Single Agent Se

UNIVERSITY of DENVER
DANIEL FELIX RITCHIE SCHOOL OF ENGINEERING & COMPUTER SCIENCE

Single Agent Search

Lecture 1
Intro to State Spaces & Algorithms

UNIVERSITY of DENVER
DANIEL FELIX RITCHIE SCHOOL OF ENGINEERING & COMPUTER SCIENCE

UNIVERSITY of DENVER
DANIEL FELIX RITCHIE SCHOOL OF ENGINEERING & COMPUTER SCIENCE

Missionaries and Cannibals

Problem due to Saul Amarel
Also known as jealous husbands

- Three missionaries and three cannibals must cross a river using a boat which can carry at most two people.
  - For both banks, if there are missionaries present on the bank, they cannot be outnumbered by cannibals (if they were, the cannibals would eat the missionaries).
  - The boat cannot cross the river by itself with no people on board.
- How do you get everyone across the river?

Solution

Left Bank	Right Bank
3M 3C	0M 0C
3M 1C	0M 2C
3M 2C	0M 1C
3M 0C	0M 3C
3M 1C	0M 2C
1M 1C	2M 2C
2M 2C	1M 1C
0M 2C	3M 1C
0M 3C	3M 0C
0M 1C	3M 2C
0M 2C	3M 1C
0M 0C	3M 3C

Domain Analysis

- How many states in the state space?
- How many states are illegal?
- Problem representation?

Search Problem

- Input:
  - State space representation
  - Start state
  - Goal state (may be implicit)
- Output
  - (Optimal) path between start and goal

State Space Representation

- State space is defined by either:
  - Successor function
  - Operator(s)
- Successor function
  - Given input state provides list of legal successors
- Operators
  - Given input state provides list of legal operators

State Space Representation

- Abstractly, state space is a graph G = (V, E)
  - Nodes/vertices are states
  - Edge for each successor
  - Find path in graph from start to goal
- All search algorithms are pathfinding in a graph
  - Graph might not fit in memory
- Search space can be *explicit* or *implicit*
- Goal state can be *explicit* or *implicit*

Path

- Set of states:
  - $s_1, s_2, \ldots s_i, s_{i+1}, \ldots s_N$
  - Where $s_{i+1} \in$successors(s_i)
- A *solution* if s_1 = start and s_N = goal
- *Optimal* path is the shortest path be
  states

Useful paradigm

- Often the state of a program is not easily encapsulated and/or testable
- If we can:
  - query the state for possible actions
  - apply and undo actions
- It is far easier to verify the correctness of an implementation

Domain Classification

- Space implicit or explicit?
- Goal implicit or explicit?
- State?
- Operator?
- Cost function?

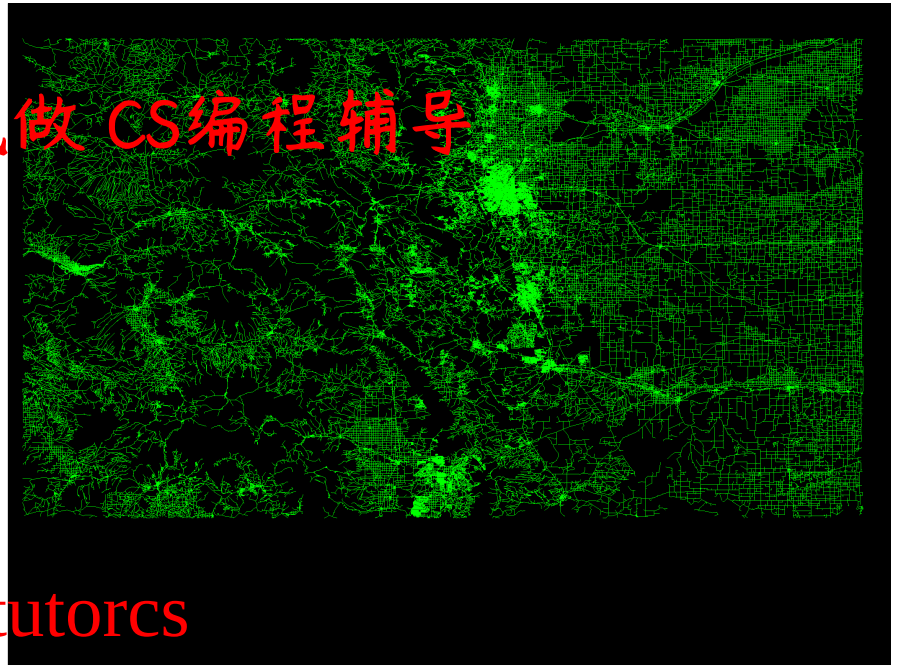
Pathfinding

- Pathfinding
  - Commonly used in robotics, computer games, map applications
  - Find path on 2D map between start and goal
- Potentially have multiple units and/or cooperation
- Potential have additional dimensions
  - Speed, heading

UNIVERSITY of DENVER
DANIEL FELIX RITCHIE SCHOOL OF ENGINEERING & COMPUTER SCIENCE

Planning in Road Networks

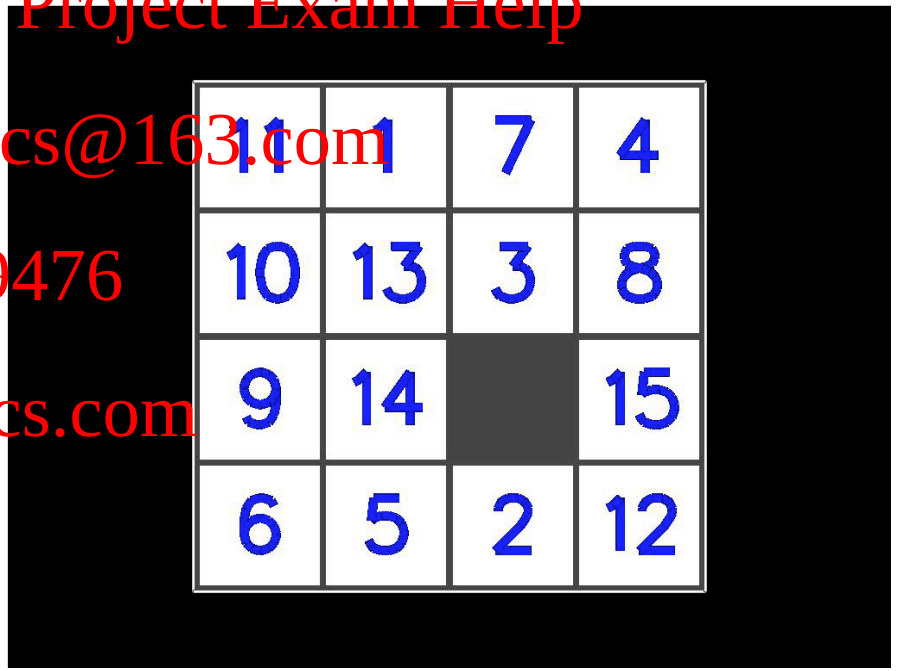
- European graph has ~30million nodes
- Find optimal path/length in <1ms
- For some techniques, longer paths



UNIVERSITY of DENVER
DANIEL FELIX RITCHIE SCHOOL OF ENGINEERING & COMPUTER SCIENCE

Sliding Tile Puzzle

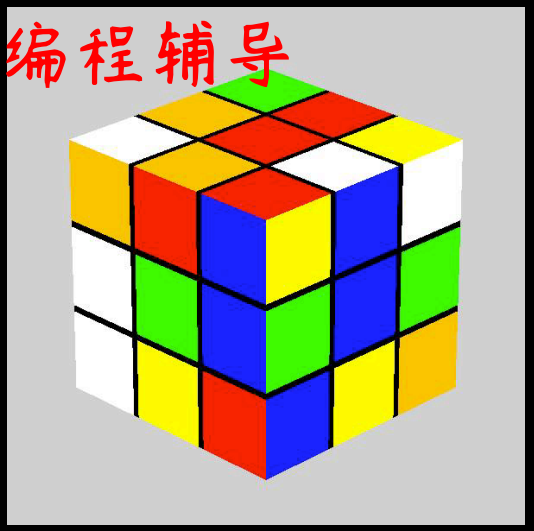
- Simple to represent, difficult to solve
- 8-puzzle has 9! = 362,880 states
- 15-puzzle has 16! = 10^{13} states
  - Half unreachable from the start
- Sam Loyd claimed to invent in the 1870's (*false*)
  - $1000 cash prize for solving

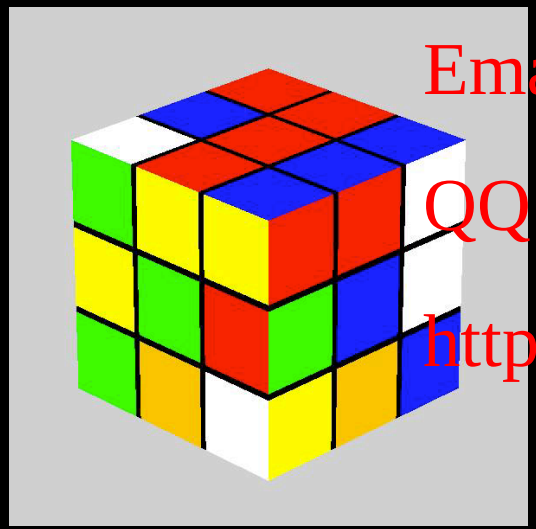


UNIVERSITY of DENVER
DANIEL FELIX RITCHIE SCHOOL OF ENGINEERING & COMPUTER SCIENCE

Rubik's Cube

- Invented in 1974 by Ernő Rubik
- 4.3×10^{19} states
- First solved optimally in 1997 (Korf)
- Up to 17 CPU-days to solve one ins
  - 100 CPU-years if using previous
- ≤ 20 moves to solve **any** position

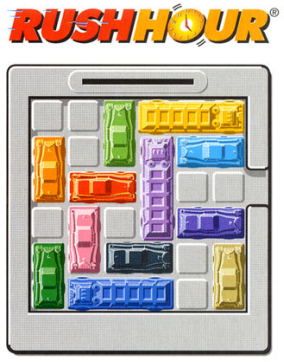




UNIVERSITY of DENVER
DANIEL FELIX RITCHIE SCHOOL OF ENGINEERING & COMPUTER SCIENCE

Rush Hour

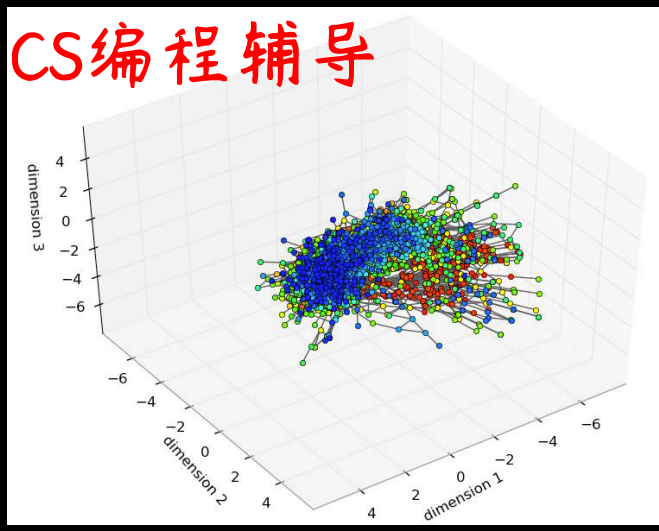
- Get red car out of maze
- Relatively easy for computer to solve



Edit Distance

- FOUR
- FIVE

FOUR -> FOUL -> FOIL -> FAIL -> FALL -> FILL -> FILE -> FIVE



Multiple Sequence Alignment

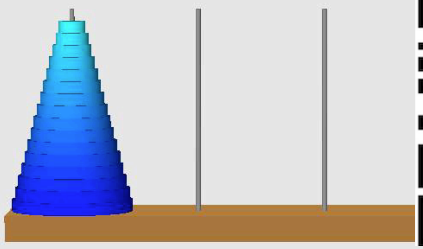
- A set of N sequences of DNA/RNA/protein
- A cost for matching, mismatching, blank
- Find optimal path through n-dimensional hyper-cube

```
      5'-GAATGTCCTTTCTCTAAGTCCTAAG
3'-GGAGACTTACAGGAAAGAGATTCAGGATTCAGGAGGCCTACCATGAAGATCAAG-5'

      5'-GAATGTCCTTTCTCTAAGTCCTAAGTCCTCCG
3'-GGAGACTTACAGGAAAGAGATTCAGGATTCAGGAGGCCTACCATGAAGATCAAG-5'

      5'-GAATGTCCTTTCTCTAAGTCCTAAGTCCTCCGG
3'-GGAGACTTACAGGAAAGAGATTCAGGATTCAGGAGGCCTACCATGAAGATCAAG-5'

      5'-GAATGTCCTTTCTCTAAGTCCTAAGTCCTCCGGATG
3'-GGAGACTTACAGGAAAGAGATTCAGGATTCAGGAGGCCTACCATGAAGATCAAG-5'

      5'-GAATGTCCTTTCTCTAAGTCCTAAGTCCTCCGGATGG
3'-GGAGACTTACAGGAAAGAGATTCAGGATTCAGGAGGCCTACCATGAAGATCAAG-5'

      5'-GAATGTCCTTTCTCTAAGTCCTAAGTCCTCCGGATGGTACTTCTAG
3'-GGAGACTTACAGGAAAGAGATTCAGGATTCAGGAGGCCTACCATGAAGATCAAG-5'
```


Top Spin

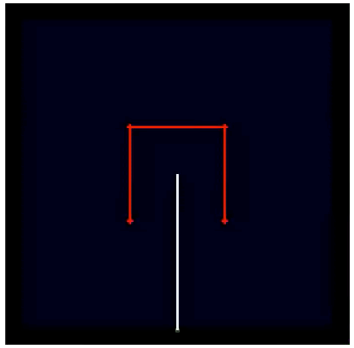
- Get tiles in order

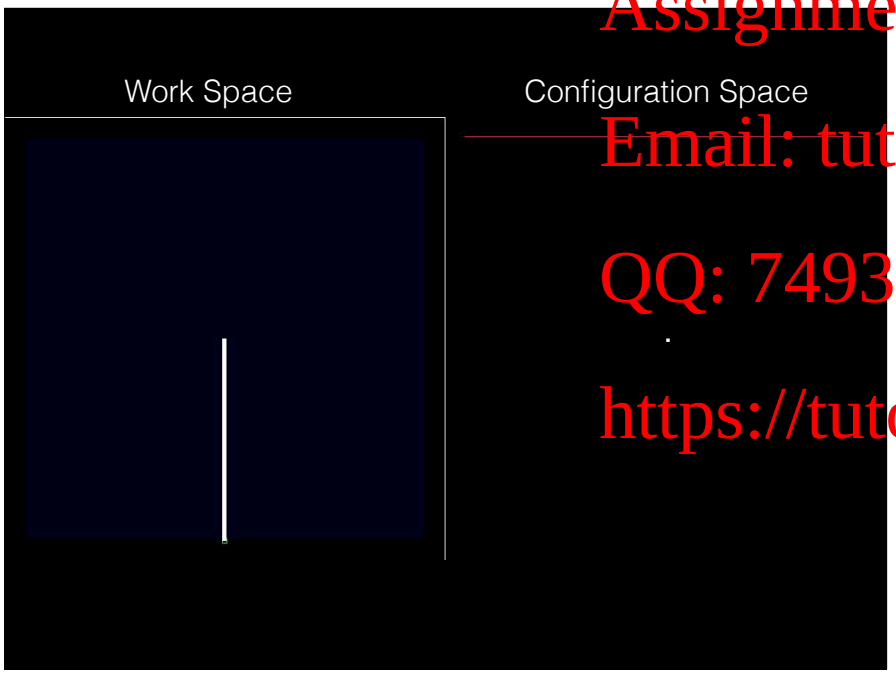
Towers of Hanoi



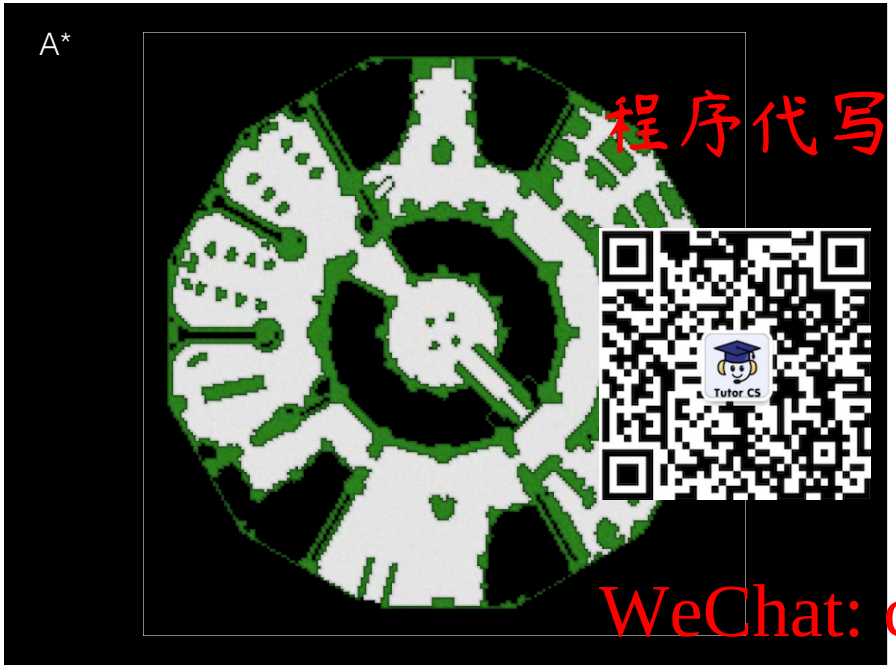
Robotic Arm

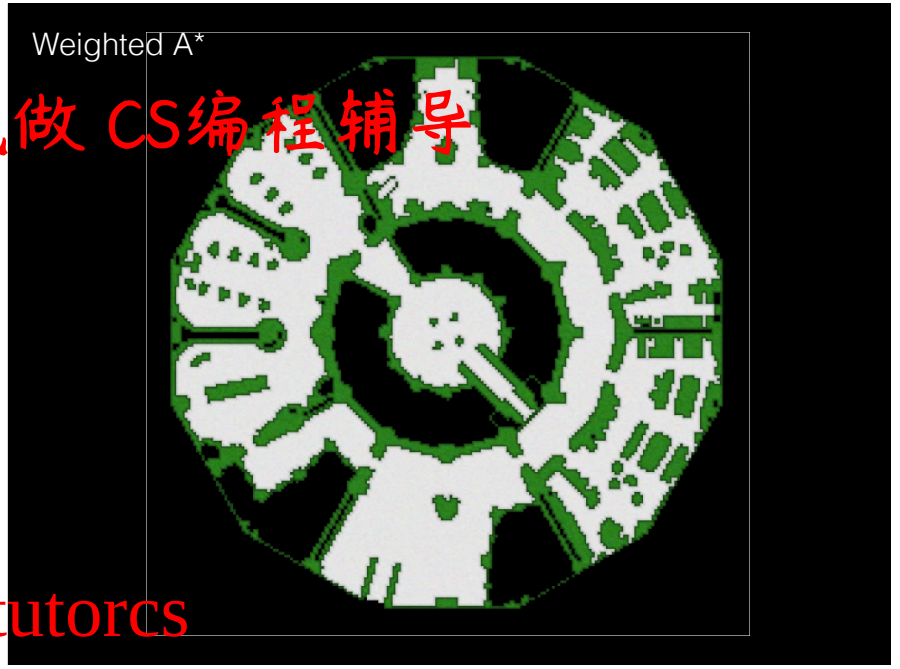
- Move the tip of the arm to a desired location
- State is location of the arms

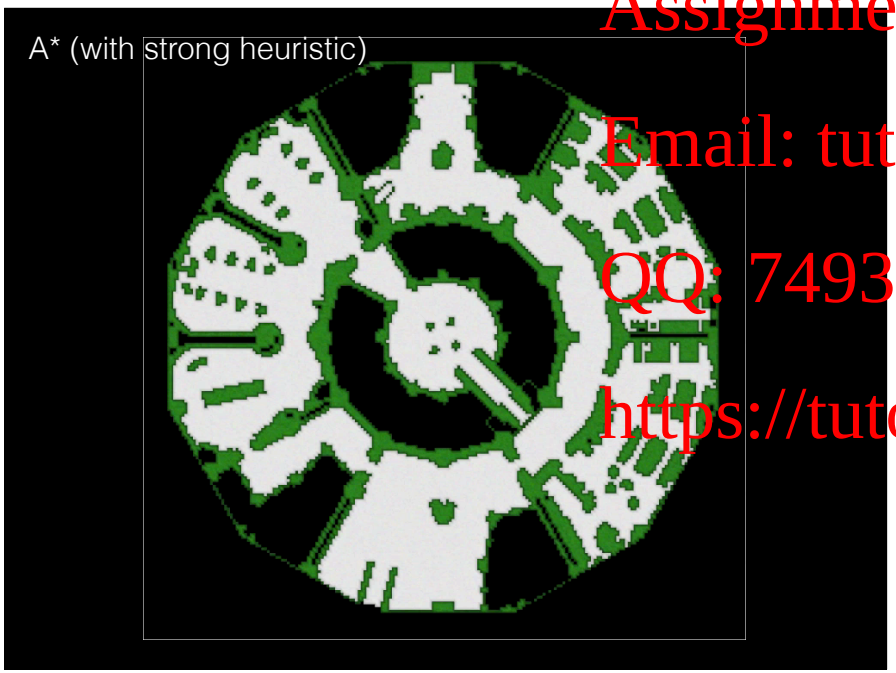




Sample Algorithms







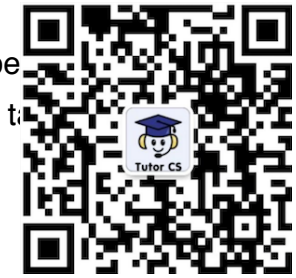
UNIVERSITY OF DENVER
DANIEL FELIX RITCHIE SCHOOL OF ENGINEERING & COMPUTER SCIENCE

CS (Graph Theory) vs. AI

- Graph theory also looks at **explicit** graphs
  - |V| vertices
  - |E| edges (At most $O(|V|^2)$ edges)
- AI often uses **implicit** graphs
  - Duplicate nodes hard to detect
- AI often assumes a constant branching factor

CS (Graph Theory) vs. AI

- CS considers time polynomial in (nodes + edges) efficient
- AI (usually) considers this too expe
  - representation of implicit states t
    log(nodes+edges)



CS (Graph Theory) vs. AI

- CS is willing to have everything stored in memory
  - polynomial in (nodes+edges)
- AI considers this exponential memory

Single Agent Search

Lecture 2
Search on Trees, Best-first search, Dijkstra

Tree Algorithms

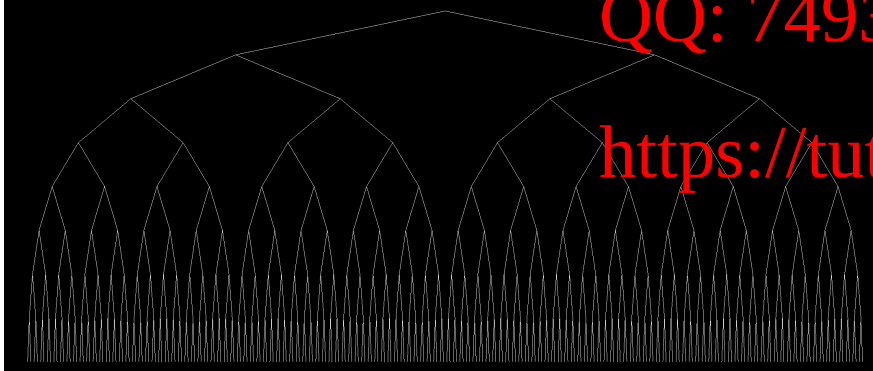
Initial Assumptions

- State space is a tree (no cycles or transpositions)
  - Branching Factor b
  - Depth d
  - Optimal solution cost C*
  - Single Goal state
- Uniform edge costs (assume 1)

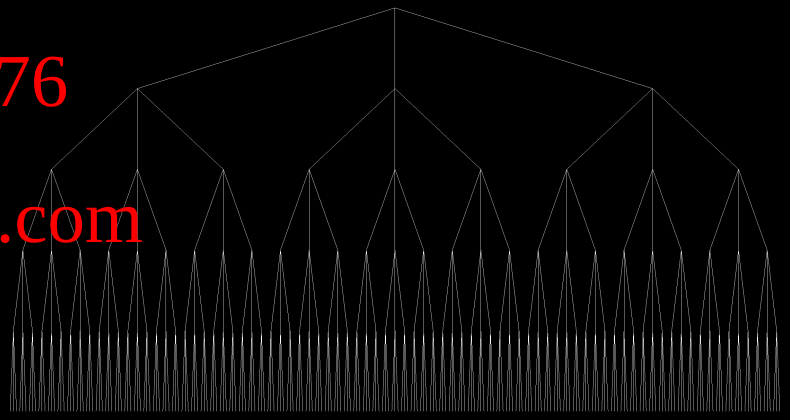
Initial Metrics

- Complete (finds solution if it exists)
- Optimal (finds optimal solution)
- Solution Quality (as ratio of optimal)
- Time Complexity
- Space Complexity

How many nodes in this tree?



How many nodes in this tree?



$$N(b,d) = 1 + b + b^2 + ... + b^{d-1} + b^d$$
$$bN(b,d) = b + b^2 + b^3 + ... + b^d + b^{d+1}$$
$$bN(b,d) - N(b,d) = b^{d+1} - 1$$
$$(b-1)N(b,d) = b^{d+1} - 1$$
$$N(b,d) = \frac{b^{d+1} - 1}{(b-1)}$$
$$N(b,d) = \frac{b(b^d) - 1}{(b-1)}$$
$$N(b,d) = \frac{b}{b-1}(b^d) - \frac{1}{b-1}$$
$$N(b,d) \approx \frac{b}{b-1}(b^d)$$

UNIVERSITY of DENVER
DANIEL FELIX RITCHIE SCHOOL OF ENGINEERING & COMPUTER SCIENCE

Algorithm 1: BFS (tree)

```
bool BFS(state from, state to)
{
  queue.push_back(from);

  while (queue.size() != 0)
  {
      if (env->GoalTest(queue.front(), to)) // later than necessary
        return true;

      env->GetSuccessors(queue.front(), succ);
      queue.pop_front();
      for (int x = 0; x < succ.size(); x++)
      {
        queue.push_back(succ[x]);
      }
  }
  return false;
}
```


UNIVERSITY of DENVER
DANIEL FELIX RITCHIE SCHOOL OF ENGINEERING & COMPUTER SCIENCE

BFS (tree) Complexity

- Complete - yes
- Optimal - yes
- Solution Quality - (optimal)
- Time Complexity - $O(b^d)$
- Space Complexity - $O(b^d)$

- Is it possible to do better?

UNIVERSITY of DENVER
DANIEL FELIX RITCHIE SCHOOL OF ENGINEERING & COMPUTER SCIENCE

Algorithm 2: DFS (tree)

```
bool DFS(state from, state to)
{
  queue.push_back(from);

  while (queue.size() != 0)
  {
      if (env->GoalTest(queue.back(), to))
        return true;

      env->GetSuccessors(queue.back(), succ);
      queue.pop_back();
      for (int x = succ.size()-1; x >= 0; x--)
      {
        queue.push_back(succ[x]);
      }
  }
  return false;
}
```

Implementation

- What do you keep in memory for BFS?
  - Have to keep copy of each state
- What do you keep in memory for DFS
  - Recursive formulation
  - Only have to keep a single copy of
  - Apply & undo moves to state

DFS (tree) Complexity

- Complete - yes
- Optimal - yes (*What if there is more than 1 goal?*)
- Solution Quality - (optimal)
- Time Complexity - $O(b^d)$
- Space Complexity - $O(b \cdot d)$

- Is it possible to do better?

DFS (general) Complexity

- Complete - yes
- Optimal - no
- Solution Quality - $O(b^d/d)$ (worst case)
- Time Complexity - $O(b^d)$ (or worse with duplicates)
- Space Complexity - $O(b^d)$ (worst case)

- Is it possible to do better?

Still assume unit edge costs

DFID

- Run DFS but bound the depth of the tree
  - 1, 2, 3…*d*

DFID Complexity

- Complete - yes
- Optimal - yes
- Solution Quality - (optimal)
- Time Complexity - $O(b^d)$
- Space Complexity - $O(b \cdot d)$

- Is it possible to do better?

DFID Time Complexity

$$DFID(b,d) = \sum_{i=0}^{d} N(b,d)$$

$$DFID(b,d) = \frac{b}{b-1}b^0 + \cdots + \frac{b}{b-1}b^{d-1} + \frac{b}{b-1}b^d$$

$$DFID(b,d) = \frac{b}{b-1}(b^0 + \cdots + b^{d-1} + b^d)$$

$$DFID(b,d) \approx \left(\frac{b}{b-1}\right)^2 (b^d)$$

$$DFID(b,d) = O(b^d)$$

DFID Time Optimality

- DFID is a time-optimal brute-force algorithm
- Proof by contradiction
  - b^d nodes at the level of the solution
  - If algorithm A uses less than b^d expansions it most not expand some node at level d
  - Create new problem -- swap goal and the unexpanded node -- then A won't find the solution

DFID Space Optimality

- O(d) space
- How much space must an algorithm use?
- If it has b^d time, must have $\log(b^d)$ space
  - d log(b) -- assume b is constant
- If algorithm is a FSM and runs K steps, each step must have unique state
  - At least a counter to distinguish between states

Underlying Assumptions

- What about problems with cycles?
- Pathfinding in a grid?
  - BFS - $O(r^2)$
  - DFID - $O(4^r)$
- Removing short cycles?
- Difference between explicit/implicit (m

- See how to improve this later in grids

Reconstruct Path

- How to reconstruct DFS path?
  - Path is sitting on stack
- How to reconstruct BFS path?
  - Can't do it with naive implementation

Activity

- DFS, DFID, BFS Demo
  - Run algorithms independently
    - Change goal and observe behavior
  - Run DFS and DFID
    - Best goal for DFS?
    - Best goal for DFID?
  - How does branching factor impact performance?

Best First Search

Best First Search

- Broad class of algorithms
- Can handle much more general probl
- Many different metrics of “best”
  - f(n) often represents the priority of

Best First Search

- Open List
  - All states that have been generated, but not expanded
- Closed List
  - All states that have been expanded
- Generate a state
  - Parent’s successors generated, placed on open
- Expand a state
  - State taken from open
  - Successors generated
  - Put on closed

Best First Search Notes

- Open and Closed lists aren’t actually lists
  - Open is usually a priority queue
    - Backed by a hash table for lookup
  - Closed is usually a hash table
- Implementation:
  - Start with lists (slow)
  - Then implement faster data structures

Pseudo-Code (1)

- Best-First Algorithm Pseudo-Code
  - Put start on OPEN
  - While(OPEN is not empty)
    - Pop best node n from OPEN
    - if (n == goal) return path(n, goal)
    - for each child of n // generate children
    - Update(n) // see next slide
  - Return NO PATH

Pseudo-Code (2)

- Update(n)
  - if n on closed
    - skip
  - if n on open
    - if found shorter path
      - update cost on open
  - else
    - add n to open

Algs as Best First Search

- DFS is Best First Search when
  - f(n) = depth (larger before smaller)
- BFS is Best First Search when
  - f(n) = depth (smaller before larger)

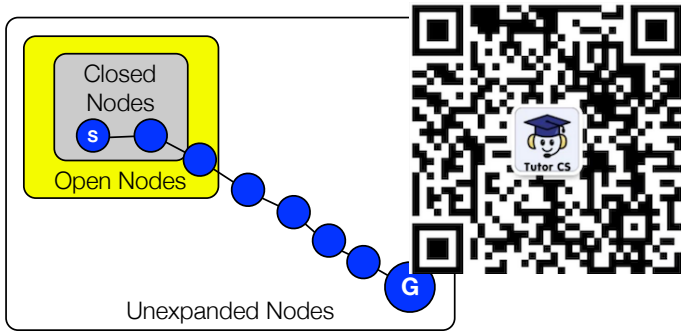
Dijkstra Algorithm

- g-cost is path cost to a node [written g(n)]
- Dijkstra is best-first search
  - f(n) = g(n)

Dijkstra Performance

- Complete - yes (Finite graph or minimum edge cost)
- Optimal - yes (Non-negative edges [for now])
- Solution Quality - (optimal)
- Time Complexity - $O(b^d)$ [?]
- Space Complexity - $O(b^d)$ [?]

High-Level View



Dijkstra Optimality

- Property #1
  - g-costs along any path are monotonically increasing
    - Assume non-negative costs
    - Therefore adding a node to a path, increases the cost

Dijkstra Optimality

- Property #2
  - A node on the optimal path (to any reachable state) is always on the open list with its optimal g-cost from the start. (Proof by induction)
    - Initially start is on OPEN
    - Assume step n; step n+1:
      - If node at n+1 is on optimal path, its successor will be on open
      - If not, the previous node will still be on OPEN

Single Agent Search

Lecture 3
Dijkstra, Heuristics

Dijkstra's Algor

Dijkstra Optimality

- Property #1
  - g-costs along any path are monotonically increasing
    - Assume non-negative costs
    - Therefore adding a node to a path, increases the cost

Dijkstra Optimality

- Property #2
  - A node on the optimal path (to any reachable state) is always on the open list with its optimal g-cost from the start. (Proof by induction)
    - Initially start is on OPEN
    - Assume step n; step n+1:
      - If node at n+1 is on optimal path, its successor will be on open
      - If not, the previous node will still be on OPEN

Dijkstra Optimality

- Property #3
  - Every closed node has optimal g-cost [induction]
  - Initially no states on closed
  - Assume at step n all states on closed have optimal g-cost
  - What if the state expanded at step n didn't have optimal g-cost?
    - There would be a node with lower g-cost (Prop. #1)
    - We would expand that node next
    - Therefore the node at n didn't have minimal g-cost

Dijkstra Optimality

- Property #4
  - Every node with finite cost will ever expanded (assume min edge cost
  - For any given depth *d*, there are at nodes at depth <= d.
  - This is finite, so for any finite cost o number of steps are needed.

$$N(b, d) = \frac{b^{d+1} - 1}{(b-1)}$$

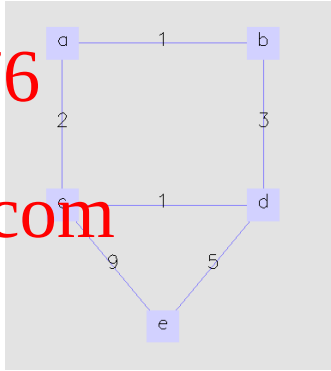
Dijkstra Optimality

- What can happen:
  - Run out of nodes on open
    - Can't happen if path exists (property #2)
  - Expand nodes forever and never reach goal
    - Assume minimum edge cost
    - Assume bounded branching factor
  - Expand the goal
    - Found it with optimal cost (property #3)
    - Find in finite # of steps (property #4)

Dijkstra Performance

- Complete - yes (Finite graph or minimum edge cost)
- Optimal - yes (Non-negative edges [for now])
- Solution Quality - (optimal)
- Time Complexity - worst case $O(b^d)$
- Space Complexity - worst case $O(b^d)$
  - Replace d with c/e if:
    - c is cost of solution
    - e is min edge cost
    - d is depth (# edges) in solution

Demo

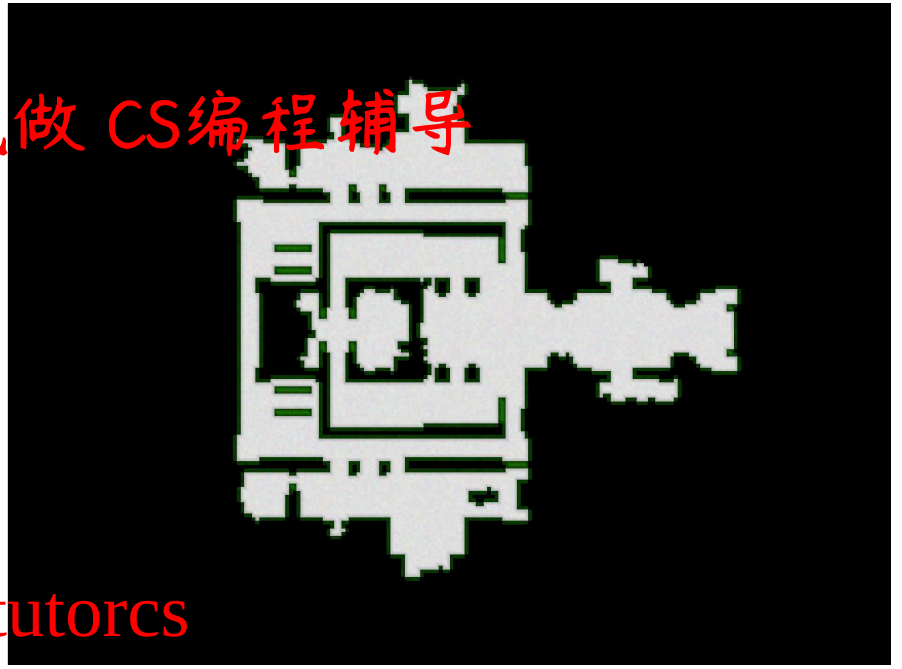


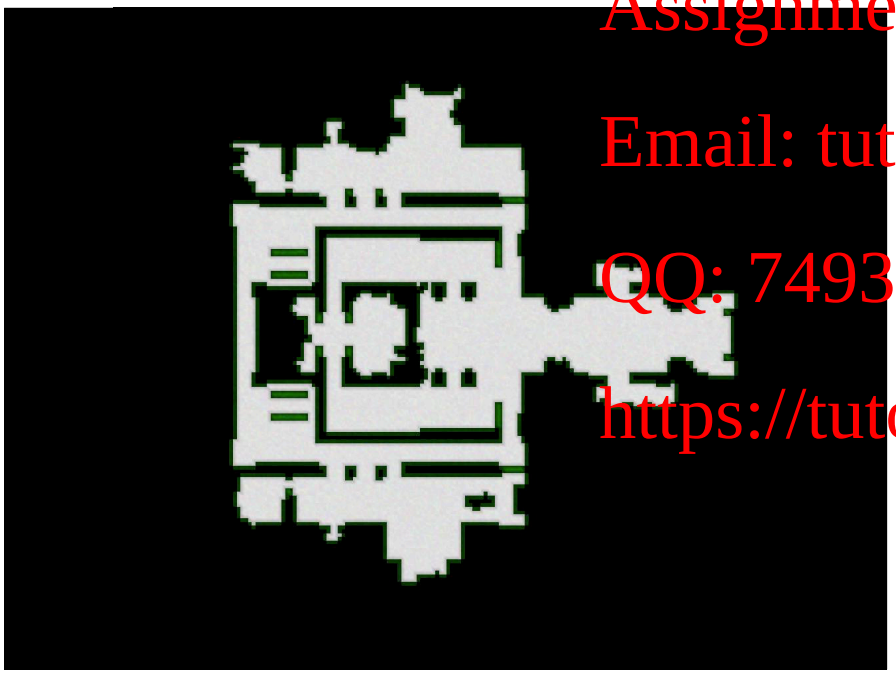
Create an example where Dijkstra's algorithm updates the cost of a state on open after it is generated

UNIVERSITY of DENVER
DANIEL FELIX RITCHIE SCHOOL OF ENGINEERING & COMPUTER SCIENCE

Motivation

- Search is uninformed
  - How can we make the search more





Heuristics

Heuristics

- What is a heuristic?
- Where do they come from?
  - Example heuristics
- Heuristic Properties

What is a heuristic?

- Distance estimate between two states
  - h*(a, b) = perfect heuristic

Distance Estimates

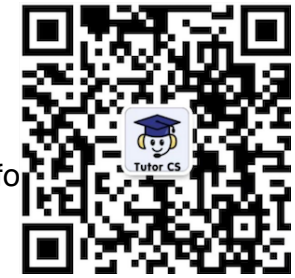
- Pathfinding on a 2D grid map?
- Sliding-Tile Puzzle?
- Pancake Puzzle?
- Rubik's Cube?

Where do they come from?

- Relaxations of original problem
  - Example STP
    - Swap(a, b)
  - Pre-conditions:
    - Blank(b)
    - Adjacent(a, b)
  - Post-conditions:
    - Blank(a), In(b, a)

Where do they come from?

- Abstractions of original problem
  - Blank out all tiles except the 1
  - Solve exactly
- Triangulation from other distance info



Heuristic Properties

- Distance estimate between two states
  - h*(a, b) = perfect heuristic
- Admissible heuristic - (non-overestimating)
  - $\forall_{a,b} h(a,b) \leq h^*(a,b)$
- Consistent heuristic (cannot drop more than cost of edge)

$$\forall_{a,b,s \in succ(a)} h(a,b) \leq c(a,s) + h(s,b)$$

$$|h(a,b) - h(s,b)| \leq c(a,s)$$

Consistency / Admissibility

- consistent → admissibility (yes)
- admissibility → consistency (no)

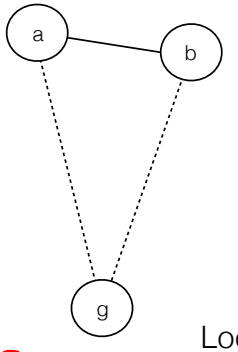
Single Agent Search

Lecture 4
Heuristics, A*, IDA*

UNIVERSITY of DENVER
DANIEL FELIX RITCHIE SCHOOL OF ENGINEERING & COMPUTER SCIENCE

Final Heuristic D

Consistency (Triangle Inequality)

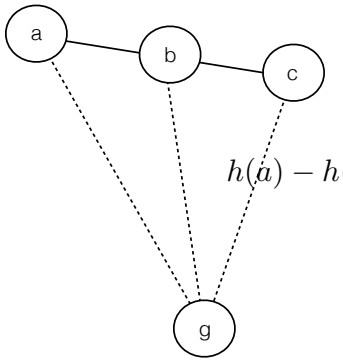


$$h(a) \leq c(a,b) + h(b)$$

$$h(a) - h(b) \leq c(a,b)$$

Local, since a & b are neighbors

Consistency (Triangle Inequality)



$$h(a) - h(b) \leq c(a,b)$$

$$h(b) - h(c) \leq c(b,c)$$

$$h(a) - h(b) + h(b) - h(c) \leq c(a,b) + c(b,c)$$

$$h(a) - h(c) \leq c(a,c)$$

This step requires c(a, c)= c(a, b)+c(b, c)

Global, since a & c are not neighbors

UNIVERSITY of DENVER
DANIEL FELIX RITCHIE SCHOOL OF ENGINEERING & COMPUTER SCIENCE

Admissibility from Consistency

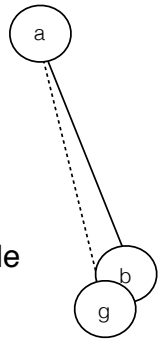
- What if b and g are the same node?
- Local consistency:

$$h(a) \leq h(b) + c(a,b)$$

$$h(a) \leq 0 + c(a,b)$$

- Global consistency is the same

- Consistent heuristics must be admissible



Pure Heuristic Search

- Best-First Algorithm
- f = h()
- Complete? / Will it find a solution?
  - Only on finite graph
- Optimal?
  - No
- Space, Time?
  - Undetermined

A*

A*

- Best-First Search
  - f = g+h
  - f is an estimate of the complete path length
- Optimality?
  - Not optimal unless we guarantee properties of our heuristic

A* Optimality

- Property #1
  - f-costs along any optimal path are monotonically increasing
    - Assume non-negative costs
    - Assume consistent heuristic

A* + Consistency

- f-cost is monotonic non-decreasing

$$h(a) \leq h(b) + c(a,b) \quad \text{(consistency}$$
$$g(a) + h(a) \leq g(a) + h(b) + c(a,b)$$
$$f(a) \leq g(b) + h(b)$$
$$f(a) \leq f(b)$$

Or, we won't follow the path from *a* to *b* in the search.

A* Optimality

- Property #1(a)
  - The f-cost of the start (and all states) is non-overestimating
    - Follows from admissibility/consistency

A* Optimality

- Property #2
  - A node on the optimal path (to any reachable state) is always on the open list with its optimal g-cost from the start. (Proof by induction)
    - Initially start is on OPEN
    - Assume step n; step n+1:
      - If node at n+1 is on optimal path, its successor will be on open
      - If not, the previous node will still be on OPEN

A* Optimality

- Property #3
  - Every closed node has optimal g-cost [induction]
    - Initially no states on closed
    - Assume at step n all closed states have optimal g-cost
      - What if the state expanded at step n didn't have optimal g-cost?
        - There would be a node with lower f-cost (Prop. #1)
        - We would expand that node next
        - Therefore the node at n didn't have minimal g-cost

A* Optimality

- Property #4
  - Every node with finite cost will ever expanded (assume min edge cost
  - For any given depth *d*, there are at nodes at depth <= d.
  - This is finite, so for any finite cost o number of steps are needed.

$$N(b, d) = \frac{b^{d+1} - 1}{(b-1)}$$

A* Optimality

- What can happen:
  - Run out of nodes on open
    - Can't happen if path exists (property #2)
  - Expand nodes forever and never reach goal
    - Assume minimum edge cost
    - Assume bounded branching factor
  - Expand the goal
    - Found it with optimal cost (property #3)
    - Find in finite # of steps (property #4)

A* Optimality Proof (1)

- Assume there exists an algorithm B that expands less nodes that A* on problem P
  - There must be some node *m* not expanded by B, but expanded by A*
  - f(m) = g(m) + h(m) < c
  - Create a new problem P' with new goal g'
  - Add an edge with cost h(*m*) to g'

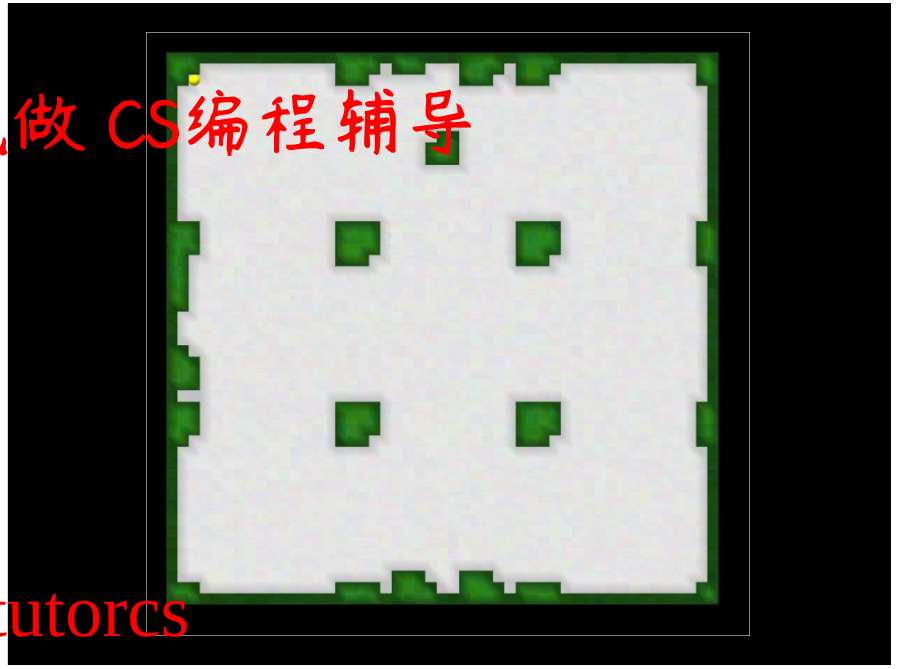
A* Optimality Proof (2)

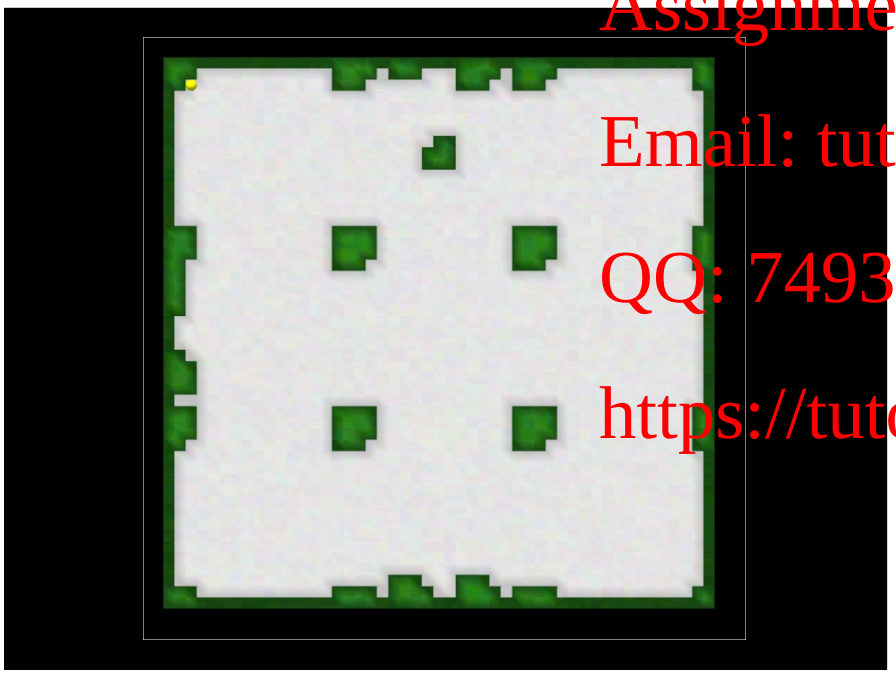
- Goal g' is now shorter than the original goal
  - g(g') = g(m) + c(m, g') = g(m) + h(m) = f(m) < c
  - g' will not be found by B, but will be found by A*
- Is our heuristic still consistent/admissible?
  - |h(m)-h(g')| ≤ c(m, g')
  - |c(m, g') - 0| ≤ c(m, g')

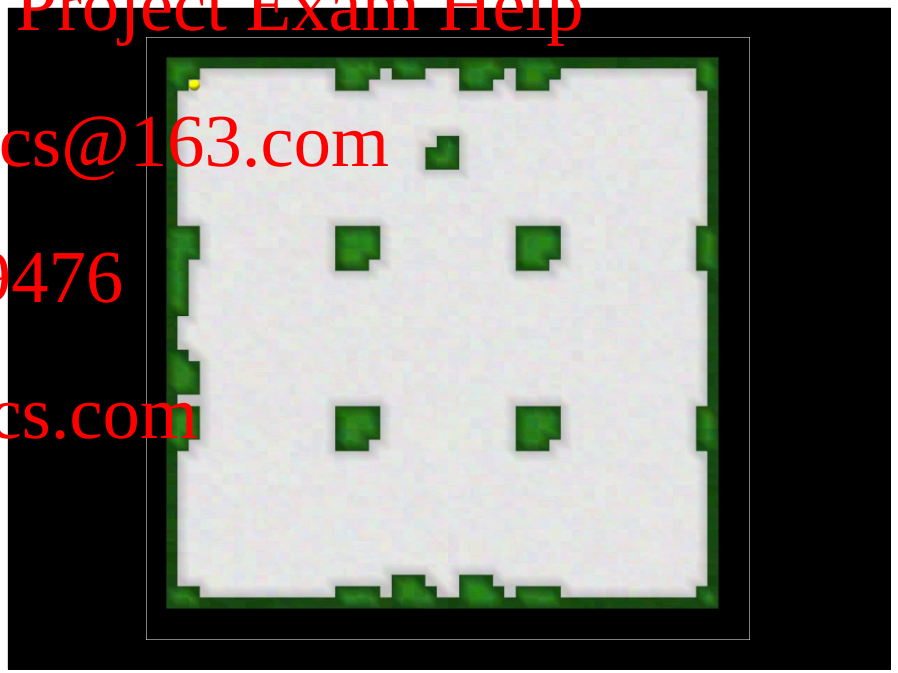
UNIVERSITY of DENVER
DANIEL FELIX RITCHIE SCHOOL OF ENGINEERING & COMPUTER SCIENCE

Can we make A* go faster?

- A*: f(n) = g(n) + h(n)
- Weighted A*: F(n) = (1-*w*)·g(*n*) + *w*·
- If *w* = 1?
  - Pure Heuristic Search
- If *w* = 0?
  - Dijkstra's







IDA*

IDA*

- Previously, BFS → DFS → DFID
- Now, A* → IDA*
- Perform DFS within f-cost limits
- Korf, 1985

IDA* Pseudo-Code

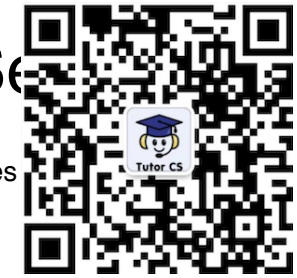
```
IDA*(start, goal)
limit ← f-cost(start)
do
   path = cost-limited-DFS(start, goal, limit)
   limit ← newlimit
while (!path)
return path
```


Cost limits

- How do we determine the next cost limit?
  - Keep track of the minimum f-cost larger than limit found during search
  - This is the next limit

Single Agent Se

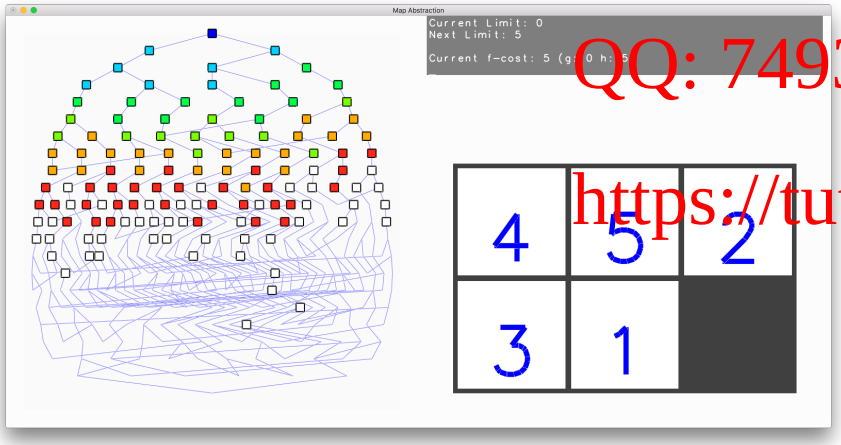
Lecture 5
IDA*, Pattern Databases



UNIVERSITY of DENVER
DANIEL FELIX RITCHIE SCHOOL OF ENGINEERING & COMPUTER SCIENCE

IDA*

IDA* Demo



UNIVERSITY of DENVER
DANIEL FELIX RITCHIE SCHOOL OF ENGINEERING & COMPUTER SCIENCE

Example

- [0 1 2 | 3 4 5]
- [0 1 2 | 4 3 5]
- Admissible heuristic

Termination (2)

- All path costs are strictly increasing
- All nodes with a given cost are exp
  iteration
  - Cost-limit strictly increasing
- At least 1 new node expanded eac
- No infinite-length paths of finite cos
- Must eventually expand the goal

Optimality

- Frontier -- nodes which have been generated but not expanded
  - Frontier always contains node on optimal path to goal
- Cost thresholds are monotonically increasing
- No thresholds > optimal path length
  - f(n) < optimal solution cost
- Goal has f(n) = g(n) -- no shorter solution
  - Cannot run with a threshold > g(goal)

Space Complexity

- Assume goal has cost c, minimum edge cost e
- Maximum depth of c/e (+1 for expanding at this depth)
- *e* is constant, so space is O(c)

Node Expansions

- How much work on last iteration of IDA*?
- Same set of nodes as A*
  - Except for tie-breaking

Node Expansions

- How much time in previous iterations?
  - Assume that the number of node cost *x* is N(*x*)
  - Then we usually assume N(x)/N(
  - The number of nodes grows exp factor of *b*) with each iteration
  - DFID analysis applies

Node Expansions (2)

- Worst-Case performance?
  - 1 more node expanded each step
  - 1, 2, 3, … b^d - 1, b^d = $O(b^{2d})$

Other Limitations of IDA*

- A* expands every state with cost < c
- IDA* expands every *node* with cost < c
  - eg. turns a graph into a tree -- doesn't detect duplicates
- What problems will IDA* work well in?
  - Sliding tile puzzle -- few cycles
- What problems will it not work well in?
  - Pathfinding -- √2 edge costs and lots of cycles

Pattern Databases

Sources of Heuristics

- Modern sources of heuristics:
  - Pattern Databases
  - True-Distance Heuristics
- Where do these work well?
- Where don't they work?

Heuristics as Relaxations

- Consider TSP - In solution:
  - All cities must be included in path
  - Each city must have two incident edges
  - Graph must be connected
- What happens if we relax/remove condition…
  - Use a MST
  - Solve sub-problems independently

Heuristics as Relaxations

- Consider route finding on a map<=>graph
  - Must travel edges
  - Otherwise just go straight to goal (euclidean)
- Another example?

Logic representation

- International planning competitions represent problems in generic language(s)
- STRIPS (Stanford Research Institute Problem Solver) – became name of description language
  - Preconditions -- things that must be true to apply an action
  - Postconditions (effects) -- how the state changes when an action is applied
  - represented as add and delete lists

2x2 sliding tile puzzle

- adjacent([0, 0], [0, 1])
- adjacent([0, 0], [1, 0])
- adjacent([1, 0], [1, 1])
- adjacent([0, 1], [1, 1])

- at([1, 0], 2)
- at([0, 1
- at([1, 1

- at([0, 0], 3)

Goal

- at([0, 0], 0)
- at([1, 0], 1)
- at([0, 1], 2)
- at([1, 1], 3)

Action: Move(x, loc_1, loc_2)

- Preconditions:
  - at([loc_1], 0)
  - at([loc_2], x)
  - adjacent([loc_1], [loc_2])
- Postconditions/effects:
  - at([loc_2], 0)
  - ¬at([loc_1], 0)
  - at([loc_1], x)
  - ¬at([loc_2], x)

How do we build heuristics?

- First method:
  - Relax preconditions & solve exactly
  - What happens if we relax:
    - at([loc_1], 0)?
    - at([loc_2], x)?
    - adjacent([loc_1], [loc_2])?

How do build heuristics?

- Second method?
  - Ignore “delete” effects of postcon
  - What happens to state?
    - Tile can be in multiple position
  - Apply all possible moves at each

Properties

- Will these methods produce admissible heuristics?
  - Consider that the search space is a graph
  - These methods add edges to the graph
  - Never remove edges
- Therefore, the result must be an admissible heuristic

Abstraction

- One generalized type of abstraction is one where edges are added into the search space (S)
  - Form an “edge supergraph” (T)
  - T contains all the edges in S plus possibly additional edges

Valtorta’s Theorem

- Theorem: If T is an edge super graph of S, and distances in T are computed by BFS, and A* with distances in T as its heuristic is used to solve problem P, then for any s∈S that is necessarily expanded if BFS is used to solve P, either:
  - s is expanded by A* in S, or
  - s is expanded by BFS in T
  - (BFS is reverse search)

How can we make this work

- Possibilities:
  - Pre-compute abstraction values
  - Decompose the heuristic comput
  - Use a different type of abstractio

Review

- Valtorta's Theorem
  - Every node expanded by BFS in the original graph will be expanded by either the BFS in the supergraph or by A* in the original graph
- Let ϕ be a mapping from states to abstract states
  - ϕ should be a surjective function

Generalized Valtorta's Theorem (Holte)

- If ϕ(S) is any abstraction of S, for any s∈S that is necessarily expanded if BFS is used to solve problem P, if A* is used to solve P using distances in ϕ(S) computed by BFS as its heuristic, then either:
  - s is expanded by A* in S, or
  - ϕ(s) is expanded by BFS in ϕ(S)

How do we get savings?

- If a large number of states are mapped into a single abstract state, there is a large reduction in search in the abstract state space
  - We only have to touch 1 node in the abstract space instead of many nodes

Single Agent Se

Lecture 6
Pattern Databases

UNIVERSITY of DENVER
DANIEL FELIX RITCHIE SCHOOL OF ENGINEERING & COMPUTER SCIENCE

Extra Bonus: Proving Unreachable States

- Sliding tile puzzle parity
  - Find any function which is invariant for all moves
  - Show that two states have different values for that function
- 8-puzzle is the number of swapped pairs
  - How many tiles to the "right" are smaller
  - (excluding the blank)
- 15-puzzle is swapped pairs plus row of blank

PDBs

Generalized Valtorta's Theorem (Holte)

- If $\phi(S)$ is any abstraction of S, for any $s \in S$ that is necessarily expanded if BFS is used to solve problem P, if A* is used to solve P using distances in $\phi(S)$ computed by BFS as its heuristic, then either:
  - s is expanded by A* in S, or
  - $\phi(s)$ is expanded by BFS in $\phi(S)$

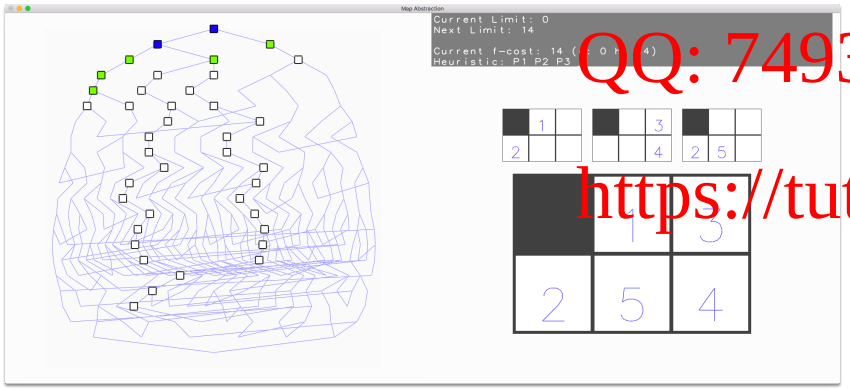
Domain Abstraction

- Take states and replace some values with blanks / colors
  - (0 1 2 3 4 5 6 7 8 9)
  - (0 1 2 3 4 5 6 7 8 9)
  - (0 * 2 * 4 * 6 * 8 *)
- Extreme example
  - (0 * * * * * * * * *)
- ϕ(S) is this mapping function

PDB Idea

- Apply a domain abstraction
  - In the abstract state space, perform a BFS from the goal
- To get a heuristic from state *s*, apply domain abstraction to *s* and lookup cost in BFS
  - Need an efficient way to store and lookup results of BFS

Demo



Why does this work

- If the problem space grows as b^d
  - The solution length grows with d
  - Suppose *w* is the state space width
- Uniformly abstract *k* states together
  - What is the new width?
  - $b^h = b^w/k$

Maximum heuristic after abstraction

- $b^h = b^w/k$
- $\log(b^h) = \log(b^d)-\log(k)$
- $h \cdot \log(b) = d \cdot \log(b)-\log(k)$
- $h = d - \log(k)/\log(b)$
- $h = d-\log_b(k)$

Methodology

- Pattern Database
  - Precomputation of values
  - Single goal state
  - BFS from goal in abstract state space

Ranking/Unranking

Hash Functions

- Where do I need a hash function?
  - Store closed/open list, pattern databases
- Given a problem state, how do we compute a perfect hash function?
  - NxM map? y·N+x
  - Sliding tile puzzle?

Sliding Tile Puzzle (1)

- Use 8-puzzle as example
  - 9 tiles (0...8)
  - Use fixed size for each tile
  - [7 4 1 5 3 0 6 2 8]
  - 8 + 9*(2 + 9*(6 + 9*(0 + 9*(3 + 9*(5 + 9*(1
  - $9^0 \cdot 8 + 9^1 \cdot 2 + 9^2 \cdot 6 + 9^3 \cdot 0 + 9^4 \cdot 3 + 9^5 \cdot 5 +$
  - = 321,305,804
  - but, only 9! = 362,880 states!

Ranking / Unranking

- A ranking function converts a permutation into an index
- An unranking function converts and index into a permutation

Small Example

- [1 2 3 0] ⇒ 01 10 11 00
  - $1 \cdot 4^3 + 2 \cdot 4^2 + 3 \cdot 4^1 + 0 \cdot 4^0$
- [3 0 1 2] ⇒ 11 00 01 10
  - $3 \cdot 4^3 + 0 \cdot 4^2 + 1 \cdot 4^1 + 2 \cdot 4^0$
- 8 bits -- 256 combinations
- 4! -- 24 actual combinations

Small Example (cont'd)

- [1 2 3 0] ⇒ 1·3! + 2·2! + 3·1! + 0·0!
  - Not quite right -- need to reduce values!
    - 1·3! + 1·2! + 1·1! + 0·0! = 6+2+1=9
- [3 0 1 2] ⇒ 3·3! + 0·2! + 0·1! + 0·0! = 18
- [0 1 2 3] ⇒ 0, [3,2,1,0] ⇒ 3·3! + 2·2! + 1·1! = 23
- This is a ranking function

Unranking?

- Unrank 18?
  - 18/3! = 3; 18mod3! = 0
  - 0/2! = 0; 0mod2! = 0
  - 0/1! = 0; 0mod1! = 0
  - last digit always 0
  - 3 0 0 0 ⇒ 3 0 1 2

Unranking

- Unrank 9
  - 9/3! = 1; 9mod3! = 3
  - 3/2! = 1; 3mod2! = 1
  - 1/1! = 1; 0mod1! = 0
  - last digit always 0
  - 1 1 1 0 ⇒ 1 2 3 0

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

Time complexity?

- We start in our representation
- $O(n^2)$ time to convert
  - Values in array must continually be re-numbered

Change of Topic (not really)

- How do I randomize order of elts in an array
- From array size N
  - select random element
  - move to position N
  - randomize array size N-1
- Time?
  - O(N)

Use this idea.

- What if my random number generator isn't random
  - Instead, use the ranking to tell us
- N = 5, r = 81, π = [0, 1, 2, 3, 4]
- 81mod5 = 1, π = [0, 4, 2, 3, 1]
- ⌊81/5⌋ =16
- N = 4, r = 16, π = [0, 4 2, 3]

N = 5, r = 81, π = [0, 1, 2, 3, 4]

81mod5 = 1, π = [0, 4, 2, 3, 1], ⌊81/5⌋ = 16

N = 4, r = 16, π = [0, 4, 2, 3, 1]

16mod4 = 0, π = [3, 4, 2, 0, 1], ⌊16/4⌋ = 4

N = 3, r = 4, π = [3, 4, 2, 0, 1]

4mod3 = 1, π = [3, 2, 4, 0, 1], ⌊4/3⌋ = 1

N=2, r=1, π = [3, 2, 4, 0, 1]

1mod2 = 1, π = [3, 2, 4, 0, 1], ⌊1/2⌋ =0

Pseudo-code

n = # of elements in permutation

r = ranking (hash key)

π = [0, 1, 2, …, *n*]

unrank1(*n*, *r*, π)

```
if (n > 0)
    swap(π[n-1], π[r mod n]);
    unrank1( n-1, ⌊r/n⌋ , π)
```


Ranking a permutation

- Slightly harder code to do the reverse process

Idea:

- Turn permutation into the identity permutation
- Reverses the process that we followed before

Ranking a permutation

- π = original permutation
- $\pi^{-1}[\ \pi[i]\ ] = i$ for $i = 0 \ldots n-1$
  - Essentially setting "which index c
  - π = [3, 2, 4, 0, 1]
  - π^{-1} = [3, 4, 1, 0, 2]

Pseudo-code

n = # of elements in permutation

π = permutation, π^{-1} = inverse

rank1(n, π, π^{-1})

```
if (n == 1) return 0;
    s = π[n-1]
    swap(π[n-1], π[ π^-1[n-1] ]);
    swap(π^-1[s], π^-1[n-1]);
    return s + n·rank1(n-1, π, π^-1)
```


N = 5, π = [3, 2, 4, 0, 1], π^{-1} = [3, 4, 1, 0, 2]

s = 1, swap(π[4], π[2]), swap(π^{-1}[1], π^{-1}[4])

N = 4, π = [3, 2, 1, 0, 4], π^{-1} = [3, 2, 1, 0, 4]

s = 0, swap(π[3], π[0]), swap(π^{-1}[0], π^{-1}[3])

N = 3, π = [0, 2, 1, 3, 4], π^{-1} = [0, 2, 1, 3, 4]

s = 1, swap(π[2], π[1]), swap(π^{-1}[1], π^{-1}[2])

N = 2, π = [0, 1, 2, 3, 4], π^{-1} = [0, 1, 2, 3, 4]

s = 1, swap(π[1], π[1]), swap(π^{-1}[1], π^{-1}[1])

N = 1 (return 0)

1 + 5*[0+4*[1+3*[1+2*0]]]

PDB computation

- We just want to rank the given set of tiles
- Note there are n!/(n-k)! states in the PDB when you have n items in the permutation and you only keep k of them
- Use an alternate dual / representation
- For each of the tiles in our pattern, rank the location
  - Where is the first tile in my pattern found?
- Ignore other tiles

PDB Example 1

- PDB Tiles: (0 5)
- Original permutation: (5 1 4 0 2 3)
- Dual (relative to tiles): (3 0)
  - 0 is in location 3, 5 is in location 0
- Mixed Radix representation: (3_6 0_5)
- Ranking:
  - 3·5!/4! + 0·4!/4!

PDB Example 2

- PDB Tiles: (0 3 5)
- Original permutation: (4 0 1 5 3 2)
- Dual (relative to tiles): (1 4 2)
  - 0 in location 1, 3 in location 4, 5 in location 2
- Mixed Radix representation: (1_6 3_5 1_4)
- Ranking:
  - 1·5!/3! + 3·4!/3! + 1

Single Agent Search

Lecture 7
Enhanced Pattern Databases

In class give out an example of a PDB. Have people do the compression (DIV / MOD). Have people do a delta computation. Let the see the difference directly.

UNIVERSITY of DENVER
DANIEL FELIX RITCHIE SCHOOL OF ENGINEERING & COMPUTER SCIENCE

Enhanced PDBs

- Additive
  - Add values together from disjoint PDBs
- Compressed
  - Min Compression
  - Partial Pattern Databases
  - Bloom Filters
- Dual
- Symmetry

Additive PDBs

- In regular PDBs we counted all actions as cost 1 whether or not they moved a tile in th
- Alternatively: Only count action costs tile in the pattern
  - If every action only moves a single results in an additive PDB
- Complicates building the PDB, becau 0-cost actions
  - Need to avoid cycles in 0-cost actions

Example

- Manhattan Distance
  - Can be seen as relaxation of logical formulation
  - Can also be seen as N-1 additive PDBs
    - Pre-compute and store distance from each tile to its goal location
    - Add the results together (instead of max)

Min Compression

- Reduce the memory used by a PDB
- Choose a function which maps the range of ranks into a smaller range of ranks
  - Could use hash function, div, mod
  - Take the min of all entries that map to the smaller range to preserve admissibility

Min Compression: DIV

- To compress by a factor of N:
  - Compute rank, divide by N
  - Example: N = 2

Original PDB:

1	2	3	4	5	6	7	8

Compressed PDB:

1	2	3	4	5	6	7	8

1	3	5	7

Min Compression: MOD

- To compress to PDB size N:
  - Compute rank, mod by N
- Example: N = 4

Original PDB:

1	2	3	4	5	6	7

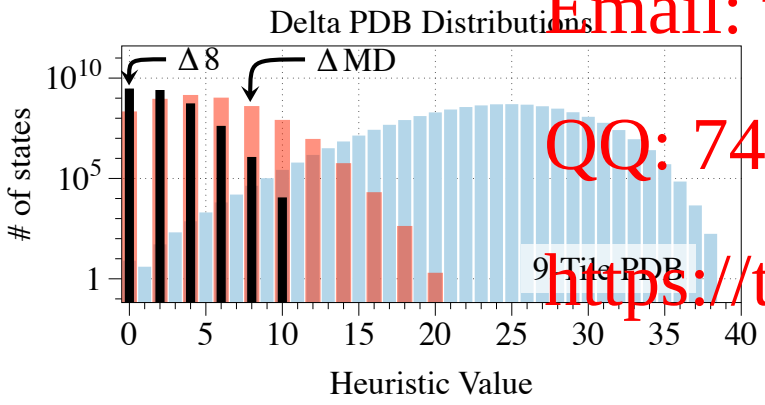
Compressed PDB:

1	2	3	4	5	6	7	8

1	2	3	4

Min Compression

- Need entries compressed together to be as similar as possible (DIV better than MOD in example)
- We can manipulate values to be more similar
- Delta Heuristic
  - Subtract a smaller heuristic from larger heuristic
  - Then compress Larger heuristic
  - Sliding Tile Puzzle with Manhattan Distance
    - Compute PDB; Subtract MD; Compress PDB
    - Better than just compressing PDB



Partial Pattern Databases

- In regular search, low values are most important
- High values are not used very often in the search (especially at larger depths)
  - Only store the low values in the search
  - Initial implementation: Anderson, et al

Bloom Filters

- Bloom filters test set membership
  - Always return true if set membersh
  - May return true when membership
- Can be used for heuristics
  - Use a bloom filter for each depth, s
  - If we get the incorrect depth, result admissible
- In practice use hash table first, then bloom filter

Bloom Filters

- Bloom filters can implement PPDBs
  - Don't need to store all levels of the PDB
  - Only store shallow levels
  - Get most of the gain in practice

Dual Lookups

- In pure permutations, we can ask what permutation takes us from the current state to the goal
  - Know this even if we don't know the set of actions that achieve it
- If we apply this permutation to the goal, we get the dual state
  - Guaranteed to be the same distance from the goal
  - Can perform another lookup from this state

Dual Lookups

- In practice dual representation is easy to compute
  - Location-based versus tile-based representation
- Can represent this state as:
  - [1 3 0 2]
    - Each entry represents the tile in the location
    - Location 1 has the 3
  - [2 0 3 1]
    - Each entry represents the location of the tile (0 to 3)
    - The 0 is in location 2

1	3
0	2

Dual Lookups

- Dual lookups can be inconsistent
  - The neighbors of the dual of a state are not the duals of the neighbor
- Only works on pure permutations
  - Available actions cannot depend on the location of the tiles
  - Doesn't work for sliding-tile puzzle
    - Depends on the blank

Symmetry

- In some states spaces other types of symmetry exists
  - Alternate states that are guaranteed to be at the same solution depth
    - eg flip sliding-tile puzzle around a diagonal
  - Performing symmetric lookup improves the average heuristic value
    - See original PDB paper for examples

Single Agent Search

Lecture 8
IDA* Performance

State Space Analysis

- What is "*b*"
  - Used the value, but haven't analyzed where it comes from
  - Analyze problems to compute *b*
  - Predict performance of IDA* (node exp)

Easy Example

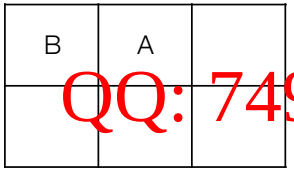
- 2x2 sliding tile puzzle (3-puzzle)
- Branching factor?
  - 2 if we include parents in analysis
  - 1 if we don't

Tougher Example

- 3x2 5-puzzle
  - 2/6 states are edge states
    - branching factor is 3 (2 w/o parent)
  - 4/6 states are corner states
    - branching factor is 2 (1 w/o parent)
  - b = 2/6*3 + 4/6*2 = 2.3333
    - Assumes we are equally likely to be any state

Time spent in each state

- Label states with types A/B
- Without pruning parents
  - A → 2B + 1A
  - B → 1A + 1B
- Use this to determine branching factor

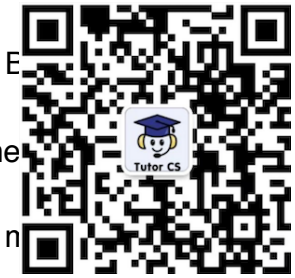


Asymptotic branching factor

- Consider a tree from any particular start state
  - Compute the number of nodes at each level
  - *b* is the ratio of nodes at level *n* to *n*+1
    - (As n gets large)
- (Demonstrate example on 5-puzzle)

Asymptotic Branching Factor

- At any particular level of the tree how many states of type A and of type B are there?
  - Eventually the fraction of A and B stabilize
  - The number of A nodes at the next level is the same as the previous level
  - The actual count will differ by a n



Asymptotic branching factor

- So, we know:
  - b $f_a = f_a + f_b$
  - b $f_b = 2f_a + f_b$
  - $f_a + f_b = 1$
- Solve for *b* and get b = 1+√2

Removing reverse operators

- Now we need 4 variables
  - A - center (from a center)
  - B - center (from a corner)
  - C - corner (from a center)
  - D - corner (from a corner)

Removing reverse operators

- A → 2C
- B → A + C
- C → D
- D → B

- b $f_a = f_b$
- b $f_b = f_d$
- b $f_c = 2f_a + f_b$
- b $f_d = f_c$
- $f_a + f_b + f_c + f_d = 1$
- Solve for *b* = 1.35

Rubik's Cube

- Naive approach:
  - 18 moves (90°, -90°, 180° turns
- Better approach:
  - 15 moves (disallow previous face
- Even better approach:
  - Restrict successive turns of face, and then first face

Rubik's Cube - Best approach

- Label 3 faces "first faces"
  - Faces should all border each other
- Other faces are "second faces"
- Disallow turning a first face after turning its associated second face
  - $b = 13.3487$

Predicting performance of IDA*

- Want to predict the performance of IDA* in general
- What is the asymptotic effect of having a heuristic?
  - Pearl, 84 predicted:
    - Search size b^d
    - With better heuristic a^d with $a < b$
- Analysis isn't necessarily incorrect; depends on assumptions

Predicting IDA* performance

- What do we need to determine performance?
  - Assume consistent heuristic
    - Analysis has been performed for inconsistent heuristics, but more complicated
  - Assume cost threshold of *c* on last iteration
    - All nodes with f(n) < c must be expanded
    - All nodes with f(n) ≤ c may be expanded
  - Information about the heuristic

Characterizing a heuristic - initial assessment

- Suppose there are N states in the world
  - How many states have h(a) = a f
    - Define this as d(a)
  - D(h) is: (∑d(a)) / N for a = 1…h
    - The percentage of nodes with less than or equal to h
- **Heuristic distribution**

Characterizing heuristic & search space

- Heuristic distribution may not represent how we exactly encounter states in our search
- This is controlled by the **equilibrium distribution**
  - What is the actual distribution of states we'll see in practice
    - A bit strange, because we search from a given start state
    - Cleaned up in later analysis

Equilibrium distribution

- Consider 5-puzzle, blank is:
  - 35.321% of time in side position
  - 64.679% of time in corner position
- Consider the heuristic distribution according to where the blank is

Equilibrium Distribution

- P(h)
  - The probability of a node having a heuristic ≤ h
  - P(h(n) < h | side)·P(side) + P(h(n) < h | corner)·P(corner)
- Example table with manhattan distance (next slide)
  - P(h) is different from D(h)

Table 2
Heuristic distributions for Manhattan distance on the Five Puzzle

h	States	Sum	$D(h)$	Corner	Side	Csum	Ssum	$P(h)$
0	1	1	0.002778	1	0	1		
1	2	3	0.008333	1	1	2		
2	3	6	0.016667	1	2	3		
3	6	12	0.033333	5	1	8		
4	30	42	0.116667	25	5	33		
5	58	100	0.277778	38	20	71		
6	61	161	0.447222	38	23	109		
7	58	219	0.608333	41	17	150		
8	60	279	0.775000	44	16	194		
9	48	327	0.908333	31	17	225	102	0.906594
10	24	351	0.975000	11	13	236	115	0.974503
11	8	359	0.997222	4	4	240	119	0.997057
12	1	360	1.000000	0	1	240	120	1.000000

What does a tree look like?

- Given a node & consistent heuristic:
  - h-value is either 1 greater than or 1 less than the original heuristic value
- Divide children into buckets
  - Sample tree
    - h = 0…3, c = 5

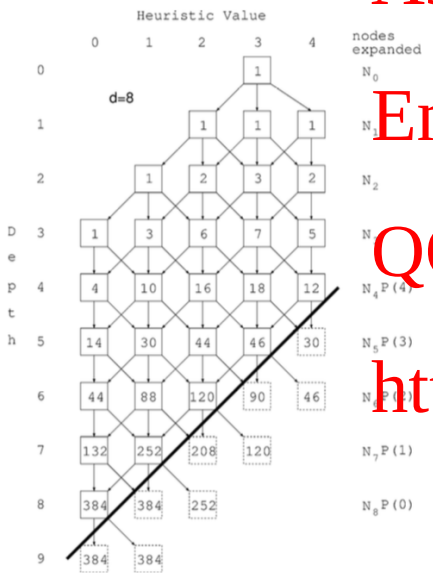


Fig. 4. Sample tree for analysis of IDA*.

What is the total work?

- Expected number of nodes expanded

$$E(N, c, P) = \sum_{i=0}^{c} N_i P(c - i)$$

  - N_i is simply b^i where b is the asymptotic branching factor
  - *Using this, what is the heuristic branching factor?*

Analysis

- Heuristic branching factor:

$$\frac{E(N,c,P)}{E(N,c-1,P)} = \frac{\sum_{i=0}^{c} N_i P(c}{\sum_{i=0}^{c-1} N_i P(c -}$$

$$= \frac{b^0 P(c) + b^1 P(c-1) + b^2 P(c-2) +}{b^0 P(c-1) + b^1 P(c-2) + b^2 P(c-3) +}$$

$$\approx b$$

Analysis

- In an exponential domain the effect of a heuristic is to keep the branching factor the same
- If we increase the heuristic, we decrease the level at which we get cutoffs
  - In exponential spaces, improving the heuristic just decreases the effective level of search

Discussion

- Perfectly predicts nodes expanded for 8-puzzle if:
  - Average over all starting states
  - Search to fixed depth *c* (past the goal)
- Not necessarily a good predictor for an actual problem

Single Agent Search

Lecture 9
Parallel & External Memory Search

Parallel Algorit

UNIVERSITY of DENVER
DANIEL FELIX RITCHIE SCHOOL OF ENGINEERING & COMPUTER SCIENCE

Parallel Algorithms

- For many years it looked like our problems would be solved by increases in processor speed
- Moore's law predicts the # of transistors, not the speed of the chip
  - The number of transistors continues to grow
  - Processor speed hasn't
- As of 2016 Intel has said it will no longer ensure Moore's Law holds
  - Need to maximize the use of what we have

UNIVERSITY of DENVER
DANIEL FELIX RITCHIE SCHOOL OF ENGINEERING & COMPUTER SCIENCE

Parallel Tree Search

- IDA* Approach #1 (from book)
  - One processor for each iteration
  - Total speed up?
    - At most a factor of 2
    - Work dominated by processor solving largest iteration

UNIVERSITY of DENVER
DANIEL FELIX RITCHIE SCHOOL OF ENGINEERING & COMPUTER SCIENCE

Parallel Tree Search

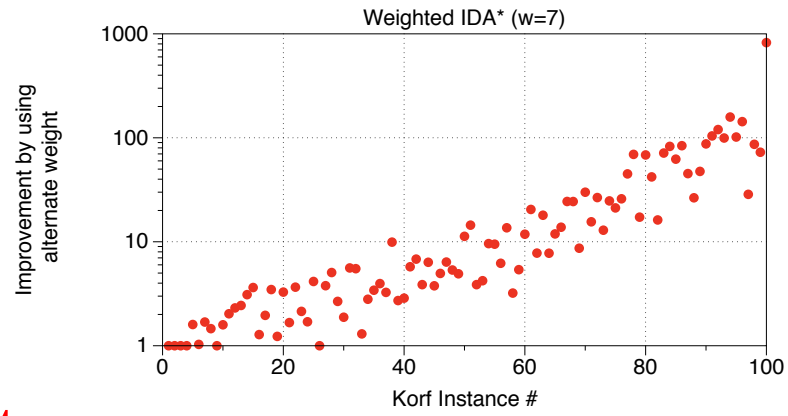
- IDA* Approach #2 (Reinefeld and Schnecke, 1994)
  - Search to depth *d*
  - Search trees at depth *d* in parallel

UNIVERSITY of DENVER
DANIEL FELIX RITCHIE SCHOOL OF ENGINEERING & COMPUTER SCIENCE

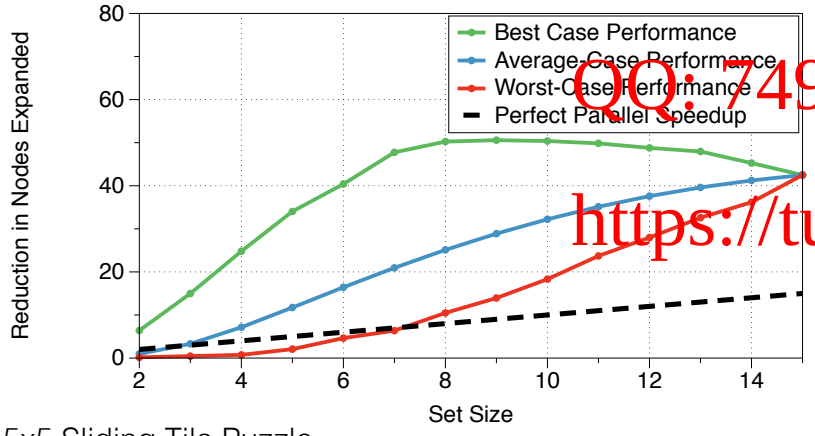
Parallel Tree Search

- IDA* Approach #3 (Valenzano)
  - Search to depth *d*
  - Search trees at depth *d* in paralle
  - If there are many possible solutic likely to find solution quickly
    - Works best on suboptimal sea

Dovetailing



Dovetailing



5x5 Sliding Tile Puzzle

External Memory Algorithms

External Memory

- Disk and/or SSD
- Characteristics
  - Slow random access
  - Fast streaming access
  - Much larger than main memory
  - Much cheaper than main memory
  - (SSD) Lower lifetime

Two Types of Search

- State spaces that fit on disk
  - For example: large PDB
  - Results are stored on disk afterwards
  - Entire state space must fit on disk
- Search to solve a single problem instance
  - State space can be far larger than disk
  - Only keep states in search on disk

Implicit vs Explicit

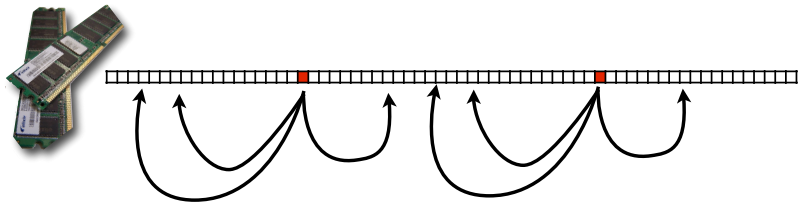
- Implicit representation
  - State determined by offset on disk
  - Data about state stored at that offset
  - Cheaper per state; need to store many states to amortize cost
- Explicit Representation
  - Full state representation stored on disk
  - More expensive per state
  - Better when storing small fraction of state space

BFS On Disk (Implicit Representaion)

What do we store?

- Two bits
  - Unseen, Next, Current, Closed
- Depth
  - Can be stored modulo 4 (2-bits)
    - Distinguishes unseen and 3 dista
    - Recover actual distances from n

In-Memory approach



Buckets

- Divide state space into pieces that fit into memory
  - Each chunk is called a bucket
- Load an entire bucket into RAM
  - Process the bucket
- Repeat until full space covered / BFS finished

Processing Buckets (TBBFS)

- Load bucket into RAM & expand states
  - Successors in same bucket updated in RAM
  - Successors in other buckets write to disk (explicitly)
- When loading new bucket, process any (explicit) buckets waiting to be processed on disk
- After all buckets expanded, may still have to process (explicit) data on disk
- If disk fills up process (all) explicit data on disk

Processing Buckets (WMBFS)

- For next bucket, initialize bit-array in RAM representing states at the next depth
- Scan space (implicit on disk) and exp
  - If successors fall into bit-array in R
  - Otherwise discard
- Write changed bits back to disk (& ch
- Repeat for every bucket

WMBFS vs TBBFS

- WMBFS
  - Uses less disk; writes far less to disk
  - Expands more states
  - Most effective when very few buckets
- TBBFS
  - Benefits from locality of state space
  - Expands fewer states
  - More effective when many buckets

Chinese Checkers

	WMBFS	TBBFS
Total Time	2,632,266 seconds 30.5 days	2,410,963 seconds 27.9 days
Nodes Expanded	1,837,185,821,822	1,072,763,999,648
Total Writes	4.85 TB	88+ TB
Total Reads	10.80 TB	88+ TB

Rubik's Cube

	WMBFS	TBBFS
Total Time	586,433 seconds 6.8 days	2,099,746 seconds 24.3 days
Nodes Expanded	1,961,990,553,104	980,995,276,800
Total Writes	2.68 TB	60.5+ TB
Total Reads	6.85 TB	60.5+ TB

BFS On Dis
(Explicit Represe

UNIVERSITY of DENVER
DANIEL FELIX RITCHIE SCHOOL OF ENGINEERING & COMPUTER SCIENCE

Depth-Layered BFS

- General BFS approach
  - Expand all states at the current depth
  - Check for duplicates
    - Current depth, Previous depth
  - Write successors to next depth
- Easy if there are no duplicates
- Divide states until buckets and do DD on buckets

UNIVERSITY of DENVER
DANIEL FELIX RITCHIE SCHOOL OF ENGINEERING & COMPUTER SCIENCE

Delayed Duplicate Detection

- Don't check for duplicates until all states are expanded in a layer
  - Random access is expensive
  - Allows us to amortize disk access

UNIVERSITY of DENVER
DANIEL FELIX RITCHIE SCHOOL OF ENGINEERING & COMPUTER SCIENCE

Frontier Search

- With every state maintain information about operators that have been reached
  - Avoid following operators that have already been explored
  - Prevents search from reaching closed states
  - No need for DD against previous layers
- Most effective with low branching factors and unit-cost operators

Sorting-Based Duplicate Detection

- Sort every bucket
  - Duplicates move next to each other
  - Can handle large buckets

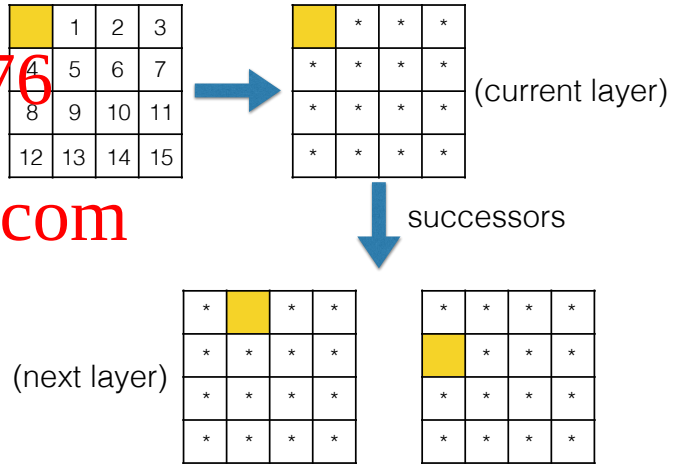
Hash-Based Duplicate Detection

- After layer finished:
  - Load each bucket into hash table in RAM
    - Removes duplicates
  - Write results back to disk

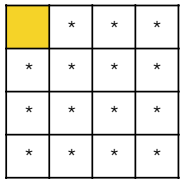
Structured Duplicate Detection

- Use structure of state space to detect duplicates
- Use abstraction (eg PDB pattern) to determine buckets for states
  - The successors of a state will go into predictable buckets
  - Load all successor buckets into RAM when processing & immediately remove duplicates
- Uses less memory (fewer duplicates in RAM)

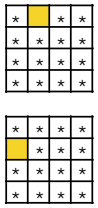
SDD Example



SDD Parallelism



Can these states be processed in parallel?



Yes - duplicate detection scopes do not overlap.

Single Agent Search

Lecture 10
Bidirectional Search

UNIVERSITY of DENVER
DANIEL FELIX RITCHIE SCHOOL OF ENGINEERING & COMPUTER SCIENCE

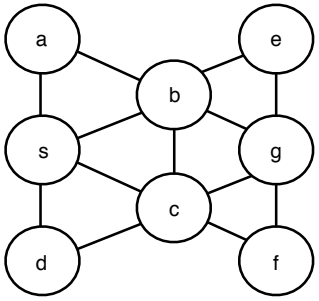
SFBDS

UNIVERSITY of DENVER
DANIEL FELIX RITCHIE SCHOOL OF ENGINEERING & COMPUTER SCIENCE

Single Frontier Search

- View each state as a task:
  - Get from *s* to *g*
  - We can choose:
    - Expand *s* or expand *g*
  - Can use with A*, IDA*, etc.
  - May be inefficient in A*
- Proposed by Pohl; not explored until later

SFBDS



Tasks:

(s,)

(a,)

(b,)

(c,)

(d,)

(a,)

(a,)

(a,)

(a, f)

…

MM

SFBDS

- Exponential model:
  - Branching factor b everywhere
  - Solution will be found at depth d
  - No overhead, but no gain
- How we can gain:
  - Different branching factor (expand lowest)
  - Different heuristic values (expand higher)

General Pseudo-Code

- Choose a frontier to expand
- Choose best node to expand
  - Place children on open in that direction
  - Check children against open in opposite direction
  - Store best solution found
- Repeat until we can terminate

How MM works

$$pr(n)=\max\begin{cases} g(n)+h(n) & \text{(case 1)} \\ 2\times g(n) \end{cases}$$

$$pr(n)=g(n)+\max\{g(n),h(n)\}$$

- Expand a node (on either sides) with minimal pr
- When a node *n* is generated, check if *n* is in OPE side
- Remember the cheapest path found (cost = U).
- MM stops when U ≤ Lower Bound

Bounding solution cost

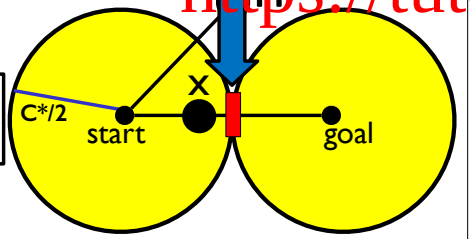
- Measures:
  - C = minimum priority node in each direction
  - $fmin_F$ = minimum f-cost in forward direction
  - $fmin_B$ = minimum f-cost in backward direction
  - $gmin_F+gmin_B+epsilon$ = estimate solution cost through frontier
- Terminate when U ≤ (max of all of these)

Main Lemma
MM never expands nodes with g(n) > C*/2

Proof:

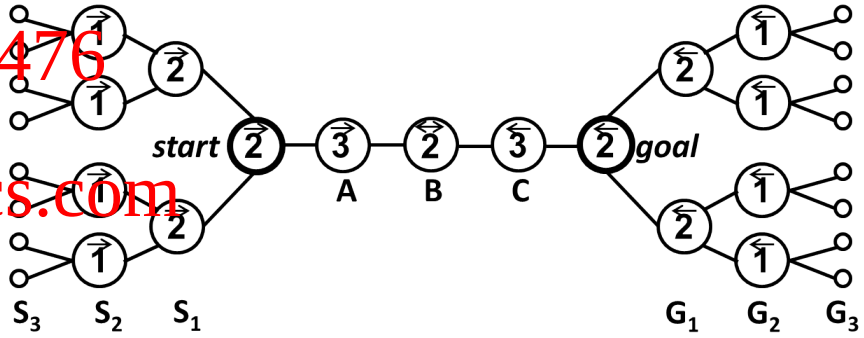
- Let **g(n)>C*/2**
  - case 1: **If g(n)<h(n) then pr(n)=g(n)+h(n) > C***
  - case 2: **If g(n)>h(n) then pr(n)= 2g(n) > C***
- **OPEN** always includes a node **x** on the optimal path with **pr(x)≤C***

Result: must meet in the middle



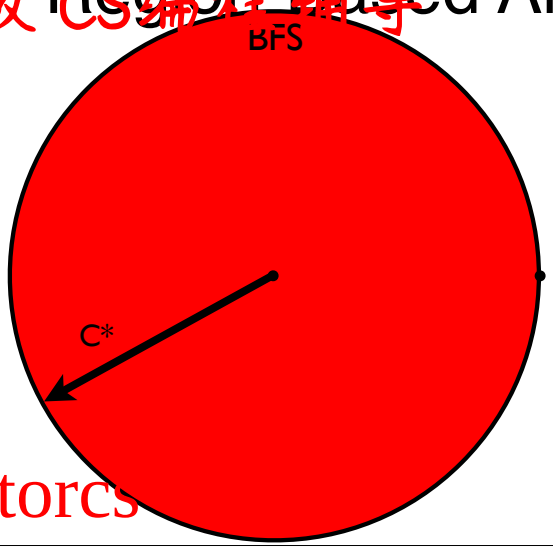
Upper Bound for MM:
$2b^{C^*/2}$

MM Example

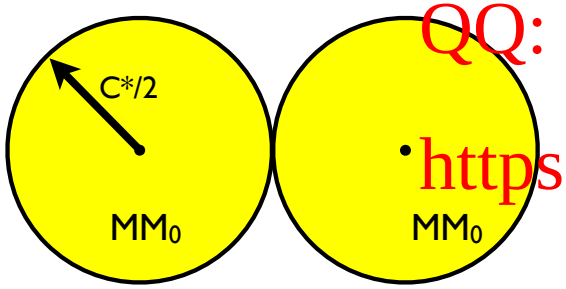


Brute-force sea
(BFS vs MM0

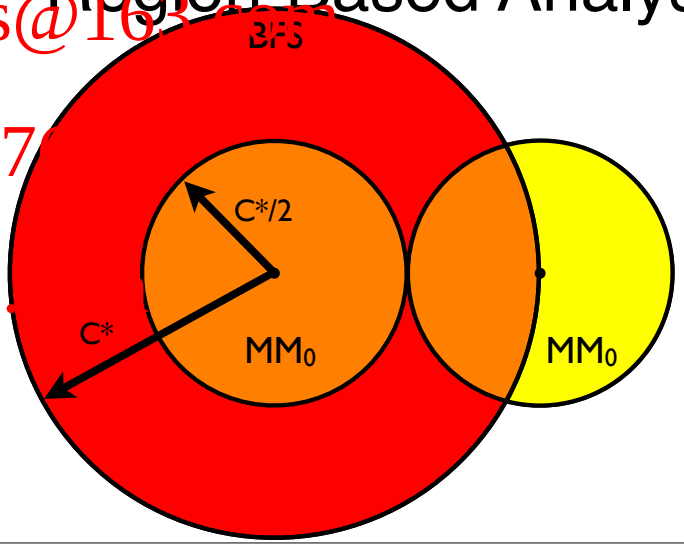
Region-Based Analysis

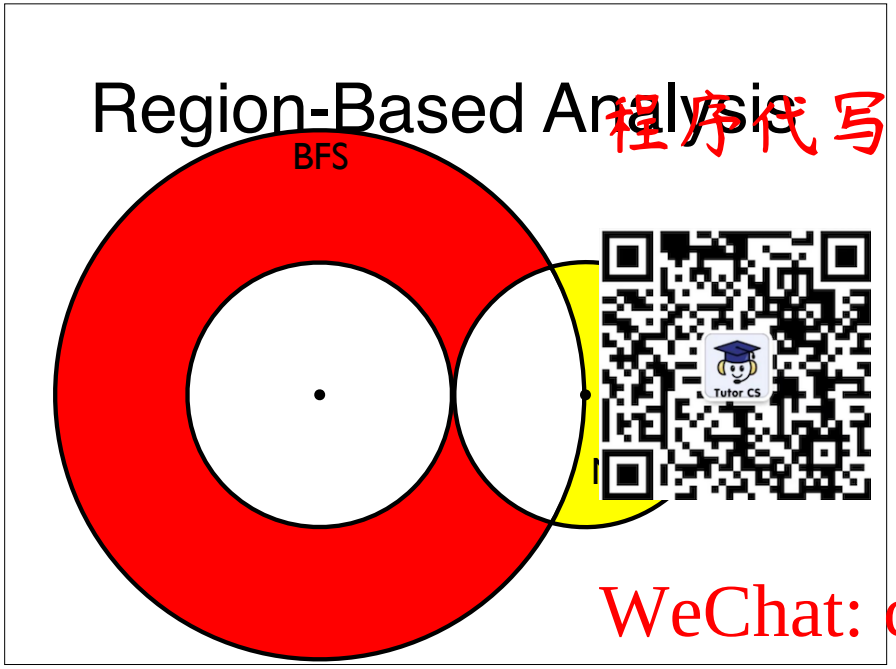


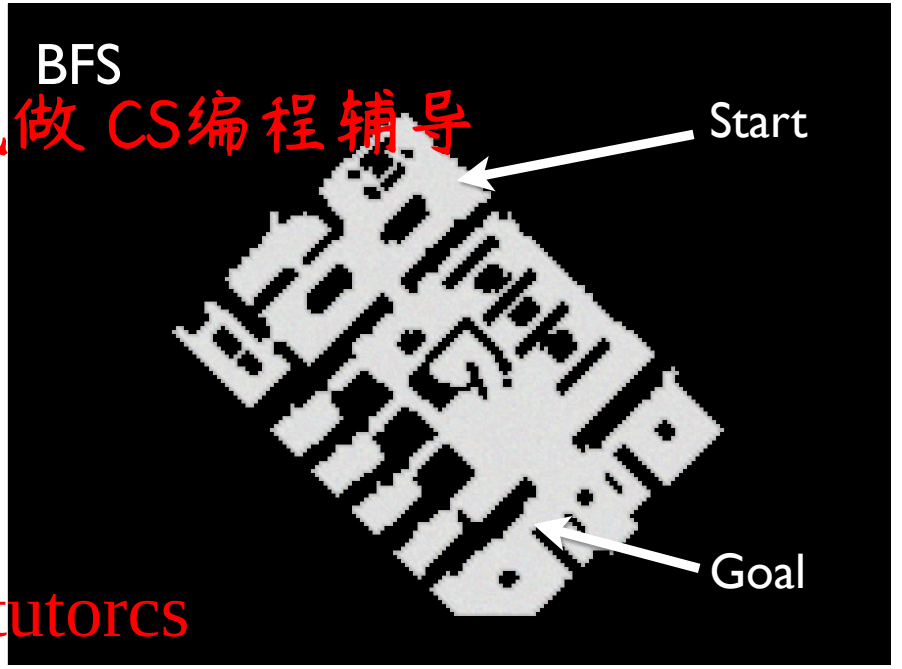
Region-Based Analysis

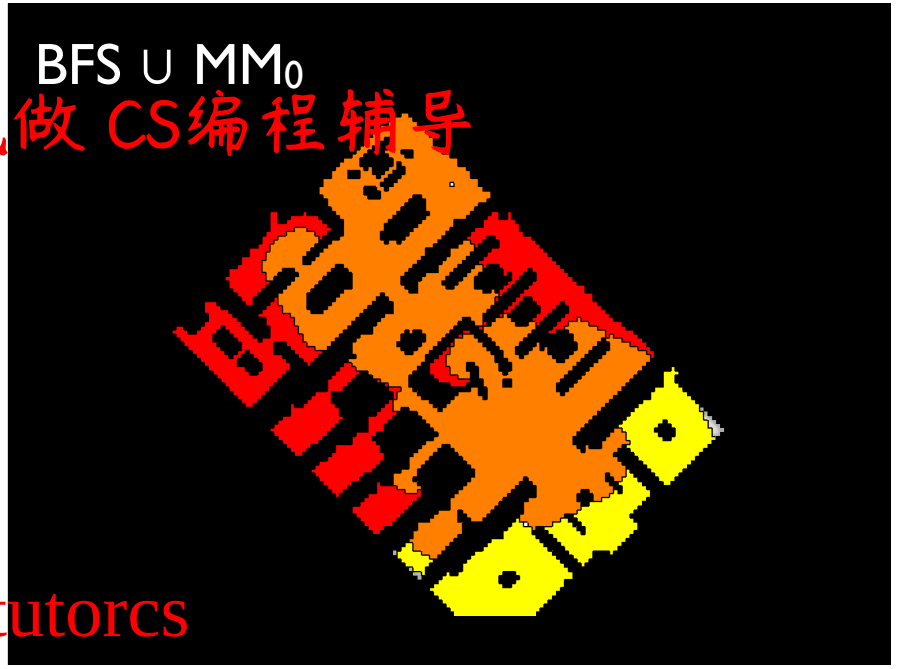


Region-Based Analysis

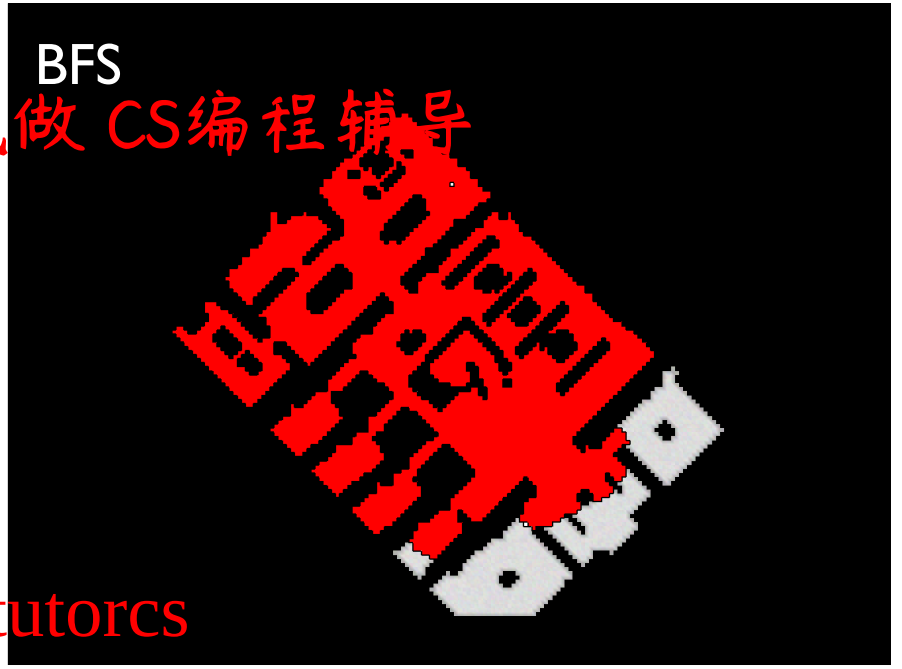


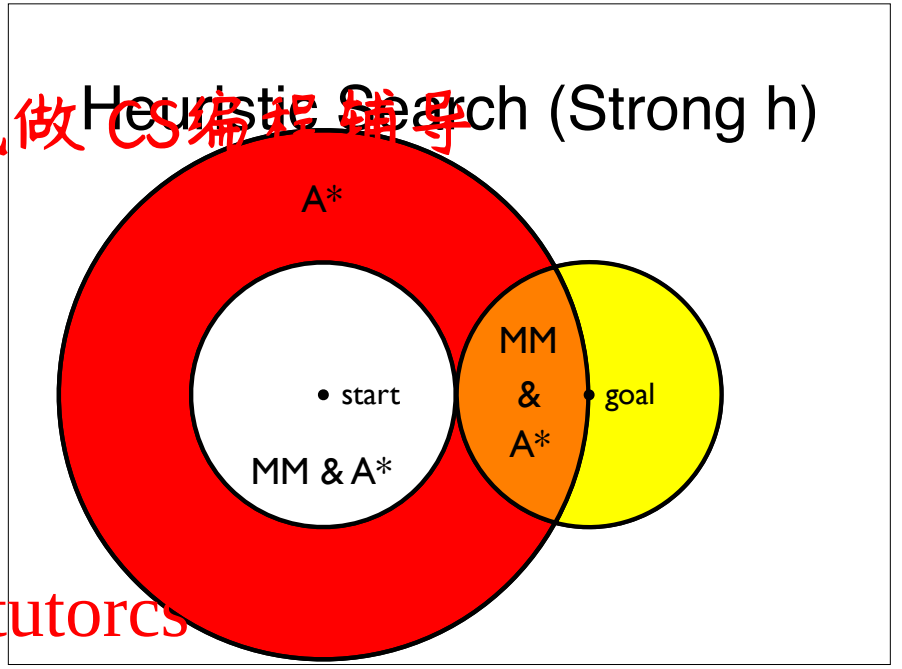


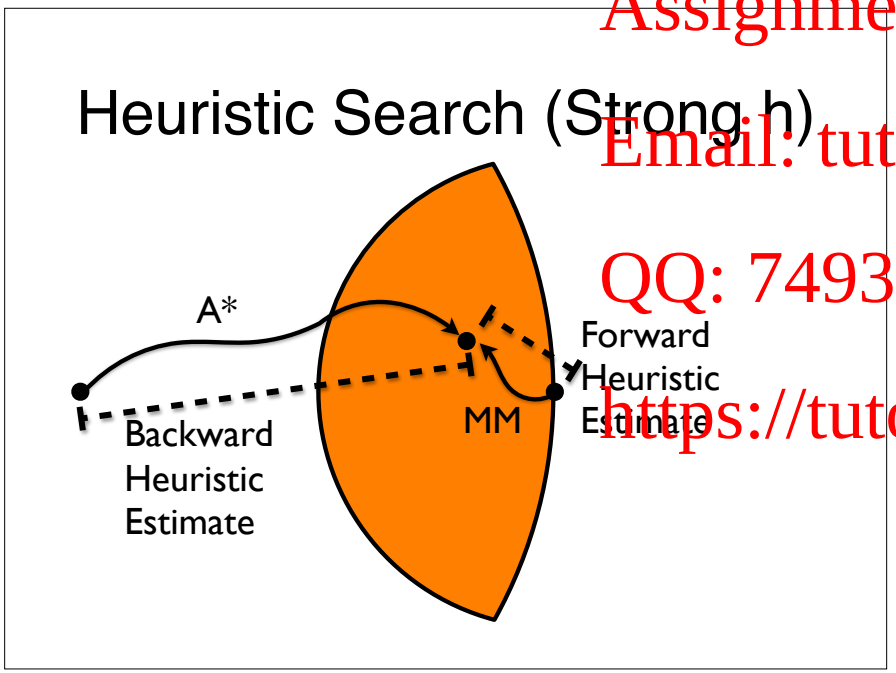




Heuristic sear
MM vs A*

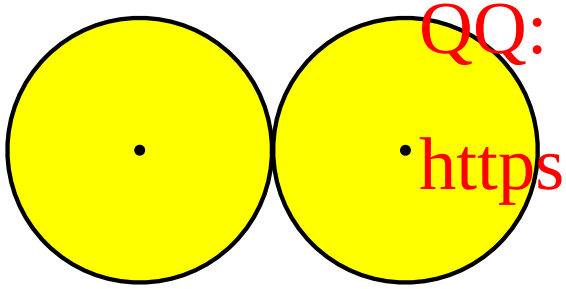




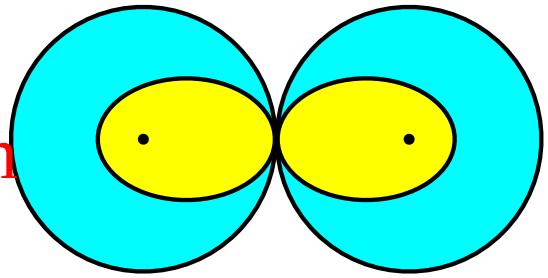


Bidirectional Search
(MM_0 vs MM)

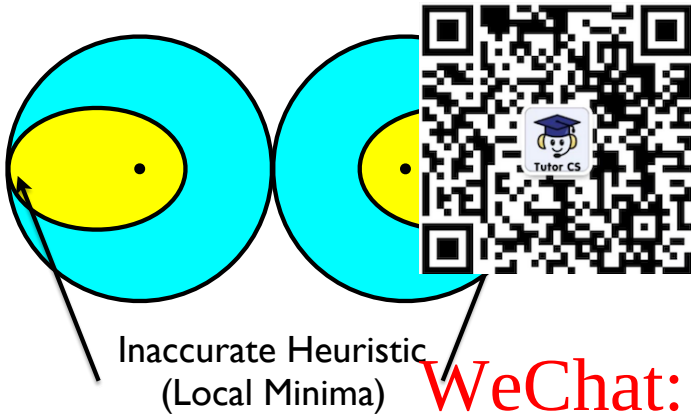
Comparing MM_0 and MM



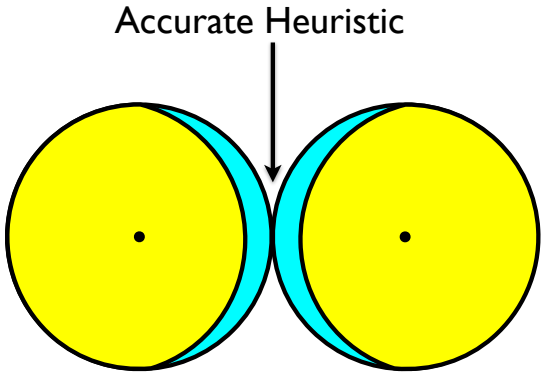
Ideal MM Expansion Order



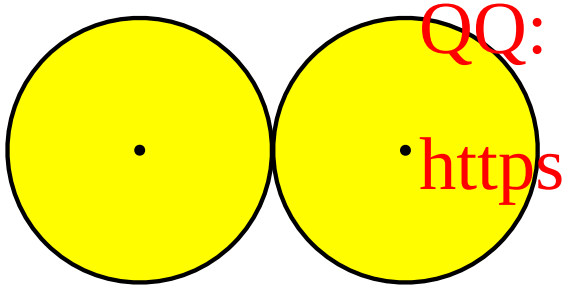
MM Expansion Order (Weak h)



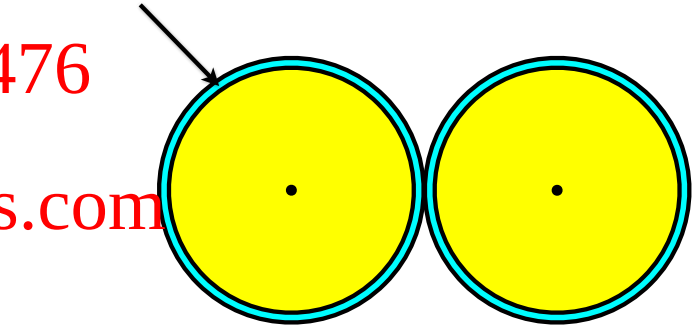
MM Expansion Order (Weak h)



MM Expansion Order (Weak h)



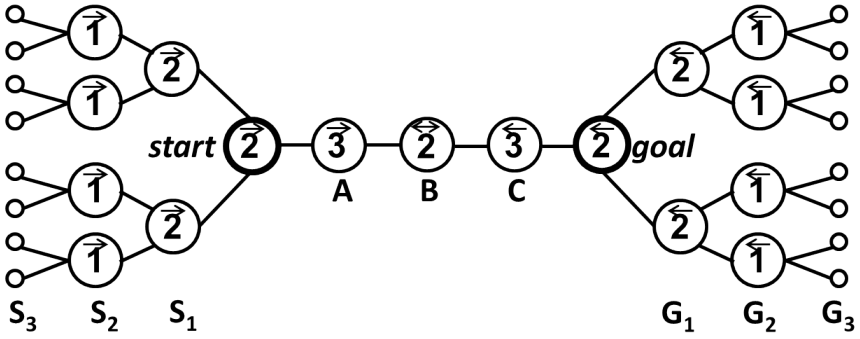
MM_0 ϵ-Termination Rule



New

- MM ϵ-expansion rule
  - Expand states based on:
    - max(f, 2g+ϵ)
  - Better performance on some proble

MM ϵ-epsilon rule



Our Conjectures

1. With a ***strong*** heuristic: A*
   < MM, MM_0
2. With a ***weak*** heuristic:
   MM_0 < A*, MM
3. With a ***moderately accurate*** heuristic
   MM < A*, MM_0

Experiments

- Domains
  - 10-Pancake Puzzle (small)
  - A*, MM, MM_0
  - 3x3x3 Rubik's Cube (large)
  - IDA*, MM, MM_0 (disk)
- Vary heuristic accuracy / problem difficulty
- Only measure nodes expanded

10-Pancake Puzzle, C*=10

	Better Heuristic Accuracy →			
Algorithm	GAP-3	GAP-2	GA	
A*	97,644	27,162	4,2	
MM	7,507	6,723	2,4	
MM_0	5,551	5,551	5,551	5,551

Rubik's Cube

7-9-7 Korf Heuristic (82MB)

	Problem Difficulty →									
	1	2	3	4	5	6	7	8	9	10
IDA*	0.26	1.51	8.13	6.56	29.7	15.4	41.6	45.9	58.4	70.3
MM	0.17	1.00	1.14	0.96	4.71	4.84	5.89	4.85	3.01	4.25
MM_0	0.22	1.55	1.55	1.55	2.88	2.88	2.88	2.88	2.88	2.88

Rubik's Cube

8-8-8 Heuristic (5 GB)

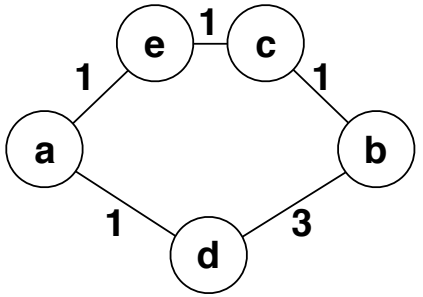
	Problem Difficulty →									
	1	2	3	4	5	6	7	8	9	10
IDA*	0.02	0.11	0.68	0.47	2.49	1.10	3.16	3.77	5.13	4.82
MM	0.10	1.01	1.02	0.39	3.58	3.51	4.01	3.67	2.87	3.29
MM_0	0.22	1.55	1.55	1.55	2.88	2.88	2.88	2.88	2.88	2.88

External Memory MM

External Memory MM

- Most operations can be run in the same way
- But:
  - Need delayed duplicate detection
- New operation:
  - Solution detection
  - Need delayed solution detection

Failure of epsilon



Single Agent Search

Lecture 11
Inconsistent Heuristics

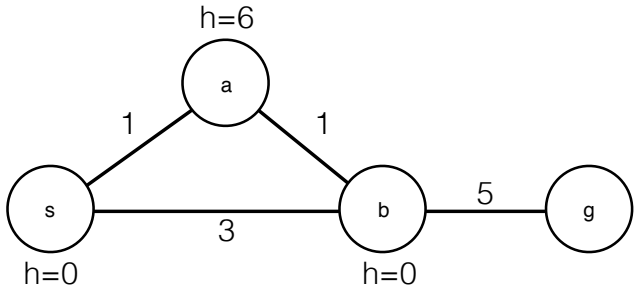
Inconsistent Heuristics

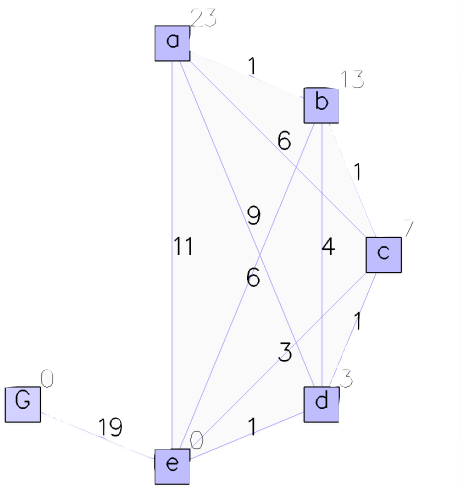
UNIVERSITY of DENVER
DANIEL FELIX RITCHIE SCHOOL OF ENGINEERING & COMPUTER SCIENCE

When A* doesn't work well

- What happens when we have inconsistent heuristics?
  - First path to goal is still optimal if
  - First path to other nodes is not n optimal
  - Can re-open closed nodes

Inconsistency Example





node	f	g	h
A	23	0	23
E	11	11	0
B	22	9	13
E	10	10	0
C	13	6	7
E	9	9	0
D	10	7	3
E	8	8	0
B	14	1	13
E	7	7	0
D	8	5	3
E	6	6	0
C	9	2	7
E	5	5	0
D	6	3	3
E	4	4	0
G	23	23	0

Definition

- Let N be the number of nodes expanded by A*
  - N is the number of *nodes* not the number of *expansions*
  - N is $O(b^{c/e})$
- A* can re-expand nodes
  - Express in terms of N

Analysis

- (1+0.5*2^N) nodes expanded!
  - 6 nodes
  - "E" expanded 8 times
  - "D" expanded 4 times
  - "C" expanded 2 times
  - "A", "B", "G" expanded once

Martelli

- Suggested Algorithm "B"
  - Maintain global "F" value
  - Maximum f-value opened so far
  - If there are nodes on OPEN with f < F
    - Open in order of increasing g-cost
    - Dijkstra's algorithm

Algorithm A*

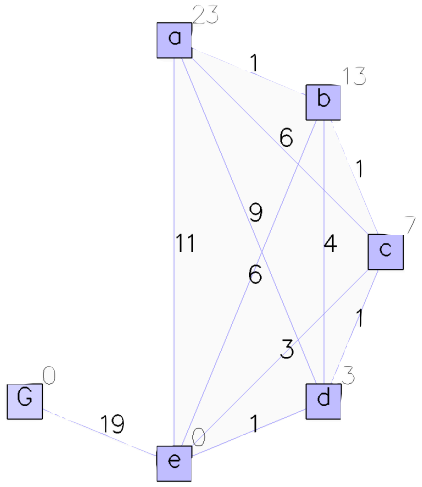
1. Put start on OPEN, g(start) = 0 f(start) ← h(start)
2. if OPEN is empty return *failure*
3. Remove lowest *f*-cost node *n* from OPEN
4. If *n* is goal, return path from start → goal
5. Expand *n* generating successors n_1 . . n_i
6. Add n_i to OPEN or update costs on OPEN/CLOSED

Algorithm B

- Two additional steps:

1' F ← 0

3' If there is a node in open with f < F, choose among them node with smallest g-cost. Otherwise set F ← f(*n*)

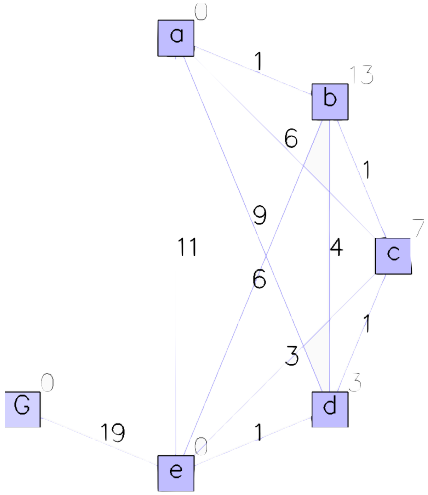


node	f	g	h
A	23	0	23
B	14	1	13
C	9	2	7
D	6	3	3
E	4	4	0

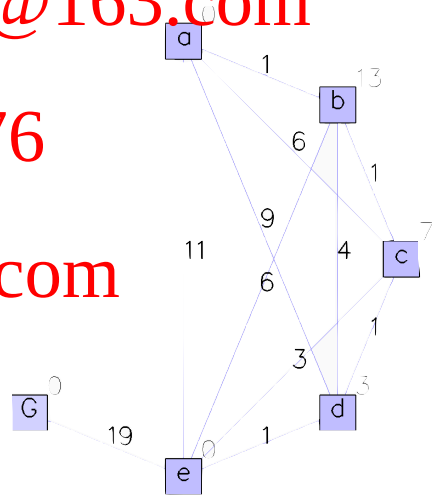
UNIVERSITY of DENVER
DANIEL FELIX RITCHIE SCHOOL OF ENGINEERING & COMPUTER SCIENCE

Martelli

- Algorithm works well, does it fix everything?
  - No -- worst case still $O(n^2)$
  - Just lower cost of start heuristic to 0



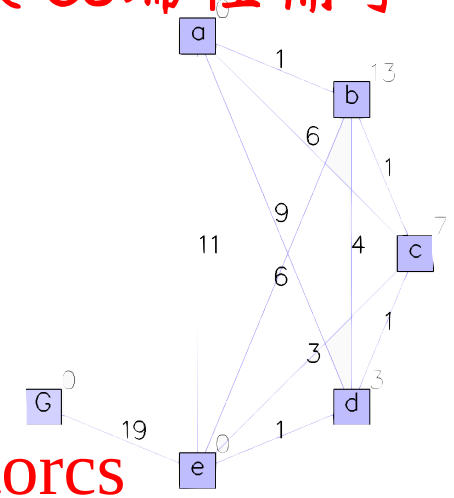
node	f	g	h	F
A	0	0	0	0
E	11	11	0	11
D	12	9	3	12
E	10	10	0	12
C	13	6	7	13
D	10	7	3	13
E	8	8	0	13
B	14	1	13	14
C	9	2	7	14
D	6	3	3	14
E	4	4	0	14
G	23	23	0	23



node	f	g	h	F
A	0	0	0	0
E	11	11	0	11
D	12	9	3	12
E	10	10	0	12
C	13	6	7	13
D	10	7	3	13
E	8	8	0	13
B	14	1	13	14
C	9	2	7	14
D	6	3	3	14
E	4	4	0	14
G	23	23	0	23

Review

- Inconsistent heuristics can lead to an exponential number of re-expansions with A*
- Algorithm B interleaves Dijkstra sea when inconsistencies are found
  - Reduces worst-case to $O(N^2)$



node	f	g	h	F
A	0	0	0	0
E	11	11	0	11
D	12	9	3	12
E	10	10	0	12
C	13	6	7	13
D	10	7	3	13
E	8	8	0	13
B	14	1	13	14
C	9	2	7	14
D	6	3	3	14
E	4	4	0	14
G	23	23	0	23

Worst case proof

- Worst case proof
  - $O(n^2)$ node expansions
- Each node can be opened with a maximum f-cost at most once
  - Between these openings, we are running Dijkstra's algorithm
  - We will do at most n openings based on g-cost
- $O(n^2)$

Solution - Mero 84 - B'

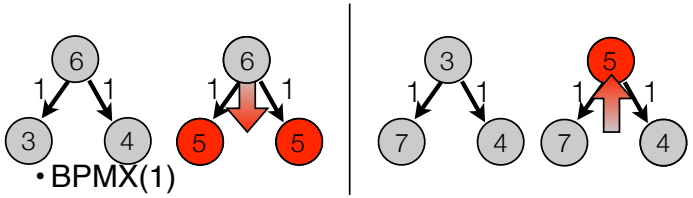
- Pathmax
  - When generating a node:
  - h(n) = h(p) - c(n, p)
  - h(p) = min(h(c) + c(c, p)) over all children *c*

Algorithm B'

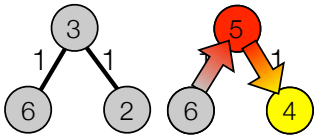
- Enhance B/A* with these steps:
  - 3(a) For each child c_i of n, if $h(c_i)$ then set $h(c_i) \leftarrow h(n) - c(n, c_i)$
  - 3(b) Given child c_i with min. cost set $h(n) \leftarrow \max(h(n), h(c_i) + c(c_i,$
- These are the pathmax rules
  - BPMX for undirected graphs app simultaneously / repeatedly

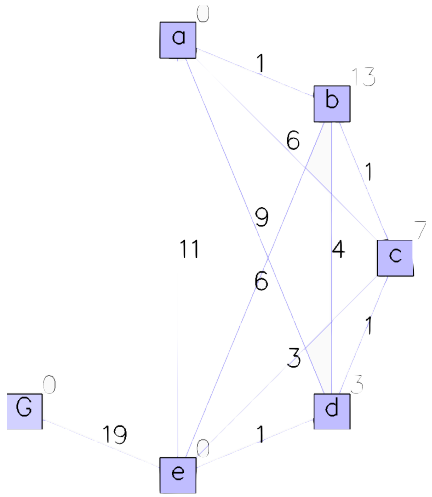
Pathmax rules

- Pathmax:



- BPMX(1)

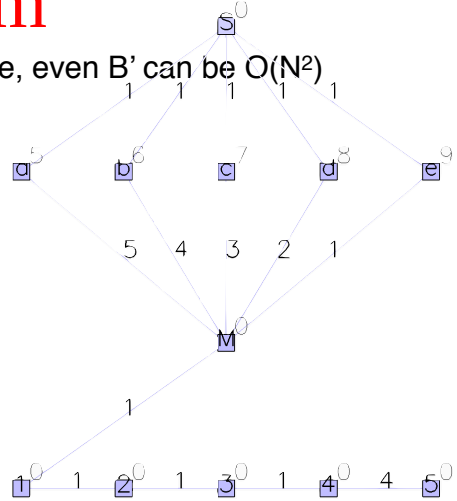




node	f	g	h	F
A	11	0	11	11
E	30	11	19	12
H	29	9	20	13
C	27	6	21	14
B	23	1	22	23
F	23	2	21	23
D	23	3	20	23
E	23	4	19	23
G	23	23	0	23

So...how bad?

- In the worst case, even B' can be $O(N^2)$



General Inconsistency Bounds

- Suppose A* performs $\phi(N) > N$ expansions
  - Some node must be re-opened (times
    - Pigeon-hole principle
  - If Δ is the minimum change in h-
    - h-cost is at least $\Delta \cdot (\phi(N) - N)$/
    - Solution cost is also at least Δ·
      - We never open a node with f-cost > solution

General Inconsistency Bounds

- Problem that grows exponentially
  - $\Delta = 1$, $N = b^d$, max heuristic is h_{max}
  - $h_{max} \geq \Delta \cdot (\phi(N) - N)/N$
  - $h_{max} \geq (\phi(N) - b^d)/b^d$
  - $\phi(N) \leq h_{max} \cdot b^d + b^d = (h^* + 1) \cdot b^d = O(d \cdot N)$

General Inconsistency Bounds

- Problem that grows polynomially
  - $\Delta = 1$, $N = d^2$, max heuristic is h_{max} / d
  - $h_{max} = \Delta \cdot (\phi(N) - N)/N$
  - $h_{max} = (\phi(N) - d^2)/d^2$
  - $\phi(N) = h_{max} \cdot d^2 + d^2 = O(d^3) = O(N^{1.5})$

How do we get inconsistency in practice

- Special properties (duality)
- Compression
- Max of multiple heuristics
  - Too expensive to use all heuristics, so use random subset of heuristics

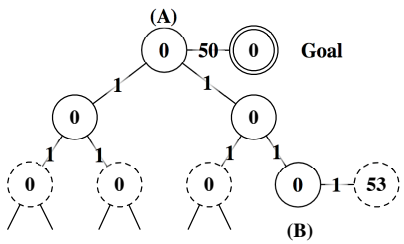
What is good/bad inconsistency

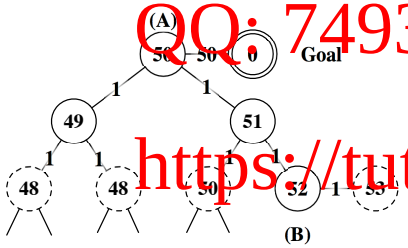
- "Good" inconsistency
  - There are always good heuristics
  - 1-step BPMX to 'fix' bad values
  - Improve the run-time distribution
- "Bad" inconsistency
  - Misleading values (worst path ha
- Note: with no cycles, inconsistency

BPMX in A*/IDA*

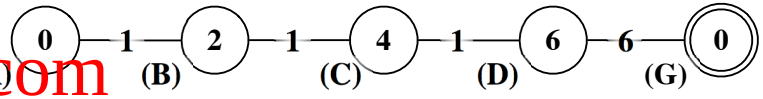
- BPMX is free in IDA*
- More expensive in A*
  - We don't naturally backtrack through closed list
  - Choice:
    - Backup as far as possible
      - $O(N^2)$ cost or unbounded savings
    - Backup only *k* steps O(kN) cost

BPMX Best Case





BPMX Worst Case



Single Agent Se

Lecture 12
Euclidean Embeddings

DO more work in this lecture to help see what good and bad placements are. (See 2016 final exam.)
Practice selecting points and testing which ones have high/low heuristic values

UNIVERSITY of DENVER
DANIEL FELIX RITCHIE SCHOOL OF ENGINEERING & COMPUTER SCIENCE

Heuristic Approaches

- Different heuristics in polynomial/exponential domains
  - Global Network Positioning (Ng & Zhang, 2001)
  - ALT (Goldberg & Harrelson, 2005)
  - True-Distance Heuristics (Sturtevant, et al, 2009)
  - 1-Dimensional Euclidean Heuristic (Rayner, et al, 2011)
  - Compressed DH (Goldenberg et al, 2011)
  - Subset Selection of Search Heuristics (Rayner et al, 2013)

Heuristic Values in PDBs

- $b^h = b^w/k$
- $\log(b^h) = \log(b^d)-\log(k)$
- $h \cdot \log(b) = d \cdot \log(b)-\log(k)$
- $h = d - \log(k)/\log(b)$
- $h = d-\log_b(k)$

Heuristic Values in Maps

- $h^d = r^d/k$ [d = dimension of map]
- $h = r/(\sqrt[d]{k})$
  - if d = 2, k = 4, h = r/2
- Building abstract graph for heuristics will not be effect
- Need to store exact distances

Distance Approximation

- Suppose undirected graph
- Suppose we know d(p_1, x) for all x
  - How can we use this to estimate th between any states a & b?
- Triangle inequality:
  - d(a, b) ≥ |d(a, p_1) - d(b, p_1)|
- Key question:
  - Where to place p_i

Subset Selection of Heuristics

Heuristic Selection Problem

- Choose *d* heuristics from a set:

$$H \subseteq C = \{h_1, \ldots, h_{|C|}\} \quad |H| = d$$

- Heuristic is max of all of these plus a default heuristic

$$h^H(i,j) = \max_{h_x \in H \cup D} h_x(i,j)$$

Optimization Formulation

- Minimize the loss when selecting heuristics:

$$\mathcal{L}(H) = \sum_{i=1}^{n} \sum_{j=1}^{n} W_{ij} |\delta(i,j) - h^H(i,j)|$$

- Equivalent to maximizing the heuristic value:

$$\mathcal{U}(H) \equiv \sum_{i,j} W_{ij}\, h^H(i,j)$$

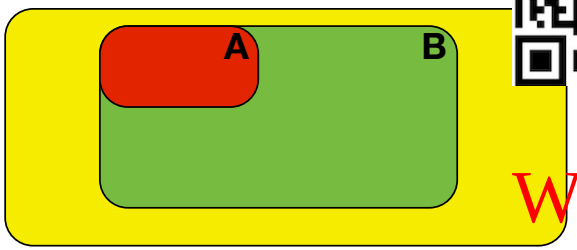
$\mathcal{U}(H)$ is monotonic

$\mathcal{U}(H)$ is submodular

Monotonicity

$f : 2^S \to \mathbb{R}$

$A \subseteq B \qquad f(A) \leq f($

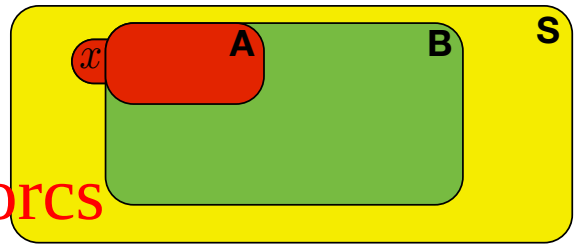


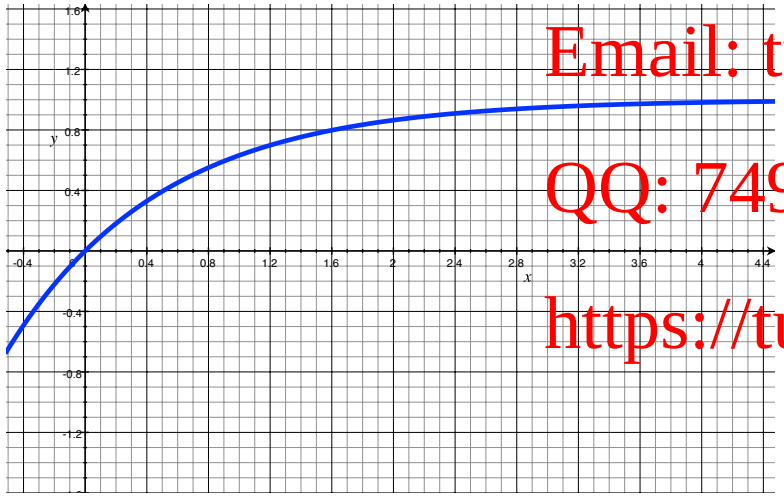
Submodularity

$f : 2^S \to \mathbb{R}$

$A \subseteq B \subseteq S \text{ and } x \in S, x \notin B$

$f(A \cup \{x\}) - f(A) \geq f(B \cup \{x\}) - f(B)$





UNIVERSITY OF DENVER
DANIEL FELIX RITCHIE SCHOOL OF ENGINEERING & COMPUTER SCIENCE

Approximation Algorithm

- Sample every heuristic and choose the one that maximizes the gain in the overall heuristic
- Sampling all heuristics is too expensive; can sample possible heuristic choices

Repeat *d* times:

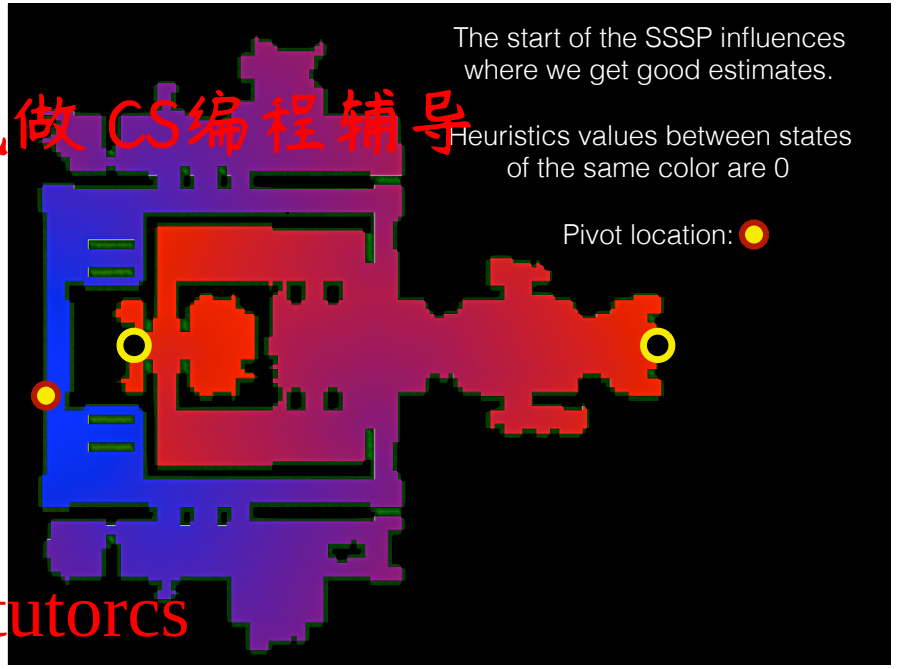
For all $h \in C \backslash H_{t-1}$

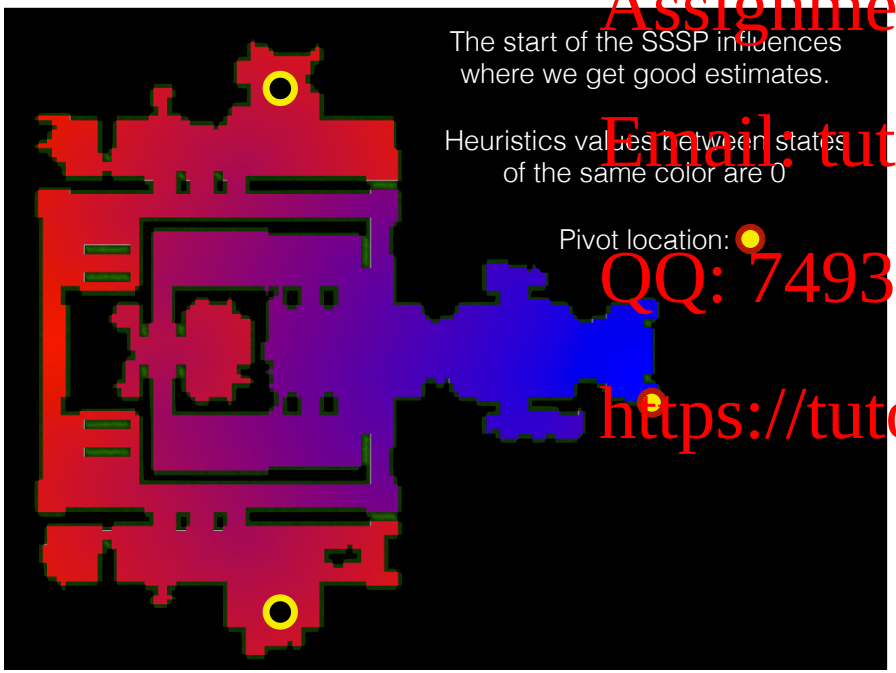
Compute:

$$\sum_{i,j}^{n} |h(i,j) - h^{H_{t-1}}(i,j)|^{+}$$

Add greedily to *H* to get: $(1 - e^{-1}) \approx 0.63$

Compressed

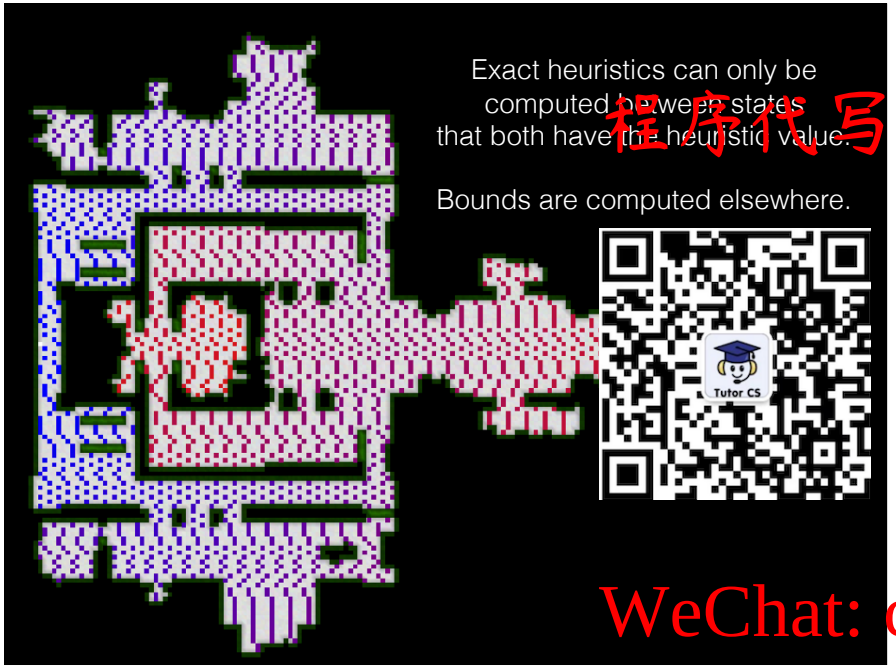


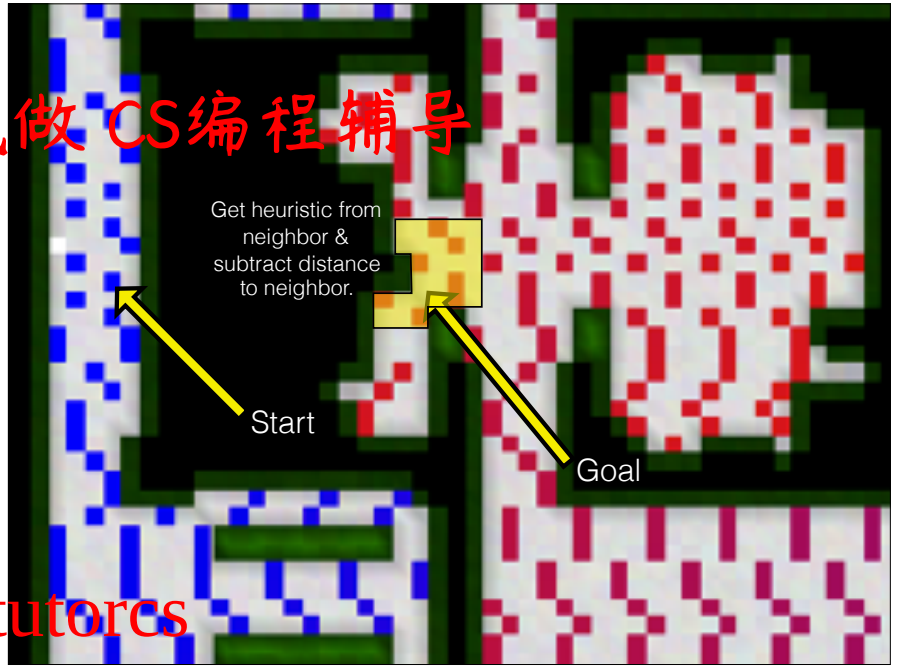


UNIVERSITY of DENVER
DANIEL FELIX RITCHIE SCHOOL OF ENGINEERING & COMPUTER SCIENCE

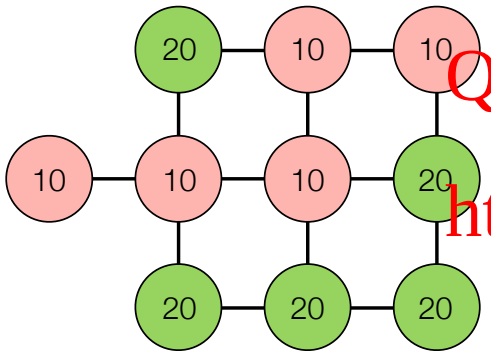
Compression

- Instead of performing min compression (ala PDBs)
  - Throw out some percentage of the heuristic values
  - Interpolate back the distances
- When computing h(a, b)
  - Only lookup the heuristics that are available at state *a*
  - If they aren't available at *b*, compute at a neighbor of *b* that does have the heuristic available
    - Subtract the distance to this state from the heuristic
    - These distance computations only need to be done once at the beginning of search
  - h(a, b) = |d(a, p) - d(p, b')| - d(b', b)

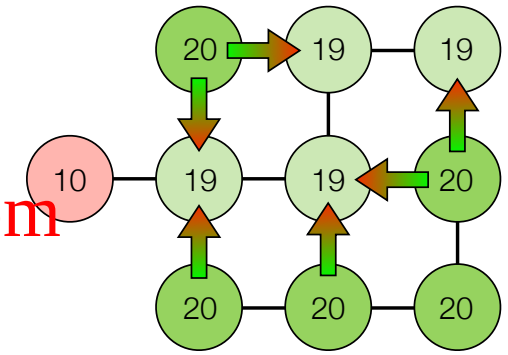




May result in paths of low heuristic values



Heuristic Repair via BPMX



Single Agent Se

Lecture 13
Grid-Based Search

UNIVERSITY of DENVER
DANIEL FELIX RITCHIE SCHOOL OF ENGINEERING & COMPUTER SCIENCE

Canonical Orderings

UNIVERSITY of DENVER
DANIEL FELIX RITCHIE SCHOOL OF ENGINEERING & COMPUTER SCIENCE

Avoiding Duplicates

- In Rubik's Cube we devised ordering rules to avoid duplicates
- Can we devise similar rules for grid pathfinding?
  - Many, many duplicates prevents use with IDA*

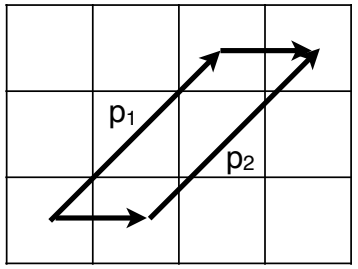
UNIVERSITY of DENVER
DANIEL FELIX RITCHIE SCHOOL OF ENGINEERING & COMPUTER SCIENCE

Canonical Ordering of Paths

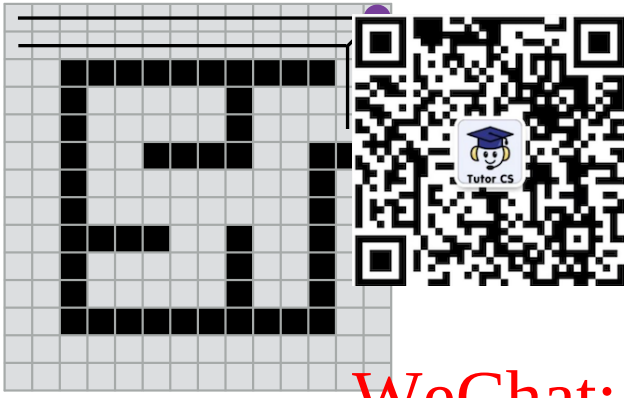
- Order all optimal paths:

Path p_1 is preferred over path p_2 if

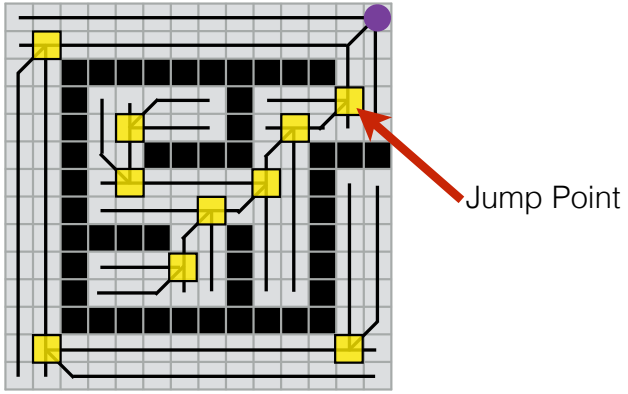
- p_1 has diagonal actions prior to cardinal actions



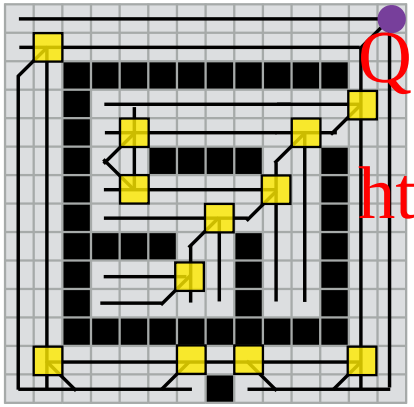
Basic Canonical Ordering



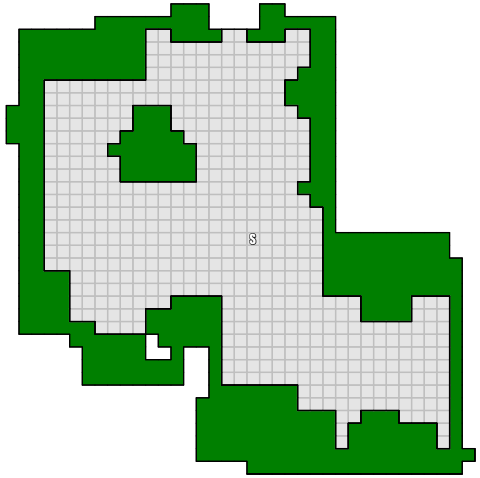
Full Canonical Ordering (Tree)



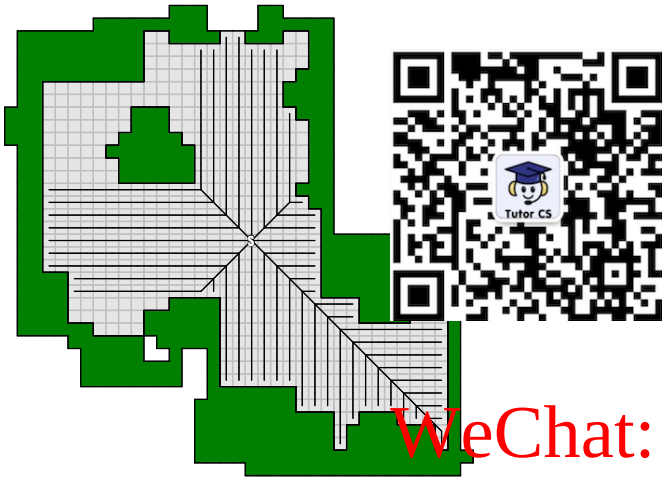
Full Canonical Ordering (graph)



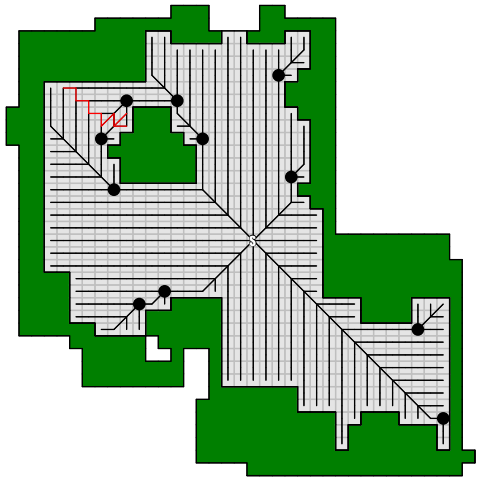
Basic Map



Simple Canonical Ordering



Canonical Ordering



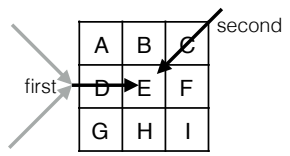
UNIVERSITY of DENVER
DANIEL FELIX RITCHIE SCHOOL OF ENGINEERING & COMPUTER SCIENCE

Correctness

- Canonical orderings are guaranteed to be correct
- Might have to put same node into the open list based on the direction that we used to generate the state
- Luckily this isn't true for this canonical ordering
  - Happens in general graphs

Proof Excerpt

What if we reach E on the second path
with the same cost (arguments also apply for higher cost)

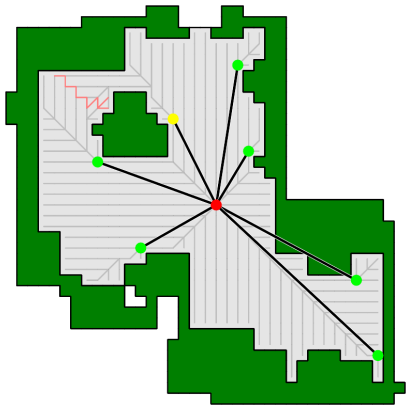


In this case we only have to worry about states D, G & H which the second path will not reach if we do not inherit the parents of the second path at E.

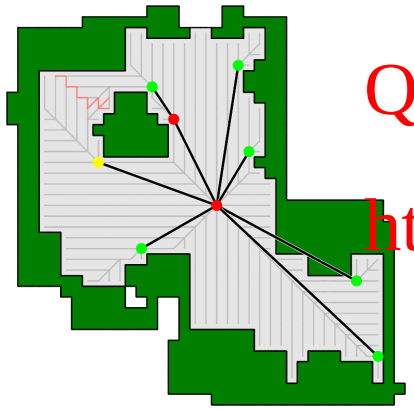
- D is already reached by the first path at a lower cost than second path.
- G will be reached at a lower cost than the second path by one of the two diagonals that come prior to the first path. (If the state left of G is blocked, G will be reached directly from D at the same cost as E.)
- H will be reached at the same or lesser cost as the second path. From the upper/left origin of the first path

JPS

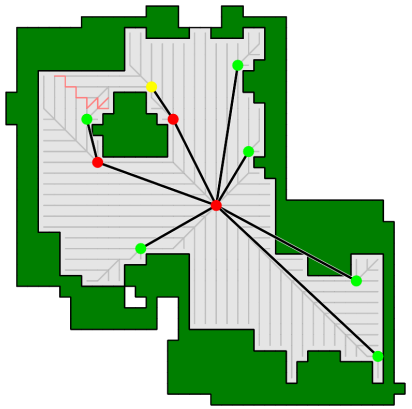
JPS



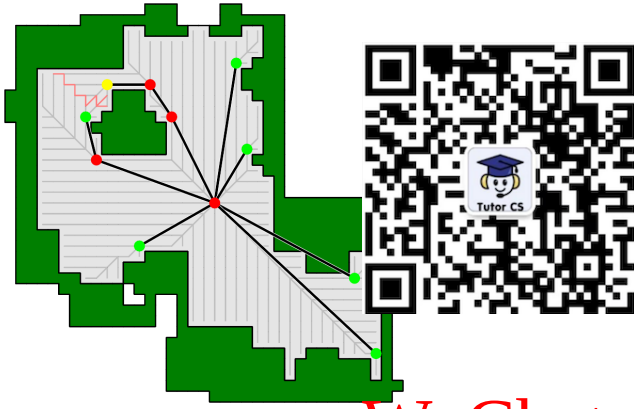
JPS



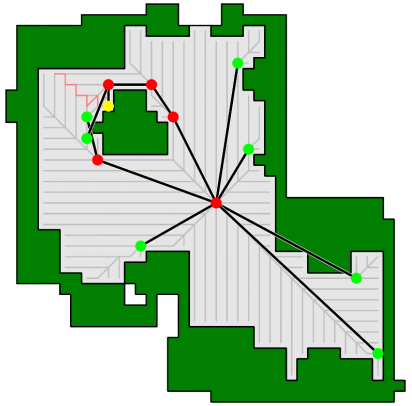
JPS



JPS



JPS



Algorithms on Canonical Orderings

UNIVERSITY of DENVER
DANIEL FELIX RITCHIE SCHOOL OF ENGINEERING & COMPUTER SCIENCE

Canonical A*

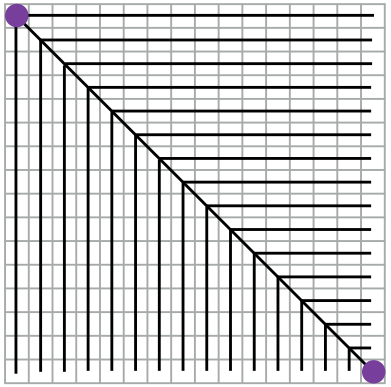
- Use canonical ordering with A*
- Will expand the same number of states
  - Small differences in tie-breaking when reaching the goal
- Will generate far fewer states
- Far less open/closed list operations

Jump Point Search

- Canonical Ordering
  - Only put jump points and the goal c
  - May generate states that A* does n
  - Expands far fewer states
  - Far fewer open list operations

Jump Point Search worst case

- Bound the jumping



Bounded JPS

- Bound how far JPS is allowed to jump
  - Reduces the number of states generated in JPS
  - Increases the number of open list operations
- Parameterized algorithm between Canonical A* (no jumping) and JPS (full jumping)

Weighted JPS

- Falls under category of weighted A*
  - Will cover this in a few lectures

Single Agent Se

Lecture 14
Constraints in Search

UNIVERSITY of DENVER
DANIEL FELIX RITCHIE SCHOOL OF ENGINEERING & COMPUTER SCIENCE

UNIVERSITY of DENVER
DANIEL FELIX RITCHIE SCHOOL OF ENGINEERING & COMPUTER SCIENCE

Constraints

- Previously we have used heuristics to prune search
- We can also build constraints that cannot easily be formulated into heuristics

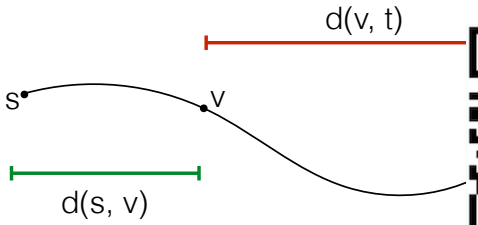
Reach

UNIVERSITY of DENVER
DANIEL FELIX RITCHIE SCHOOL OF ENGINEERING & COMPUTER SCIENCE

Reach

- [Goldberg, Kaplan & Werneck, 2005]
- Given a path P from *s* to *t* and a vertex *v* on P, the reach of *v* with respect to P is the minimum of the length of the prefix of P (the subpath from *s* to *v*) and the length of the suffix of P (the subpath from *v* to *t*).
- The reach of *v*, r(*v*), is the maximum, over all shortest paths P through *v*, of the reach of *v* with respect to P.

Reach

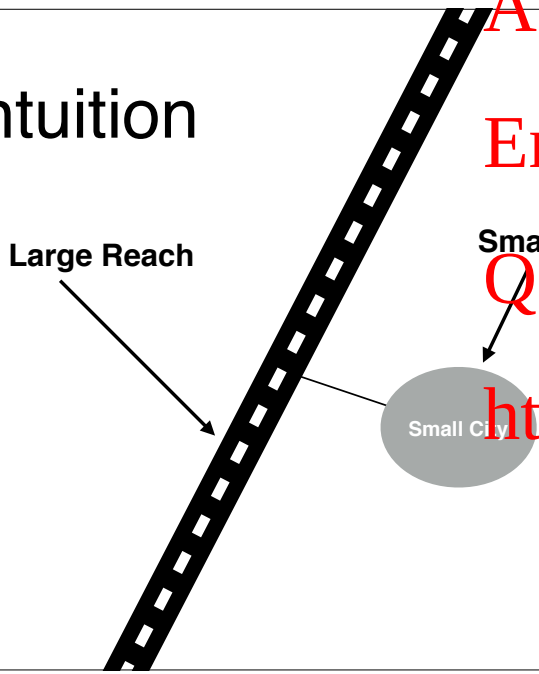


- Interpretation: If this vertex is on a shortest path, the start or goal must be within the reach of the vertex.
  - Defines a radius around this state

Reach during search

- When expanding a vertex *v*
  - If g(v) ≤ r(v), we must expand v
  - else if h(v, goal) ≤ r(v), we must expand v
    - Improving heuristic estimates will improve reach

Intuition



Bounding Boxes

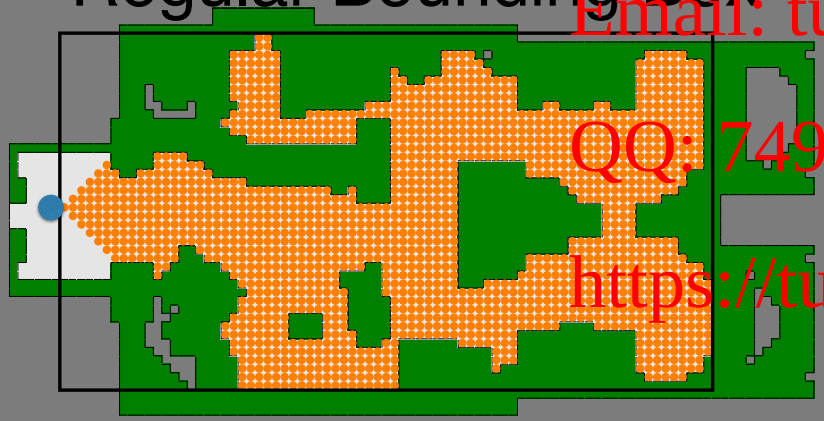
Bounding Boxes

- Pre-compute information constraining optimal paths
- Geometric containers (Wagner et al, 2
  - Bounding Boxes (Rabin & Sturteva

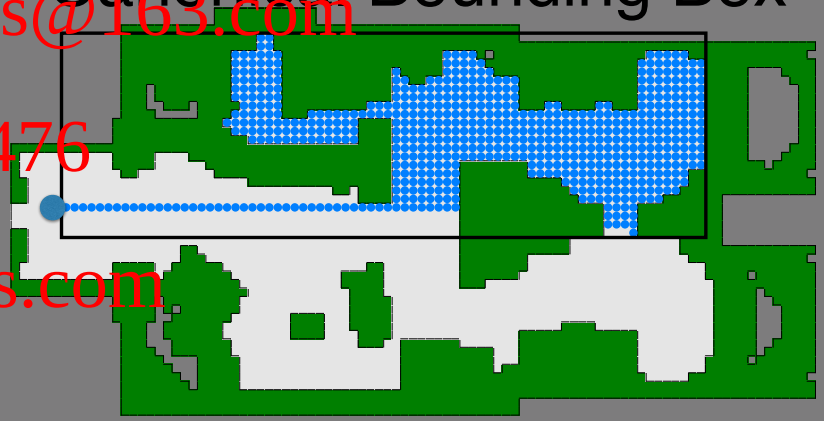
Bounding Boxes

- For each state/action pair:
  - Compute all states that can be reached optimally
  - Summarize these states in a bounding box
- When we search, from state *s*
  - If the goal isn't in the bounding box for an action prune it from the search

Regular Bounding Box



Canonical Bounding Box



Swamps / Dead

- Swamps
  - Nir Pochter, Aviv Zohar, Jeffrey S. Rosenschein, Ariel Felner [2008 - 2010]
- Dead-end heuristic
  - Yngvi Björnsson and Kári Halldórsson, 2006

Single Agent Search

Lecture 15
Exact Shortest Path Queries

UNIVERSITY of DENVER
DANIEL FELIX RITCHIE SCHOOL OF ENGINEERING & COMPUTER SCIENCE

Canonical Heuristics

Heuristics v1

- Choose set of canonical points in the world
- Every state in the world computes its nearest canonical point
- Compute the shortest path between a points
  - The distance between arbitrary poi
  - d(a, b) = d(c(a), c(b)) - d(a, c(a)) - c

Heuristics v2

- Choose set of canonical points in the world
  - Every point is assigned a radius
  - Distance to border
- Every state in the world computes its distance to the border
- Compute the shortest path between all borders
  - The distance between arbitrary points is:
  - d(a, b) = d(b(a), b(b)) + d(a, b(a)) + d(b, b(b))

Canonical Heuristics

- General ideas
- Can be generally applied
  - Even to Towers of Hanoi with compressed PDBs
- But!
  - With domain specific information we can do better

Highway Dimension & Transit Routing

Highway Dimension

- For a given point p:
  - How many points at radius r are on paths to all states at radius 4r?
- What are the implications of low highw
  - Most states have low reach
    - Why?
  - Transit Routing

Transit Routing

- Compute Transit Points
  - All points at a small radius en route to larger radii
- Compute APSP between transit points
- Store the distance to all transit points with each state
- Shortest s-t path length:
  - $\min_{ij}(d(s, t_i(s)) + d(t_i(s), t_j(t)) + d(t_j(t), t))$

Single Agent Search

Lecture 16
Abstraction / Refinement

Practice building abstractions on maps in class (see 2016 exam question)

UNIVERSITY of DENVER
DANIEL FELIX RITCHIE SCHOOL OF ENGINEERING & COMPUTER SCIENCE

Abstraction

- We have seen abstraction for heuristics
- What about for pathfinding?

- Need right type of abstraction
  - Downward refinement property
  - Serializable subgoals

- Are paths in PDBs refineable?

Abstraction

- Group strongly connected states
- Add edges between abstract states if reachable in low-level graph
- A path exists in the high-level graph if low-level graph

Refinement

- Given an abstract path it can be refined:
  - Incrementally
  - Guarantee completeness
  - No guarantee of optimality
- Lecture continues in DAO slides

Single Agent Search

Lecture 17
Suboptimal Search

Weighted A*

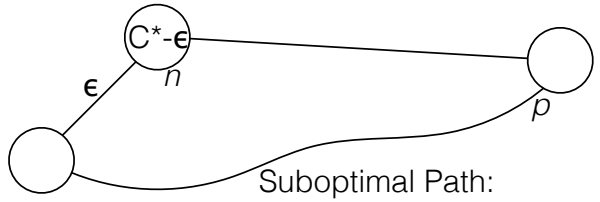
- A*
  - f(n) = g(n) + h(n)
- Weighted A*
  - (a) f(n) = (1-w)·g(n) + w·h(n)
  - (b) f(n) = g(n) + w·h(n)
- Pure Heuristic Search (a) with w=1
- Dijkstra's w=0
- Weighted A*: (b) with w>1

Optimality of WA*

- Complete? yes (finite graph)
- Optimal? no
- Consider optimal path from start to gc
  - Let *n* be the first node on open (opt
- Consider path found via *p*
- *p* was chosen over *n*, so:
  - f'(p) ≤ f'(n)
  - g(p) ≤ g(n)+w·h(n) ≤ w(g(n)+h(n)) ≤ w·f(n) ≤ w·C*

Example

First Node on Open
on Optimal Path:
f = C*, f' = ϵ+w(C*-ϵ)



Suboptimal Path:
Cost: f' ≈ w·C*

Observation

- Slack in bound: (w-1)g(n)
  - Worst case occurs if the search only expands a single state of the optimal path
  - The more of the optimal path expanded, the lower the actual bound
- We get much better paths than the bound in practice

Observation (Pearl)

- We can separate the proof of the suboptimality bound from the search for the suboptimal solution
- When a regular A* search has a f-cost of F, we know that the optimal solution must be at least F.
  - A w-optimal solution can have cost at most w·F
- This is an important and general principle

Optimistic Search

- Thayer & Ruml, 2008
- Maintains two open lists
  - One for an A* search
  - One for a weighted A* search with l
    than the actual bound desired in pr
- Perform the weighted search until the
- Continue expanding states on the A*
  bound is proven

Optimistic Search

- Notes:
  - Same states on both open lists on all times
  - So, must re-expands states when shorter path found
  - In many domains re-expansions can be very expensive
  - Works best on domains with variable edge weights or with other unique costs functions