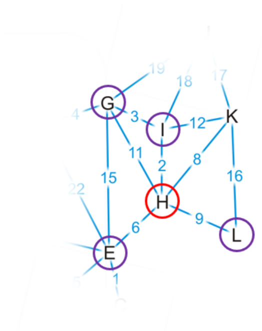
上海科技大学
ShanghaiTech University

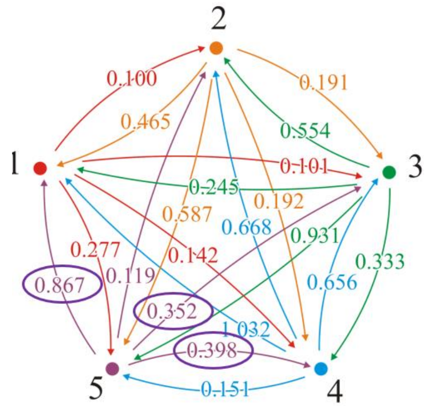
Algorithms and Data Structures

Week 10

Shortest Path



Vertex	Visited	Distance	Previous
A	F	∞	Ø
B	F	∞	Ø
C	F	∞	Ø
D	F	∞	Ø
E	F	**14**	**H**
F	F	∞	Ø
G	F	**19**	**H**
H	**T**	**8**	**K**
I	F	**10**	**H**
J	F	17	K
K	T	0	Ø
L	F	16	K



Acknowledgement:

The slide is modified from

- *Fan Rui's CS140@ShanghaiTech, Fall 2018*
- *TA discussion CS101@ShanghaiTech, Fall 2018*
- *Lecture CS101@ShanghaiTech, Fall 2019*
- *KeWei Tu's CS181@ShanghaiTech, Fall 2018*

Shortest Path

Today’s topic:

- SSSP: The Dijkstra’s algorithm
  - Introduce the motivation of relaxation and what it is
  - Introduce the algorithm and its time complexity with binary heap
  - Introduce the correctness of it(prepare for Prof. Dengji, Zhao ‘s algorithm section)
- APSP: The Floyd-Warshall algorithm
  - Introduce the algorithm
- A*: An improvement of Dijkstra’s algorithm
  - A quick proof
  - Applications

Shortest Path

Today’s topic:

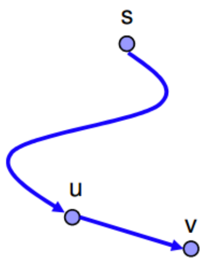
- SSSP: The Dijkstra’s algorithm
  - Introduce the motivation of relaxation and what it is
  - Introduce the algorithm and its time complexity with binary heap
  - Introduce the correctness of it(prepare for Prof. Dengji, Zhao ‘s algorithm section)
- APSP: The Floyd-Warshall algorithm
  - Introduce the algorithm
- A*: An improvement of Dijkstra’s algorithm
  - A quick proof
  - Applications

Shortest Path

- Given a weighted directed graph, one common problem is finding the shortest path between two given vertices.
- This is different from the minimum spanning tree.

SSSP: Dijkstra’s algorithm

Idea *shortest distance from s to v ≤ some distance from s to u + distance from u to v*



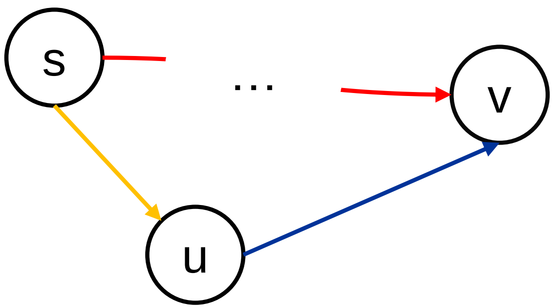
Given source s and node v

- Let $\delta(s, v)$ be length of a shortest path from s to v
- Let $\mathrm{d}(s, v)$ be an estimate of $\delta(s, v)$, where $\mathrm{d}(\mathrm{s}, \mathrm{v}) \geq \delta(s, v)$.
  - $\mathrm{d}(s, v)$ is the shortest distance as for our best knowledge!
  - Intuitively, we can update $\mathrm{d}(s, v)$ by iterations to “push” it to $\delta(s, v)$.
  - How? By relaxation!
    - In lecture, Prof. Yuyao, Zhang has guided you how it works by specific example of Dijkstra’s algorithm(find the shortcut!)

SSSP: Dijkstra's algorithm

Relaxation Given a neighbor u of v,

$$d(s,v) \leftarrow \min(d(s,v),\ d(s,u) + w(u,v))$$




```
RELAX(u, v, w)
if v.d > u.d + w(u, v)
    v.d = u.d + w(u, v)
    v.π = u
```

- $\delta(s,v)$ *be length of a shortest path from s to v*
- $d(s,v)$ *is the shortest distance we known*

Relaxation compares two paths and picks a better one:

- A shortest path we known(an estimate)
- Another one path which passes u, then reaches v directly

SSSP: Dijkstra's algorithm

SSSP: single source shortest paths

We will iterate $|V|$ times:

- Find the unvisited vertex ***v*** that has a minimum distance to it
- Mark it as visited
- Consider its every adjacent vertex ***w*** that is unvisited:
  - Is the distance to ***v*** plus the weight of the edge (***v***, ***w***) less than our currently known shortest distance to ***w*** ?
  - If so, update the shortest distance(relaxation) to ***w*** and record ***v*** as the previous pointer

Continue iterating until all vertices are visited or all remaining vertices have a distance of infinity

```
DIJKSTRA(G, w, s)
INITIALIZE-SINGLE-SOURCE(G, s)
S = Ø
Q = G.V
while Q ≠ Ø
    u = EXTRACT-MIN(Q)
    S = S ∪ {u}
    for each vertex v ∈ G.Adj[u]
        RELAX(u, v, w)
```

SSSP: Dijkstra's algorithm

Questions:

- What if at some point, all unvisited vertices have a distance ∞?
  - This means that the graph is unconnected
  - We have found the shortest paths to all vertices in the connected subgraph containing the source vertex
- What if we just want to find the shortest path between vertices v_j and v_k?
  - Apply the same algorithm, but stop when we are <u>visiting</u> vertex v_k
- Does the algorithm change if we have a directed graph?
  - No

SSSP: Dijkstra's algorithm

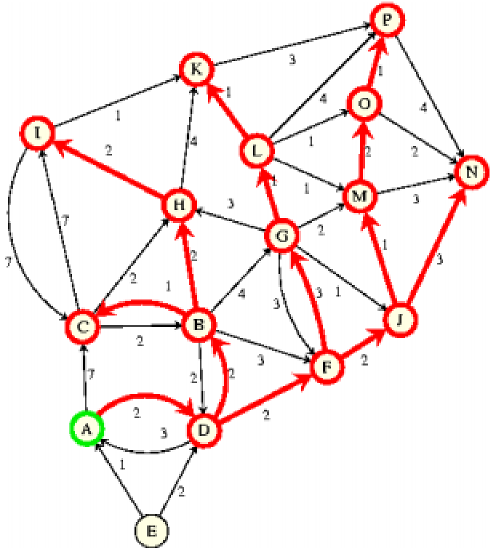
Questions:

- How to re-construct the shortest path?
  - Recall we record previous pointer when each relaxation happens

Shortest path tree There exists a tree T rooted at s such that the shortest path from s to any node v lies in T.

- Each node v has a parent in the tree.
- By following parent pointers starting from v,

we find shortest path from s to v.



SSSP: Dijkstra's algorithm

The initialization requires $\Theta(|V|)$ memory and run time

- Iterating through the table requires $\Theta(|V|)$ time
- Each time we find a vertex, we must check all of its neighbors
  - With an adjacency matrix, the run time is
    - $\Theta(\ |V|(|V|(\text{find closest}) + |V|(\text{relaxation}))\ ) = \Theta(|V|^2)$
  - With an adjacency list, the run time is
    - $\Theta(\ |V|^2\ (\text{find closest}) + |E|(\text{relaxation})\ ) = \Theta(|V|^2)$ as $|E| = O(|V|^2)$

SSSP: Dijkstra's algorithm

Can we do better?

- How about using a priority queue to find the closest vertex?
  - Assume we are using a binary heap
  - Thus, the total run time is O(|V| ln(|V|) + |E| ln(|V|))
  - O(|V| ln(|V|)) : find closet vertex and pop it(recall time complexity for a binary heap)
  - O(|E| ln(|V|))) : for each edge, we do a relaxation, which influences the estimated distance for a vertex. We should maintain the heap.
    - Naïve solution: delete it, decrease it then insert it.
      - It is called decrease the key in heap context (the key is the estimated distance in shortest path context. Relaxation decreases the estimation)

SSSP: Dijkstra's algorithm

Can we do better then better?

Dijkstra's algorithm for positive edge weights, in $O((|V| + |E|) \log |V|)$ time by binary heap(has discussed in last page).

Improve to $O(|V| \log |V| + |E|)$ using Fibonacci heap

Using a Fibonacci heap, the $O(|E|)$ decrease keys each take $O(1)$, on average, which is used to be $\log |V|$ (omit details here. Clarifying it refers to an advanced topic: amortization analysis).

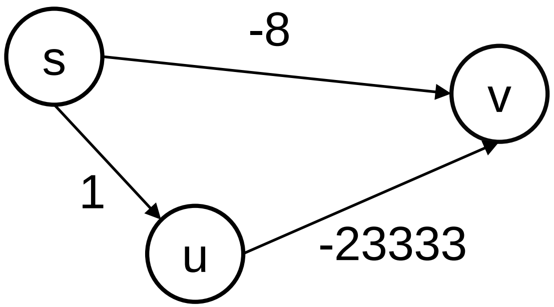
SSSP: Dijkstra's algorithm

Negative cycles

- A negative weight cycle is a cycle in the graph, s.t. the sum of all weights on cycle is negative.

- If a graph has a negative weight cycle reachable from the source, then shortest paths are not well defined
  - We can repeated go around the cycle to get arbitrarily short paths

- Dijkstra's algorithm assumes all weights are nonnegative(why?)
- Counter example:



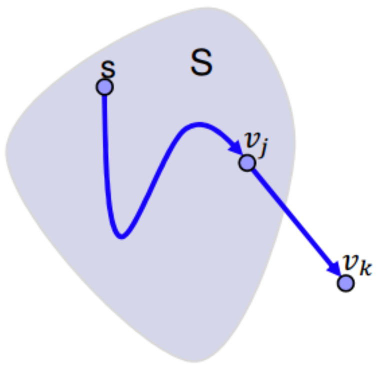
SSSP: Dijkstra's algorithm

Correctness

Idea Order nodes by increasing distance from source, as $s, v_1, \cdots, v_n$. s is source. The larger index indicates larger distance of that vertex to the source.

$$\delta(v_1, s) \leq \delta(v_2, s) \leq \cdots \leq \delta(v_n, s)$$

Lemma In round k of Dijkstra's algorithm, node v_k is settled(founded shortest path, visited), and S = $\{s, v_1, \cdots, v_k\}$. S is the set of visited vertexes.



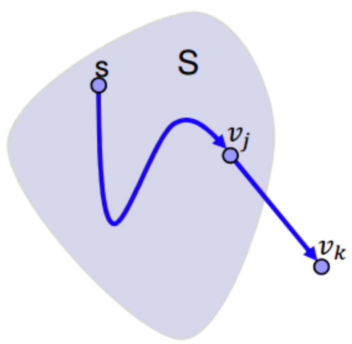
SSSP: Dijkstra's algorithm

Correctness

Lemma In round k of Dijkstra's algorithm, node v_k is settled(founded shortest path, visited), and S = $\{s, v_1, \cdots, v_k\}$. S is the set of visited vertexes.

Proof

1. In round 0, s is settled.
2. Assume lemma on round k-1 establishes, then we prove lemma on k (induction).



Consider shortest path from s to v_k, and let u be node preceding v_k in path.

- Then $\delta(s, v_k) = \delta(s, u) + w(u, v_k)$

Since w(.) > 0, the equation indicates that:

- $\delta(s, v_k) \geq \delta(s, u)$

By increasing distance notation, $\exists\ j < k$: $u = v_j$.

Otherwise, vertex k is not connected to source(trivial, not consider)

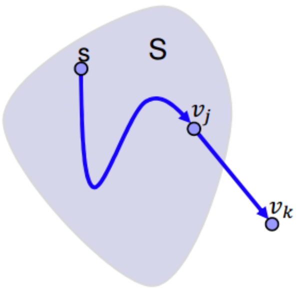
SSSP: Dijkstra's algorithm

Correctness

Lemma In round k of Dijkstra's algorithm, node v_k is settled(founded shortest path, visited), and S = $\{s, v_1, \cdots, v_k\}$. S is the set of visited vertexes.

Proof (cont.)

2. Assume lemma on round k-1 establishes, then we prove lemma on k (induction).



By induction, $v_j.d$ (the estimated distance) = $\delta(s, v_j)$ (the optimal distance).

- v_j is processed in round j (induction, j<k). v_k is its neighbor and being relax:
    - $v_k.d \leq v_j.d(= \delta(s, v_j)) + w(v_j, v_k)$ since $\mathrm{v_k.d} = \min(v_k.d, v_k.d + w(v_j, v_k))$.

- Since $u = v_j$ therefore, $v_k.d \leq \delta(s, u) + w(v_k, u) = \delta(s, v_k) \Rightarrow \mathrm{v_k.d} = \delta(s, v_k)$ after relaxation in round k.

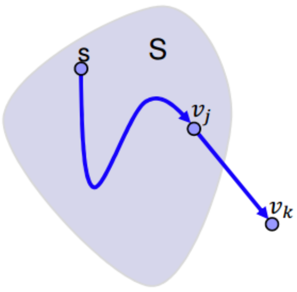
SSSP: Dijkstra's algorithm

Correctness

Lemma In round k of Dijkstra's algorithm, node v_k is settled(founded shortest path, visited), and S = $\{s, v_1, \cdots, v_k\}$. S is the set of visited vertexes.

Proof (cont.)

2. Assume lemma on round k-1 establishes, then we prove lemma on k (induction).



$\mathrm{v_k.d} = \delta(s, v_k)$ after relaxation in round k.

And in round k, v_k is sure to be selected.

- Since by induction, $V - S = \{v_k, v_{k+1}, \cdots\}$ in round k before selection of closet vertex. S = $\{s, v_1, \cdots, v_k\}$

q.e.d

Recursively apply Lemma, then correctness of Dijkstra's algorithm is guaranteed

Shortest Path

Today's topic:

- SSSP: The Dijkstra's algorithm
  - Introduce the motivation of relaxation and what it is
  - Introduce the algorithm and its time complexity with binary heap
  - Introduce the correctness of it(prepare for Prof. Dengji, Zhao 's algorithm section)
- APSP: The Floyd-Warshall algorithm
  - Introduce the algorithm
- A*: An improvement of Dijkstra's algorithm
  - A quick proof
  - Applications

APSP: Floyd-Warshall algorithm

ASSP: all pairs shortest paths

First, let's consider only edges that connect vertices directly:

$$d_{i,j}^{(0)} = \begin{cases} 0 & \text{If } i = j \\ w_{i,j} & \text{If there is an edge from } i \text{ to } j \\ \infty & \text{Otherwise} \end{cases}$$

Here, $w_{i,j}$ is the weight of the edge connecting vertices i and j

– Note, this can be a directed graph; *i.e.*, it may be that

APSP: Floyd-Warshall algorithm

The calculation is straight forward:

```
for ( int i = 0; i < num_vertices; ++i ) {
    for ( int j = 0; j < num_vertices; ++j ) {
        d[i][j] = std::min( d[i][j], d[i][k-1] + d[k-1][j] );
    }
}
```

APSP: Floyd-Warshall algorithm

What Is the Shortest Path?

Let us store the next vertex in the shortest path. Initially:

$$p_{i,j} = \begin{cases} \varnothing & \text{If } i = j \\ j & \text{If there is an edge from } i \text{ to } j \\ \varnothing & \text{Otherwise} \end{cases}$$

APSP: Floyd-Warshall algorithm

When we find a shorter path, update the next node(relaxation):

$$p_{i,j} = p_{i,k}$$

```
// Initialize the matrix p
//    ...

for ( int k = 0; k < num_vertices; ++k ) {
    for ( int i = 0; i < num_vertices; ++i ) {
        for ( int j = 0; j < num_vertices; ++j ) {
            if ( d[i][j] > d[i][k] + d[k][j] ) {
                p[i][j] = p[i][k];
                d[i][j] = d[i][k] + d[k][j];
            }
        }
    }
}
```

Takes O(n^3) time overall.

Shortest Path

Today's topic:

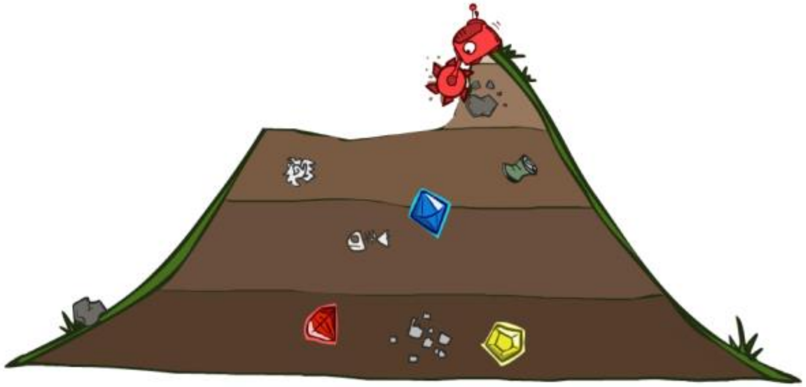
- SSSP: The Dijkstra's algorithm
  - Introduce the motivation of relaxation and what it is
  - Introduce the algorithm and its time complexity with binary heap
  - Introduce the correctness of it(prepare for Prof. Dengji, Zhao 's algorithm section)
- APSP: The Floyd-Warshall algorithm
  - Introduce the algorithm
- A*: An improvement of Dijkstra's algorithm
  - A quick proof
  - Applications

A* Search

SSSP is to find shortest paths from source to all vertexes in graph. When there is a ***goal*** vertex and regard weights as ***cost***, it can be reframed as a searching problem:

Search the goal with lowest cost

(Dijkstra halts when visit the goal vertex)



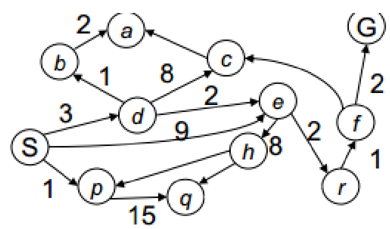
A* Search

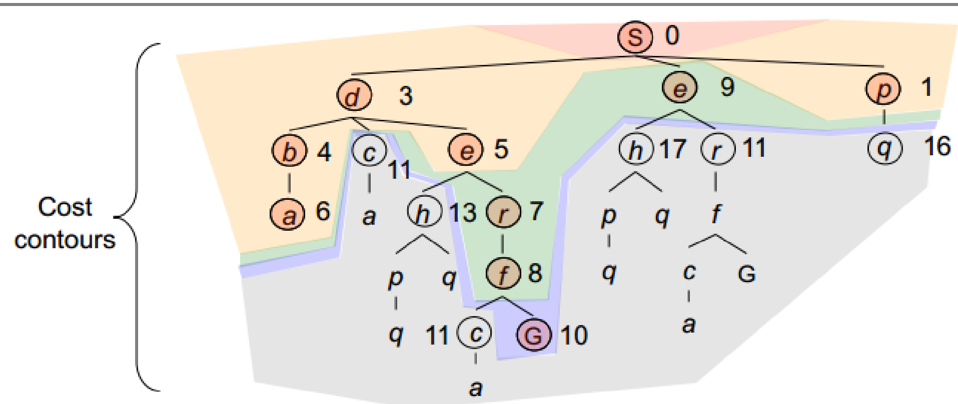
The strategy employed by Dijkstra: expand the cheapest node first is also called:

Uniform Cost Search

Strategy: expand a cheapest node first:

Fringe is a priority queue (priority: cumulative cost)





In some context, UCS can access visited vertexes(trivial). 'Visit' is called 'expand' here.

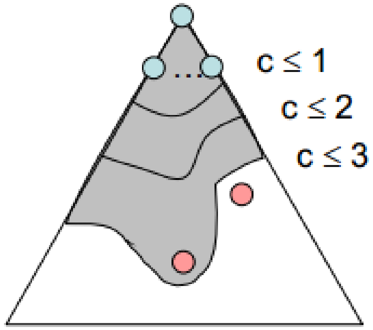
A* Search

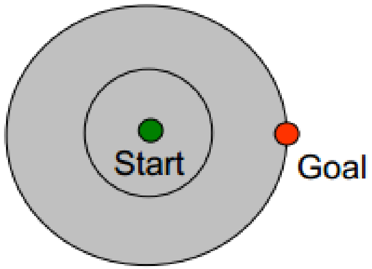
Uniform Cost Search

- The good: UCS is complete and optimal!

- The bad:
  - Explores options in every “direction”
  - No information about goal location

- We’ll fix that soon!

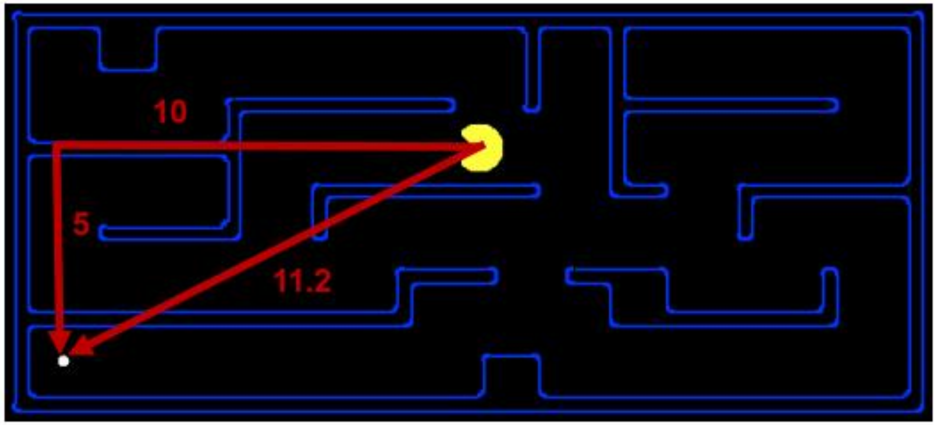


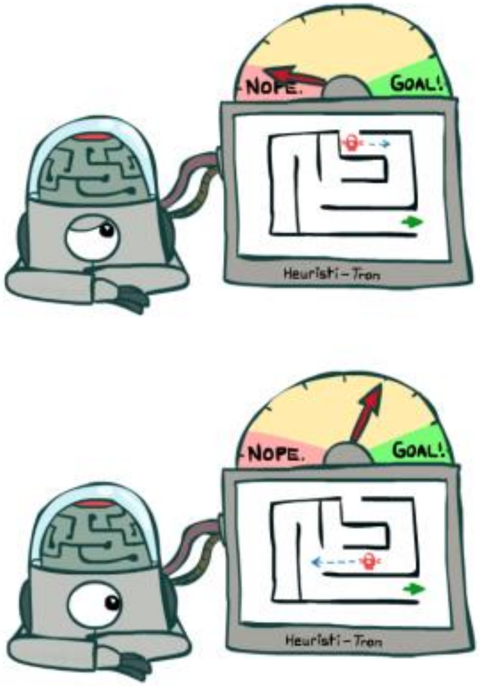


A* Search

Search heuristics

- A heuristic is:
  - A function that *estimates* how close a state is to a goal
  - Designed for a particular search problem
  - Examples: Manhattan distance, Euclidean distance for pathing

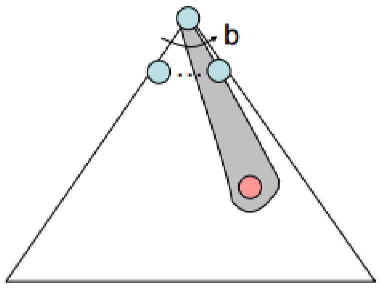




A* Search

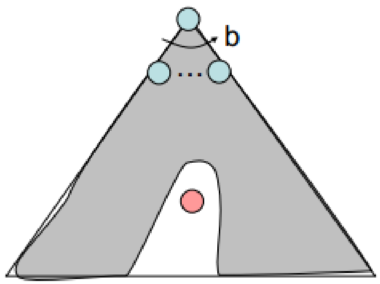
Greed Search

Strategy: visit(expand) the node that you think is closet to a goal state (lowest heuristics value).



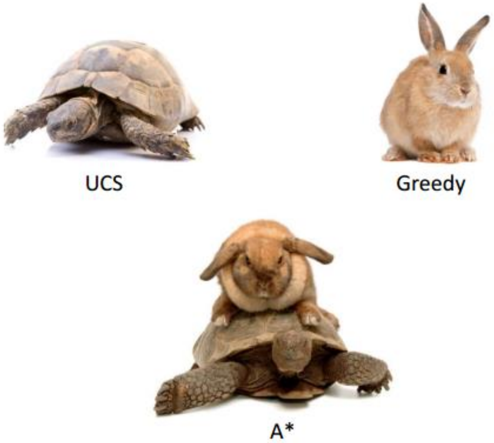
The ideal scenario: best-first takes you straight to the goal.

Worst-case: like a badly-guided DFS



A* Search

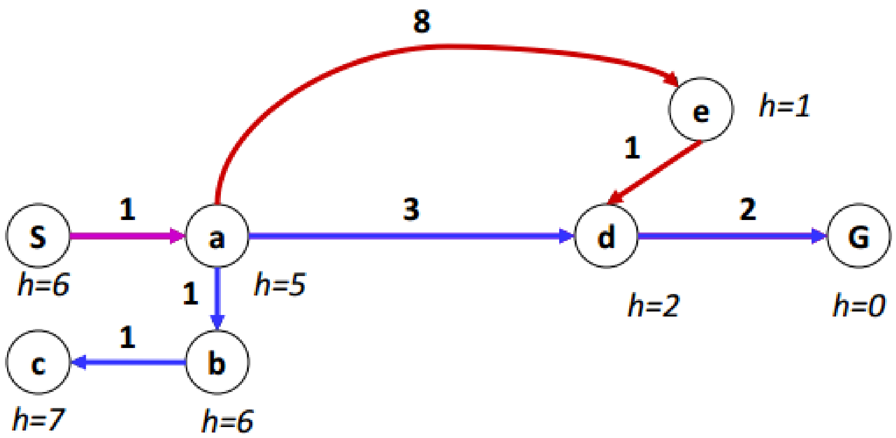
A* Search

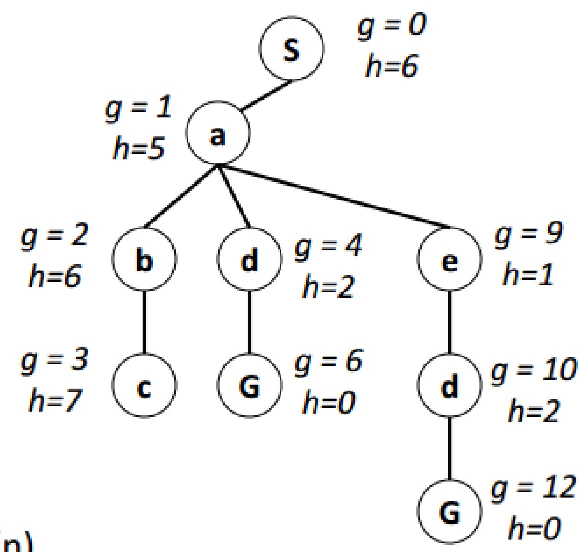


A* Search

UCS v.s. Greedy

- Uniform-cost orders by path cost, or *backward cost* g(n)
- Greedy orders by goal proximity, or *forward cost* h(n)



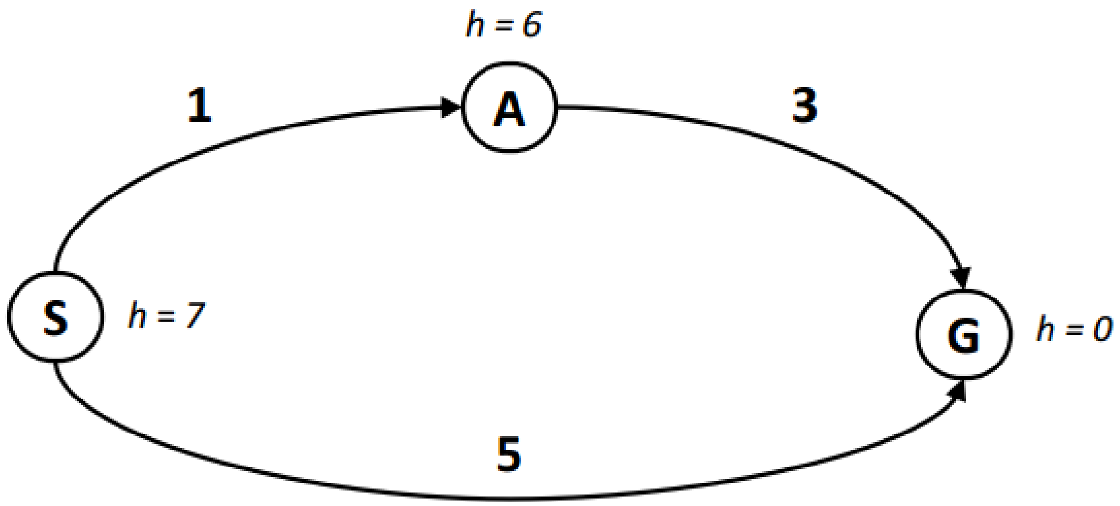


- A* Search orders by the sum: f(n) = g(n) + h(n)

Example: Teg Grenager

A* Search

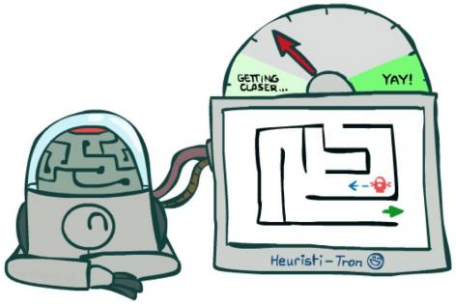
Optimal?



- What went wrong?
- Over-estimated goal cost

A* Search

Admissible Heuristics

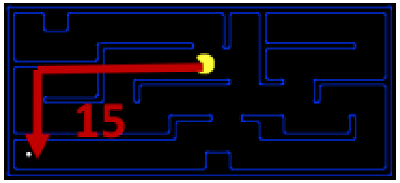


- A heuristic h is *admissible* (optimistic) if:

$$0 \leq h(n) \leq h^*(n)$$

  where $h^*(n)$ is the true cost to a nearest goal

- Examples:



- Coming up with admissible heuristics is most of what's involved in using A* in practice.

A* Search

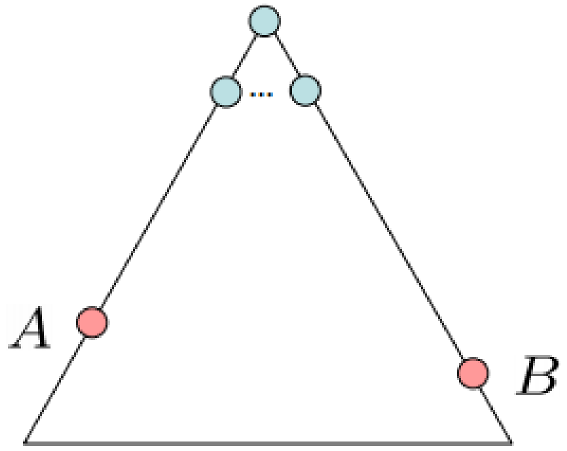
Optimality of A* with admissible heuristics

Assume:

- A is an optimal goal node
- B is a suboptimal goal node
- h is admissible

Claim:

- A will exit the fringe before B



Remark1: fringe is the vertexes(nodes) adjacent to visited vertexes

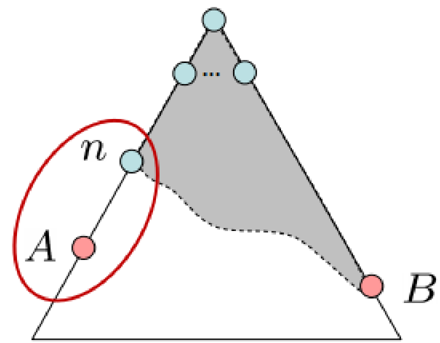
Remark2: A, B here may mean different paths to goal(relaxed from different parents). Also, goal could also be a set of nodes in A* setting.

A* Search

Optimality of A* with admissible heuristics

Proof:

- Imagine B is on the fringe
- Some ancestor *n* of A is on the fringe, too (maybe A!)
- Claim: *n* will be expanded before B
  1. f(n) is less or equal to f(A)



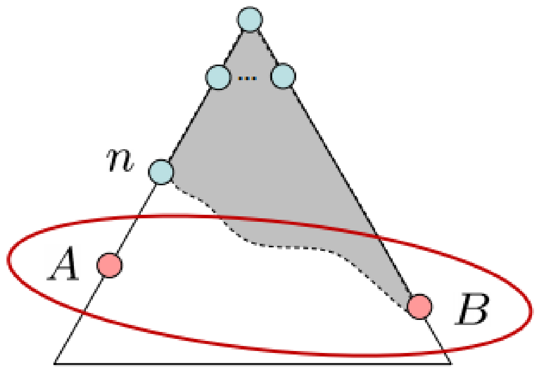
$f(n) = g(n) + h(n)$	Definition of f-cost
$f(n) \leq g(A)$	Admissibility of h
$g(A) = f(A)$	h = 0 at a goal

A* Search

Optimality of A* with admissible heuristics

Proof:

- Imagine B is on the fringe
- Some ancestor *n* of A is on the fringe, too (maybe A!)
- Claim: *n* will be expanded before B
  1. f(n) is less or equal to f(A)
  2. f(A) is less than f(B)



$g(A) < g(B)$ B is suboptimal

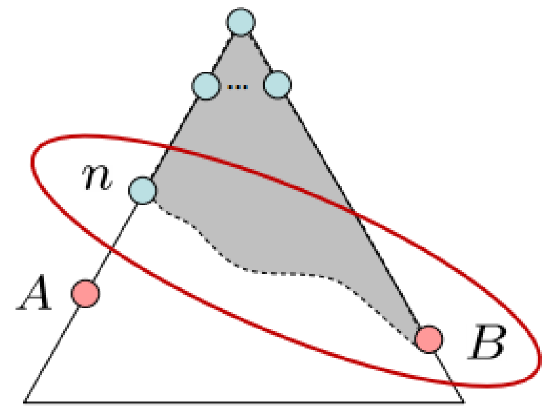
$f(A) < f(B)$ h = 0 at a goal

A* Search

Optimality of A* with admissible heuristics

Proof:

- Imagine B is on the fringe
- Some ancestor *n* of A is on the fringe, too (maybe A!)
- Claim: *n* will be expanded before B
  1. f(n) is less or equal to f(A)
  2. f(A) is less than f(B)
  3. *n* expands before B
- All ancestors of A expand before B
- A expands before B
- A* search is optimal

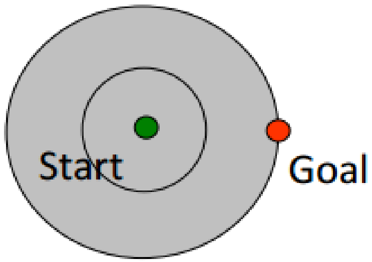


$$f(n) \leq f(A) < f(B)$$

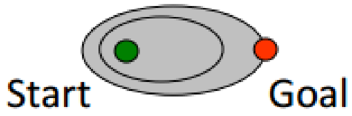
A* Search

Optimality of A* with admissible heuristics

- Uniform-cost expands equally in all "directions"



- A* expands mainly toward the goal, but does hedge its bets to ensure optimality



A* Search

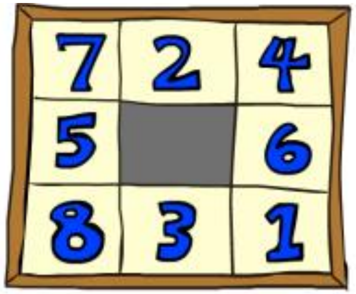
Applications

- Video games
- Pathing / routing problems
- Resource planning problems
- Robot motion planning
- Language analysis
- Machine translation
- Speech recognition
- …

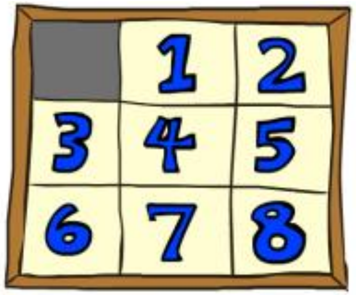
A* Search

Applications

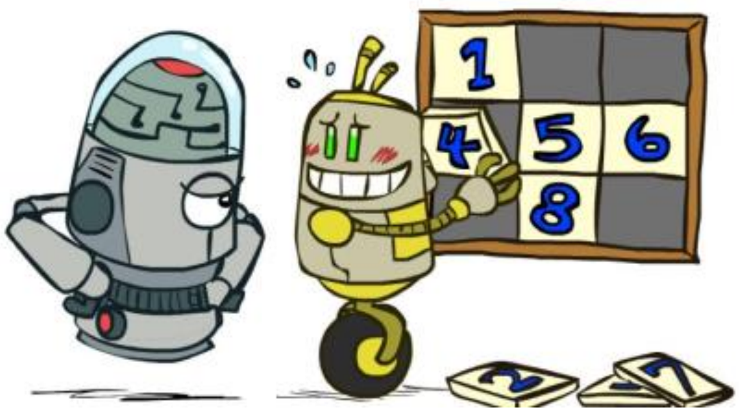
- Heuristic: Number of tiles misplaced
- h(start) = 8
- Is it admissible?
- This is a *relaxed-problem* heuristic



Start State



Goal State



	Average nodes expanded when the optimal path has…		
	…4 steps	…8 steps	…12 steps
UCS	112	6,300	3.6×10^6
TILES	13	39	227

Thank you !

Goodbye for data structure section and say hello world to algorithm part!