Advanced Games Programming (AI) Part 1: Introduction, Map Representations, Path Planning

Michael Buro

December 23, 2024

[Under Construction]

Change Log

- [Oct-03] added proofs related to consistent heuristic claims
- [Sep-15] fixed typo in Dijkstra's algorithm
- [Sep-04] created

Outline

1. Course Introduction
2. Path Planning
3. Map Representations
4. Graphs
5. Reachability: Breadth-First Search
6. Find Shortest Path to All Other Vertices: Dijkstra's Algorithm
7. Find Shortest Path Between Two Vertices: A*

Course Logistics [AI-Lec 1 L1]

Instructor: Michael Buro (ATH 337, **mburo@ualberta.ca**)

Course web page: **skatgame.net/mburo/courses/350**

Content id/password: **c350 bar350z**

Three **face-to-face** sessions per week:

- Lectures **CAB 273, TR 11:00-12:20**
- Labs **CSC 153/159, T 14:00-16:50** (starting next week)

Consider taking notes — electronic material may be incomplete

[going through course outline on web page first ...]

Game AI Overview

In this course we concentrate on AI methods that can be applied to abstract and modern video games

Main AI Objectives:

- Aiding human players by delegating cumbersome tasks to AI system. E.g.,
  - Move group of units from X to Y along shortest path
  - Workers gather resources (commute between minerals and command center)
  - Attack town with a large number of units from multiple angles
- Create challenging adversaries and collaborators
  - Strategic AI (e.g., where to build what when?)
  - Tactical AI (e.g., small-scale combat)
  - Push the state-of-the-art of AI in general, challenge human experts
  - Can help designers to balance game parameters and find bugs

Reality Check (1)

State of video game AI? Big A - small i ...

Video game AI systems are currently mostly scripted

I.e., programmer/designer thinks about strategy and tactical behaviour and implements it in form of if-then-else rules, finite state machines, hierarchical task networks, behaviour trees, etc.

E.g., “If I am attacked, then gather forces and launch counter attack”

Advantages:

- doesn’t require lots of CPU time ...
  Important, because gfx still gets majority of cycles
- can mimic expert behaviour directly

Problems: predictable, can’t deal well with unforeseen events

Reality Check (2)

We don’t have good AI solutions that require only a few CPU cycles, yet

But the games industry wants to create challenging games NOW, and still invests most CPU/GPU cycles into graphics and physics simulations

Pragmatic "Solution":

> Let the AI cheat (e.g., faster movement, fewer resources required, leak game state information (no fog of war), etc.)

Not satisfactory from an academic viewpoint

How to Improve Game AI?

Organize video game AI competitions!

- Fair, no cheating
- Play games that people enjoy. This way we can benchmark programs by playing against human experts.
- Build strong tatical and strategic AI systems that play video games well
- Apply lessons learned to real-world problems

In the coming game AI lectures we will focus on

- Map Representations
- Path Planning
- Search Space Abstractions
- Adversarial Search, Imperfect Information, MCTS, ...

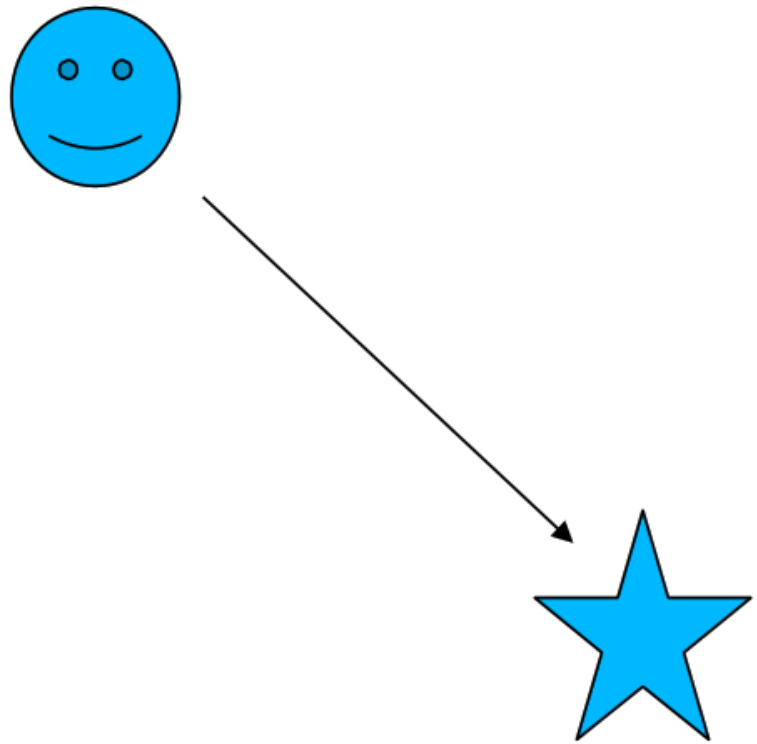
and see how these techniques can be applied to video games

Why C++?

- A better C
  - Much easier to write bug-free C++ code (e.g., constr./destructors)
  - Easier maintenance of big projects (classes support data/method encapsulation)
  - Large libraries available (e.g., generic container classes and algorithms)
- Fast program execution
  (no interpreter or JIT compiler, like Python/Java)
  - More memory efficient than Java (object data stored sequentially)
  - 2-3x faster than Java in typical AI applications
  - Typically 10+ faster than Python
- Language of choice for game engines

Path Planning [AI-Lec 2 L2]

- Want to get some object from one point to another, avoiding obstacles
- Robotics: non-point object, needs to avoid obstacles by some margin
- Games: needs to be very fast and use little memory



Goal of Path Planning Algorithms

- Find (nearly) optimal path, where optimal usually means quickest
- Obey constraints (e.g., object size, fuel limit, exposure to enemy fire, real-time)
- Terrain features and some interactions with the environment can be expressed in terms of gaining or losing time
  - Moving on highways vs. swamps
  - Destructible obstacles along the way
- Tradeoff between search complexity and path quality

Local Path Planning

- Path planning algorithms must be able to deal with dynamic obstacles
- Adding / removing objects can be expensive in abstractions or geometry-based systems
- Can use simple object avoidance methods that try to follow high-level paths and resolve local conflicts

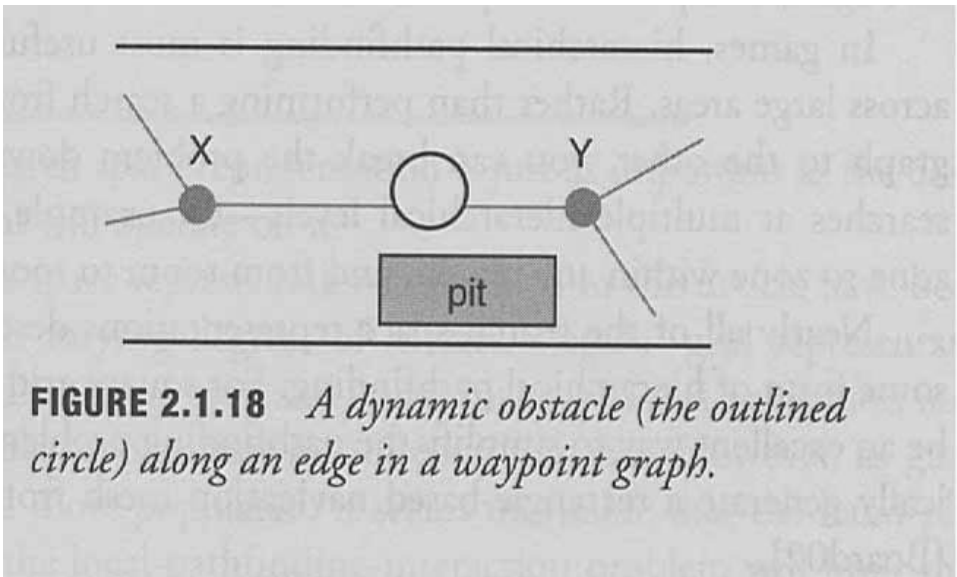


FIGURE 2.1.18 *A dynamic obstacle (the outlined circle) along an edge in a waypoint graph.*

Map Representations

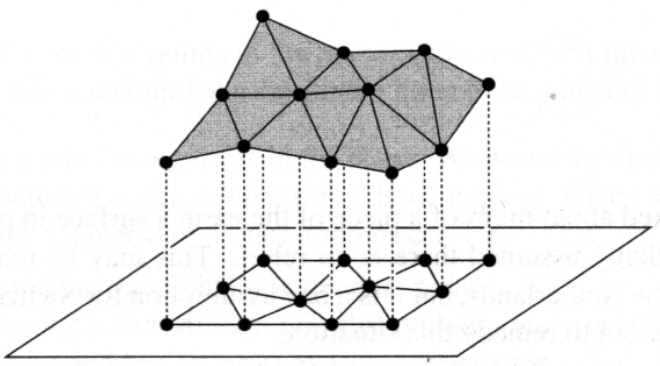
- Path planning algorithm is only half the picture
- Underlying map representation and data structures are just as important
- Important design questions:
  - Are optimal paths required?
  - Is the world static or dynamic?
  - Are worlds known ahead of time?
  - Are there real-time constraints?
  - How much memory is available?

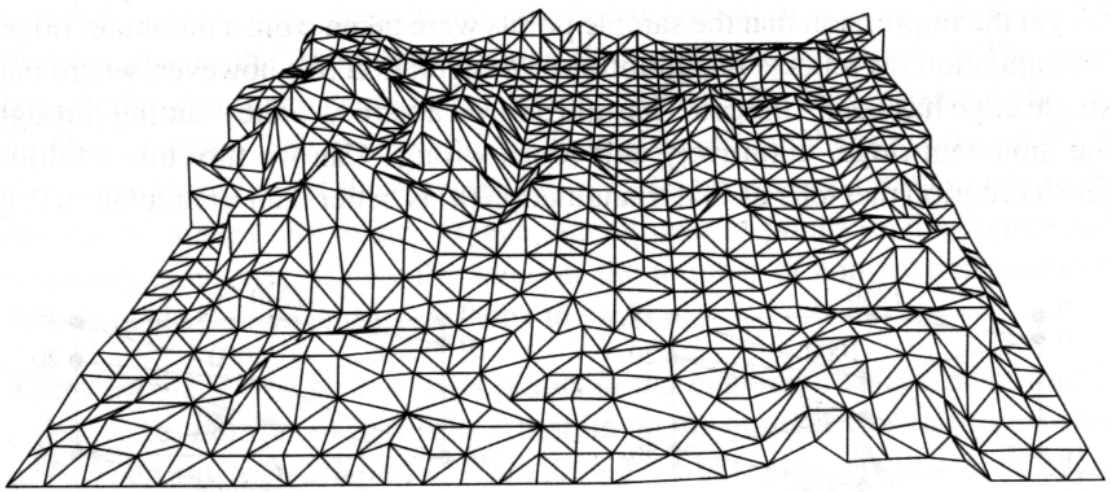
State Space Generation

- Worlds can be huge
- Like to avoid cumbersome task of picking way points or room abstractions manually
- Should be automatically generated from world geometry

Planning Paths in Continuous Spaces

- Main approach: discretize continuous height field to create search graph
- Objects move on 2d surface, so mapping height field to plane is sufficient





Regular Grids

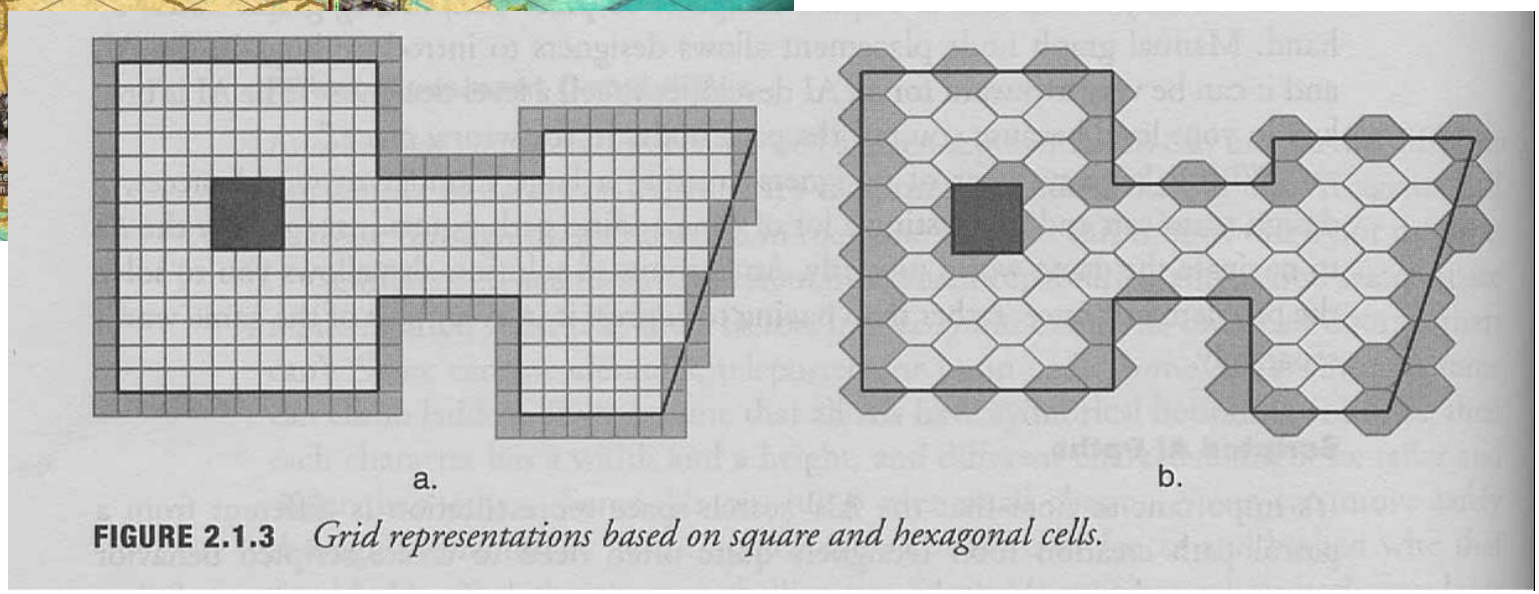
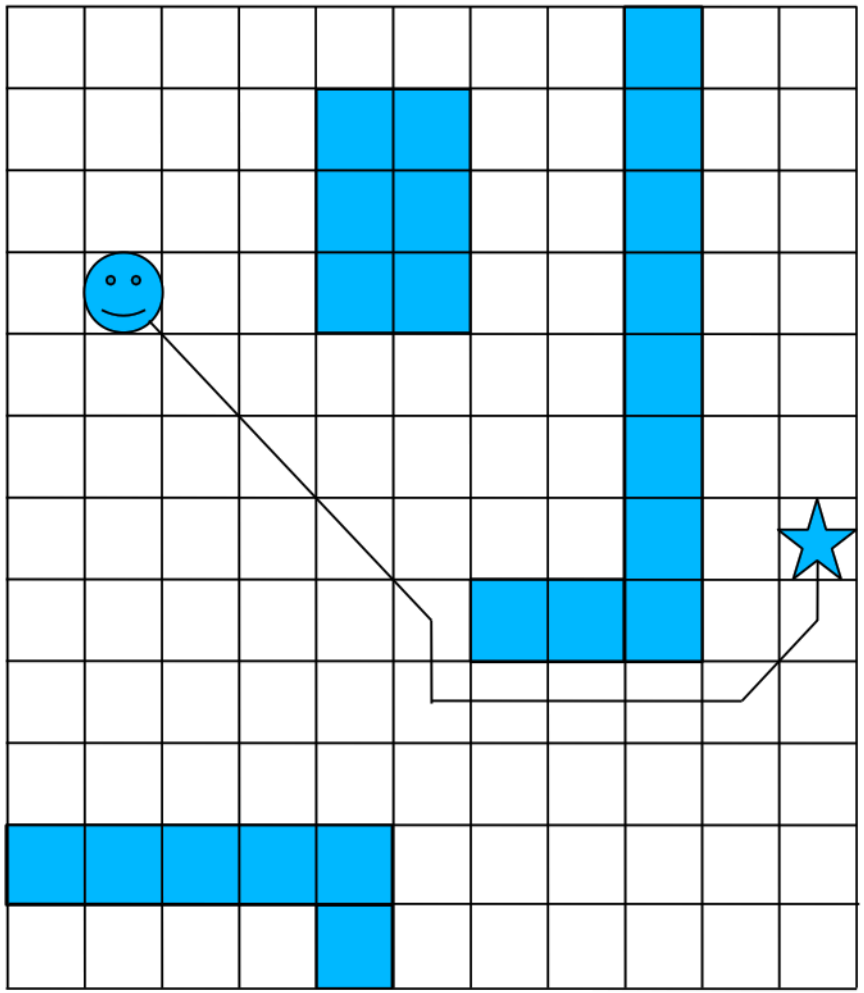


FIGURE 2.1.3 *Grid representations based on square and hexagonal cells.*

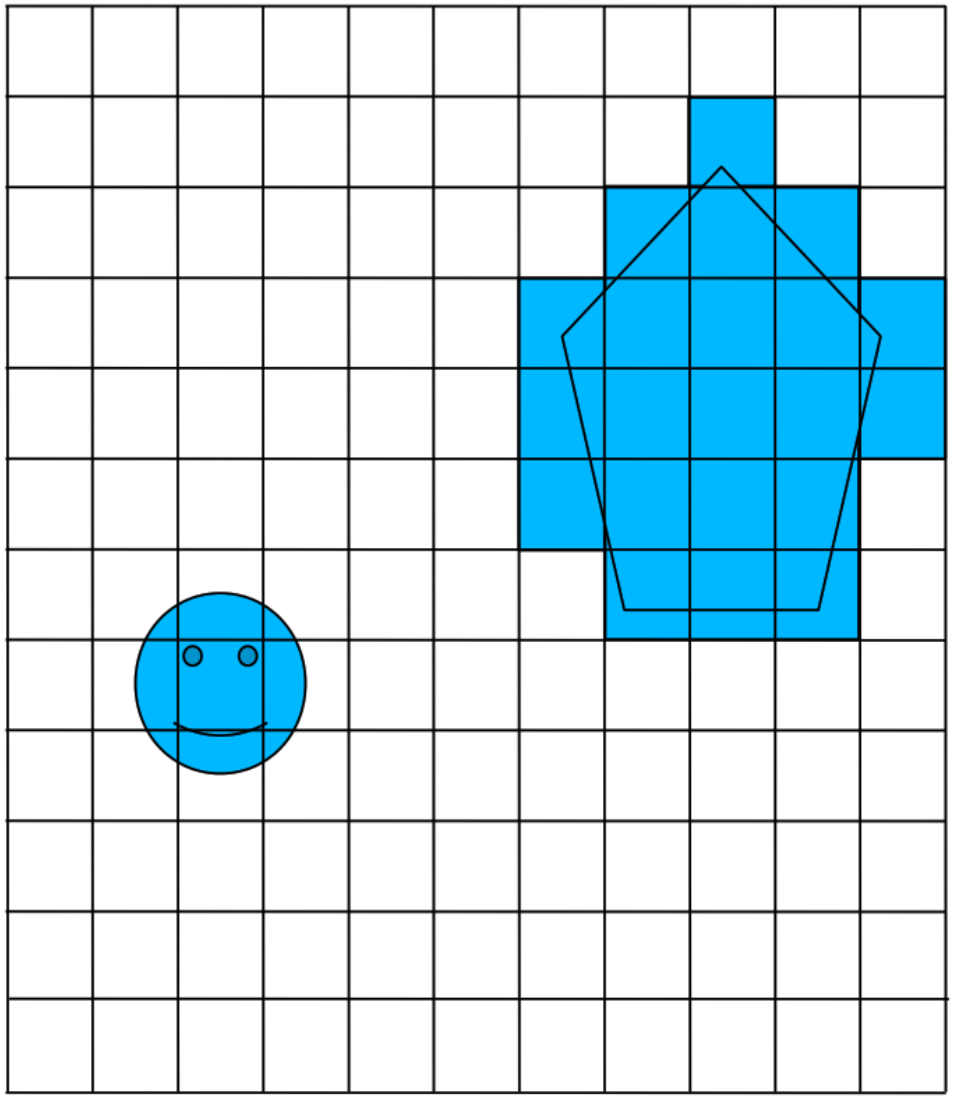
Grid-Based Methods: Advantages

- Conceptually simple representation
- Local changes have only local effects —- well-suited for dynamic environments
- Perfectly represents tile-based environments
- Exact paths easy to determine for cell-sized objects



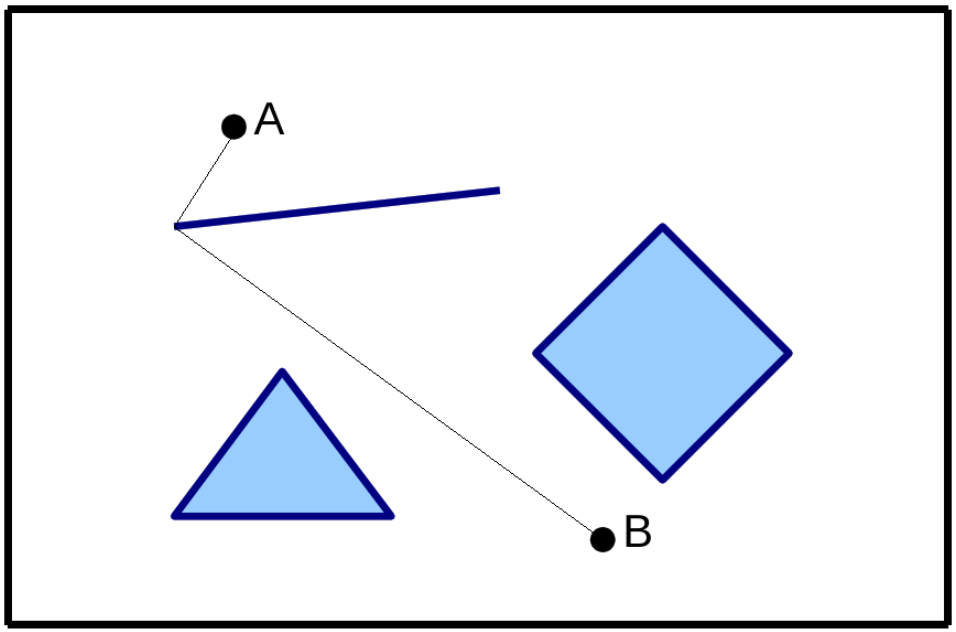
Grid-Based Methods: Disadvantages

- Imprecise representation of arbitrary barriers
- Increased precision in one area increases complexity everywhere – potentially large memory footprint
- Awkward for objects that are not tile-sized and shaped
- Need to post-process paths if environment allows arbitrary motion angles (or tweak A*)



Geometric Representations

- World is an initially empty simple shape
- Represent obstacles as polygons, i.e., sequences of line segments (also called constraints)
- Find path between two points that does not cross constraints



Geometric Methods

Advantages

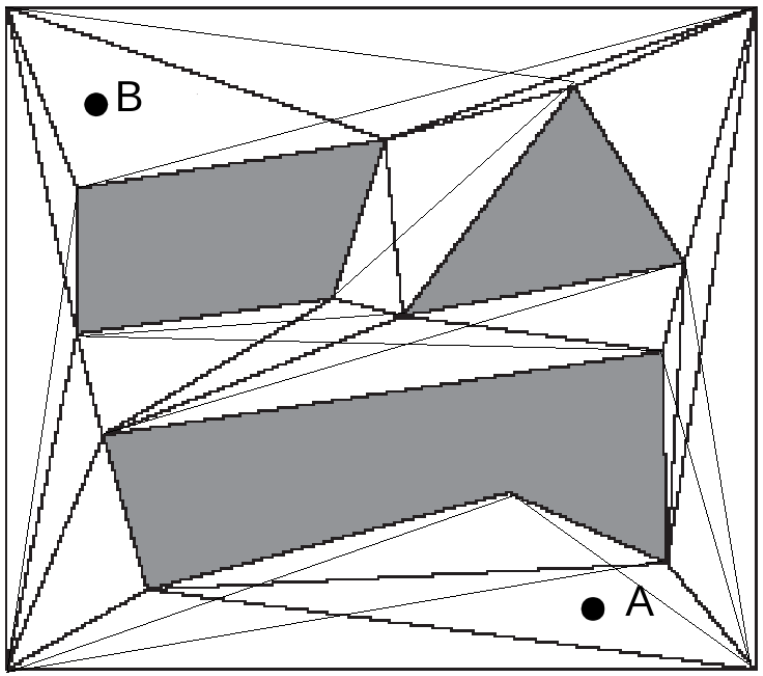
- Arbitrary polygon obstacles
- Arbitrary motion angles
- Memory efficient
- Finding optimal paths for circular objects isn't hard
- Topological abstractions

Disadvantages

- Complex code
- Robustness issues
- Point localization no longer takes constant time

Visibility Graphs

- Place nodes at corners of obstacles
- Place edges between nodes that can “see” each other
- Find path from *A* to *B*:
  - add these nodes to graph, connect to visible nodes
  - run pathfinding algorithm on resulting graph
- Path provably optimal
- But adding and changing world can be expensive as graph can be dense



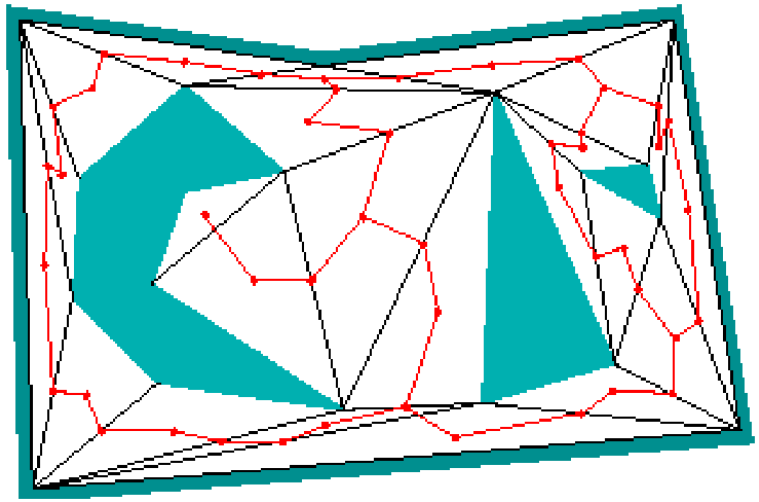
Visibility Graph Issues

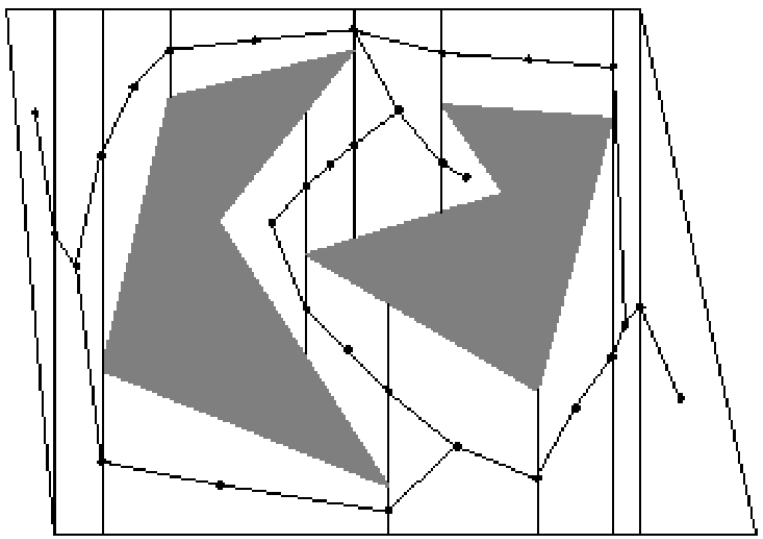
- Memory requirements: every node can potentially “see” every other, leading to quadratic space and time requirements
- Connecting the start/end nodes to the graph can already take linear time
- Also, corner-corner visibility test isn’t cheap

Not practical ...

Free-Space Decompositions

- Decompose empty areas into simple convex shapes (e.g. triangles, trapezoids)
- Create way point graph by placing nodes on unconstrained edges and in the face interior, if needed
- Connect nodes according to direct reachability
- Find path from A to B:
- Locate faces in which A, B reside
- Connect A, B to all face nodes
- Find path, smooth path





Graphs

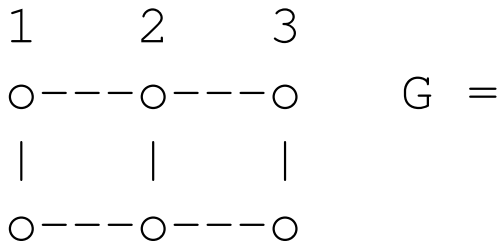
Maps in video games can be represented as graphs

Definition:

An (undirected) graph $G = (V, E)$ is a discrete mathematical object comprised of a set of vertices (or nodes) V and a set of edges (or arcs) E. An undirected edge is a set of at most two vertices $\{u, v\}$. A directed edge from u to v is an ordered pair (u, v).

Graphs describe binary relations between vertices: if edge $\{u, v\}$ is in E then we say u and v are related. Otherwise, they are unrelated. Graphs can be easily visualized:

Graph Example



$G = (V,E)$

$V = \{ 1,2,3,4,5,6 \}$

$E = \{ \{1,2\}, \{2,3\}, \{3,4\}, \{4,5\}, \{5,6\}, \{6,1\}, \{2,5\} \}$

$|V| = 6$ (6 vertices or nodes)

$|E| = 7$ (7 edges or arcs)

($|S|$ = number of elements in set S)

Step 1: Reachability [AI-Lec 3 L3]

Task:

Given a graph $G = (V, E)$ consisting of vertex set V and edge set E and two vertices s (start), and g (goal), determine whether g is reachable from s

Or more general: compute all nodes in V that are reachable from s

In games, this information is sometimes sufficient, if we don't need to know shortest paths

E.g., answering "Can I reach the base by land, or do I need to build a transport ship?"

The simpler test usually runs much faster than computing the minimal distance

Graph Traversal

Discussed

- Graph types and representations in memory ("toolbox" document)
- How breadth-first search (BFS) and depth-first search (DFS) work in principle

Queues

A queue is an abstract data structure that provides the following operations:

- isEmpty() : returns true iff (if-and-only-if) no element left in Q
- enqueue(v) : add element at the end
- dequeue() : remove front element and return it

```
| 1 | 2 | 3 |  enqueue(4) ->  | 1 | 2 | 3 | 4 |

          ... dequeue()  ->  | 2 | 3 | 4 |
```

Queues can be implemented based on doubly linked lists, arrays, vectors, circular buffers, etc.

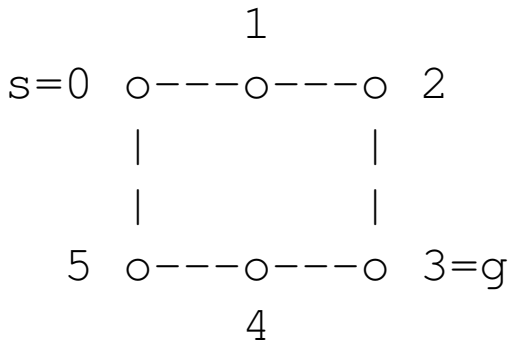
C++ provides template type `std::queue` (covered later)

Breadth-First Search (BFS)

```
// input:  graph G, start vertex s, goal vertex g
// output: whether we can reach g from s
function BFS(G,s,g) {
  if (s = g) { return true }
  create queue Q     // fringe of nodes yet to be expanded
  create set R       // set of reached nodes
  enqueue s onto Q
  add s to R
  while (Q not empty) {                  // (*)
    t ← dequeue element from Q           // (1)
    for each { t, u } in E[G] {
      if (u = g) { return true }
      if (u not in R) {
        add u to R                       // (2)
        enqueue u onto Q
      }
    }
  }
  return false
}
```

BFS Example

Here is an example of BFS in action:




```
(*)     Q       R
 1.    [0]    {0}
 2.    [1,5]  {0,1,5}
 3.    [5,2]  {0,1,2,5}
 4.    [2,4]  {0,1,2,4,5}
       g reached (neighbour of 2) => return true
```

Asymptotic Growth Rate Refresher (1)

$f(n) \in O(g(n))$ means: (read: "f of n is (in) big-O of g of n")

there exists $c > 0, n_0 \in \mathbb{N}$ such that for all $n \geq n_0 : |f(n)| \leq c|g(n)|$

We then say g is an asymptotic upper bound for f

E.g., $n \in O(n^2), 1 \in O(n), n^2 \notin O(n)$

IMPORTANT: big-O may not describe the true asymptotic behaviour, it's just an upper bound. Also, big-O and its relatives Θ ("big-Theta", true growth rate) and Ω ("big-Omega", lower bound) are sets of functions

Therefore, notations like $n = O(n^2)$ which are used frequently in the literature don't make much sense!

Asymptotic Growth Rate Refresher (2)

So, saying your algorithm runs in time $O(\log n)$ and mine in $O(n)$, so yours is asymptotically faster is NONSENSE. For this kind of argument you need to use Θ or Ω instead of O

E.g., if my algorithm runs in time $O(\log n)$ and yours in $\Omega(n)$ or $\Theta(\sqrt{n})$, then mine is asymptotically faster

For a more in-depth treatment of the big-O notation read my "toolbox" document, my related "CMPUT 204 Refresher" files, or Wikipedia

BFS Runtime

BFS Runtime Complexity:

For $G = (V, E)$ the runtime of BFS is $\Theta(|V| + |E|)$ because in the worst case every node is visited once in step (1) and all edges are visited twice in step (2). BFS uses $\Theta(|V|)$ space

This is good news. BFS's runtime is asymptotically optimal, because the least we have to do is accessing all input information

For a BFS correctness proof see [9]

Step 2: Find Shortest Path to All Other Vertices

Task:

Given a weighted graph $G = (V, E, w)$ and a start vertex s, determine the shortest distance to all reachable nodes. Weight function w assigns a weight (or distance) ≥ 0 to each edge (and by extension to paths)

While this is a bit more than just finding a shortest path between two given vertices, it is a good starting point for the A* algorithm we'll look at shortly

Dijkstra's algorithm [3] solves the single-source shortest path problem. In each iteration i it implicitly generates "reached set" S_i and maintains "minimal-distance" array d, such that

1. $d[v]$ is the minimal distance from s to v, if $v \in S_i$, and
2. the minimum distance from s to v if $v \notin S_i$ and all but the last node along the paths are in S_i

Dijkstra's Algorithm: Idea and Correctness

Using array d, Dijkstra's algorithm maintains current distances for every node

In iteration i it expands the set of visited nodes S_i by the closest node u that is reachable in one step (i.e., it picks u with minimal $d[u]$ value)

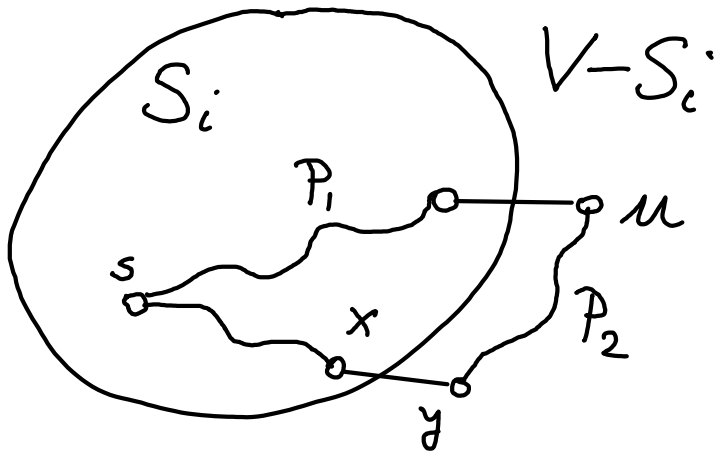
It then updates distances of the neighbours of u into S_{i+1}, which may have decreased when considering paths through u

Why is path (call it P_1) from s to u optimal?

Dijkstra's Algorithm: Idea and Correctness (continued)

Suppose there is another possible path P_2 from s to u that is shorter than P_1 (the one that directly leads to u) and also shortest overall

Then P_2 would leave the set of vertices with established minimal distance (S_i) somewhere else — say using edge (x, y)



Then:

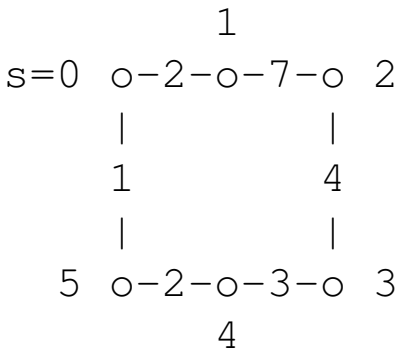
$$w(P_2) < w(P_1) \Rightarrow d[y] + w(\text{rest of } P_2 \text{ after } y) < d[u]$$
$$\Rightarrow (\text{all weights} \geq 0)\ d[y] < d[u]$$

which contradicts the algorithm's choice of u. Therefore, $w(P_2) \geq w(P_1)$, and thus P_1 is optimal

[AI-Lec 4 L5] Dijkstra's Algorithm: Pseudo Code

```
// input:  weighted directed graph G and start vertex s (weights >= 0)
// output: minimal distances from s to all vertices (d[]),
//         parent indexes for constructing shortest path (p[])
function Dijkstra(G=(V,E,w), s) {
  for each vertex v in V {    // initializations
    d[v] ← ∞                  // unknown distance from s to v
    p[v] ← -1                 // previous node on optimal path from s
  }
  d[s] ← 0                    // distance from s to s
  Q ← V                       // all nodes are unoptimized
  while (Q is not empty) {    // (*) main loop
    u ← vertex in Q with smallest distance in d
    remove u from Q
    if (d[u] = ∞) { break; }   // all remaining vertices inaccessible
    for each (u,v) in E {
      alt ← d[u] + w((u,v))    // w maps edge to distance
      if (alt < d[v]) {
        d[v] ← alt             // found shorter path to v
        p[v] ← u               // memorize parent
    } }
  }
  return (d, p)
}
```

Dijkstra's Algorithm: Example



$V = \{ 0,1,2,3,4,5 \}$

(*)	d	p	Q	S
1.	$(0,\infty,\infty,\infty,\infty,\infty)$	(-1,-1,-1,-1,-1,-1)	{0,1,2,3,4,5}	
	extract 0 (dist 0)			{0}
2.	$(0,2,\infty,\infty,\infty,1)$	(-1,0,-1,-1,-1,0)	{1,2,3,4,5}	
	extract 5 (dist 1)			{0,5}
3.	$(0,2,\infty,\infty,3,1)$	(-1,0,-1,-1,5,0)	{1,2,3,4}	
	extract 1 (dist 2)			{0,1,5}
4.	$(0,2,9,\infty,3,1)$	(-1,0,1,-1,5,0)	{2,3,4}	
	extract 4 (dist 3)			{0,1,4,5}
5.	(0,2,9,6,3,1)	(-1,0,1,4,5,0)	{2,3}	
	extract 3 (dist 6)			{0,1,3,4,5}
6.	(0,2,9,6,3,1)	(-1,0,1,4,5,0)	{2}	
	extract 2			{0,1,2,3,4,5}
7.	(0,2,9,6,3,1)	(-1,0,1,4,5,0)	{} done	

Shortest Path Tree

A shortest path tree can be constructed using the p array which stores the predecessor for each vertex (-1 indicating the root of the tree (s)):

```
  d= (0,2,9,6,3,1)     p= (-1,0,1,4,5,0)


    [0] [2] [9]
s= 0 o-2-o-7-o
     |   1   2        [x] minimal distance to s
     1
     |   4   3
   5 o-2-o-3-o
    [1] [3] [6]
```

Once a shortest path tree has been computed it can be used for instant path planning for units on a map that are ordered to congregate at the start location (assuming undirected edges)

Dijkstra's Algorithm: Runtime Complexity

If an array is used as data structure for Q, the worst-case runtime of the algorithm is $\Theta(|V|^2)$, where $|V|$ is the number of nodes in the graph. If the graph is dense (i.e., $|E| \in \Theta(|V|^2)$ then this is best possible

To speed up locating the vertex with minimum distance we can use a priority queue data structure for which extracting the minimal element and inserting a new element takes $\Theta(\log n)$ time (e.g., STL std::priority_queue) [discussed how heaps work in class]

In the implementation below, we even just keep on adding (vertex,distance) pairs to avoid having to implement decrease-key for priority queue Q for handling the case when discovering shorter paths. This comes at the cost of memory: Q can now hold up to $2|E|$ elements

With this, the worst-case runtime becomes $O(|E| \log |E|)$. For sparse graphs (i.e., $|E| \in O(|V|)$) this leads to runtime $O(|V| \log |V|)$, which is much faster than the array-based implementation

Dijkstra with Priority Queue

```
// input:  weighted directed graph G and start vertex s (weights >= 0)
// output: minimal distances from s to all vertices (d[]),
//         parent indexes for constructing shortest path (p[])
function DijkstraPQ(G=(V,E,w), s) {
  for each vertex v in V {    // initializations
    d[v] ← ∞                  // unknown distance from s to v
    p[v] ← -1                 // previous node on optimal path from s
  }
  d[s] ← 0; insert (s,0) into Q // distance from s to s = 0
  while (Q is not empty) {        // (*) main loop
    pick and remove pair (u,x) from Q with smallest distance x to s
    if (x = ∞) { break; }      // nothing else reachable
    if (x > d[u]) { continue; } // (u,x) obsolete duplicate (>= wrong!)
    for each (u,v) in E {
      alt ← d[u] + w((u,v))     // w maps edge to distance
      if (alt < d[v]) {
        d[v] ← alt              // found shorter path to v
        p[v] ← u                // memorize parent
        insert (v,d[v]) into Q  // consider v with updated distance
  } } }
  return (d, p)
}
```

[AI-Lec 5 L7] Is there a better way?

Dijkstra's original algorithm initializes the queue with all nodes in the beginning. This wouldn't work if the search space is large (or even infinite). There is another search algorithm called "Uniform Cost Search" (UCS) which is essential mimicking Dijkstra's computation using a smaller queue, which is initialised with the starting node (like our priority queue based algorithm). UCS is a special case of A* (using $h(n) = 0$), which we will look at next

Dijkstra's algorithm adds new nodes by referring to previously computed shortest distances, i.e., it only looks back

Can we do better than Dijkstra's algorithm by also looking ahead?

Perhaps we can save even more time if we are only interested in the minimal distance between TWO locations ...

Step 3: Finding Shortest Paths Between Two Vertices

Heuristic Search, also known as Single Agent Search, began in the 1960s with the A* algorithm [1], which can be thought of as a generalization of Dijkstra's algorithm that in addition to looking back, also looks ahead

For node n which represents a partial solution (or search state as opposed to problem state), we define:

$f(n) = g(n) + h(n)$, where

$g(n)$ is the cost of the path to node n so far

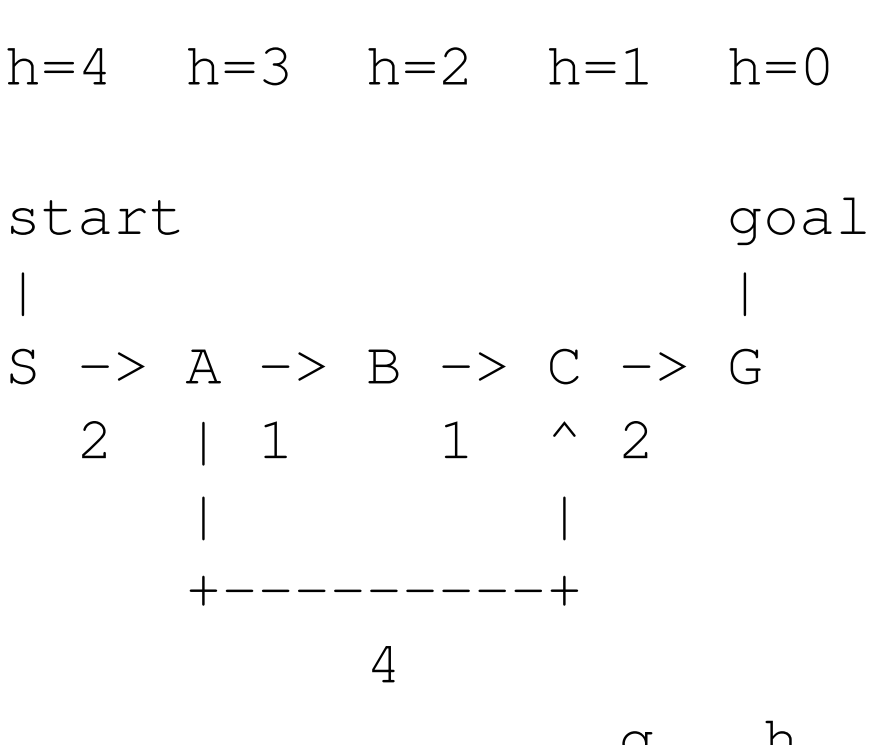
$h(n)$ is a heuristic estimate of the cost of reaching the goal from node n

A* Search: Expand node on fringe (queue) with lowest f value

$\rightarrow$ This makes A* a best-first search algorithm

For path planning, $g(n)$ is the distance traveled so far, and $h(n)$ is often the Euclidean distance between n and the goal location

A* Example



	g	h		g	h
1. Expand S (f = 0 + 4 = 4)			-> A (f = 2 + 3 = 5)		
2. Expand A -> B (f = 3 + 2 = 5), C (f = 6 + 1 = 7)					
3. Expand B -> C (f = 4 + 1 = 5 improvement, update C					
4. Expand C -> G (f = 6 + 0 = 6)			\value in queue		
5. Expand G - recognize it as goal					

A* Termination Condition

When should A* stop?

When reaching the goal OR when expanding it?

When reaching the goal for the first time there may still exist another shorter path. So we need to wait until we EXPAND the goal state (correctness proof below)

What if we revisit a state that was already expanded?
(e.g., node *C* in above example)

We might get a better solution and have to update then node's *f* value

A^* Data Structures

State:
Object that describes the current problem state. E.g., location of agent on a map for path planning, or orientation of cubies for Rubik's Cube

Node:
Object that encodes a partial problem solution

Contains: state, f, g, parent pointer (and optionally an action)

OPEN:
Container of nodes at the search fringe that still need to be expanded

CLOSED:
Container of nodes that have been expanded

OPEN + CLOSED:
Nodes we have seen so far

A* Algorithm Overview

Main Loop:

- remove lowest-*f* node n from OPEN
- n goal? yes => stop
- move n to CLOSED
- expand n: consider its children
- as far as we know, we are done with this node (but can be re-opened later)

Consider a child:

- check if state seen before (in OPEN or CLOSED):
- if state has been seen with the same or smaller *g* value, reject
- Otherwise, remove child state from OPEN and CLOSED and add corresponding node to OPEN for consideration

The OPEN/CLOSED lists act as a cache of previously seen results

A* Pseudo Code (1)

```
// input: start state, goal state
// return (success,path) if goal found, (failure,) otherwise
function A*(State s, State goal)
{
  node = new Node(state=s, g=0, h=DistEstimate(s, goal),
                  f=g+h, parent=null)
  insert node into OPEN

  while (OPEN not empty) {                    // (*)
    n = minimal-f node in OPEN
    remove n from OPEN                        // (+)
    add n to CLOSED
    s = n.state
    if (s == goal) {
      construct path from s to goal using n
      return (success, path)
    }
    for (i=0; i < NumChildren(s); ++i) {
      Consider(n, Child(s, i), goal)
    }
  }
  return (failure,)
}
```

A* Pseudo Code (2)

```
Consider(Node from, State to, State goal)
{
  newg = from.g + cost(from.state, to)

  if (node n with n.state == to in CLOSED) {
    if (n.g ≤ newg) return  // older is better
    remove n from CLOSED    // found better solution
  }
  if (node n with n.state == to in OPEN) {
    if (n.g ≤ newg) return  // older is better
    remove n from OPEN      // found better solution
  }

  node = new Node(state=to, g=newg,
            h=DistEstimate(to, goal), f=g+h, parent=from)
  add node to OPEN
}
```

Admissible Heuristics

Let $h^*(n)$ denote the true minimal cost to the goal from node n

Definition:

A heuristic h is called **admissible** if and only if $h(n) \geq 0$ and $h(n) \leq h^*(n)$ for all n

I.e., admissible heuristics never overestimate the minimal cost to the goal, they are optimisitic

Examples:

- $h(n) = 0$ is an admissible heuristic
- $h(n) = 1$ is not admissible, because $h(g)$ must be 0 for goals g
- $h(n) =$ Euclidean distance from n to goal state is admissible for finding minimal Euclidean distance paths in the plane (with obstacles or motion constraints)

[AI-Lec 6 L9] A* Correctness (1)

[Refresher: correctness proofs using loop invariants in toolbox]
In what follows, we will show that A* is correct, i.e., it terminates on all inputs and computes optimal paths when they exist

Theorem 1:

Applied to graphs with bounded node neighbourhoods and edge costs $\geq \epsilon > 0$, A* using an admissible heuristic *h* returns "success" and an cost-minimal path, if one exists, or "failure" otherwise in case the graph is finite

Proof Sketch:

We assume goal *G* is reachable from *S* (otherwise A* reports "failure" which can be proved by showing that in this case A* visits all nodes reachable from *S* – homework (induction on minimal number of steps needed to reach a node))

Define $C^* =$ minimal cost from *S* to *G*

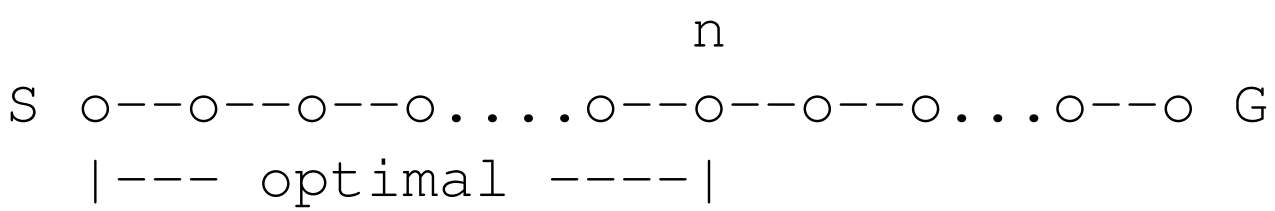
A* Correctness (2)

Lemma:

At line $(*)$ in the A* computation there is always a node n in OPEN with the following properties:

1. n is on an optimal path to G
2. A* has found an optimal path to n
3. $f(n) \leq C^*$

Illustration:



$$f(n) = g(n) + h(n) \leq C^*$$

We prove this by induction on the number of node expansions

A* Correctness (3)

Base Case:

In the beginning, S is on an optimal (empty) path and is in OPEN, and A* has found this path

Also, $h(S) \leq h^*(S)$, because h is admissible, and thus

$$f(S) = \underbrace{g(S)}_{=0} + h(S) \leq 0 + h^*(S) = C^*$$

Inductive Step:

- If the previous n is not expanded in the current step, the conditions still hold — done: just pick n again (its g value hasn't increased and thus $f(n)$ is still $\leq C^*$)
- Otherwise, if n is the goal state we are done as well (nothing to show because the loop is terminated)

A* Correctness (4)

- Otherwise, all successors will be considered and at least one (say m) will be on an optimal path because n lies on an optimal path

```
                    n   m
S o--o--o....o--o--o--o...o--o G
```

- We have found an optimal path to m, because otherwise there would be a shorter path to the goal, contradicting that we have an optimal path going through n

```
                    n   m
S o--o--o....o--o--o--o...o--o G
   \_____....._____/<- shorter? => shorter path to G!
```

- $f(m) \leq C^*$ because
  - $f(m) = g(m) + h(m)$
  - $g(m) = g^*(m)$ (minimal path length to m) because m on optimal path
  - $h(m) \leq h^*(m)$ because h is admissible

Thus: $f(m) = g(m) + h(m) \leq g^*(m) + h^*(m) = C^*$ □

A* Correctness (5)

Next we show that if A* terminates with "success" then it produced an optimal path

Suppose we expanded goal state G at which we arrived via a suboptimal path. I.e., $g(G) > C^*$, which means

$$f(G) = g(G) + h(G) = g(G) > C^* \quad (*)$$

According to the Lemma there is a node n in OPEN with $f(n) \leq C^*$, but we expanded G before n. Therefore,

$$f(G) \leq f(n) \leq C^*$$

which contradicts $(*)$. So, G will not be expanded because there is another node n in OPEN with smaller f value. As a consequence, A* when using an admissible heuristic has constructed an optimal solution when it is about to expand a goal node

A* Correctness (6)

What's left to show is that A* terminates:

- Nodes A* generates always refer to acyclic paths (no cycles) because edge weights are > 0
- In each iteration only new acyclic paths are generated because when a node is added the first time, a new path is created, and when a node is promoted a new path to that node has been found, because it is shorter than the one seen before

So, in the worst case, A* may consider every acyclic path in the search graph

Note that this graph is finite, if every node has finitely many neighbours and all edge costs are $\geq \epsilon > 0$, because the maximal distance to consider is C^* which will be reached after at most $\lceil C^*/\epsilon \rceil$ steps. Therefore, the algorithm stops

This concludes the proof sketch of Theorem 1 □

Node Expansions

Theorem 2: Using an admissible heuristic,

- A* expands all nodes n with $f(n) < C^*$
- A* might expand some of the nodes on the "goal contour" where $f(n) = C^*$
- A* does not expand any node n with $f(n) > C^*$

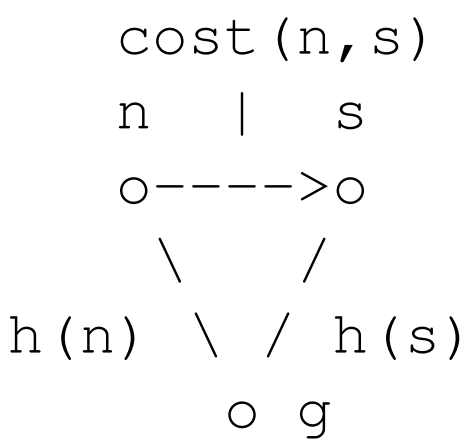
Proof: (see [1])

Consistent Heuristics (1)

Definition:

h is consistent (or monotone) iff

- $h(G) = 0$ for all goal states G and
- $h(n) \leq cost(n, s) + h(s)$ for all n and their successors s (also known as triangle inequality):



E.g., Euclidean distance in $\mathbb{R}^2$ is consistent (because of the triangle inequality)

Consistent Heuristics (2)

Theorem 3:

If $h(n)$ is consistent, then $f(n)$ is non-decreasing along any path A* considers

Proof:

Assume n' is a successor node of n. Then

$$g(n') = g(n) + \text{cost}(n, n')$$

$$\Rightarrow f(n') = g(n') + h(n') = g(n) + \text{cost}(n, n') + h(n')$$

$$\Rightarrow [\text{with } \text{cost}(n, n') + h(n') \geq h(n)] \;\; f(n') \geq g(n) + h(n) = f(n)$$

□

Consistent Heuristics (3)

Theorem 4: h consistent $\Rightarrow$ h admissible

Proof:

For some state S from which goal G is reachable assume $h(S) > h^*(S)$. Consider an optimal path from S to G. Along that path, f is monotone because h is consistent. I.e., for each node n along the path we have $f(n) \geq f(S)$, including goal G

$$
\begin{array}{rcll}
f(G) & = & g^*(G) + h(G) & \\
 & = & g^*(G) & \text{// } h \text{ consistent} \\
 & \geq & f(S) & \text{// } f \text{ monotone because } h \text{ consistent} \\
 & = & 0 + h(S) & \\
 & > & h^*(S) & \text{// assumption}
\end{array}
$$

Therefore, $g^*(G) > h^*(S)$, which is a contradiction. Consequently, our assumption was wrong, i.e. $h(S) \leq h^*(S)$ must hold for all S, and h, therefore, is admissible □

Consistent Heuristics (4)

Corollary 1: A* with consistent $h(n)$ expands nodes in non-decreasing $f(n)$ order

[Proof idea: f values are non-decreasing with each node expansion by Theorem 3, also see [8]]

Corollary 2: A* using a consistent heuristic only expands nodes it has found an optimal path to and it therefore never re-expands nodes

[This immediately follows from Corollary 1]

Consistent Heuristics (5)

Theorem 5 (#3 in [10]):

Any search algorithm based on a consistent heuristic h whose values are provided upon expanding a node will expand all nodes that are expanded by A*

In other words, A* using a consistent heuristic is an optimal search algorithm

[AI-Lec 7 L11] A* Data Structure Considerations (1)

Using linked lists or vectors for OPEN and CLOSED is certainly possible

However, the runtime of some of the queries (such as finding the minimum f value node) and updates can be linear which is often unacceptable

OPEN and CLOSED hold nodes (actually node pointers), but are queried for states. This requires complex data structures if we want those operations to run in $O(\log n)$ or even $O(1)$ time

For example, for CLOSED we could use an efficient map data structure that maps states to node pointers, thereby allowing us to quickly check whether there is a node present for a given state

For OPEN, we could use a priority queue of node pointers based on f values, in conjunction with a state-to-node-pointer map to also allow for membership queries

Data Structure Considerations (2)

If we use a consistent heuristic, A* can be simplified, because no better paths to nodes will be found by subsequent search. I.e., once closed, nodes will never be re-opened

CLOSED can be implemented as state-to-node-pointer map as before (either based on binary search trees or hash tables), but OPEN can be simplified by using a priority queue for node pointers based on f values

Simplified A* for Consistent Heuristics

These observations allow us to simplify the A* code:

```
// use priority queue Q that stores nodes and uses
// f-cost as their priority

// added right after code line (+):
if (node n with state = n.state in CLOSED) then continue

// updated Consider function
Consider(Node from, State to, State goal)
{
  if (node n with to = n.state in CLOSED) then return

  n = new Node(state=to, g=from.g + cost(from.state, to),
           h=DistEstimate(to, goal), f=g+h, parent=from)
  add n to OPEN  // allow duplicates in OPEN
}
```

Note: using $h(n) = 0$ (consistent!) this is the UCS algorithm

A* Optimizations (1)

Here are some examples of how the runtime and space requirements of A* can be optimized in practice:

- Breaking f value ties in favour of larger g values: using less heuristic approximation is empirically better
- As we have seen, priority queues allow us to find the node with minimal f value in time $O(\log n)$, where n is the number of queue elements (e.g., C++ STL priority_queue<T,Compare>)
- We can use hash tables for OPEN/CLOSED membership to avoid scanning lists, or if the entire graph fits in memory, we can associate two flags with each search state: “in OPEN?” and “in CLOSED? With this, membership test is really fast (constant time)

A* Optimizations (2) [homework]

If A* is run multiple times on the same graph (think fixed terrain and consecutive path planning operations), we can decrease the initialization cost of clearing OPEN and CLOSED (i.e., setting all flags to false) before each A* run by using the "generation counter" trick:

- each flag becomes an integer, initialized with 0 (meaning "false")
- set the generation counter (gc) to 1
- setting flag = gc indicates true, flag != gc means false
- before the next A* run, increment gc
  (this implicitly sets all flags to false in constant time!)
- set all flags to 0 if gc exceeds big value and reset gc to 1

$\Rightarrow$ virtually no initialization cost!

E.g., when using an unsigned 4 byte gc, then only after $\approx$ 4 billion A* calls all flags have to be reset. This is a nice example of trading space for time. The idea is applicable whenever one needs a very fast (constant time actually) membership test for large sets

A* Complexity

General (implicit graph): (e.g., Rubic's cube)

- Successor nodes are generated on the fly (graph defined implicitly)
- Exponential time complexity in worst case (measured in solution length)
- A good heuristic will help a lot here
- Runtime is $O(bm \cdot \log(bm))$ if the heuristic is perfect (branching factor b, length m) (why?)
- Exponential space complexity in worst case

If all nodes fit in memory (explicit graph): (e.g., video game maps)

- Time complexity often $O(n \log n)$
- (n nodes, $\log n$ for priority queue operation, assuming number of neighbours bounded by constant, and reopen operations are rare, (e.g., when using consistent heuristics))

A* in Computer Games (1)

A* can be used for path planning and single agent action planning in computer games, but there are a few problems:

- Memory: not much available, often only a few bits per node if maps are big
- Speed: need to find acceptable paths on huge maps in milli-seconds
- How about dynamic environments or opponents blocking paths?

  A simple approach for dynamic environments is to assume the world is static and to replan frequently

  However, this doesn't take into account mobile objects

  What we really want is to avoid collisions in space-time, because objects can be at the same location at different times
- ...

A^* in Computer Games (2)

However:

Finding good approximations is often sufficient and can be much faster

Idea:

Abstracting the search graph to gain speed at the cost of memory and path quality, which is next ...

References (1)

[1] P. Hart, N. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths", IEEE Trans. Syst. Sci. Cyber., Vol 4, No 2, pp 100-107, 1968
[2] Wikipedia: A* Search
[3] Wikipedia: Dijkstra's Algorithm
[4] Supreme Commander Flowfield Path Planning
https://www.youtube.com/watch?v=lOYXUktahv8
[5] Crowd Flow Path Planning
http://www.youtube.com/watch?v=lGOvYyJ6r1c
[6] Crowd Flow Paper
http://grail.cs.washington.edu/projects/crowd-flows
[7] Wikipedia: Breadth First Search
[8] More A* Proofs: see file
https://skatgame.net/mburo/courses/350/material/ai-part1-bekris_cs482_f09_notes_lec05.pdf

References (1)

[9] Cormen, Leiserson, Rivest, Stein (“CLRS”): Introduction to Algorithms
[10] R. Dechter and J. Pearl: “The Optimality of A* Revisited”, AAAI-83, pp. 95-99