads to our third algorithm, called *A^*-search*. (See the code block below.)

A-Star Search

```
A-Star(G, s, t) {
  foreach (node u) {                          // initialize
    d[u] = +infinity;  mark u undiscovered
  }
  d[s] = 0;  mark s discovered                // distance to source is 0
  repeat forever {                            // go until finding t
    let u be the discovered node that minimizes d[u] + dist(u,t)
    if (u == t) return d[t]                   // arrived at the destination
    else {
      for (each unfinished node v adjacent to u) {
        d[v] = min(d[v], d[u] + w(u,v))  // update d[v]
        mark v discovered
      }
      mark u finished                         // we're done with u
  }
}
```

While this might appear to be little more than a "tweak" of best-first search, this small change is exactly what we desire. In general, there are two properties that the heuristic function $h(u)$ must satisfy in order for the above algorithm to work.

Admissibility: The function $h(u)$ *never overestimates* the graph distance from u to t, that is $h(u) \leq \delta(u,t)$. It is easy to see that this is true, since $\delta(u,t)$ must take into account obstacles, and so can never be smaller than the straight-line distance $h(u) = \mathrm{dist}(u, t)$. A heuristic function is said to be *admissible* if this is the case.

Consistency: A second property that is desirable (for the sake of efficiency) states that, for any two nodes u' and u'', we have $h(u') \leq \delta(u', u'') + h(u'')$. Intuitively, this says that the heuristic cost of getting from u' to the destination cannot be larger than the graph cost from u' to u'' followed by the heuristic cost from u'' to the destination. Such a heuristic is said to be *consistent* (or *monotonic*). This can be viewed as a generalization of the *triangle inequality* from geometry (which states that the sum of two sides of a triangle cannot be smaller than the other side). Consistency follows for our heuristic from the triangle inequality:

$$\begin{aligned} h(u') &= \mathrm{dist}(u', t) \;\leq\; \mathrm{dist}(u', u'') + \mathrm{dist}(u'', t) \qquad \text{(by the triangle inequality)} \\ &\leq \delta(u', u'') + \mathrm{dist}(u'', t) \;=\; \delta(u', u'') + h(u''). \end{aligned}$$

It turns out that admissibility alone is sufficient to show that A* search is correct, but like a graph with negative edge weights, the search algorithm is not necessarily efficient, because we might declare a node to be "finished," but later we will discover a path of lower cost to this vertex, and will have to move it back to the "discovered" status. (This is similar to what happens in the Bellman-Ford algorithm). However, if both properties are satisfied, A* runs in essentially the same time as Dijkstra's algorithm in the worst case, and may actually run faster. The key to the efficiency of the search algorithm is that along any shortest path from s to t, the f-values are nondecreasing. To see why, consider two nodes u' and u'' along the shortest path, where u' appears before u''. Then we have

$$\begin{aligned} f(u'') &= d[u''] + h(u'') \;=\; d[u'] + \delta(u', u'') + h(u'') \\ &\geq d[u'] + h(u') \;=\; f(u'). \end{aligned}$$

Although we will not prove this formally, this is exactly the condition used in proving the correctness of Dijkstra's algorithm, and so it follows as a corollary that A* is also correct. It is interesting to note, by the way, that Dijkstra's algorithm is just a special case of A*, where $h(u) = 0$. Clearly, this is an admissible heuristic (just not a very interesting one).

Examples: Let us consider the execution of each of these algorithms on a common example. The input graph is shown in Fig. 5. For Best-First and A* we need to define the heuristic $h(u)$. To save us from dealing with square roots, we will use a different notion of geometric distance. Define the L_1 (or Manhattan) distance between two points to be the sum of the absolute values of the difference of the x and y coordinates. For example, in the figure the L_1 distance between nodes f and t is $\mathrm{dist}_1(f, t) = 3+6 = 9$. For both best-first and A* define the heuristic value for each node u to be L_1 distance from u to t. For example, $h(f) = 9$. (Because the edge weights have been chosen to match the L_1 length of the edge, it is easy to verify that $h(\cdot)$ is an admissible heuristic.)

Dijkstra's Algorithm: The table below provides a trace of Dijkstra's algorithm. For each discovered node we show the value $d[u]$. At each stage (each row of the table), the node with the lowest d-value is selected next for processing. Once processed, a node's d-value

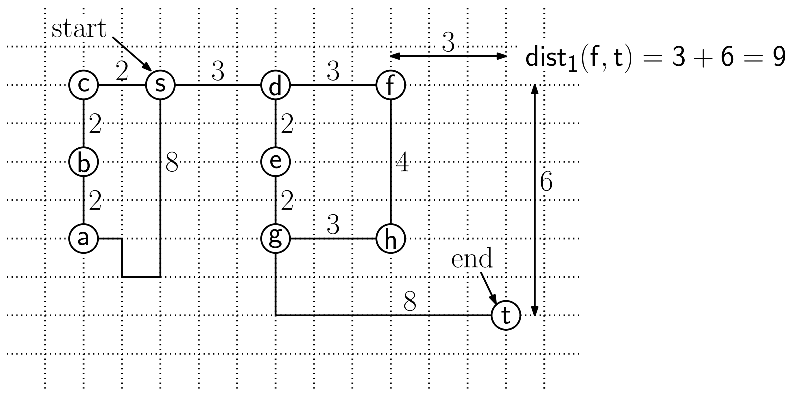

Fig. 5: The graph G used in the sample runs.

never changes (as indicated by the down arrow). Note that at Stage 6 we could have processed either a or f, since they have the same d-values.

Dijkstra's Algorithm – Each entry is $d[u]$										
Stage	$d[s]$	$d[a]$	$d[b]$	$d[c]$	$d[d]$	$d[e]$	$d[f]$	$d[g]$	$d[h]$	$d[t]$
Init	$\underline{0}$	∞	∞	∞	∞	∞	∞	∞	∞	∞
1: s	**0**	8	–	$\underline{2}$	3	–	–	–	–	–
2: c	↓	8	4	**2**	$\underline{3}$	–	–	–	–	–
3: d		8	$\underline{4}$	↓	**3**	5	6	–	–	–
4: b		6	**4**		↓	$\underline{5}$	6	–	–	–
5: e		$\underline{6}$	↓			**5**	6	7	–	–
6: a		**6**				↓	$\underline{6}$	7	–	–
7: f		↓					**6**	$\underline{7}$	10	–
8: g							↓	**7**	$\underline{10}$	15
9: h								↓	**10**	$\underline{15}$
10: t									↓	**15**
Final	**0**	**6**	**4**	**2**	**3**	**5**	**6**	**7**	**10**	**15**

An intuitive way to think about how Dijkstra's algorithm operates is to imagine fluid flooding out from the source node s along the edges simultaneously. The first node that is hit by this fluid is processed, and begins propagating the fluid along its edges as well. This idea is loosely illustrated in Fig. 6, where the different colors indicate stages of the flooding algorithm For example, first red fluid is flooded out of s. The first vertex to be hit is c (at distance 2). The next phase of flooding is indicated by dark blue. It floods out along c's edges and continues to flood along s's edges. The first vertex to be hit by the blue flood is vertex d, which is processed at a distance of 3 units. While Dijkstra's algorithm does not explicitly track these flows, it processes nodes in the order in which the flooding fluid reaches the nodes.

Best-First Search: The table below shows the trace of best-first search. For each discovered node we show the value $d[u] : h(u)$. At each stage, the discovered node with the smallest h-value (underlined) is chosen to be the next to be processed. Once processed, a node's d-value never changes (as indicated by the down arrow).

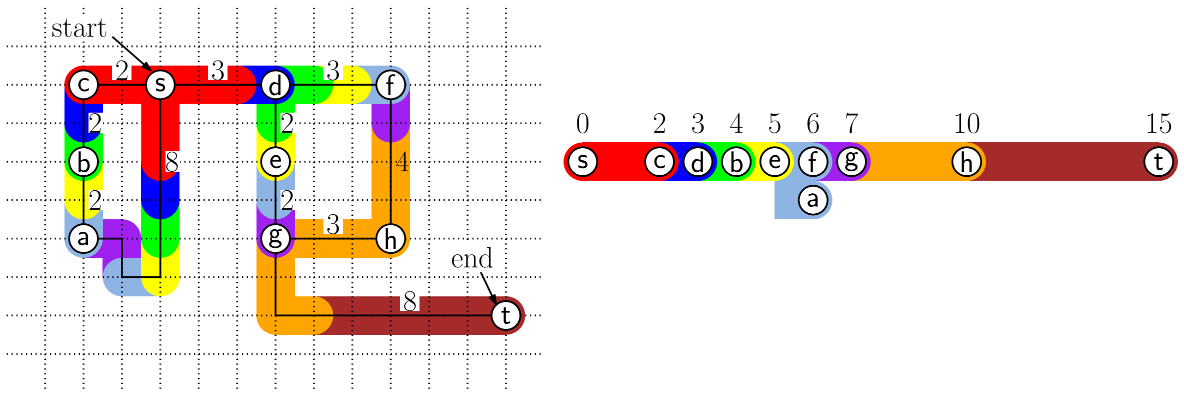

Fig. 6: Illustrating the propagation of distance information in Dijkstra's algorithm.

Best-First Search – Each entry is $d[u] : h(u)$										
Stage	$d[s]$	$d[a]$	$d[b]$	$d[c]$	$d[d]$	$d[e]$	$d[f]$	$d[g]$	$d[h]$	$d[t]$
$h(u)$	15	13	15	17	12	10	9	8	5	0
Init	0:$\underline{15}$	∞:13	∞:15	∞:17	∞:12	∞:10	∞:9	∞:$\underline{8}$	∞:5	∞:0
1: s	**0**	8:13	–	2:17	3:$\underline{12}$	–	–	–	–	–
2: d	$\downarrow$	8:13	–	2:17	**3**	5:10	6:$\underline{9}$	–	–	–
3: f		8:13	–	2:17	$\downarrow$	5:10	**6**	–	10:$\underline{5}$	–
4: h		8:13	–	2:17		5:10	$\downarrow$	13:$\underline{8}$	**10**	–
5: g		8:13	–	2:17		5:10		**13**	$\downarrow$	21:$\underline{0}$
6: t		8:13	–	2:17		5:10		$\downarrow$		**21**
Final	**0**	**8**	∞	**2**	**3**	**5**	**6**	**13**	**10**	**21**??

Not that Best-first determines that $d[t] = 21$, which is *incorrect* (it should be 15).

A* search: The table below shows the trace of the A* algorithm. For each discovered node we show the value $d[u] : h(u)$. At each stage, the discovered node with the smallest value of $d[u] + h[u]$ (underlined) is chosen to be the next to be processed. Once processed, a node's d value never changes (as indicated by the down arrow). Note that at Stages 3, 4, and 5 we have a choice of nodes to process next since there are multiple nodes with the same $d[u] + h(u)$ values.

A* Search – Each entry is $d[u] : f(u)$										
Stage	$d[s]$	$d[a]$	$d[b]$	$d[c]$	$d[d]$	$d[e]$	$d[f]$	$d[g]$	$d[h]$	$d[t]$
$h(u)$	15	13	15	17	12	10	9	8	5	0
Init	0:15	∞:13	∞:15	∞:17	∞:12	∞:10	∞:9	∞:8	∞:5	∞:0
1: s	**0**	8:13	–	2:17	$\underline{3{:}12}$	–	–	–	–	–
2: d	$\downarrow$	8:13	–	2:17	**3**	$\underline{5{:}10}$	6:9	–	–	–
3: e		8:13	–	2:17	$\downarrow$	**5**	$\underline{6{:}9}$	7:8	–	–
4: f		8:13	–	2:17		$\downarrow$	**6**	7:8	–	$\underline{15{:}0}$
5: t		8:13	–	2:17			$\downarrow$	7:8	–	**15**
Final	0	8	∞	2	3	5	6	7	∞	15

Note that the algorithm computes the correct result, but it terminates after just five stages, not ten as was the case for Dijkstra's algorithm.

A* search can also be thought of as simulating fluid propagation, but the analogy is not as clear as in Dijkstra's algorithm. The order in which nodes are processed depends now on two things. First, the distance that the fluid needs to travel from s to reach the node (as given by the value $d[u]$ for node u) and the heuristic value $h[u]$. In Fig. 7, we illustrate the fluid flows as we did for Dijkstra's algorithm, and we use a colored line to indicate the length of the $h[u]$ value. In this case, because we are using the sum of horizontal and vertical distances as the heuristic values, these heuristic paths consist of a single horizontal and a single vertical edge.

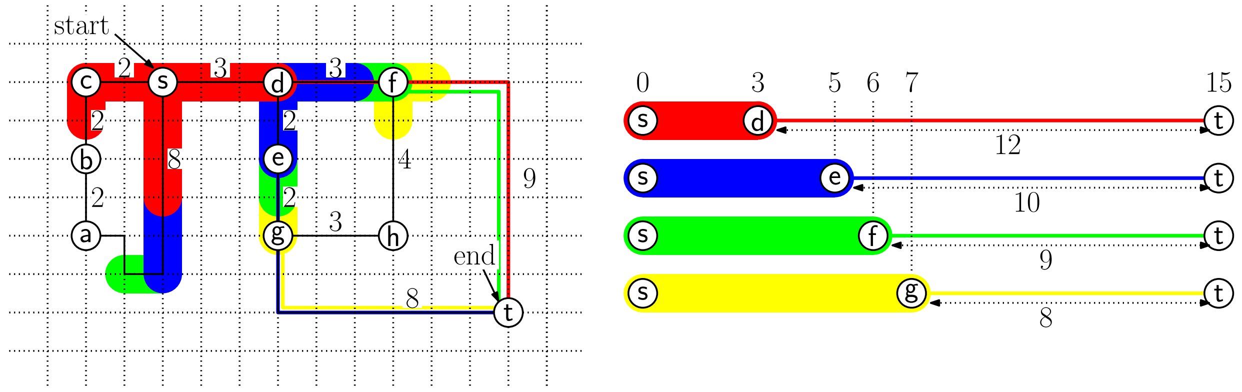

Fig. 7: Illustrating the propagation of distance information in A* search.

A* with inadmissible heuristic: The table below shows the trace of the A* algorithm using the heuristic $h'(u) = 10 \cdot \text{dist}_1(u,t)$, which is *not admissible* for this input. For each discovered node we show the value $d[u] : h'(u)$. At each stage, the discovered node with the smallest value of $d[u] + h'(u)$ (underlined) is chosen to be the next to be processed. Once processed, a node's d value never changes (as indicated by the down arrow).

A* Search – Each entry is $d[u] : f'(u)$										
Stage	$d[s]$	$d[a]$	$d[b]$	$d[c]$	$d[d]$	$d[e]$	$d[f]$	$d[g]$	$d[h]$	$d[t]$
$h(u)$	150	130	150	170	120	100	90	80	50	0
Init	0:150	∞:130	∞:150	∞:170	∞:120	∞:100	∞:90	∞:80	∞:50	∞:0
1: s	**0**	8:130	–	2:170	3:120	–	–	–	–	–
2: d	↓	8:130	–	2:170	**3**	5:100	6:90	–	–	–
3: f		8:130	–	2:170	↓	5:100	**6**	–	10:50	–
4: h		8:130	–	2:170		5:100	↓	13:80	**10**	–
5: g		8:130	–	2:170		5:100		**13**	↓	21:0
6: t		8:130	–	2:170		5:100		↓		**21**
Final	**0**	**8**	∞	**2**	**3**	**5**	**6**	**13**	**10**	21??

Observe that this heuristic boosts the h-values so high that they dominate when computing the f-values. As a result, the algorithm effectively determines which nodes to process based on the h-values alone, and so the algorithm behaves in essentially the same manner as best-first search, and computes the same incorrect result.