PATHFINDING OPTIMIZATIONS

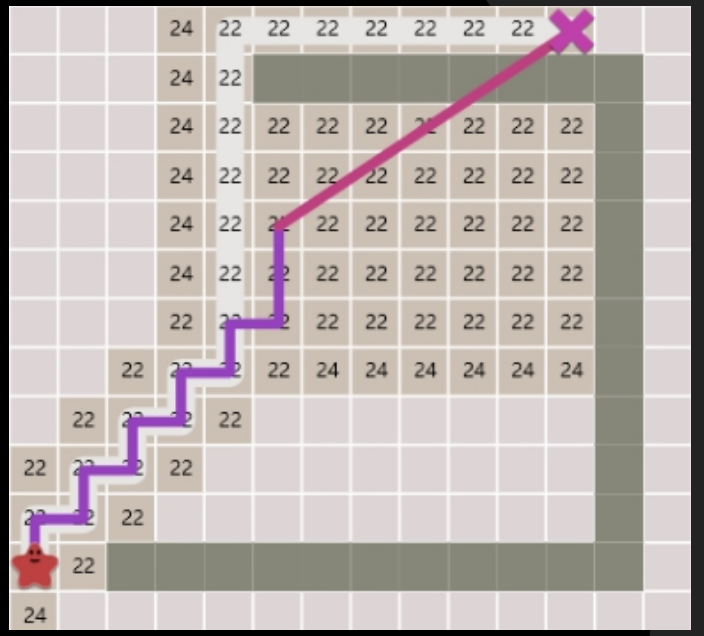
Yang Hu

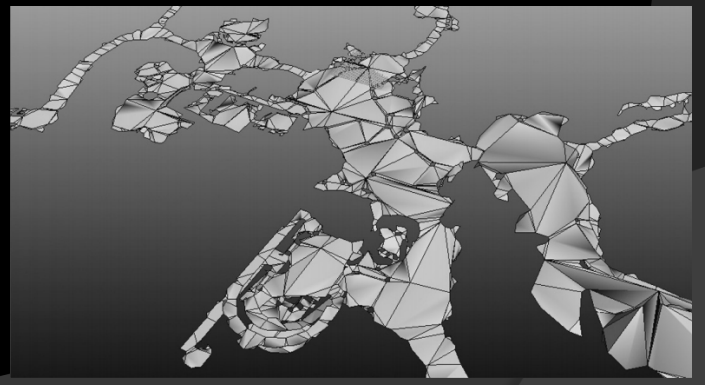
CS 5150 Course Presentation

2/17/2021

Content

- Optimizing A* pathfinding at run-time
- Optimizing A* pathfinding with precomputation
- Optimization for large map pathfinding





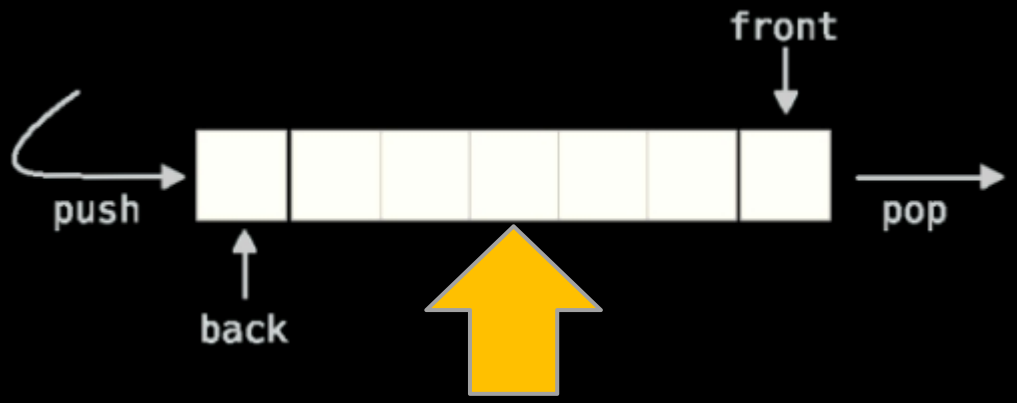
Optimizing A* pathfinding at run-time

- Use better frontier queue management
- Add heuristic coefficient
- Use better heuristic
- Avoid backtrack

```
while not frontier.empty():
   current = frontier.get()

   if current == goal:
      break

   for next in graph.neighbors(current):
      new_cost = cost_so_far[current] + graph.cost(current, next)
      if next not in cost_so_far or new_cost < cost_so_far[next]:
         cost_so_far[next] = new_cost
         priority = new_cost + heuristic(goal, next)
         frontier.put(next, priority)
         came_from[next] = current
```

Use better frontier queue management

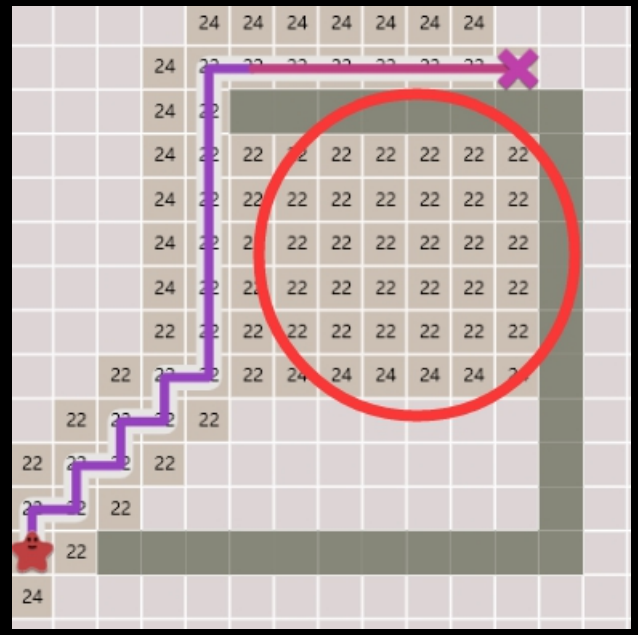
- Queue operations could be slow with large data.




```
while not frontier.empty():
   current = frontier.get()

   if current == goal:
      break

   for next in graph.neighbors(current):
      new_cost = cost_so_far[current] + graph.cost(current, next)
      if next not in cost_so_far or new_cost < cost_so_far[next]:
         cost_so_far[next] = new_cost
         priority = new_cost + heuristic(goal, next)
         frontier.put(next, priority)
         came_from[next] = current
```

Use better frontier queue management

- Cache the last inserted node in case it will be immediately used in next run.
- Put nodes into several LIFO stacks with respective costs.



COST 1

COST 2

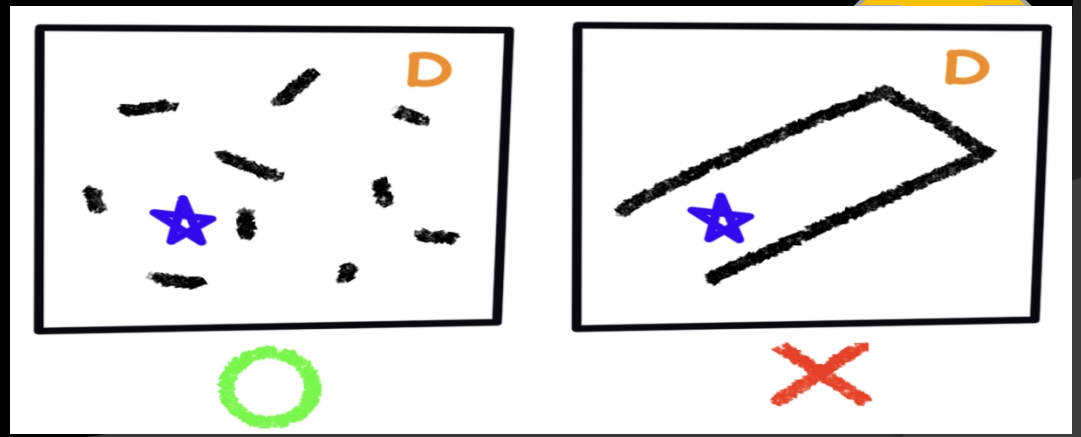
COST 3

COST 4

Add heuristic coefficient

- A* guarantees an shortest path
- Adding a >1 weight makes the algorithm more “greedy” and tremendously faster, while less accurate.
- The suitable weight should be discovered experimentally (usually 1.1 ~ 1.5).
- Use different weights in different maps.

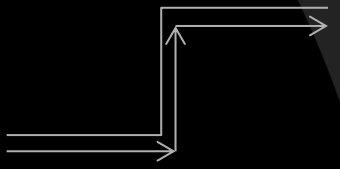
```
while not frontier.empty():
   current = frontier.get()

   if current == goal:
      break

   for next in graph.neighbors(current):
      new_cost = cost_so_far[current] + graph.cost(current, next)
      if next not in cost_so_far or new_cost < cost_so_far[next]:
         cost_so_far[next] = new_cost
         priority = new_cost + heuristic(goal, next) * weight
         frontier.put(next, priority)
         came_from[next] = current
```

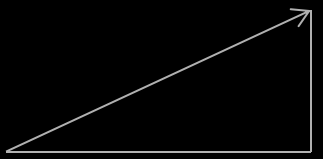


Use better heuristic

```
cost_so_far[next] = new_cost
priority = new_cost + heuristic(goal, next)
frontier.put(next, priority)
```

- Manhattan distance does not consider diagonal movement, hence overestimate distances
- Straight-line heuristic assumes that paths can take any angle hence underestimate distances.
- Use Octile heuristic, which corresponds exactly to movement in the world





$$\max(\Delta x, \Delta y) + 0.41 \cdot \min(\Delta x, \Delta y)$$

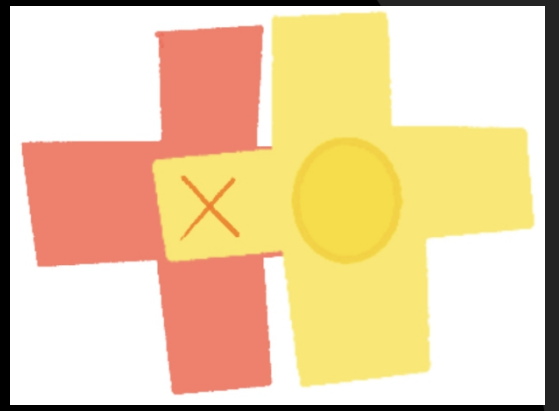
Represents diagonal move additional cost.

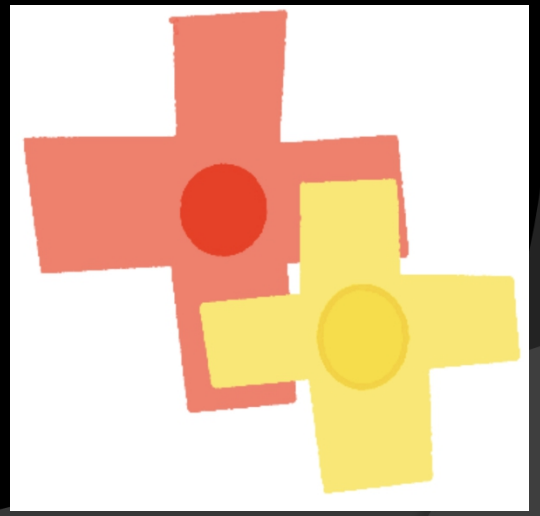
Avoid backtrack

- Happens when current node's neighbor is the parent or the parent's neighbor
- A simple fix as not considering parent node as neighboring node speed up the algorithm by

$$\frac{1}{branching\_factor}$$

$\frac{1}{8}$ for grid search space, $\frac{1}{3}$ for nav mesh



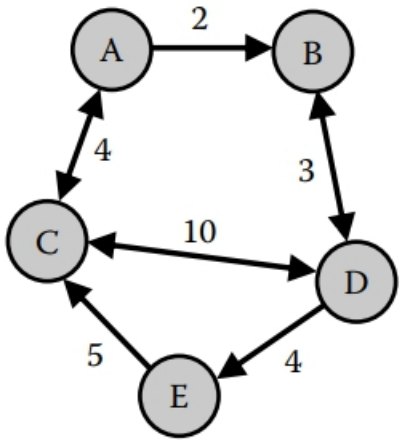


Optimizing A* pathfinding with precomputation

- Precompute all paths: Roy-Floyed-Warshall
- Precompute Pivot nodes
- Precompute neighbors
- Precompute bound boxes

Precompute all paths

- Absolutely fastest, but ridiculously memory consuming
- O(n^2) table entries for n nodes.



Next Node Look-Up Table

	A	B	C	D	E
A	A	B	C	B	B
B	D	B	D	D	D
C	A	A	C	A	A
D	E	B	E	D	E
E	C	C	C	C	E

Cost to Goal Look-Up Table

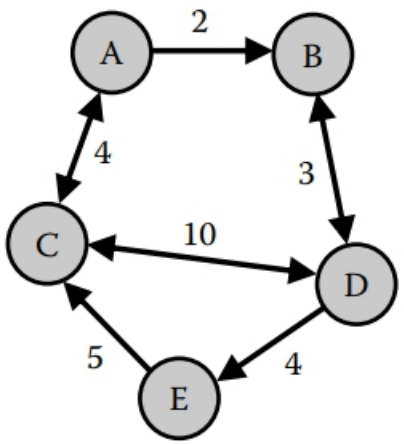
	A	B	C	D	E
A	0	2	4	5	9
B	16	0	12	3	7
C	4	6	0	9	13
D	13	3	9	0	4
E	9	11	5	14	0

Figure 17.1

A search space with its corresponding Roy–Floyd–Warshall path look-up table on the left and a very accurate heuristic cost look-up table on the right.

Precompute all paths

- We can compress losslessly by canceling duplicate paths
  - (i.e highways in GTA)
- Lossy compression – only store meaningful data



Next Node Look-Up Table

	A	B	C	D	E
A	A	B	C	B	B
C	A	A	C	A	A
D	E	B	E	D	E

Cost to Goal Look-Up Table

	A	B	C	D	E
A	0	2	4	5	9
C	4	6	0	9	13
D	13	3	9	0	4

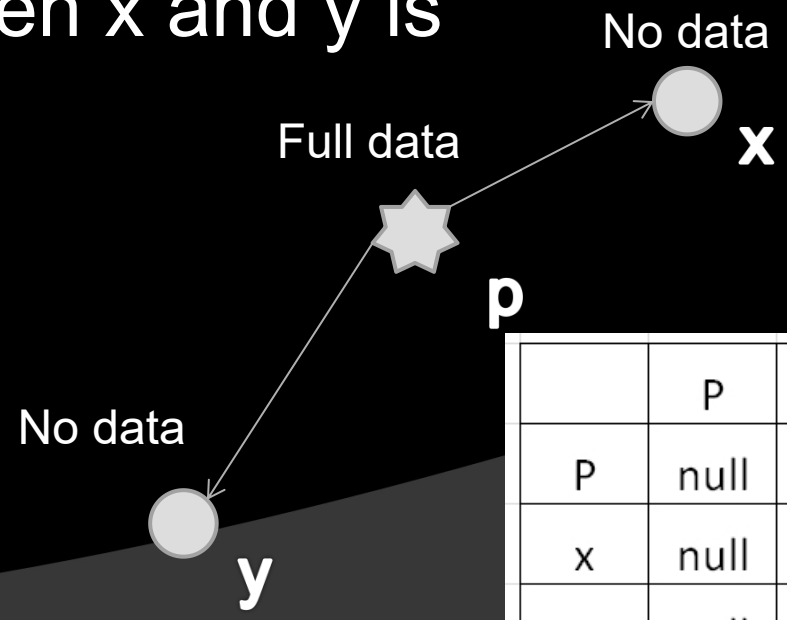
Figure 17.1

A search space with its corresponding Roy–Floyd–Warshall path look-up table on the left and a very accurate heuristic cost look-up table on the right.

Precompute Pivot nodes

- Only store complete data for few nodes - pivots
- Given such node p, we know d(p, z) for all z
- If d(x, y) is the distance between node x and y
- Then the estimated distance between x and y is

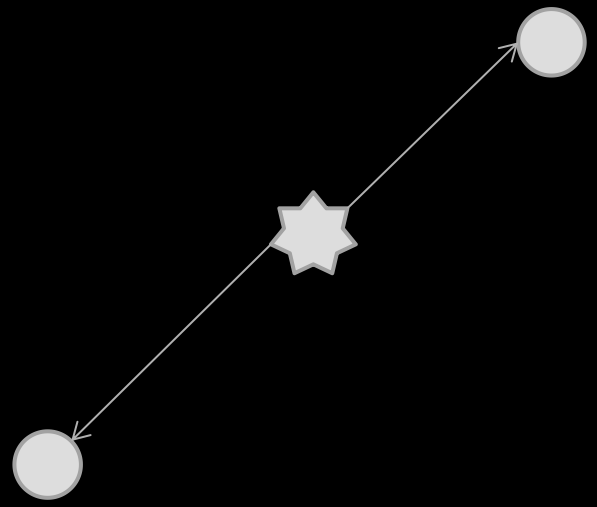
$$h(x, y) = | d(p, x) – d(p, y) |$$



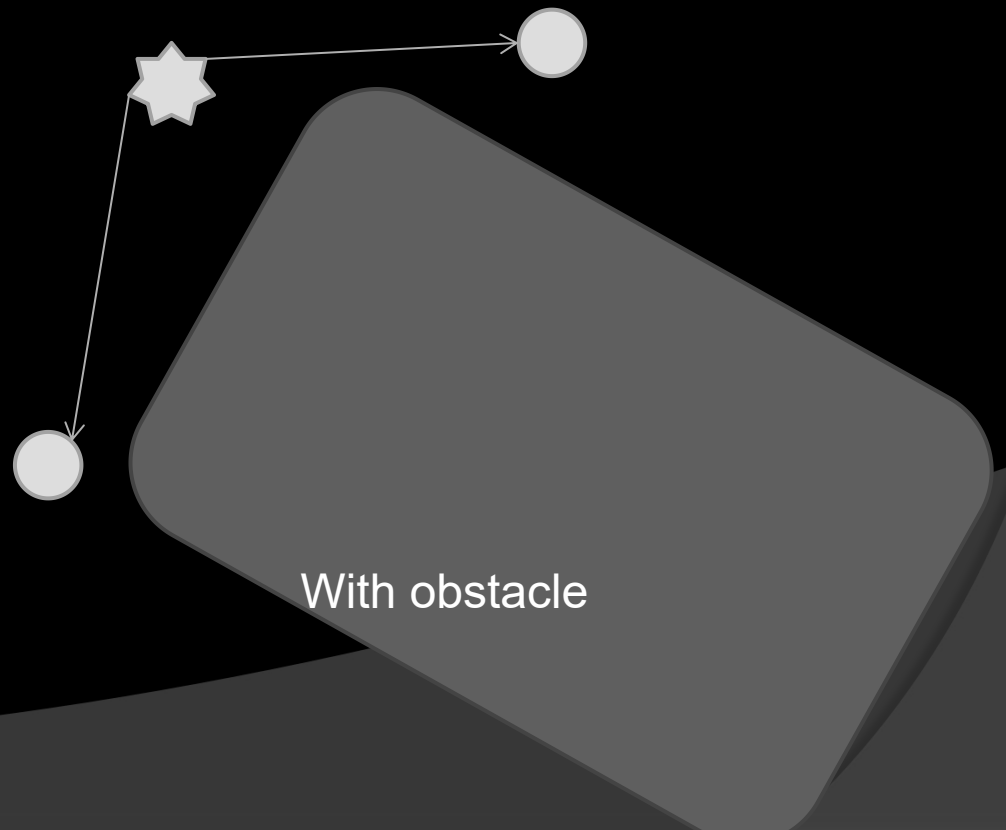
	P	x	y
P	null	2	3
x	null	null	null
y	null	null	null

Precompute Pivot nodes

- Only store complete data for few nodes



Without obstacle



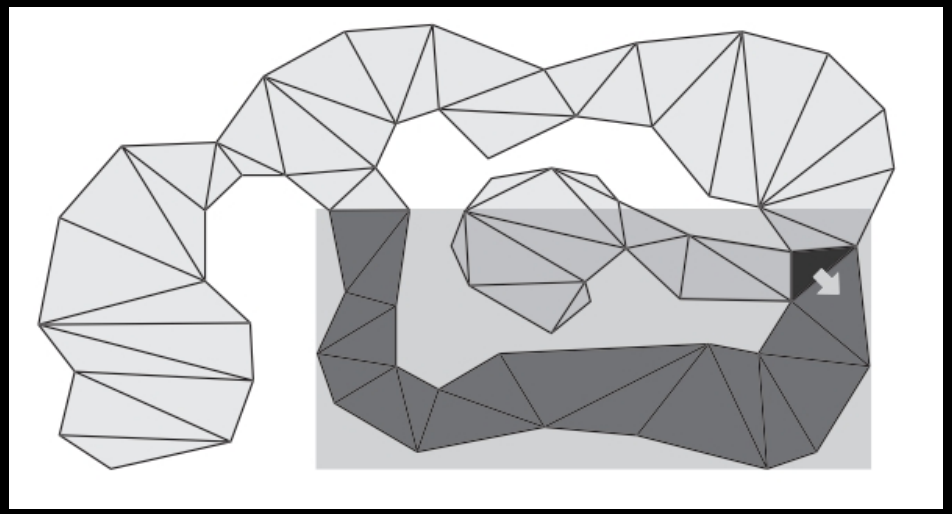
With obstacle

Precompute neighbors

- Get neighbors is one of the most common operations in an A* search
- Storing the neighbors of each node, rather than traversing more expensive data structures, can improve speed at the cost of additional memory

Precomputed bound boxes

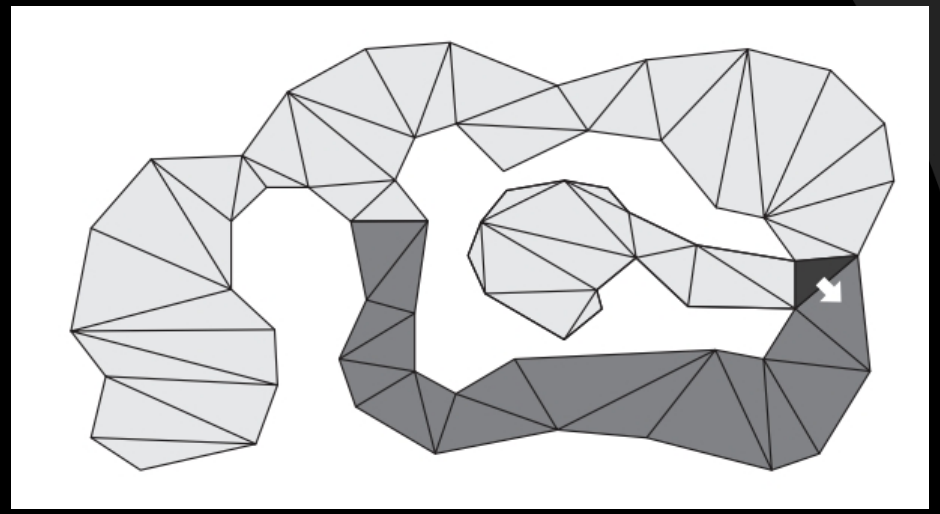
- A box area stored for an node edge.
- If the destination is within the bound box, then we should choose this direction.

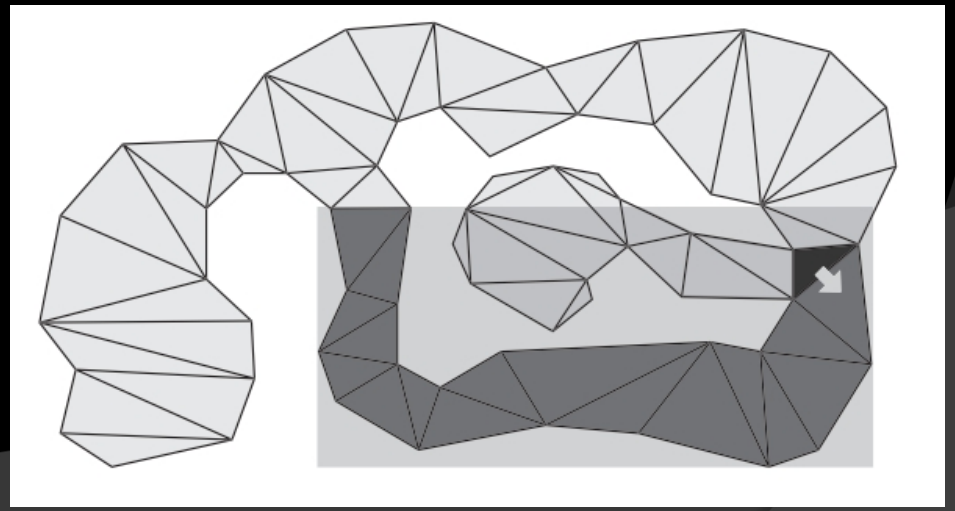


```
procedure AStarSearch(start, goal)
{
    Push (start, openlist)
    while (openlist is not empty)
    {
        n = PopLowestCost(openlist)

        if (n is goal)
            return success

        foreach (neighbor d in n)
        {
            if (WithinBoundingBox(n, d, goal))
            {
                // Process d in the standard A* manner
            }
        }

        Push (n, closedlist)
    }
    return failure
}
```

Precomputed bound boxes

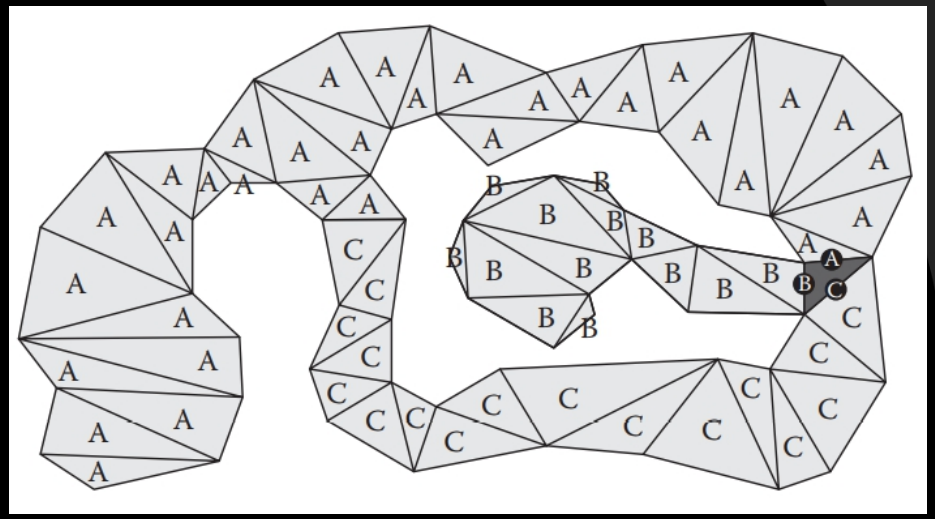
- Help the search algorithm to directly determine which neighbor should lead to shortest path.
- Generate bound boxes!

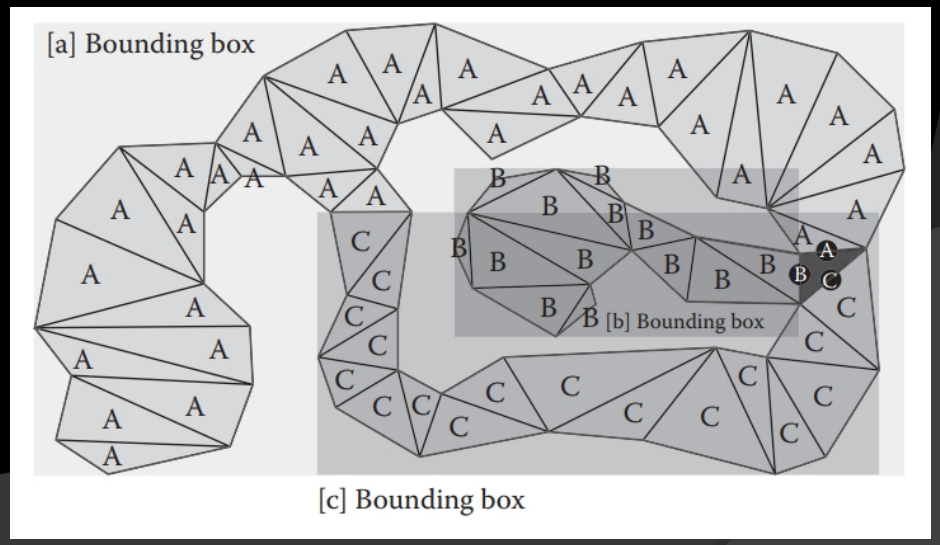




Precomputed bound boxes

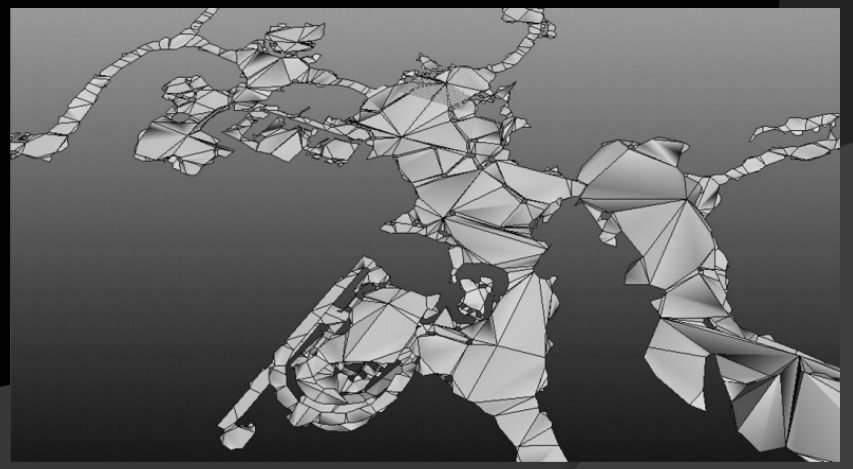
- Choose a start node, find all optimal destinations, and mark the destinations with corresponding starting edge.
- Generate a box area that contains all these destinations.
- Store the 4 vertices of the bound box to the start node.
- Update the next node





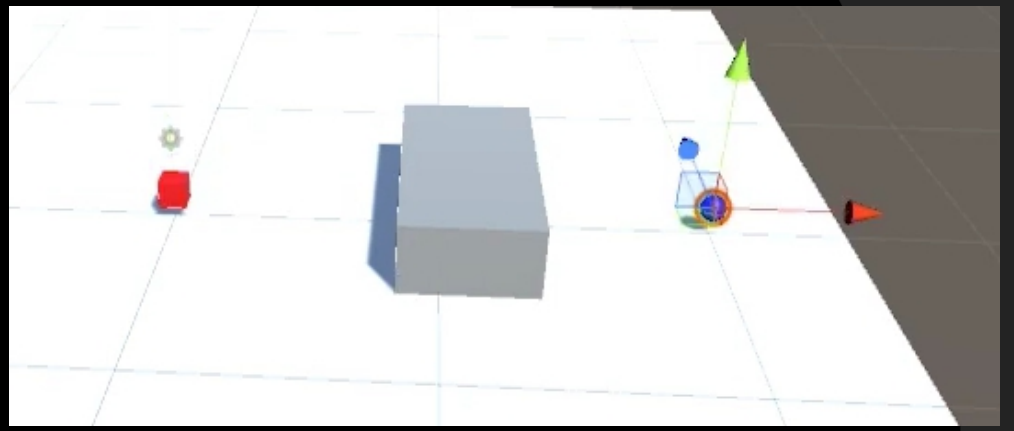
Optimization for large maps

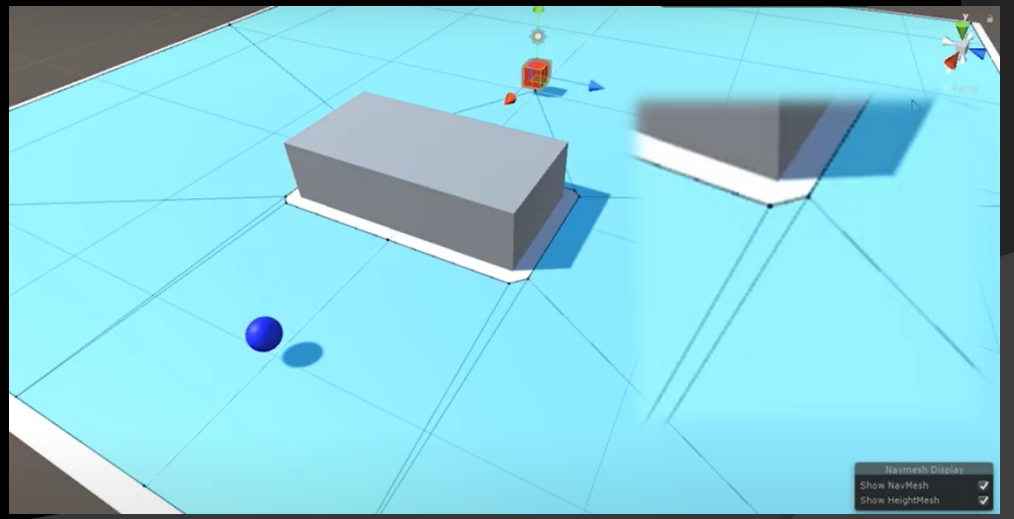
- Large game maps may not afford run-time search, especially in MMO game servers. Even bound boxes could be a huge computation burden. We must directly return the path ASAP.
- Precomputed pathfinding data becomes a common solution.
- The memory cost is unimaginable, given the scale of the map.
  - Think about the “Precompute all paths” example



Optimization for large maps

- Minimize Navmesh polygons
  - meanwhile make sure it matches the collision surfaces



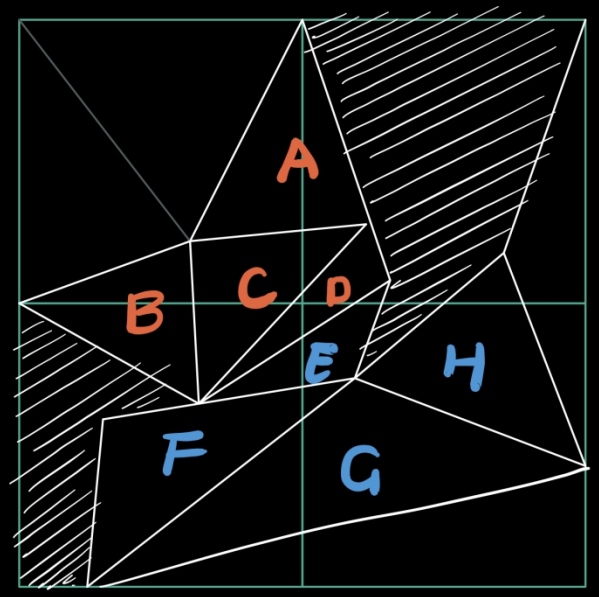


	A	B	C	D
A				
B				
C				
D				

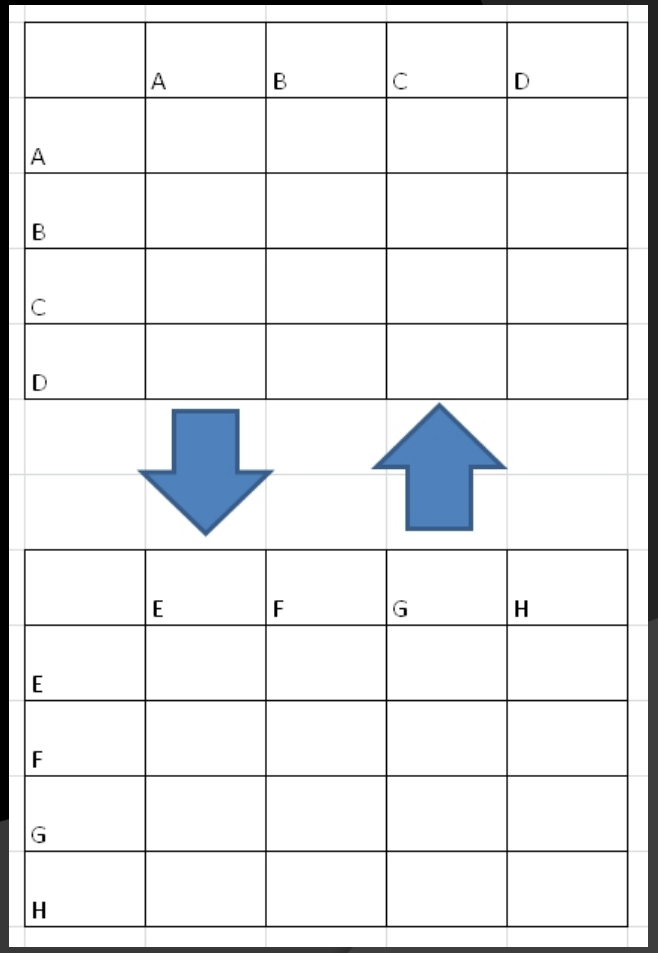
	A	B	C
A			
B			
C			

Optimization for large maps

- Group triangle nodes into larger components
- Each component has its own table
- Separate tables to store links



	A	B	C	D	E	F	G	H
A								
B								
C								
D								
E								
F								
G								
H								

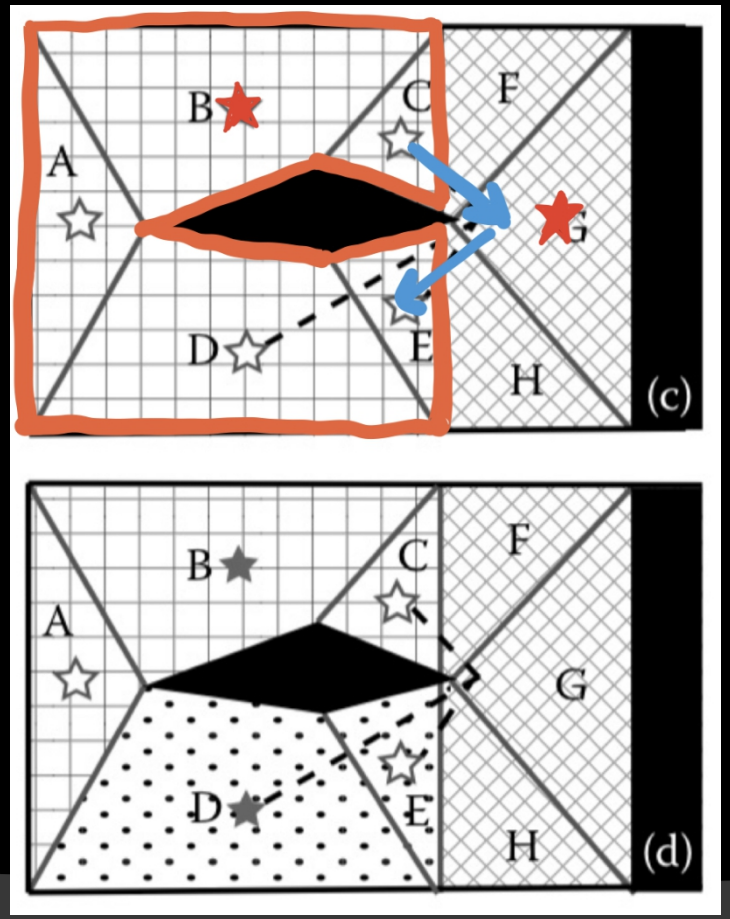


Optimization for large maps

- Hierarchical look up tables!
  - Split the data into blocks to save memory.
  - Use hierarchical structure to make search faster.
    - Say you are in Boston, and you want to travel to Beijing.
      - 1. From Boston, is Beijing in my local scope? NO
      - 2. From North America, can we go to Beijing's continent? YES (global table search)
      - 3. Goto Asia
      - 4. Zoom in and go to Beijing (local table data)

Optimization for large maps

- This might break the convex virtue of a node



- This might result in an infinite loop
  - If we want to go from C to E
  - At C: oh, E is local, lets do node-node search!
  - At F/G: what? We are out of local! Let's do global search!
  - Global search returns component center connection G to B
  - At B: here we go again

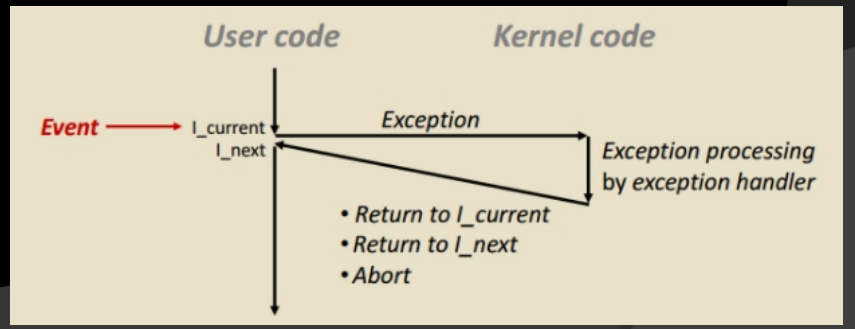
Optimization for large maps

- Optimizing falling edges to reduce table size
- Optimizing speed by using secondary goals
- How to support smooth paths
- . . .
  - You can read the paper to get the whole picture about how to build such a system!

```
ComputeConnectivity();
FallingMeshSetup();
BuildMeshTable();
for(int i = 0; i<max_level; ++i) {
    ComponentComputation(i);
    SplitProblematicComponents(i);
    ComputeComponentCenter(i);
    if (i+1 != max_level) {
        AddHierarchy(i);
        BuildHierarchicalTable(i);
    }
}
for(int i = max_level-1; i>0; --i) {
    RemoveInvalidSubLink(i);
}
```

Memory pre-allocation

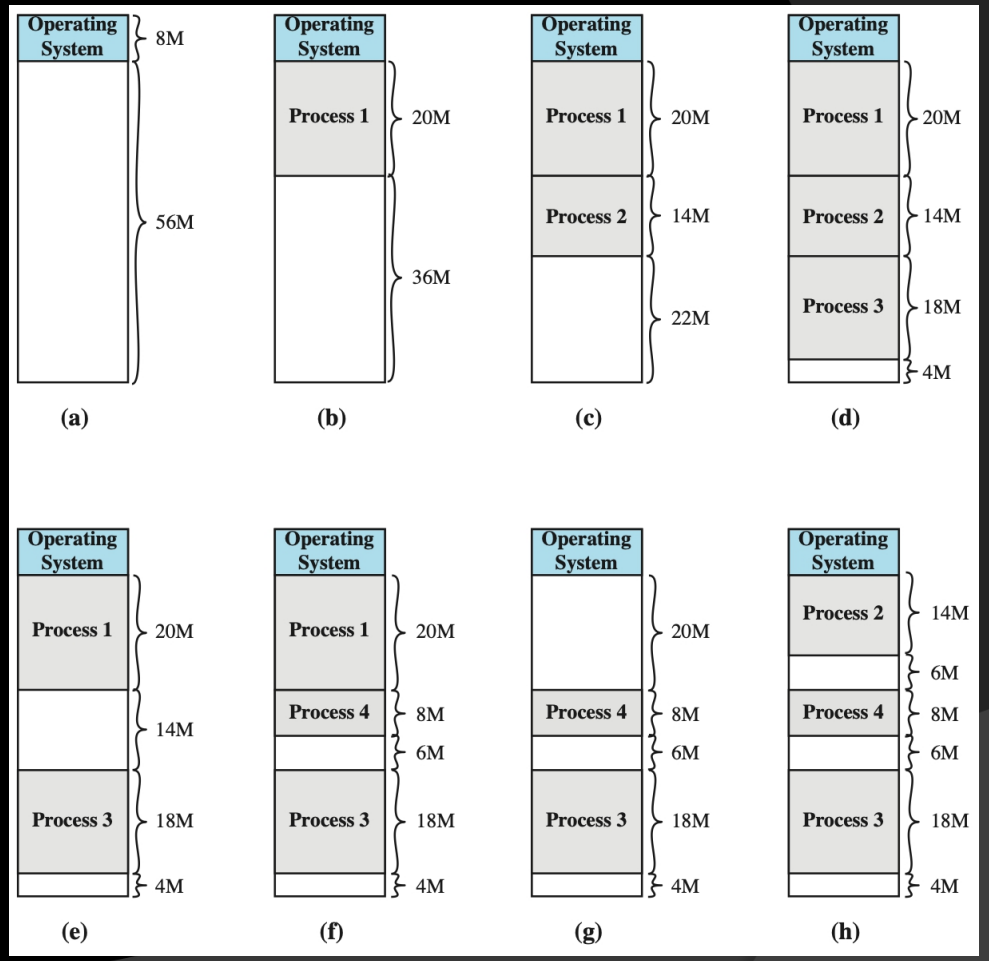
- Make sure to leave enough space for all data when game starts running, instead of allocate space for new data gradually.
- Anecdotally context switches take the order of 100s-1000s of cycles vs regular instructions taking 1-100 cycles.



Adapted from CS 5850 slide

Memory pre-allocation

- Causes memory fragment - incontineous “gaps” after allocation and deletion.



Adapted from CS 5850 slide

Thanks!

References

- http://www.gameaipro.com/GameAIPro/GameAIPro_Chapter17_Pathfinding_Architecture_Optimizations.pdf
  - A* optimizations
- http://www.gameaipro.com/GameAIPro3/GameAIPro3_Chapter22_Faster_A_Star_with_Goal_Bounding.pdf
  - Bound boxes
- http://www.gameaipro.com/GameAIPro/GameAIPro_Chapter20_Precomputed_Pathfinding_for_Large_and_Detailed_Worlds_on_MMO_Servers.pdf
  - Precomputed pathfinding for large game world
- https://www.youtube.com/watch?v=NGGoOa4BpmY
  - How to use Navmesh in Unity