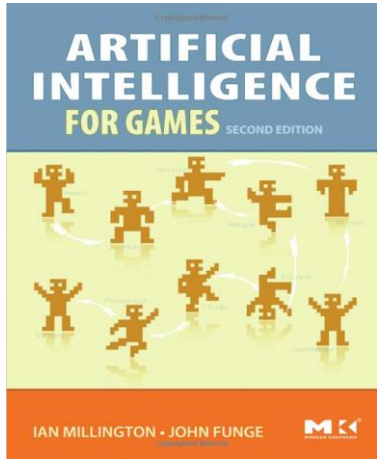
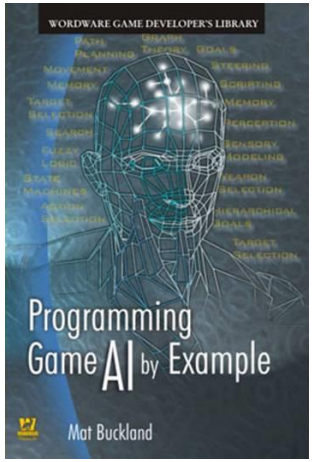


Disclaimer: I use these notes as a guide rather than a comprehensive coverage of the topic. They are neither a substitute for attending the lectures nor for reading the assigned material.



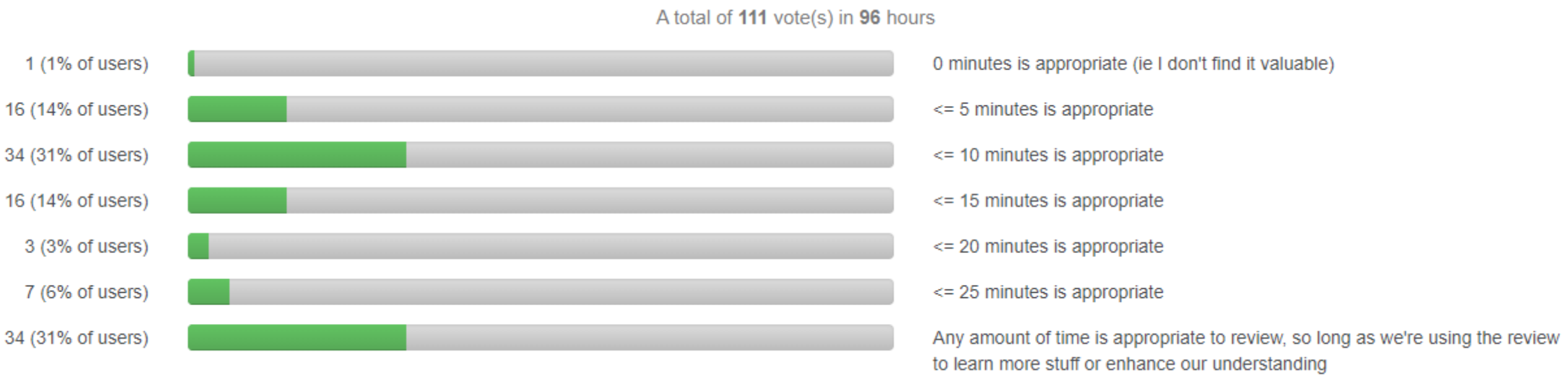
"I may not have gone where I intended to go, but I think I have ended up where I needed to be." –Douglas Adams

"All you need is the plan, the road map, and the courage to press on to your destination." –Earl Nightingale

"If I cease searching, then, woe is me, I am lost. That is how I look at it - keep going, keep going come what may." – Vincent van Gogh

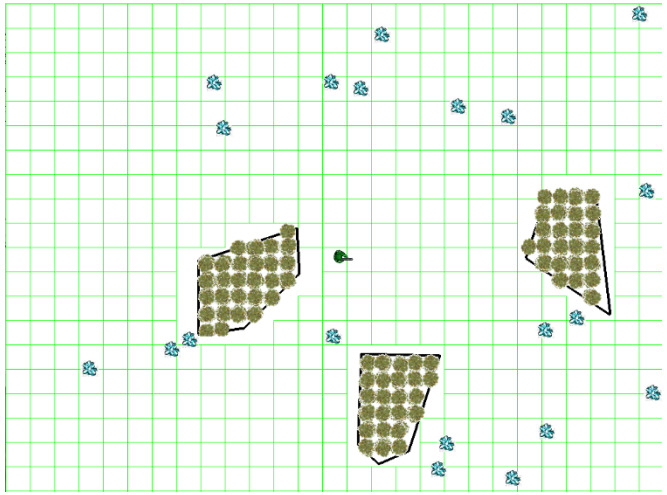
Announcements

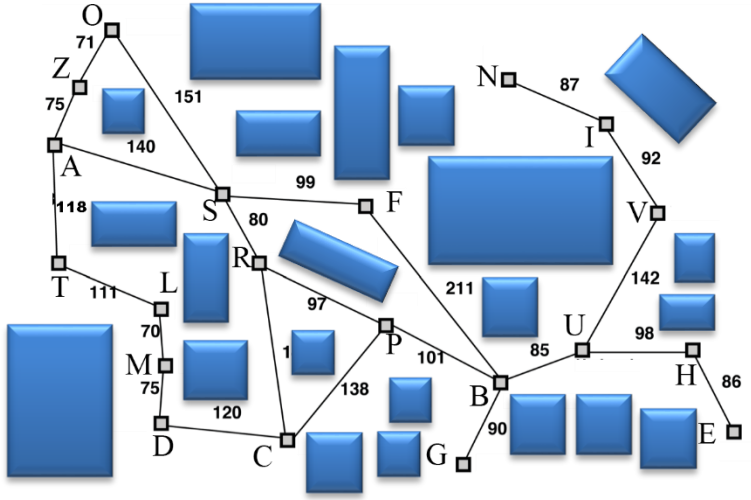
- Attendance verification: please take piazza poll!
  - https://piazza.com/class/jzhausy9bho1iu?cid=88
- Hw3 (navmesh) posted

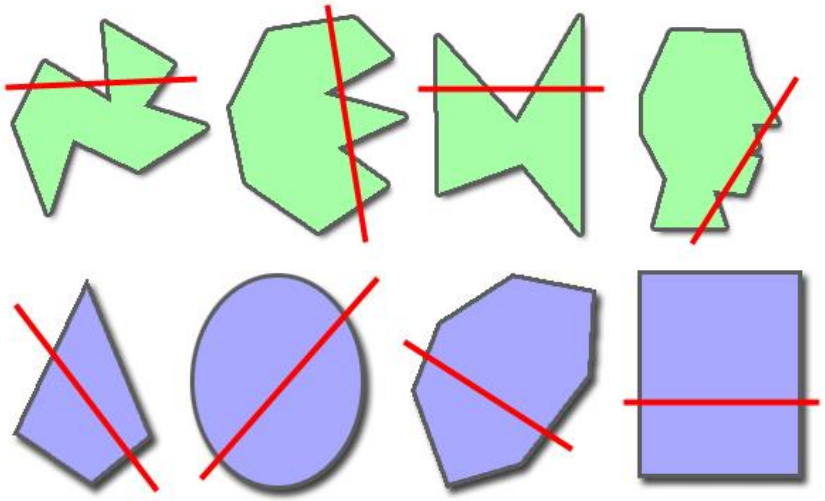


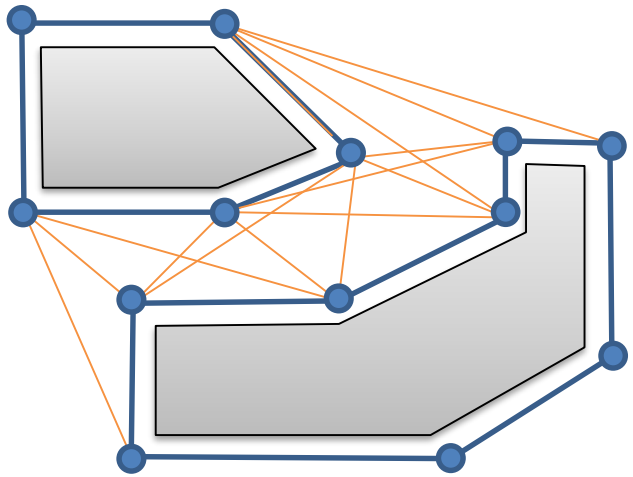
PREVIOUSLY ON…

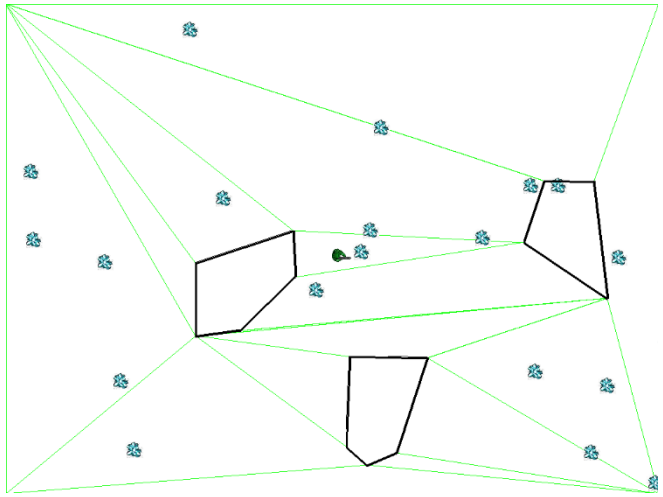
Modelling and Navigating the Game World

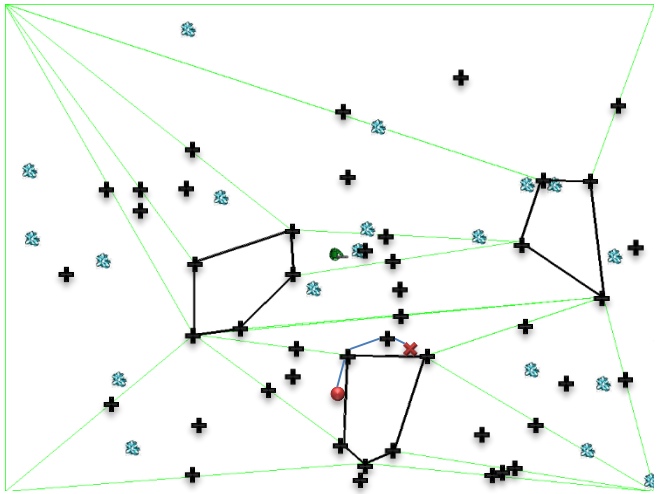




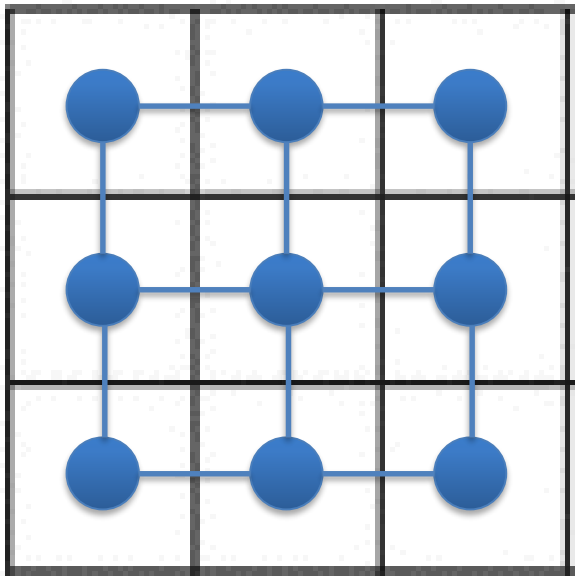


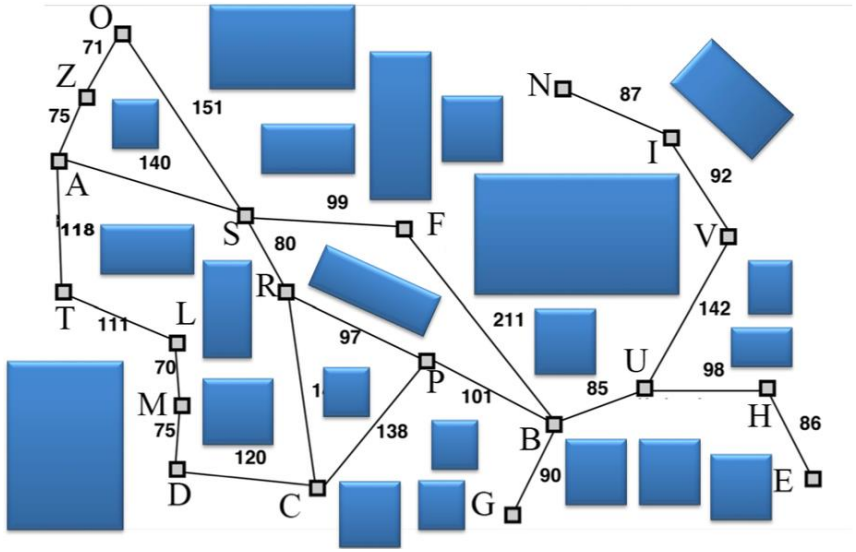


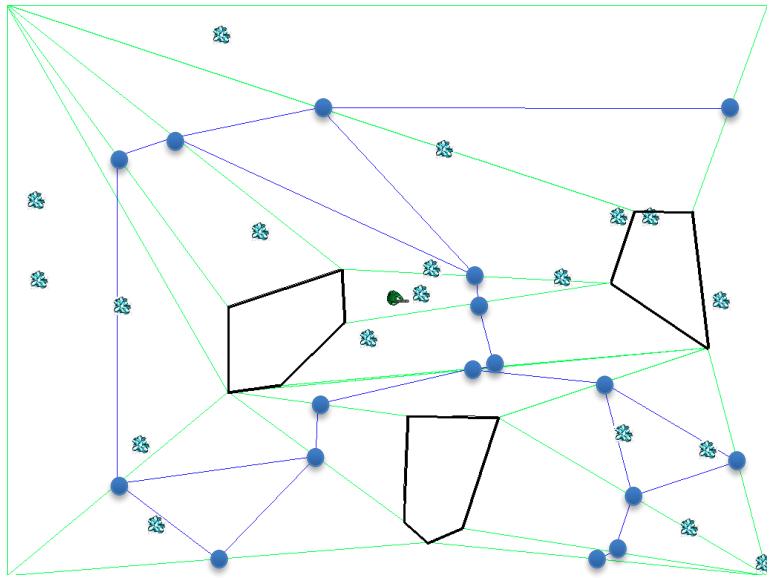




Graphs, Graphs, Graphs…







Graph Search: Sorting Successors

- Uninformed (all nodes are same)
  - DFS (stack – lifo), BFS (queue – fifo)
  - Iterative-deepening (Depth-limited)
- Informed (pick order of node expansion)
  - Greedy Best First
  - Dijkstra – guarantee shortest path ($E\log_2 N$)
  - Floyd-Warshall
  - A* (IDA*)…. Dijkstra + heuristic, Memory Bounded A*
  - D*
- Hierarchical can help

http://en.wikipedia.org/wiki/A*_search_algorithm

Greedy Search

- Expand the node that yields the minimum cost
  - Expand the node that is closest to target
  - Depth first
  - Minimize the heuristic cost function *h(n)*
- Not Complete!
- "Greedy" implies that working solution is not revised
  - Note, A* is best-first, but not greedy (incorporates distance from start)
- Local Minima/Maxima (dead end)

- https://cs.stanford.edu/people/abisee/tutorial

Iterative Deepening (DFS)

- mid 1970s
- Idea: perform depth-limited DFS repeatedly, with an increasing depth limit, until a solution is found.
- Each repetition of depth-limited DFS needlessly duplicates all prior work?
  - Duplication is not significant because a branching factor b > 1 implies that the number of nodes at depth k exactly is much greater than the total number of nodes at all depths k-1 and less.
  - That is: most nodes are in the bottom level.
- The space required by DFS is O(tree depth); BFS is O(#tree nodes ~= $\text{branching}^{\text{depth}}$); IDDFS O(depth * branching factor)

Admissible Heuristic

- An *admissible heuristic* is one that guarantees that the shortest path can be found with the search because it *never overestimates the cost of reaching the goal*
  - A heuristic that does not overestimate is *admissible*
  - Otherwise we say a heuristic is *inadmissible*
- Euclidean Distance is *admissible*
- In games: perfectly acceptable to use either *admissible* or *inadmissible*
  - **"Overestimates can make A* faster if they are almost perfect but home in on the goal more quickly"** – M&F

N-1

1. What kind of solution does greedy search find? Why might this be useful?
2. What kind of solution does A* find?
3. What are some of the insights behind A*?
4. What's a good data structure to use with A*? Why?

A* Search

- 1968: Single source, single target graph search
- Guaranteed to return the optimal path if the heuristic is admissible
- Evaluate each state: f(n) = g(n) + h(n)
- Open list: nodes that are known and waiting to be visited
- Closed list: nodes that have been visited

Pathfinding List (Open and Closed Sets)

- Critical Operations:
  - Adding an entry to the list
  - Removing an entry from the list
  - Finding the smallest element
  - Finding an entry in the list corresponding to a particular node (find() or contains())
- Must find a balance between these four operations for best performance

Excellent slides available here: https://cs.stanford.edu/people/abisee/gs.pdf

Supporting interactive animations: https://cs.stanford.edu/people/abisee/tutorial

We can also thank Stanford for A*

GRAPH SEARCH – PATH PLANNING CONCLUDED

Efficiency of A*

- "maximally efficient":
  - **For a given heuristic function, no optimal algorithm is guaranteed to do less work in terms of nodes expanded.**
- Search cost involves both the cost to expand nodes and the cost to apply heuristic function.
  - Trade-off between heuristic function cost & performance: must consider the *cost to execute the heuristic function* relative to the *cost of expanding nodes* and the *reduction in nodes expanded*.
  - Can we have a h() with perfect estimation?
- A* requires open (& close?) list for remembering nodes
  - Can lead to very large storage requirements

Tradeoffs

- IDA* is asymptotically same time as A* but only O(d) in space - versus O(bd) for A*
  - Also complete and optimal
  - ID tends to increase search time in exchange for reduced memory
  - does not go to the same depth everywhere in the search tree:
    - Begins with an f-bound equal to the f-value of the initial state; performs depth-first search bounded by the f-bound instead of a depth bound
    - ie threshold < f=g+h rather than depth

All pairs shortest path (APSP)

- We talked about this briefly as a “navigation table”
- A look-up table of the form table[node1,node2]-> node 3
  - Where node3 is the next node to go to if you want to go from node1 to node2
- Intuition: Find the shortest distance/path between all pairs of nodes
  - Use this to construct the look-up table

Floyd-Warshall algorithm

- 1962: All-pairs shortest path algorithm
- Tells you path from all nodes to all other nodes in weighted graph
- Positive or negative edge weights, but **no negative cycles** (edges sum to negative)
- Incrementally improves estimate
- $O(|V|^3)$
- Makes use of dynamic programming
- [Use Dijkstra from each starting vertex when the graph is sparse and has non-negative edges]

Given: G=(V,E), |V| = number of vertices

For each edge (u, v) do:

    dist[u][v] = weight of edge (u, v) or infinity

    next[u][v] = v

For k = 0 to |V| do: ← Intermediate node

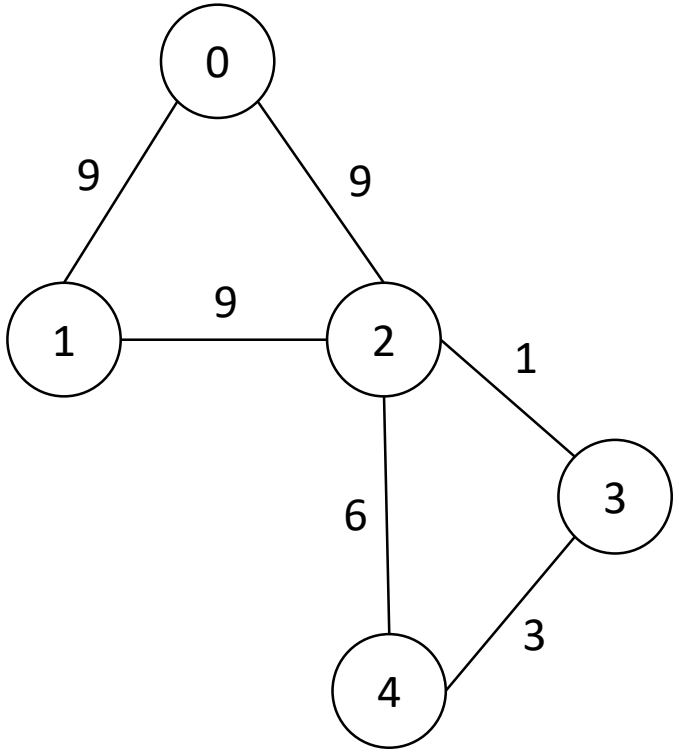
    for i = 0 to |V| do: ← Start node

        for j = 0 to |V| do: ← End node

            if dist[i][k] + dist[k][j] < dist[i][j] then:

                dist[i][j] = dist[i][k] + dist[k][j]

                next[i][j] = next[i][k]



Distance

	0	1	2	3	4
0	INF	9	9	INF	INF
1	9	INF	9	INF	INF
2	9	9	INF	1	6
3	INF	INF	1	INF	3
4	INF	INF	6	3	INF

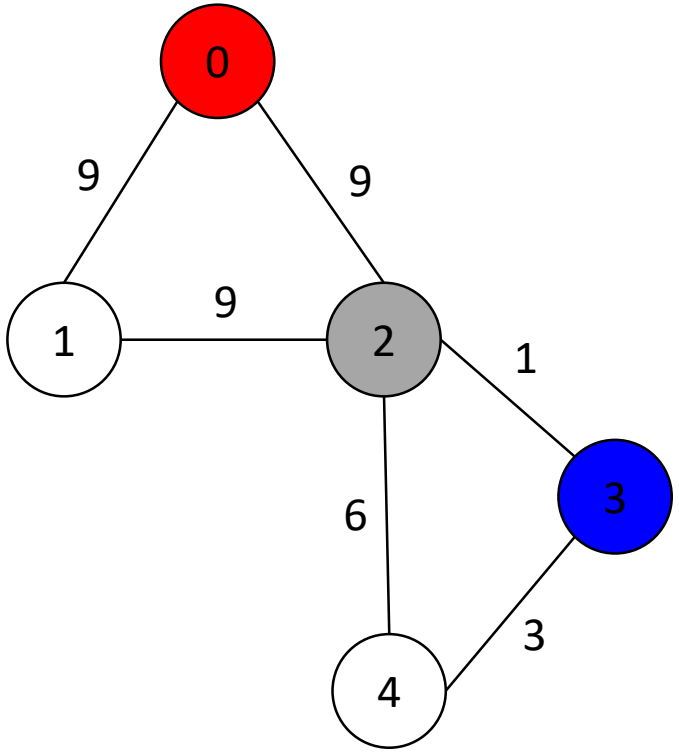
Next

	0	1	2	3	4
0		1	2		
1	0		2		
2	0	1		3	4
3			2		4
4			2	3	

k = 0

i = 0

j = 0



Distance

	0	1	2	3	4
0	INF	9	9	INF	INF
1	9	INF	9	INF	INF
2	9	9	INF	1	6
3	INF	INF	1	INF	3
4	INF	INF	6	3	INF

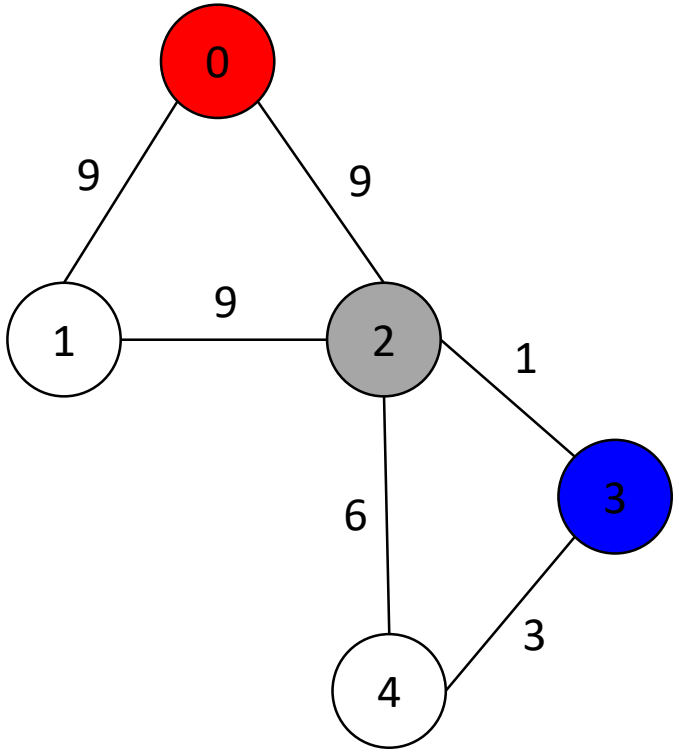
Next

	0	1	2	3	4
0		1	2		
1	0		2		
2	0	1		3	4
3			2		4
4			2	3	

k = 2

i = 0

j = 3 9 + 1 < INF



Distance

	0	1	2	3	4
0	INF	9	9	10	INF
1	9	INF	9	INF	INF
2	9	9	INF	1	6
3	INF	INF	1	INF	3
4	INF	INF	6	3	INF

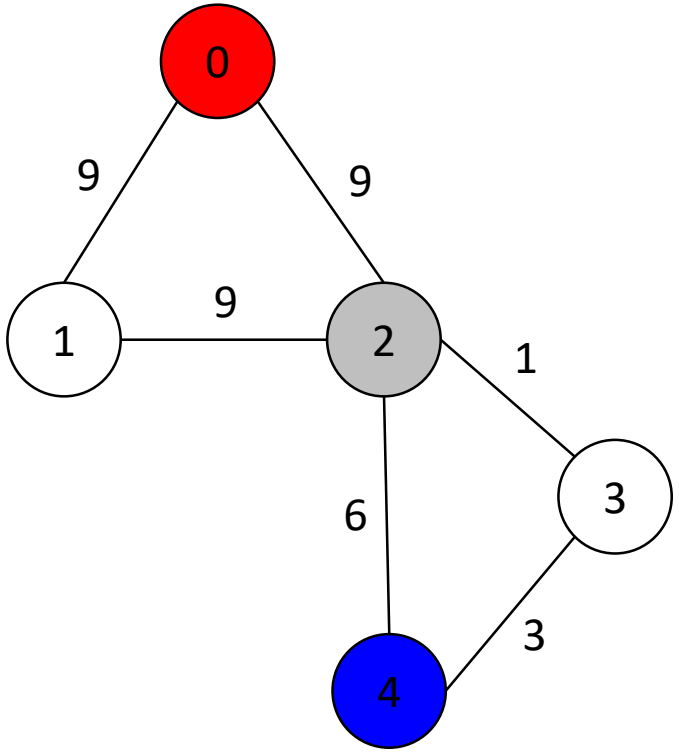
Next

	0	1	2	3	4
0		1	2	2	
1	0		2		
2	0	1		3	4
3			2		4
4			2	3	

k = 2

i = 0

j = 3 0 -> 3: dist 10, goto 2



Distance

	0	1	2	3	4
0	INF	9	9	10	INF
1	9	INF	9	INF	INF
2	9	9	INF	1	6
3	INF	INF	1	INF	3
4	INF	INF	6	3	INF

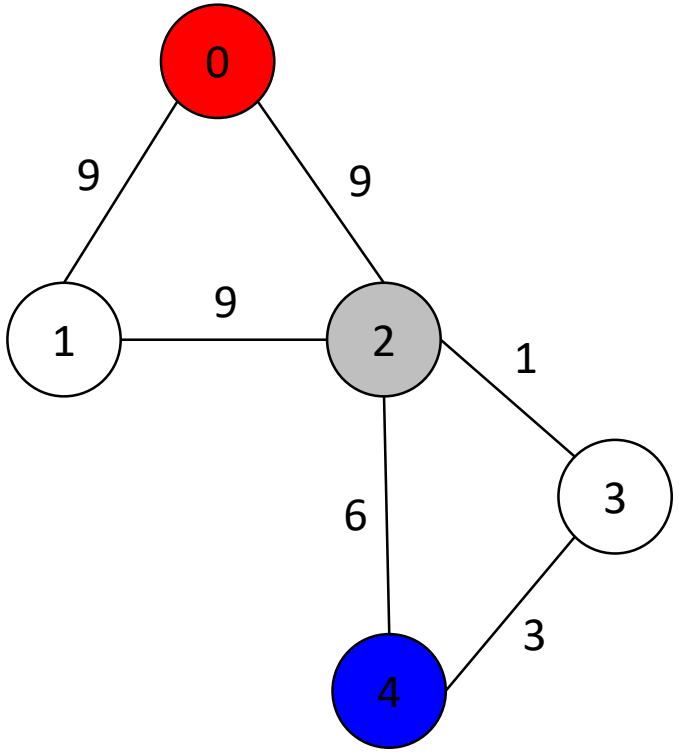
Next

	0	1	2	3	4
0		1	2	2	
1	0		2		
2	0	1		3	4
3			2		4
4			2	3	

k = 2

i = 0

j = 4 9 + 6 < INF



Distance

	0	1	2	3	4
0	INF	9	9	10	15
1	9	INF	9	INF	INF
2	9	9	INF	1	6
3	INF	INF	1	INF	3
4	INF	INF	6	3	INF

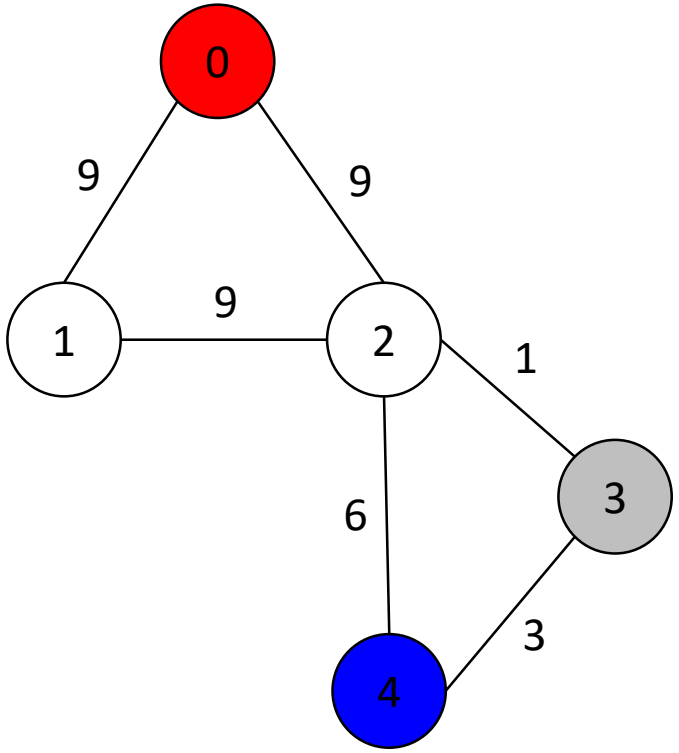
Next

	0	1	2	3	4
0		1	2	2	2
1	0		2		
2	0	1		3	4
3			2		4
4			2	3	

k = 2

i = 0

j = 4 0 -> 3: dist 10, goto 2



Distance

	0	1	2	3	4
0	INF	9	9	10	15
1	9	INF	9	INF	INF
2	9	9	INF	1	6
3	INF	INF	1	INF	3
4	INF	INF	6	3	INF

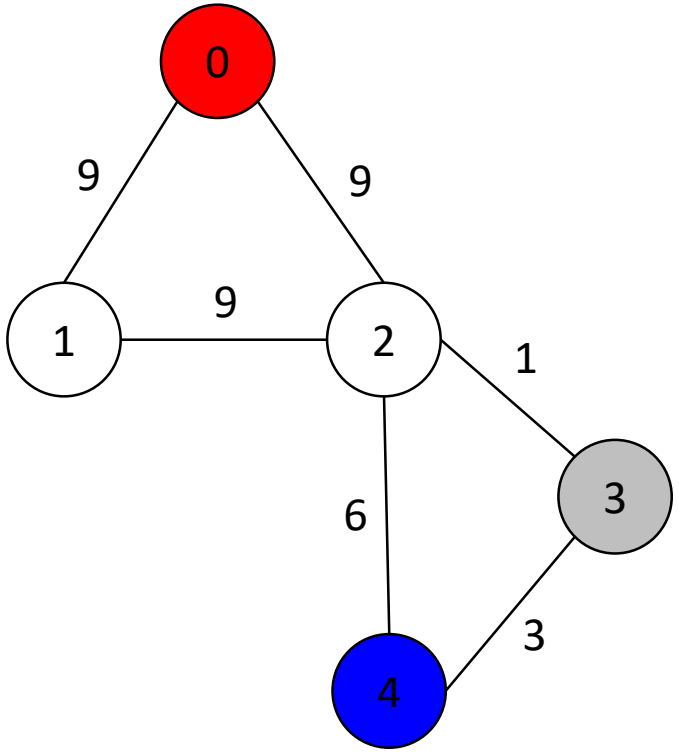
Next

	0	1	2	3	4
0		1	2	2	2
1	0		2		
2	0	1		3	4
3			2		4
4			2	3	

k = 3

i = 0

j = 4 10 + 3 < 15



Distance

	0	1	2	3	4
0	INF	9	9	10	13
1	9	INF	9	INF	INF
2	9	9	INF	1	6
3	INF	INF	1	INF	3
4	INF	INF	6	3	INF

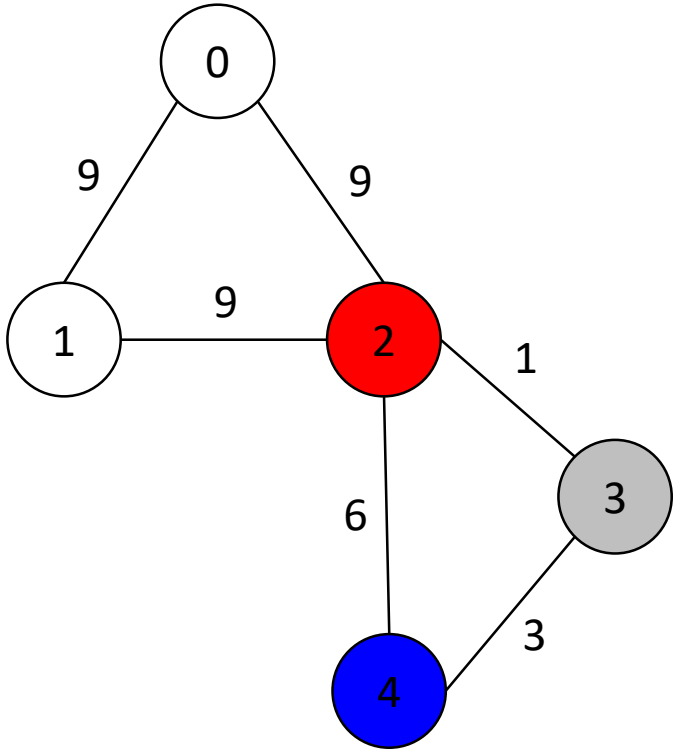
Next

	0	1	2	3	4
0		1	2	2	2
1	0		2		
2	0	1		3	4
3			2		4
4			2	3	

k = 3

i = 0

j = 4 0 -> 4: dist 13, goto 2



Distance

	0	1	2	3	4
0	INF	9	9	10	13
1	9	INF	9	INF	INF
2	9	9	INF	1	6
3	INF	INF	1	INF	3
4	INF	INF	6	3	INF

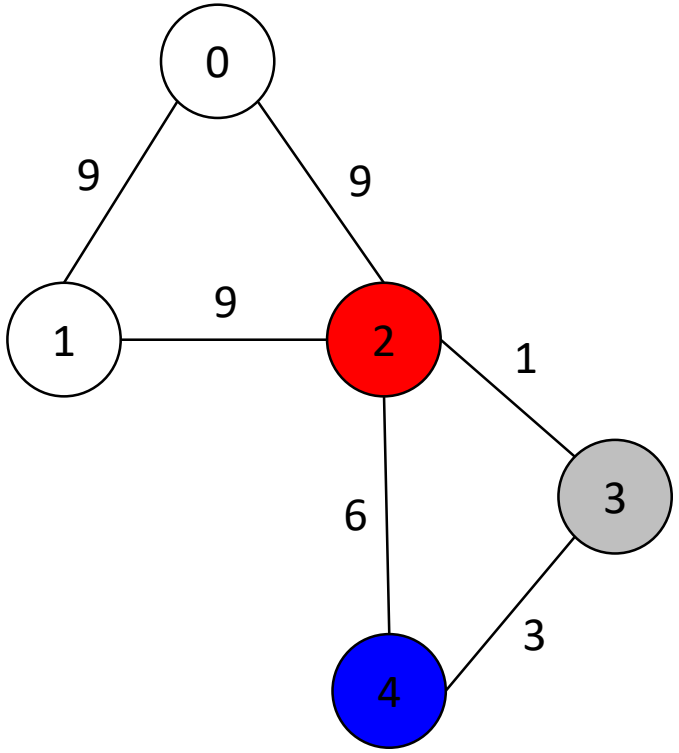
Next

	0	1	2	3	4
0		1	2	2	2
1	0		2		
2	0	1		3	4
3			2		4
4			2	3	

k = 3

i = 2

j = 4 1 + 3 < 6



Distance

	0	1	2	3	4
0	INF	9	9	10	13
1	9	INF	9	INF	INF
2	9	9	INF	1	4
3	INF	INF	1	INF	3
4	INF	INF	6	3	INF

Next

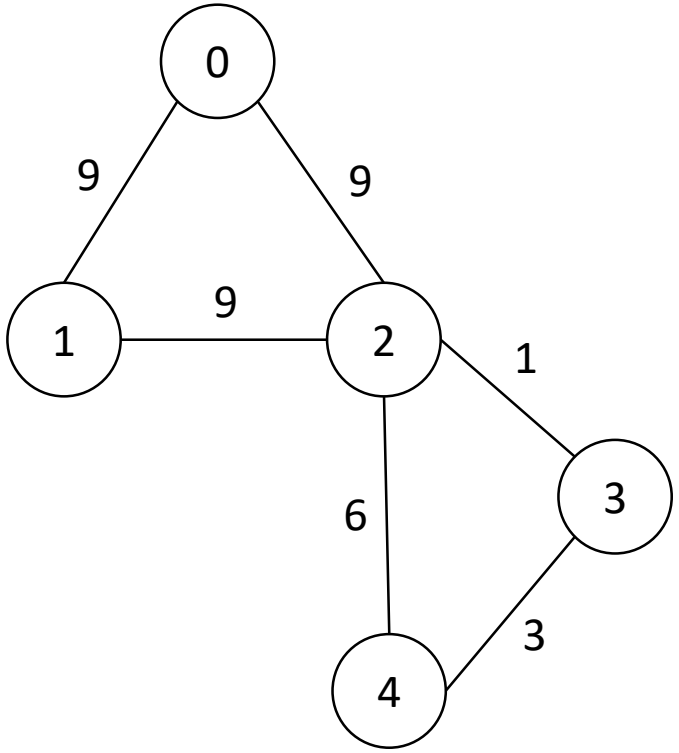
	0	1	2	3	4
0		1	2	2	2
1	0		2		
2	0	1		3	3
3			2		4
4			2	3	

k = 3

i = 2

j = 4 2 -> 4: dist 4, goto 3

Finally...



Distance

	0	1	2	3	4
0	INF	9	9	10	13
1	9	INF	9	10	13
2	9	9	INF	1	4
3	10	10	1	INF	3
4	13	13	4	3	INF

Next

	0	1	2	3	4
0		1	2	2	2
1	0		2	2	2
2	0	1		3	3
3	2	2	2		4
4	3	3	3	3	

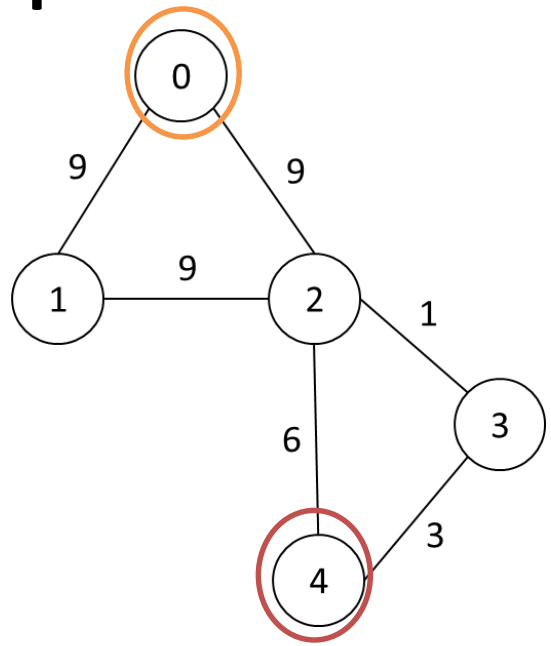
k = 3

i = 2

j = 4 2 -> 4: dist 4, goto 3

Reconstructing the path from lookup table

Want to go from u to v (E.g. u=0, v=4)



if next[u][v] is empty then return null path

path = (u)

path=0

while u <> v do:

u = next[u][v]

path.append(u)

u=next[0][4]=2; path=0,2

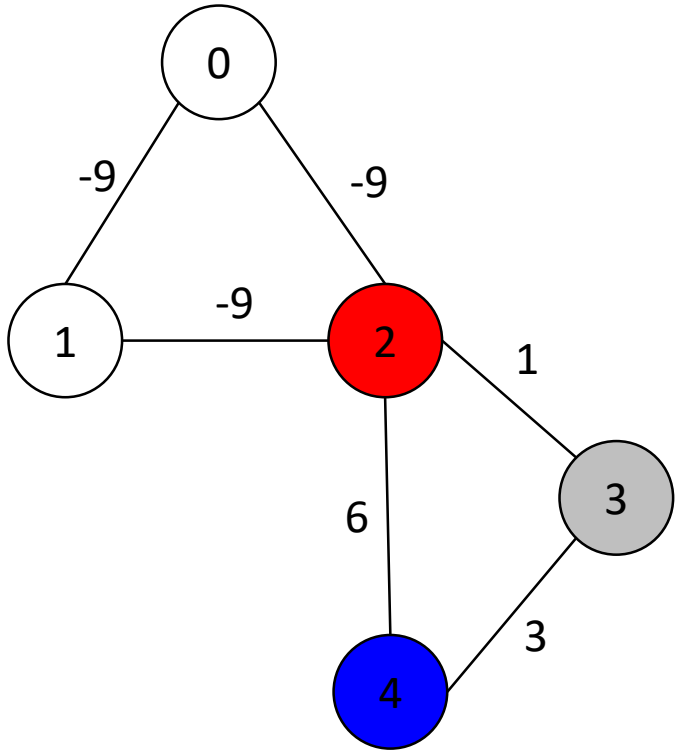
u=next[2][4]=3; path=0,2,3

u=next[3][4]=4; path=0,2,3,4

return path

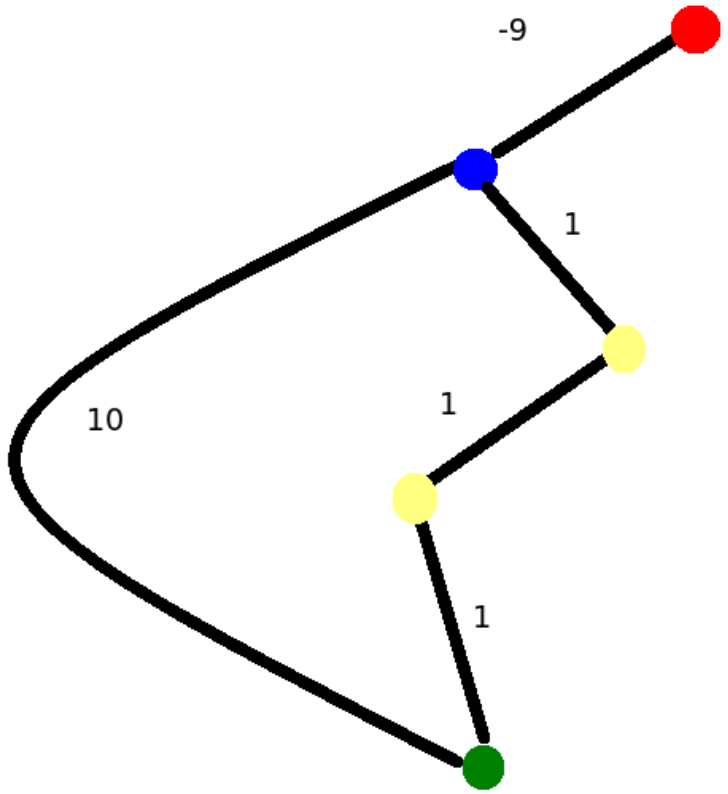
	0	1	2	3	4
0		1	2	2	2
1	0		2	2	2
2	0	1		3	3
3	2	2	2		4
4	3	3	3	3	

Detecting Negative Cycles



Distance

	0	1	2	3	4
0	-3	-9	-9	8	4
1	-9	-3	-9	INF	INF
2	-9	-9	-3	1	4
3	INF	INF	1	INF	3
4	INF	INF	6	3	INF



Reweighting is no help in networks with negative cycles: The operation does not change the weight of any cycle, so we cannot remove negative cycles by reweighting. But for networks with no negative cycles, we can seek to discover a set of vertex weights such that reweighting leads to edge weights that are nonnegative, no matter what the original edge weights. With nonnegative edge weights, we can then solve the all-pairs shortest-paths problem with the all-pairs version of Dijkstra's algorithm.

http://www.informit.com/articles/article.aspx?p=169575&seqNum=8

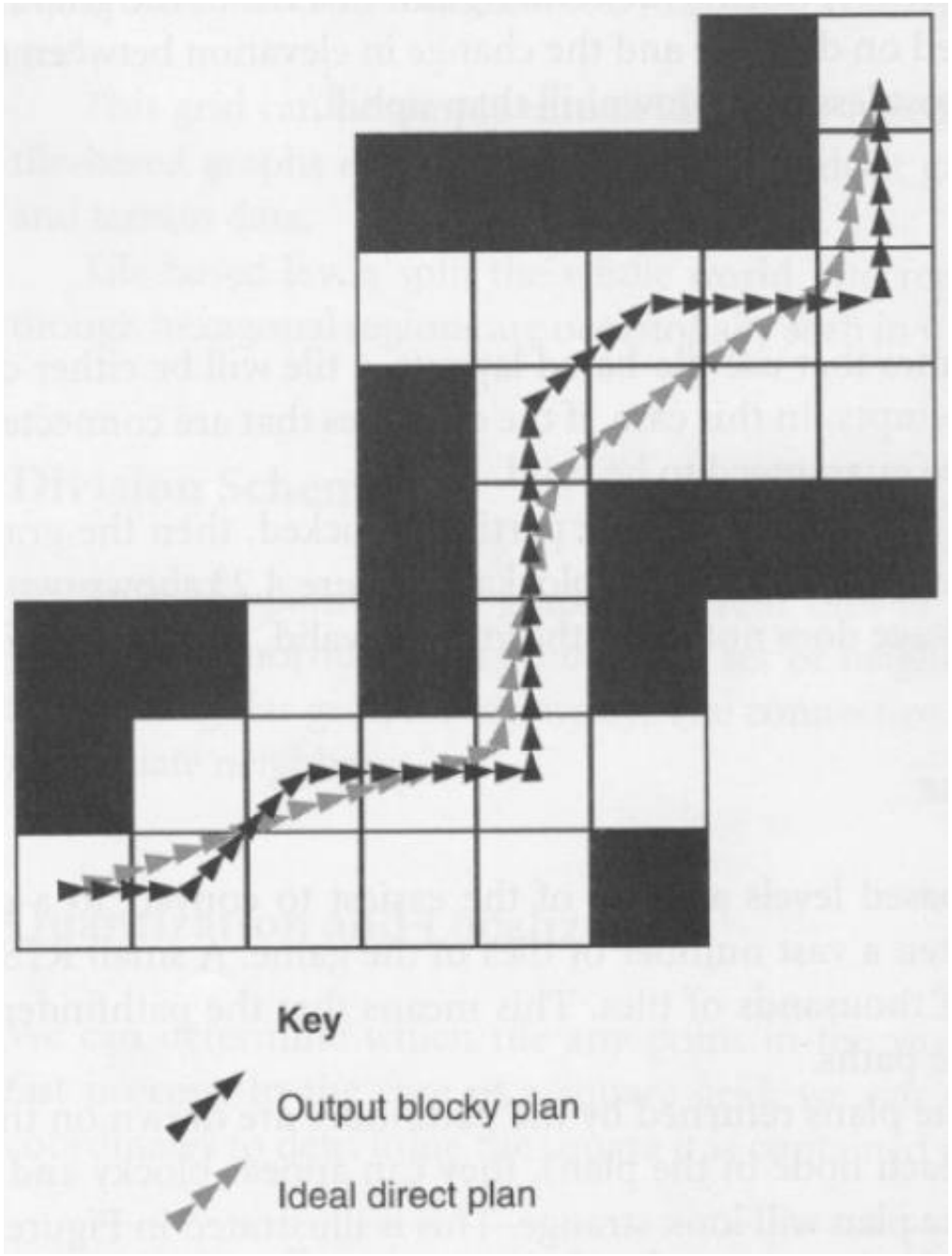
When to use A* and APSP

- If the environment is small and static?
- If the environment is dynamic?
- If the environment is large and static?
  - If runtime memory is an issue?
  - If runtime memory isn’t an issue?
- If the environment is large and dynamic?

What to do if…

- the open list has gotten so large that you are running out of memory?
- you are running out of time and you have not yet reached an answer?
- there are a number of nodes on the open list whose h value is very small?

SOLVING WEIRD FINAL PATHS

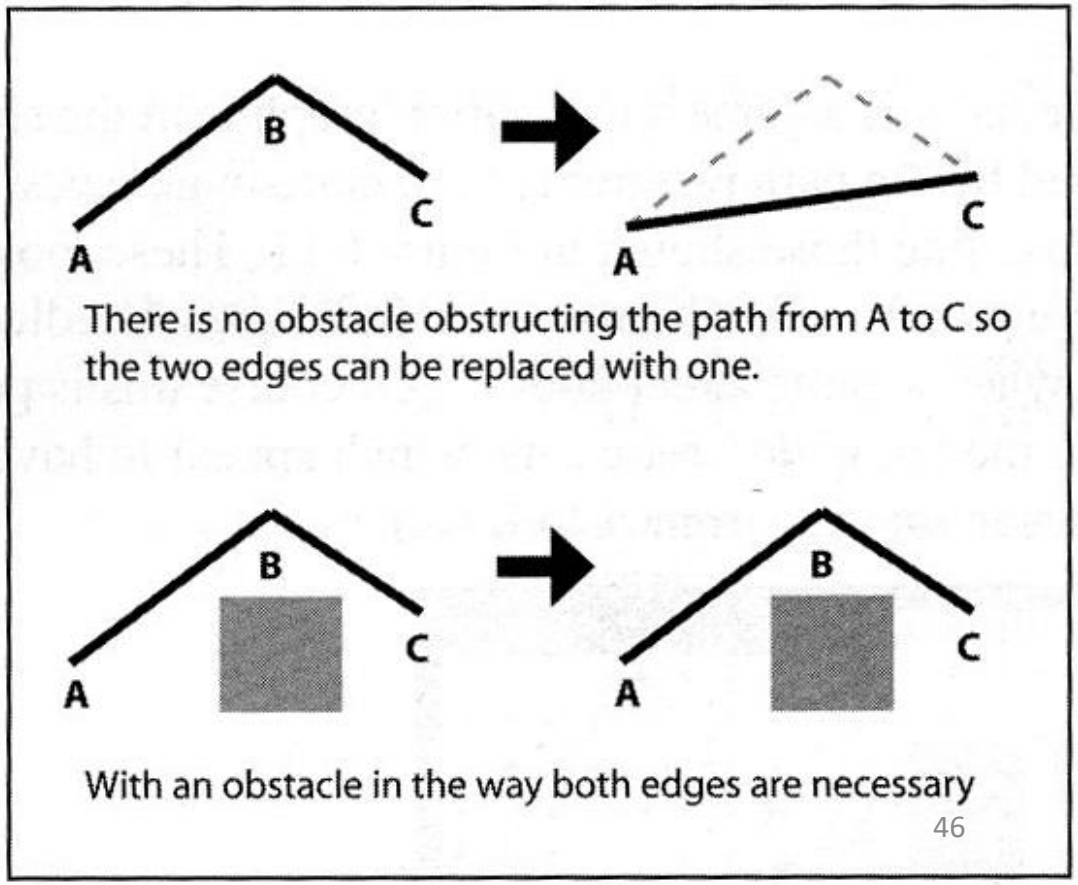


Fixing awkward agent movement:

- String pulling
- Splines
- Hierarchical A*
- Simple stupid funnel

Path Smoothing via “String pulling”

- Zig-zagging from point to point looks unnatural
- Post-search smoothing can elicit better paths



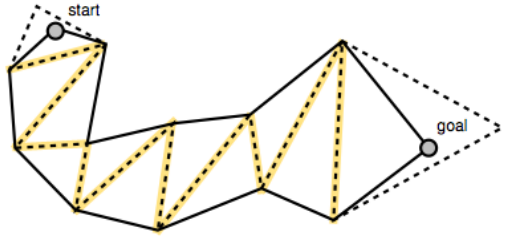
Quick Path-Smoothing

- Given a path, look at first two edges, E1 & E2
  1. Get E1_src and E2_dest
  2. If unobstructed path between the two, set E1_dest = E2_dest, then delete E2 from the path. Set next edge as E2.
  3. Else, increment E1 and E2.
  4. Repeat until E2_dest == goal.

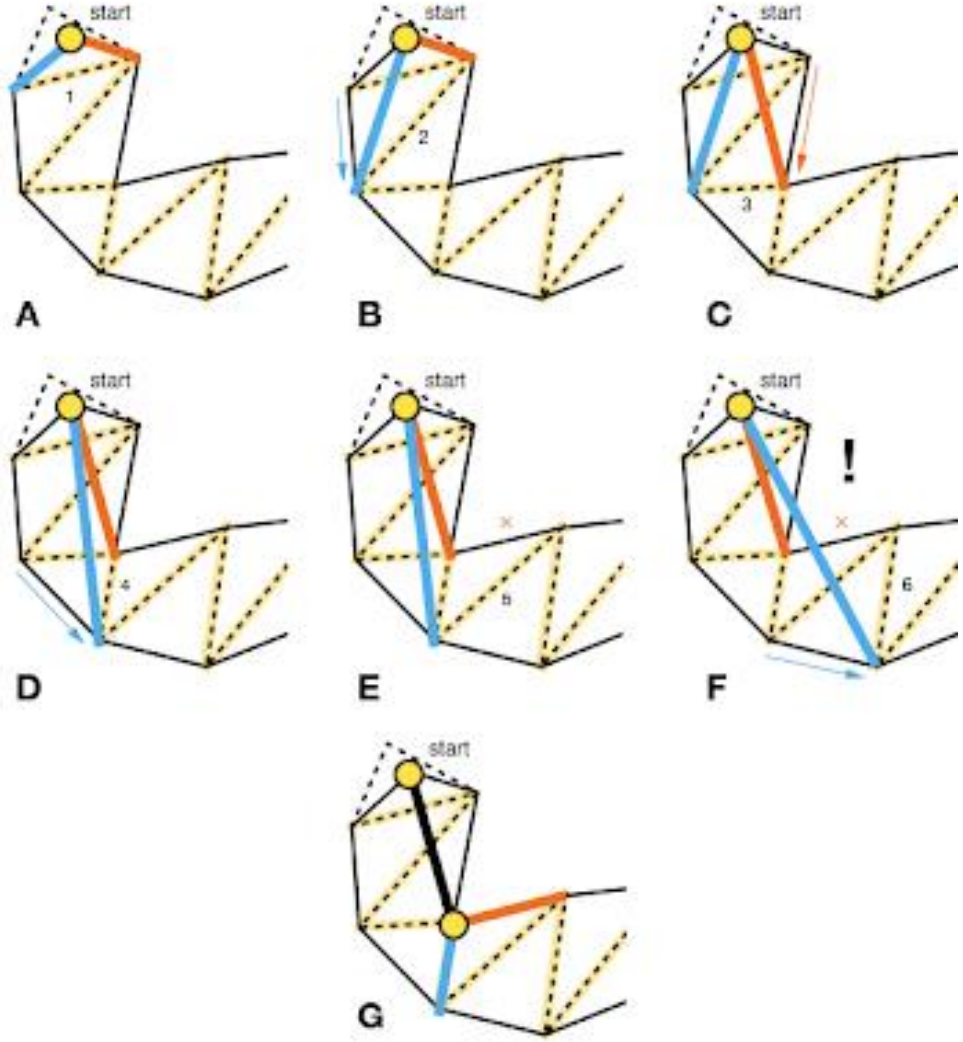
Slow Path-Smoothing

- Given a path, look at first two edges, E1 & E2
  1. Get E1_src and E2_dest
  2. If unobstructed path between the two, set E1_dest = E2_dest, then delete E2 from the path. Set E1 and E2 from beginning of path.
  3. Else, increment E1 and E2.
  4. Repeat until E2_dest == goal.

Stupid Simple Funnel



- Check if the left and right points are inside the current funnel (described by the blue and red lines),
  - if they are, we simple narrow the funnel (A-D).
- If the new left endpoint is outside the funnel, the funnel is not updated (E-F)
  If the new left end point is over right funnel edge (F),
  - add the right funnel as a corner in the path and place the apex of the funnel at the right funnel point location and restart the algorithm from there (G).



http://digestingduck.blogspot.com/2010/03/simple-stupid-funnel-algorithm.html

SOLVING LONG PATH SEARCH TIMES

Solution to Long Search Times

- Precompute paths (if you can)
  - Dijkstra: Single source shortest path (SSSP; O(E log V))
    - Run for each vertex: O(VE Log V) which can go (V^3 Log V) in worst case
  - Floyd-warshall: All pairs shortest path (APSP, $O(|V|^3)$)
- Register search requests
  - Works best with lots of agents. Prevents heavy load to CPU.
  - Let agents wander or seek toward a goal while waiting for a search response. (Although they might wander in the wrong direction)
- Hierarchical Path Planning
  - Within 1% of optimal path length, but up to 10 times faster

Precomputing Paths

- Faster than computation on the fly esp. large maps and many agents
- Use Dijkstra's or Floyd-warshall algorithm to create lookup tables
- What is the main problem with pre-computed paths?

Sticky Situations: Movable objects, fog of war, memory issues, and other burps – precomputed paths do no good

SOLVING WHEN WE CAN’T PRECOMPUTE

Sticky Situations

- Dynamic environments can ruin plans; memory issues can inhibit precomputing
- What do we do when an agent has been pushed back through a doorway that it has already “visited”?
- What do we do in “fog of war” situations?
- What if we have a moving target?

Dynamic environments

- Terrain can change: destructible environments
  - Jumpable?
  - Kickable?
  - Too big to jump/kick?
- Path network edges can be created/eliminated

Heuristic Search

- A* (Iterative deepening, memory bounded)
- Hierarchical A*
- Real-time A*
- Real-time A* with lookahead
- D* lite

Hierarchical Path Planning

- Used to reduce CPU overhead of graph search
- Plan with coarse-grained and fine-grained maps
- Example: Planning a trip to NYC based on states, then individual roads

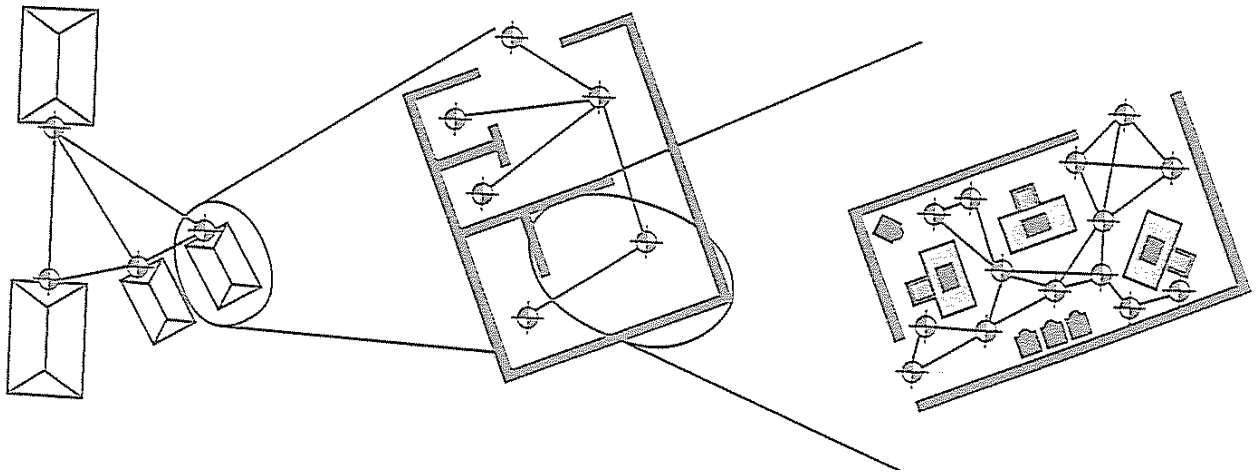
Hierarchical A*

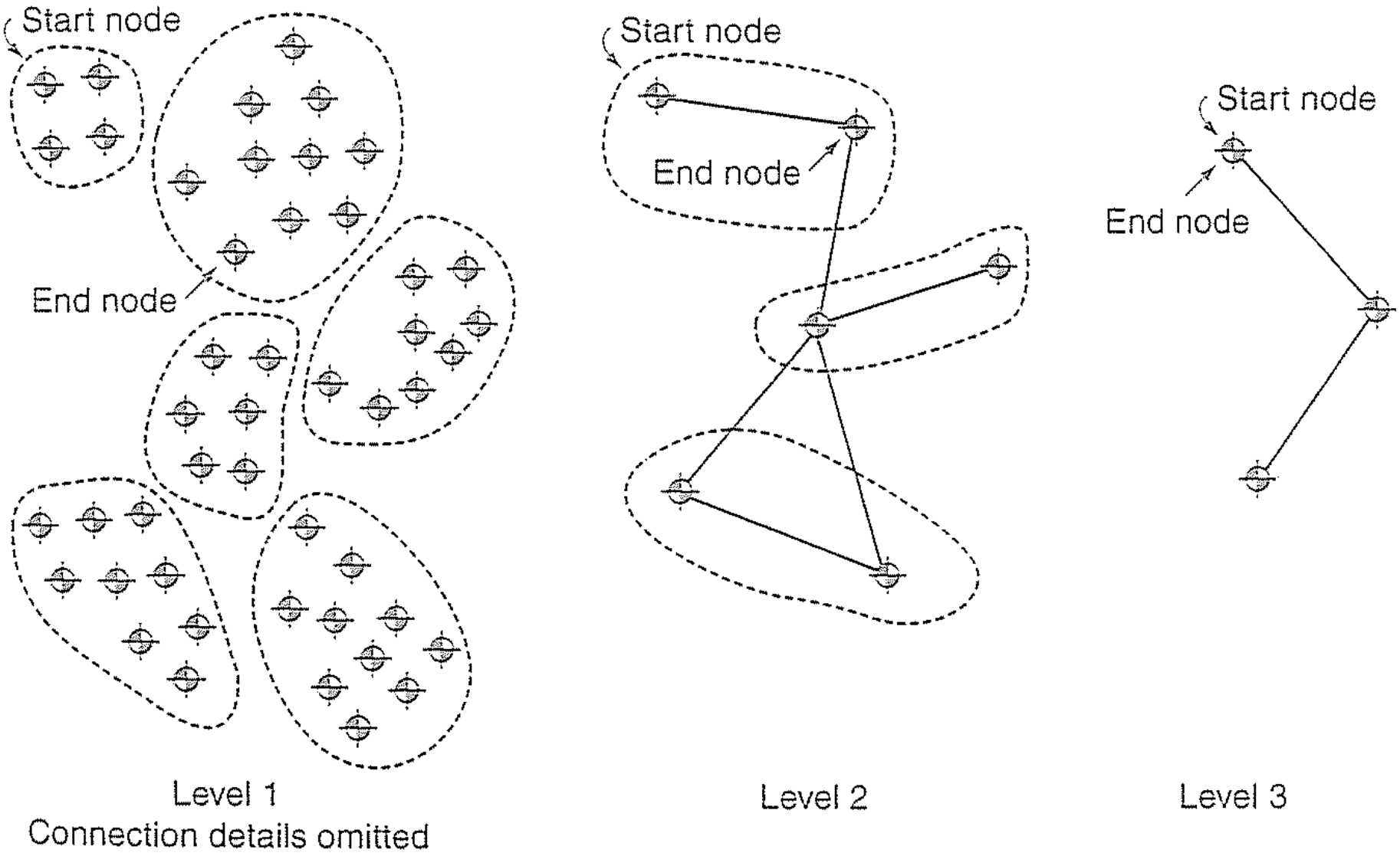
- http://www.cs.ualberta.ca/~mmueller/ps/hpastar.pdf
- http://aigamedev.com/open/review/near-optimal-hierarchical-pathfinding/
- Within 1% of optimal path length, but up to 10 times faster

Hierarchical A*

- People think hierarchically (more efficient)

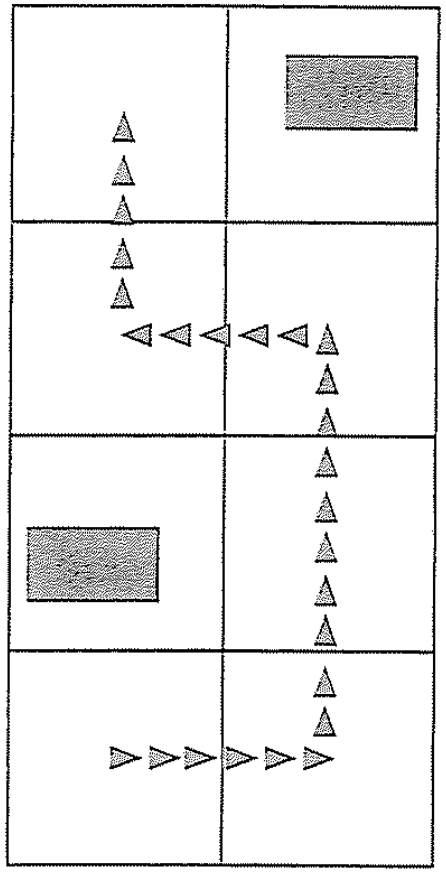
- We can prune a large number of states



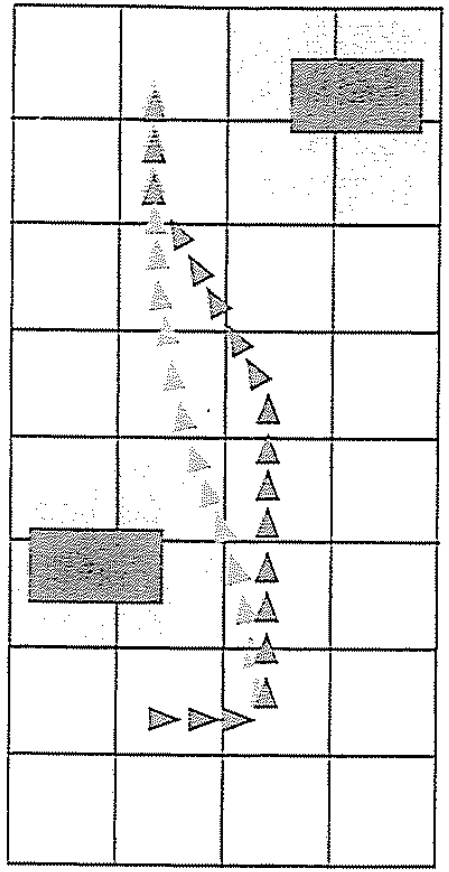


How high up do you go? As high as you can without start and end being in the same node.

Path Smoothing in Hierarchical A*

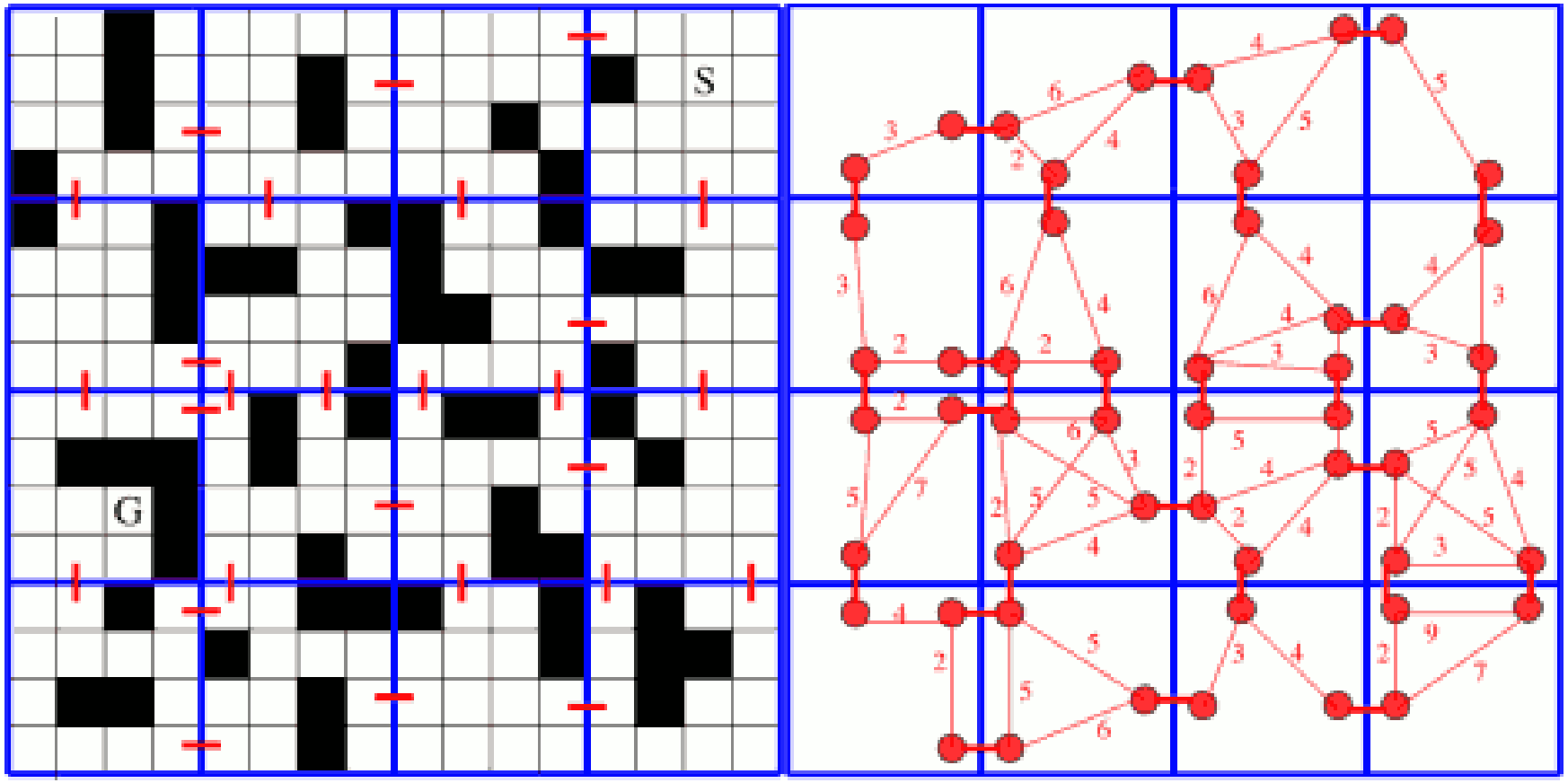


High-level plan



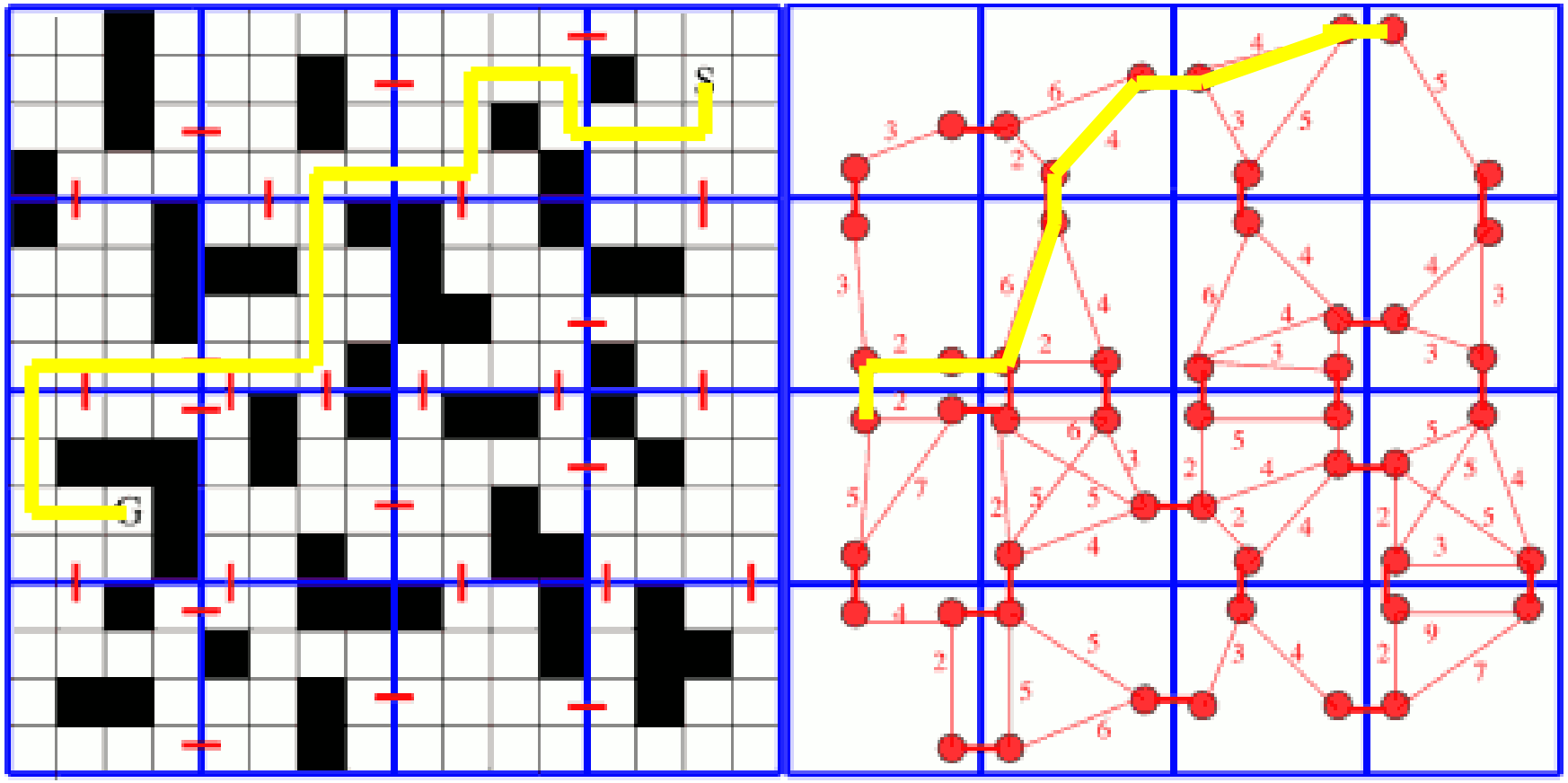
Low-level plan

1. Build clusters. Can be arbitrary
2. Find transitions, a (possibly empty) set of obstacle-free locations.
3. Inter-edges: Place a node on either side of transition, and link them (cost 1).
4. Intra-edges: Search between nodes inside cluster, record cost.
    * Can keep optimal intra-cluster paths, or discard for memory savings.



1. Start cluster: Search within cluster to the border
2. Search across clusters to the goal cluster
3. Goal cluster: Search from border to goal
4. Path smoothing

* Really just adds start and goal to the hierarchy graph



Real Time A*

- “One should backtrack to a previously visited real world state when the estimate of solving the problem from that state plus the cost of returning to that state is less than the estimated cost of going forward from the current state.” - Korf
- Tracks states/nodes that have been visited by an actual move in the real world of the problem solver

Thad StArner says: bidirectional a star path search still has research available to be done on determining your terminal case.

Real Time A*

- Online search: **execute as you search**
  - Because you can't look at a state until you get there. No backtracking
  - No open list
- Modified cost function f()
  - g(n) is actual distance from n to current state (instead of initial state)
  - Use a hash-table to keep track of h() for nodes you have visited (because you might visit them again)
- Pick node with lowest f-value from immediate successors
- Execute move immediately
- After you move, update previous location
  - h(prev) = second best f-value
  - Second best f-value represents the estimated cost of returning to the previous state (and then add g)

RTA* with lookahead

- At every node you can see some distance
- DFS, then back up the value (think of it as minimin with alpha-pruning)
- Search out to known limit
- Pick best, then move
- Repeat, because something might change in the environment that changes our assessment
  - Things we discover as our horizon moves
  - Things that change behind us

D* Lite

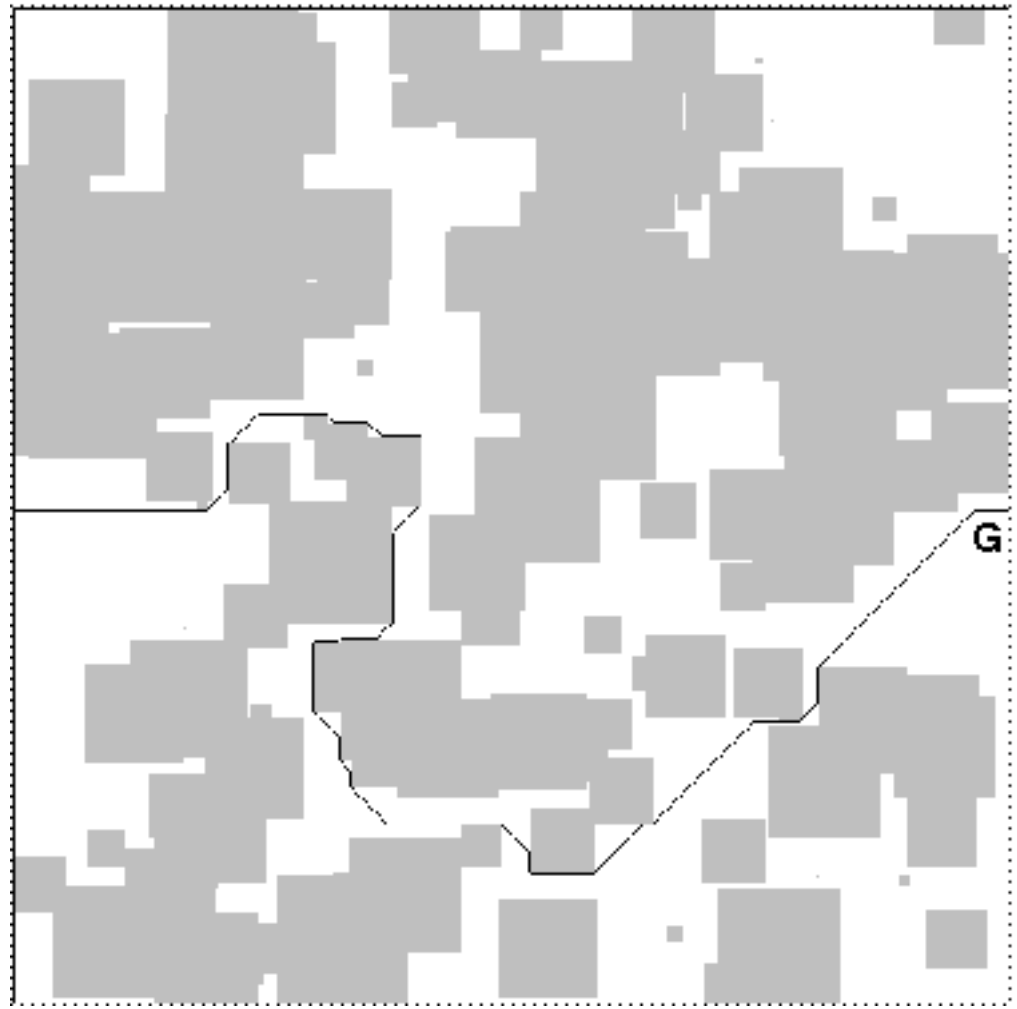
- 1994: Incremental search: replan often, but reuse search space if possible
- In unknown terrain, assume anything you don't know is clear (optimistic)
- Perform A*, execute plan until discrepancy, then replan
- D* Lite achieves 2x speedup over A* (when replanning)

D* Lite

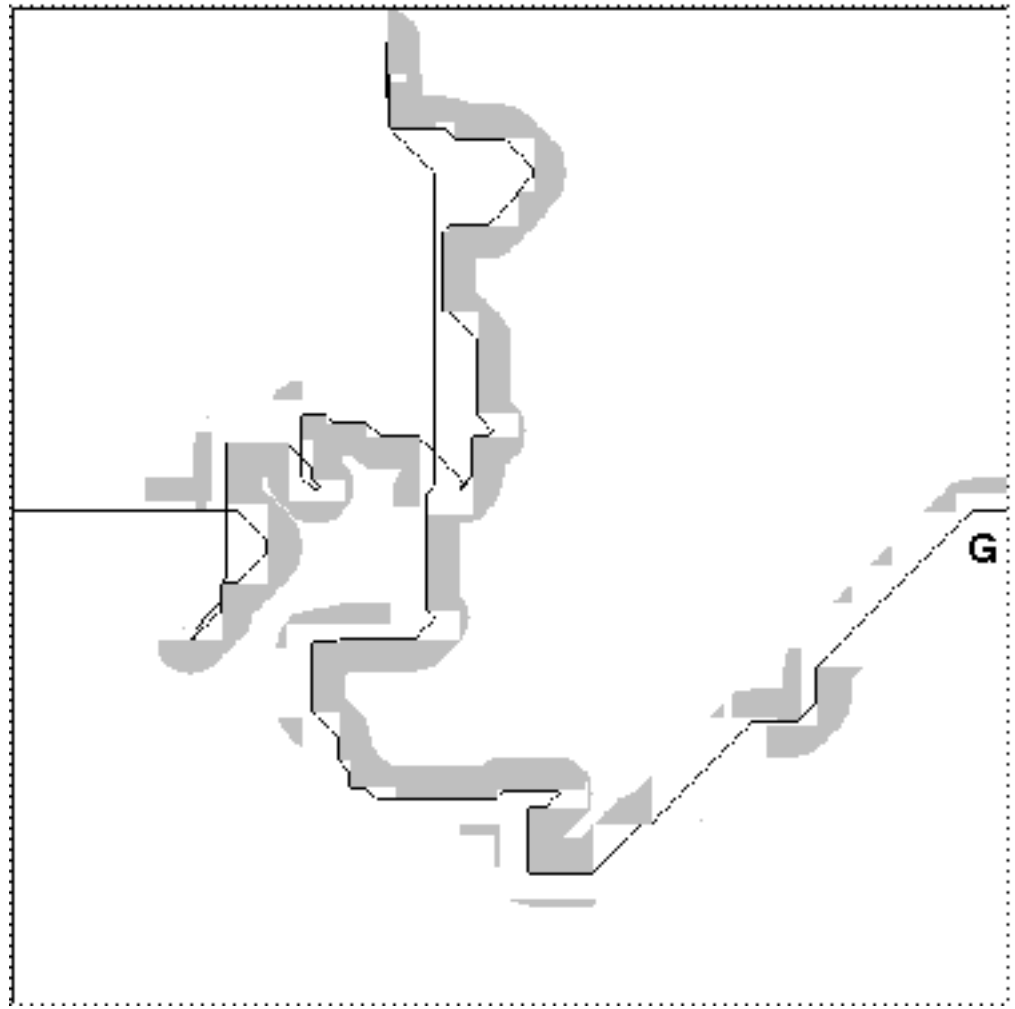
Sven Koenig
College of Computing
Georgia Institute of Technology
Atlanta, GA 30312-0280
skoenig@cc.gatech.edu

Maxim Likhachev
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
maxim+@cs.cmu.edu

“Omniscient optimal”: given complete information



“Optimistic optimal”: assume empty for parts you don’t know.



Heuristic Search Recap

- A*
  - Use when we can't precompute
    - Dynamic environments
    - Memory issues
  - Optimal when heuristic is admissible (and assuming no changes)
  - Replanning can be slow on really big maps
- Hierarchical A* is the ~same, but faster
  - Within 1% of A* optimality but up to 10x faster
- Real-time A*
  - Stumbling in the dark, 1 step lookahead
  - Replan every step, but fast!
  - Realistic? For a blind agent that knows nothing
  - Optimal when completely blind
- Real-time A* with lookahead
  - Good for fog-of-war
  - Replan every step, with fast bounded lookahead to edge of known space
  - Optimality depends on lookahead

Heuristic Search Recap

- D* Lite
  - Assume everything is open/clear
  - Replan when necessary
  - **Worst case: Runs like Real-Time A***
  - Best case: Never replan
  - Optimal including changes

See Also

- AI Game Programming wisdom 2, CH 2
- Buckland CH 8
- Millington CH 4
- Wikipedia rabbit hole
- https://cs.stanford.edu/people/abisee/gs.pdf
  - https://cs.stanford.edu/people/abisee/tutorial
- http://mas.cs.umass.edu/classes/cs683/lectures-2010/