A Survey of Suboptimal Search Algorithms

Jordan T. Thayer and Wheeler Ruml

University of New Hampshire

jtd7, ruml at cs.unh.edu

slides at: http://www.cs.unh.edu/~jtd7/papers/

This is Search

- initial state
- expand

  generates all successor states, implicitly computes $g(n)$
- goal test
- $h(n)$

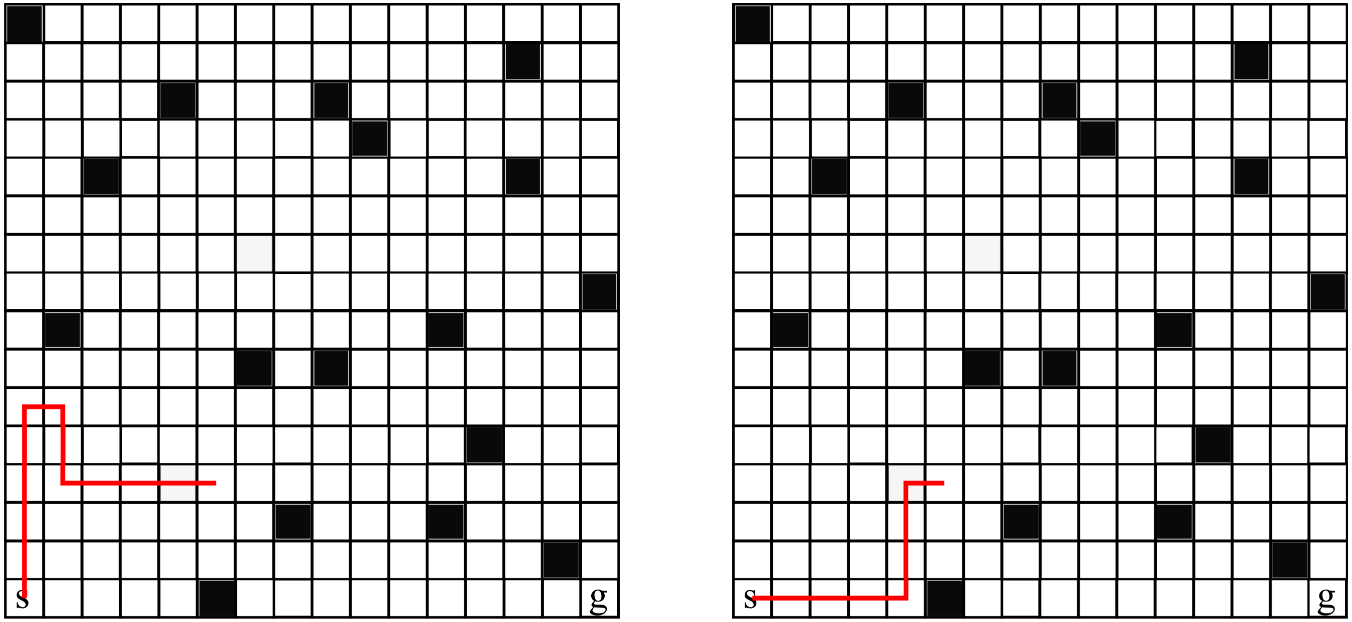
  estimates cost of reaching a goal

  admissibility: non-overestimating

  consistency: obeys triangle inequality
- state: problem data
- node: state, g, parent pointer

What is a duplicate?

1. while *open* has nodes
2. remove n from *open* with minimum $h(n)$
3. if n is a goal then return n
4. otherwise expand n, inserting its children into *open*
5. return failure



same state, different path

Common Data Structures

1. while *open* has nodes
2. remove n from *open* with minimum $h(n)$
3. if n is a goal then return n
4. otherwise expand n, inserting its children into *open*
5. return failure

- openlist:

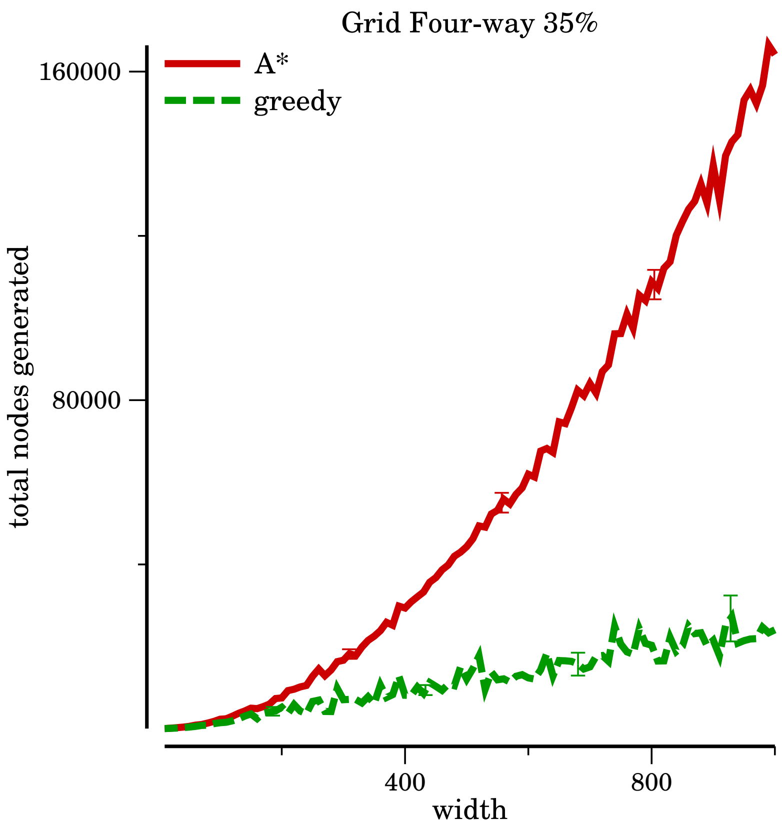
  heap: handles real costs, large ranges of values, etc

  bucket list: more efficient, requires integer values

- closed list:

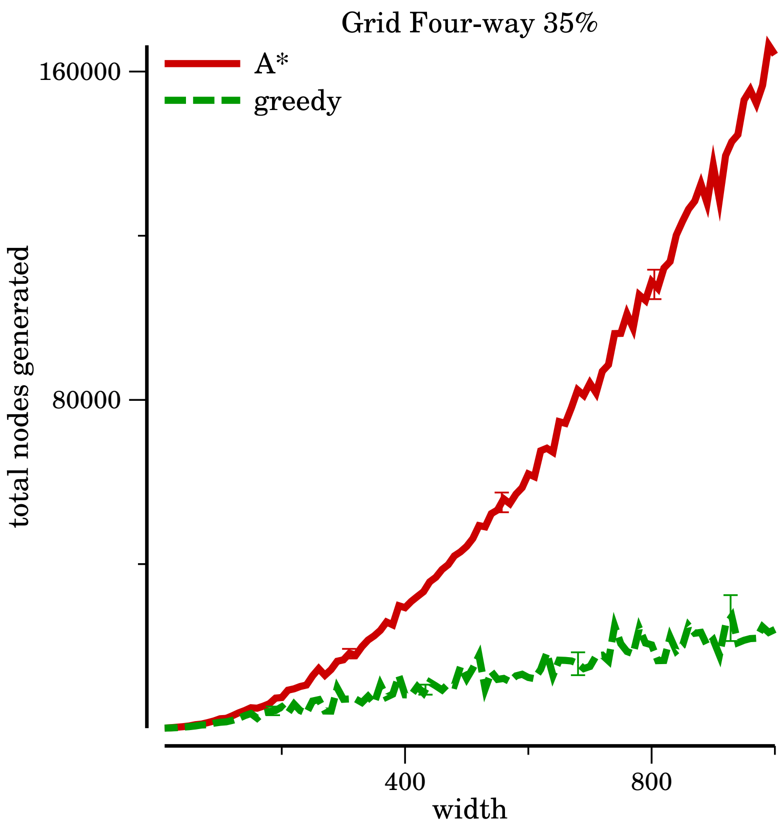
  hash table: in practice, also includes open nodes

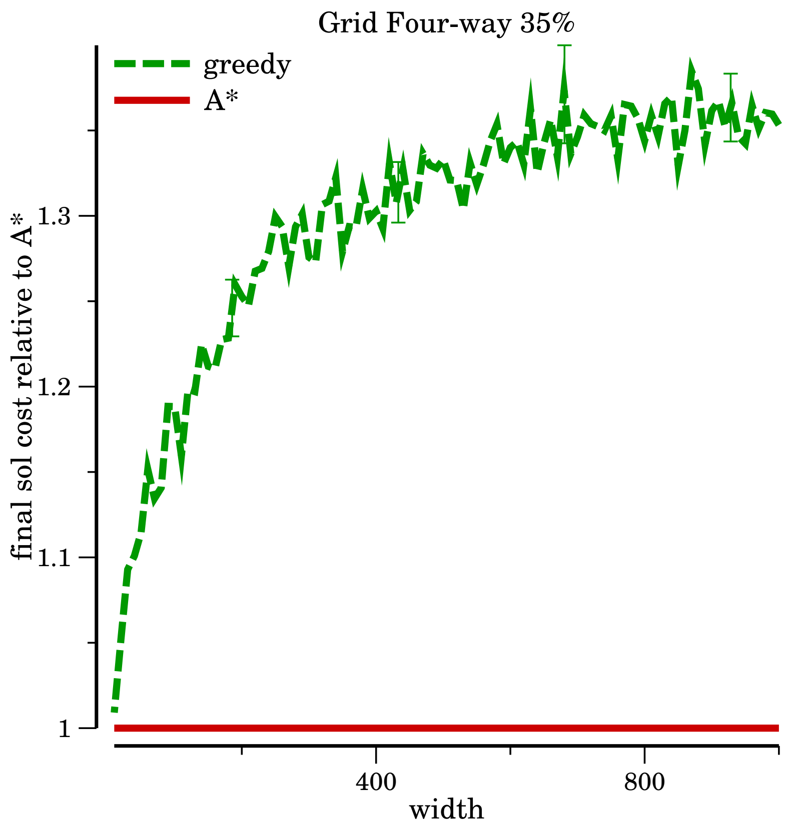
Different Kinds of Search



suboptimal search scales better than optimal search

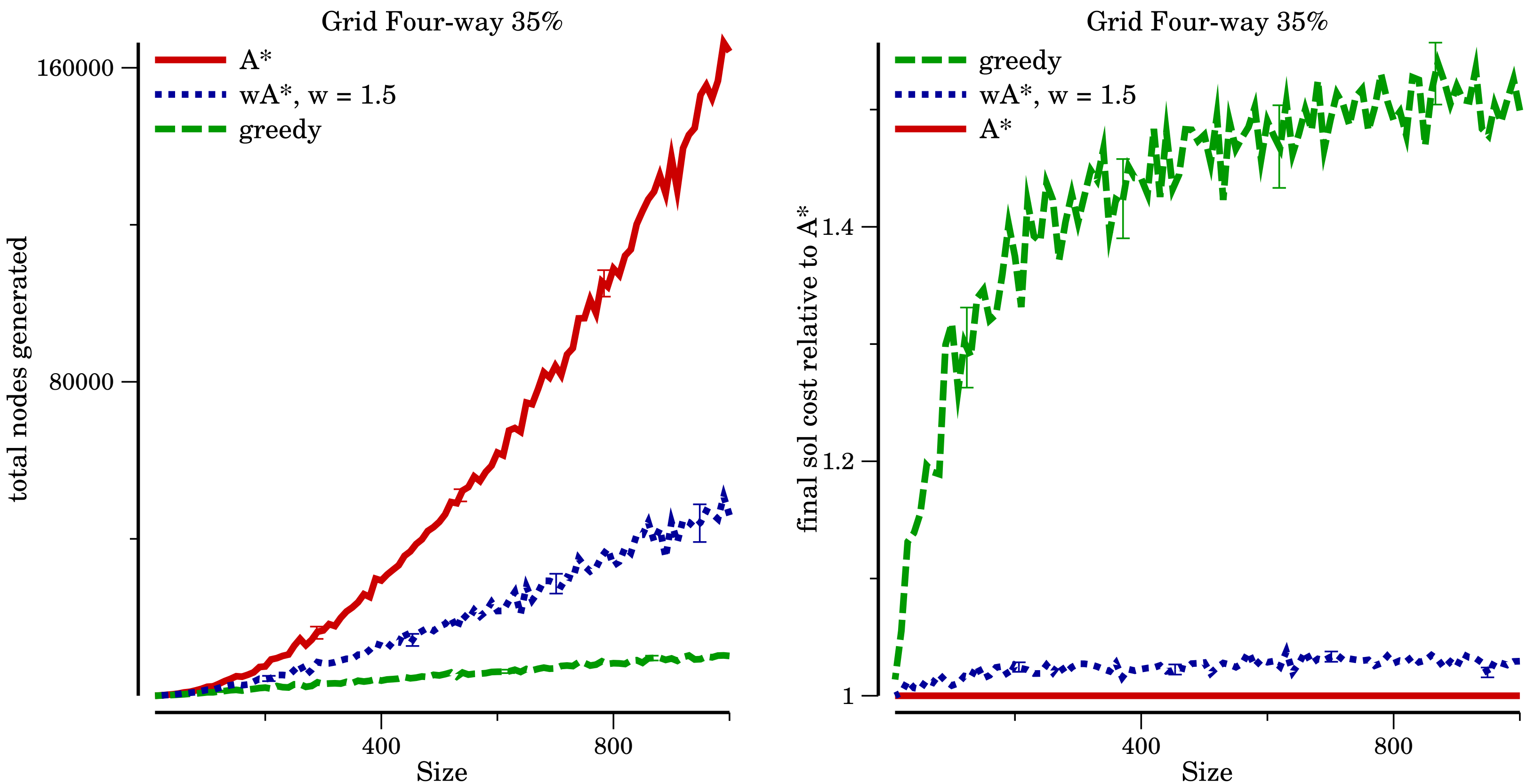
Different Kinds of Search





it does so at the cost of solution quality

Different Kinds of Search



bounded suboptimal search presents a middle ground

Simple Tips for Efficient Search

- store open nodes in 'closed list' as well

  prevents multiple copies of a state being open at once

- duplicate checking and delaying or dropping

- correct tie breaking

  varies by search, generally prefer high g

- goal test on generation

  then prune for bounded suboptimal search

- recursive expansions (to reduce heap operations)

About the Tutorial

- Jordan and I will alternate
- bibliography at the end
- the pseudo code
  - not presented during talk
  - included for later review
- not comprehensive due to time constraints
  - each section covers 'greatest hits'
  - our personal experience and biases

Things We Will Discuss

- suboptimal search — minimize solving time
  - greedy best-first search
  - beam search
  - LSS-learning real-time Search (LSS-LRTA*)
- bounded suboptimal search — balance time and cost
  - weighted A*
  - optimistic search
- anytime search — unknown deadlines
  - anytime repairing A*
  - anytime weighted A*
  - restarting weighted A*

Things We Won't Discuss

- optimal search strategies
- bounded-depth tree search
- local search strategies
- constructing heuristics
- using disk
- parallel algorithms
- anything relying on distance estimates

  (come back next session)

Unboundedly Suboptimal Search

Suboptimal Search: Outline

- greedy best-first search

  variant of best-first search

- beam search: best-first, breadth-first

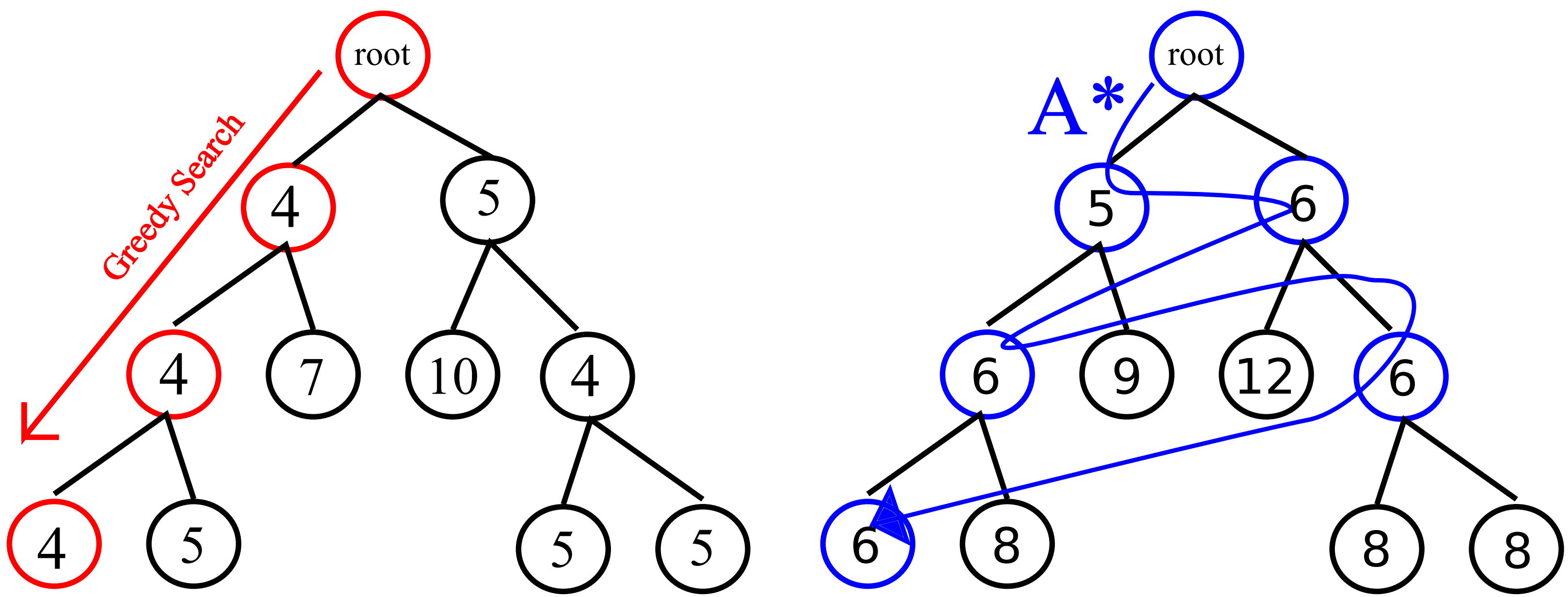
  irrevocable pruning

- real-time search: LSS-LRTA*

  interleaves planning and acting

next session: speedy search, A* on length, beam search on d.

1. Best-first Search On Cost-to-go Estimate, $h(n)$.



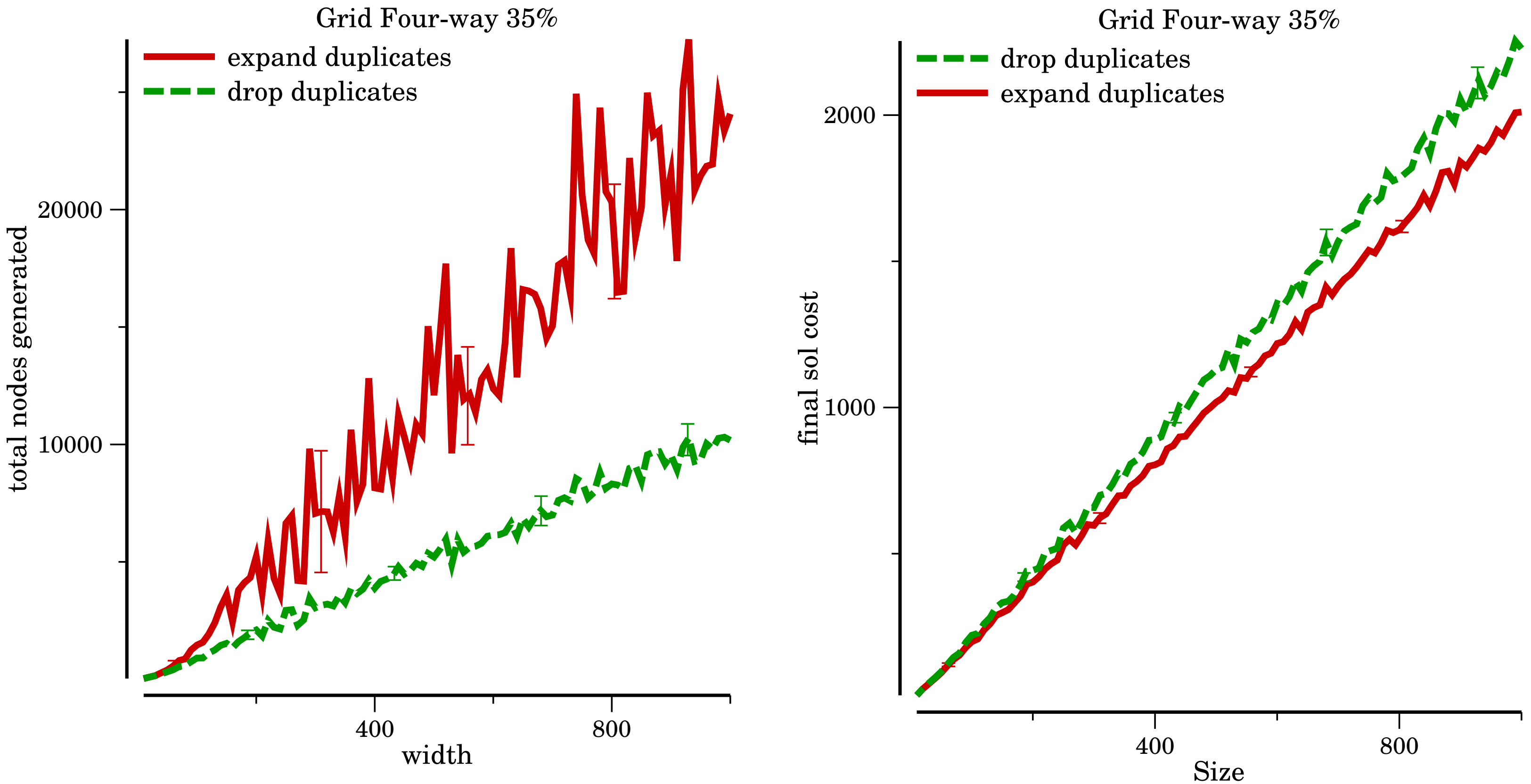
with admissible h, f rises resulting in vacillation

in optimal search, h prunes, in greedy it guides

1. while *open* has nodes
2. remove n from *open* with minimum $h(n)$
3. if n is a goal then return n
4. otherwise expand n, inserting its children into *open*
5. return failure

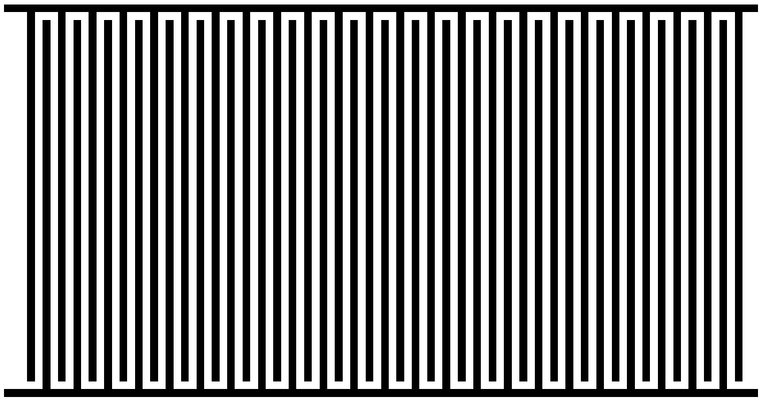
1. while *open* has nodes
2. remove n from *open* with minimum $h(n)$
2a. break ties on h in favor of low $g(n)$
3. if n is a goal then return n
4. otherwise expand n, inserting its children into *open*
5. return failure

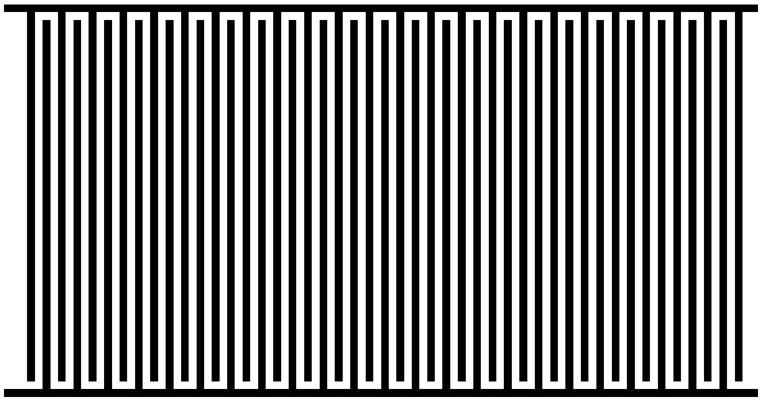
1. while *open* has nodes
2. remove n from *open* with minimum $h(n)$
2a. break ties on h in favor of low $g(n)$
3. if n is a goal then return n
4. otherwise for each child c of n
4a. if c was ever expanded, discard it
4b. if c is in *open*, keep c with smallest g
4c. otherwise insert c into *open*
5. return failure



dropping duplicates improves time, harms quality greedy best-first search: $h(n)$

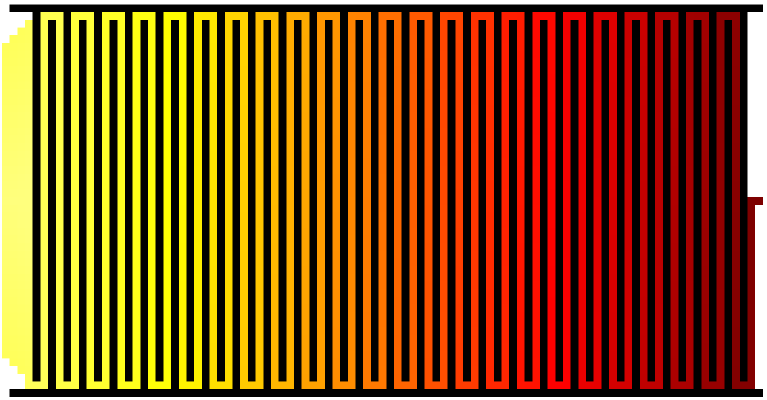
A*: $f(n) = g(n) + h(n)$

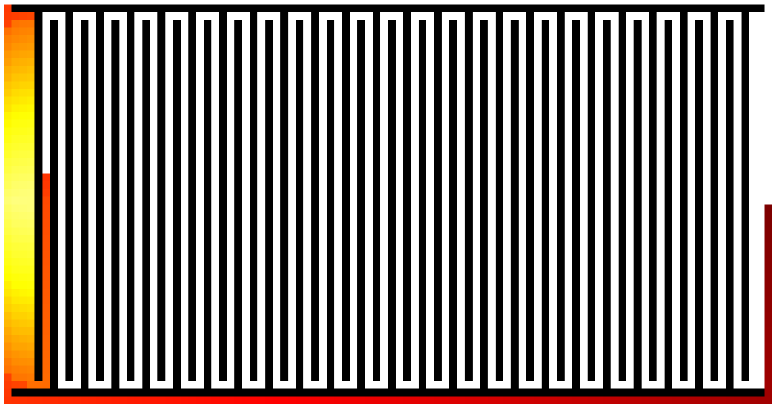




greedy best-first search: $h(n)$

A*: $f(n) = g(n) + h(n)$





basic $\Rightarrow$ brittle

Best-First Beam Search

1. run A* with a fixed sized open list
2. filter out nodes with high $f(n)$

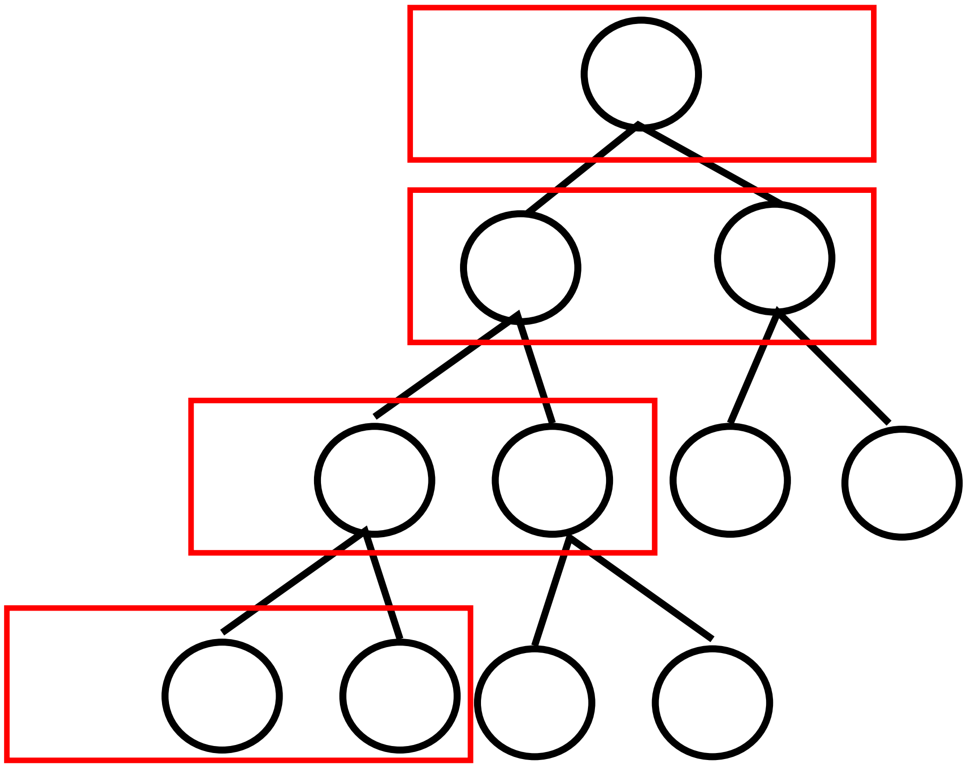
fixed memory *(sometimes)*
conflates propulsion with pruning
doesn't work well in practice Wilt et al SoCS-10

Best-First Beam Search

1. while *open* has nodes
2. remove n from *open* with minimum $f(n)$
3. if n is a goal then return n
4. otherwise for each child c of n
5. insert c into *open*
6. if *open* is larger than *width*
7. discard worst node on *open*
8. return failure

Breadth-First Beam Search

1. run breadth-first search with a fixed sized open list
2. filter out nodes with high $f(n)$



Breadth-First Beam Search

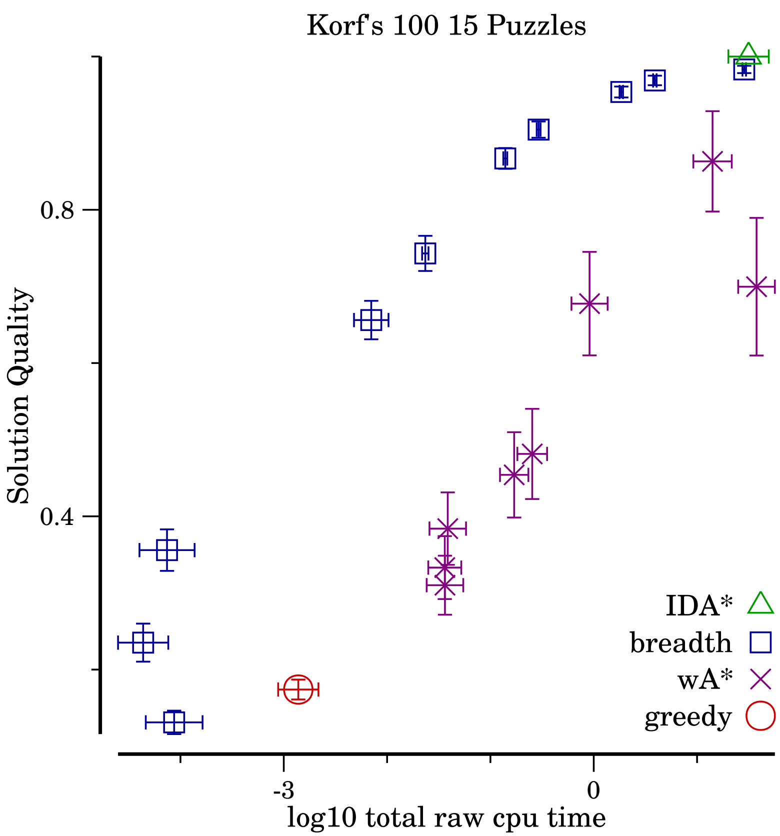
1. while *open* has nodes
2. for each $n \in$ *open*
3. if n is a goal, return n
4. otherwise expand n, adding to *children*
5. *open* becomes best *width* nodes in *children*
5a. best according to $f(n) = g(n) + h(n)$
6. return failure

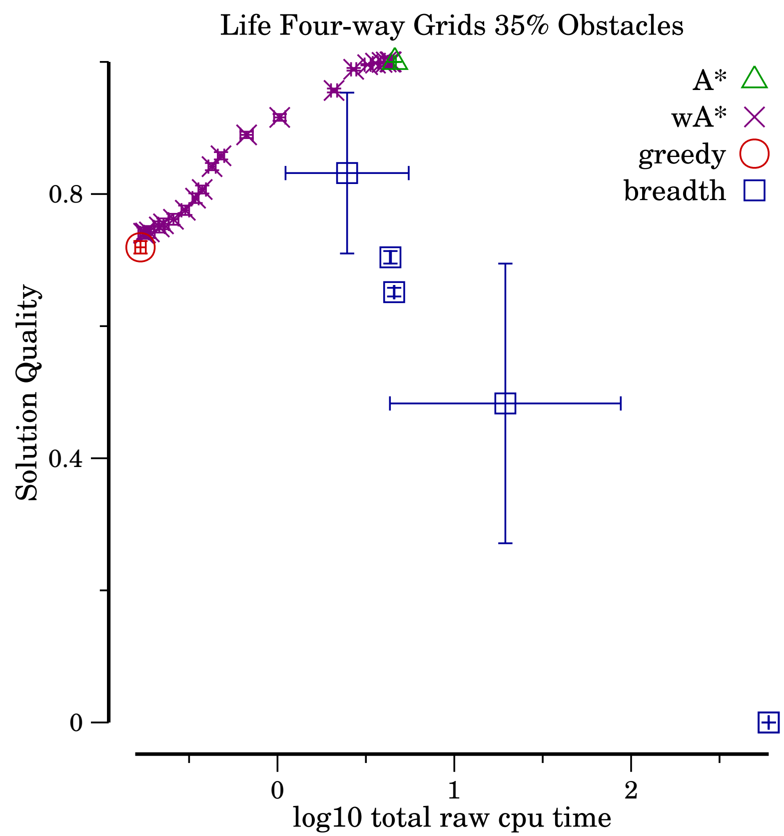
Beam Search — Rich and Knight 1991, Bisani 1992

Breadth-First Beam Search

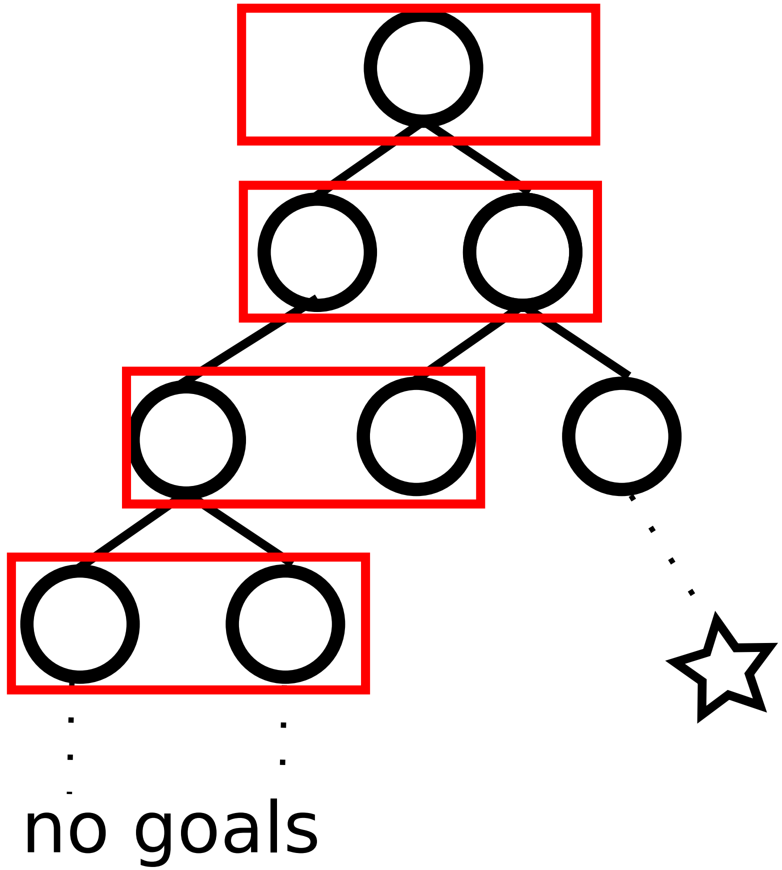
1. while *open* has nodes
2. for each $n \in$ *open*
3. for each child c of n
4. if c is a goal, return it
5. otherwise add c to *children*
6. *open* becomes best *width* nodes in *children*
6a. best according to $f(n) = g(n) + h(n)$
6b. break ties in favor of high $g(n)$
7. return failure

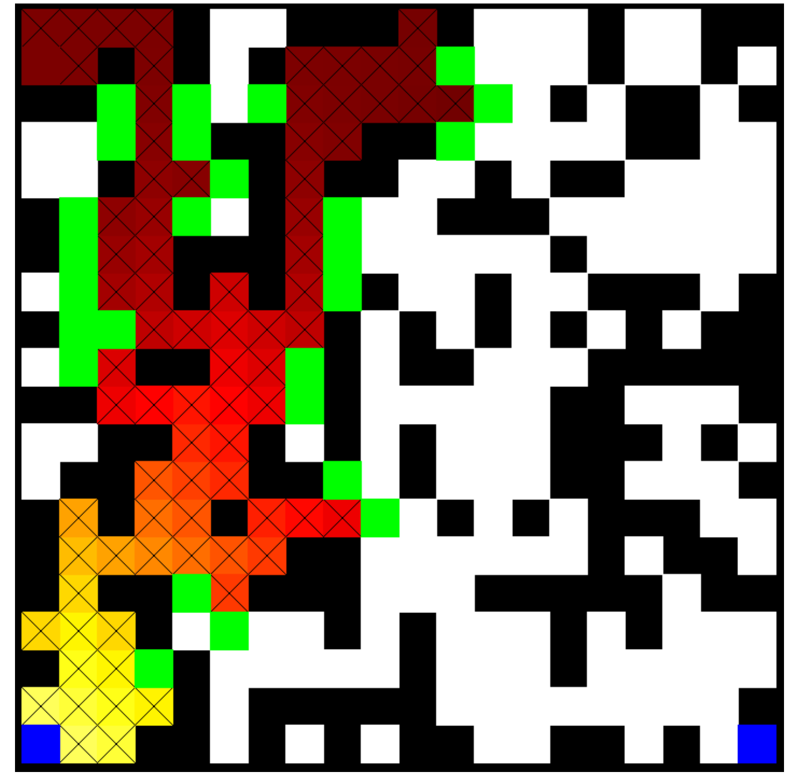
for the uninitiated, the 15 puzzle





beam search might be best approach, or it might be awful





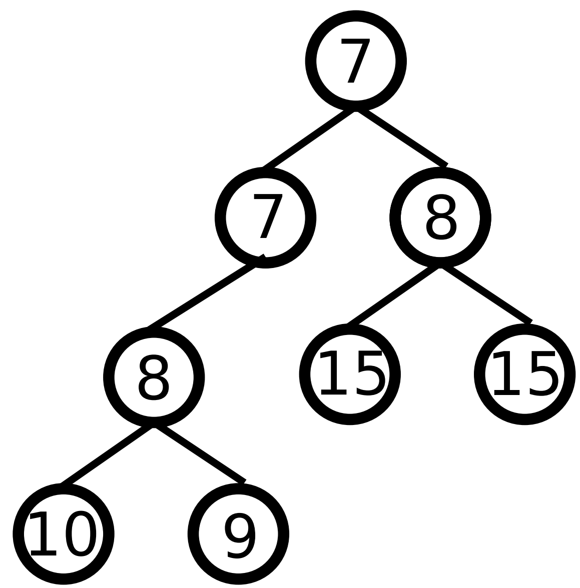
dead ends (left) and many duplicates (right) cause problems

real-time search: interleave planning and acting
time bound on planning per action

1. run A* for a fixed number of expansions
2. commit to $best_f$
3. back-up heuristic values using djikstra variant
4. act and repeat

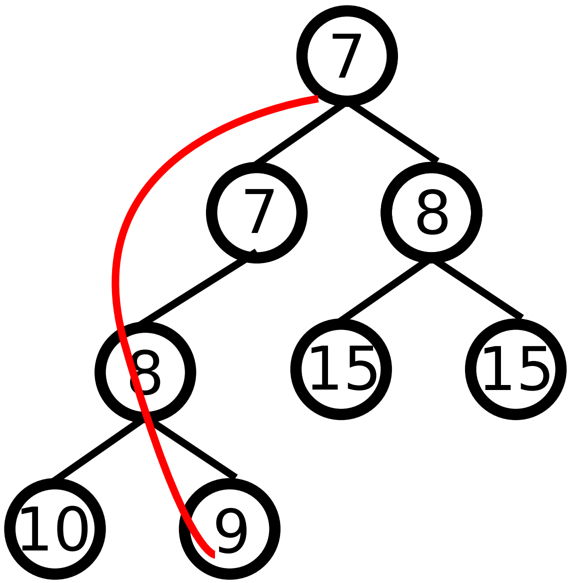
real-time search: interleave planning and acting
time bound on planning per action

1. run A* for a fixed number of expansions
2. commit to $best_f$
3. back-up heuristic values using djikstra variant
4. act and repeat



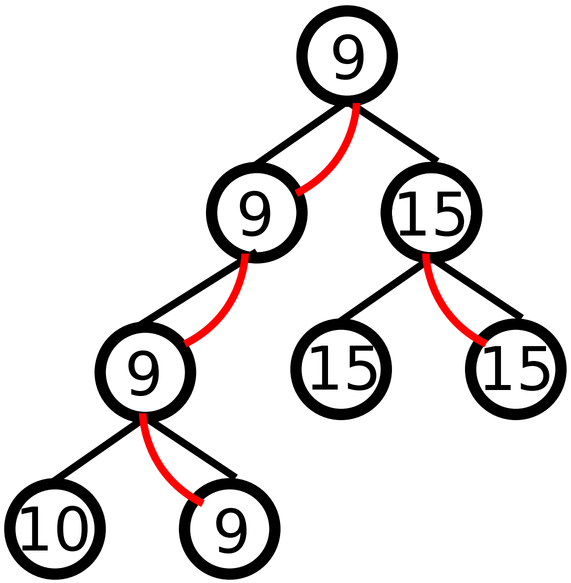
real-time search: interleave planning and acting
time bound on planning per action

1. run A* for a fixed number of expansions
2. commit to $best_f$
3. back-up heuristic values using djikstra variant
4. act and repeat



real-time search: interleave planning and acting
time bound on planning per action

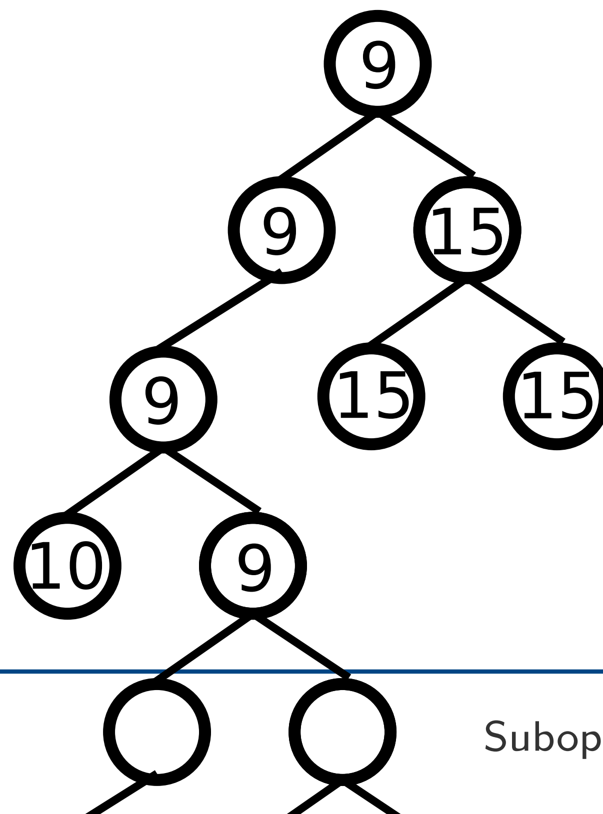
1. run A* for a fixed number of expansions
2. commit to $best_f$
3. back-up heuristic values using djikstra variant
4. act and repeat

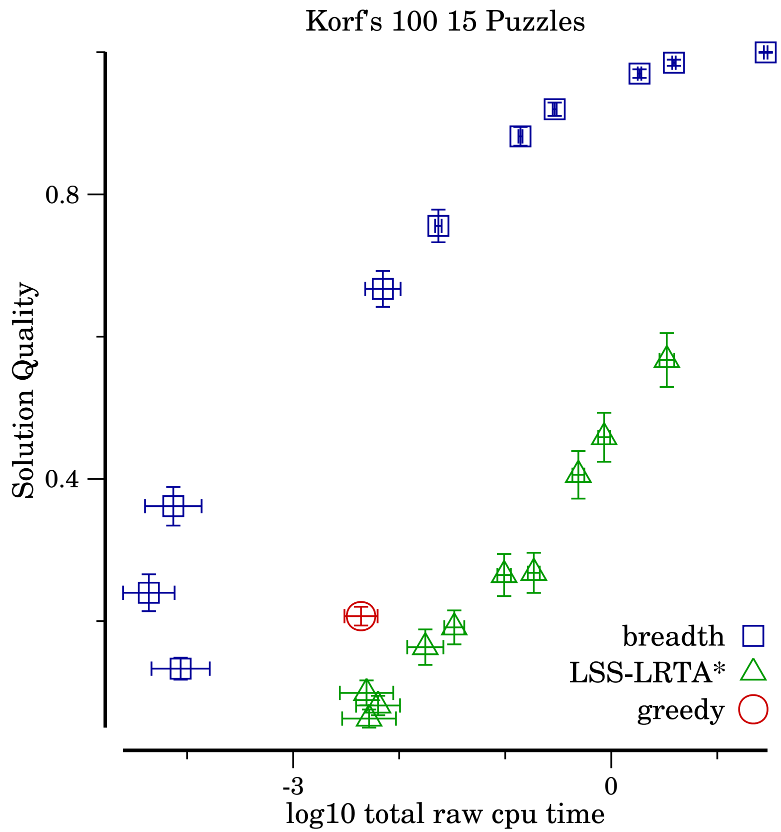


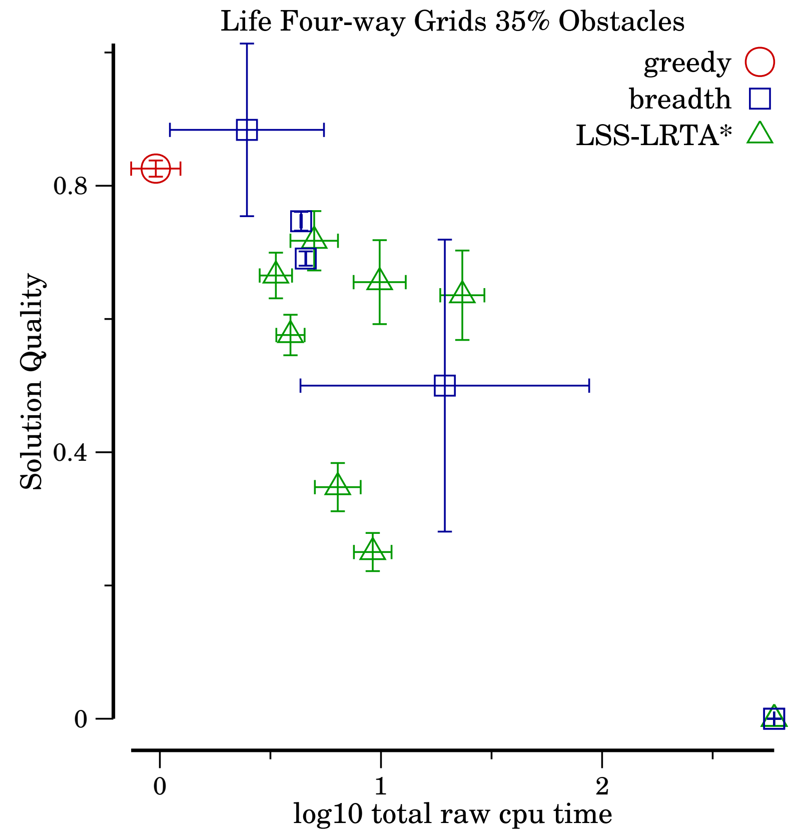
LSS-LRTA* Koenig And Sun, 2008

real-time search: interleave planning and acting
time bound on planning per action

1. run A* for a fixed number of expansions
2. commit to $best_f$
3. back-up heuristic values using djikstra variant
4. act and repeat







don't commit if you don't have to!

Section Summary

- duplicate policy affects solution cost and solving time
- beam searches
  - breadth-first generally better than best-first
  - need closed lists, see Wilt et al SoCS-10
- topology dictates algorithm choice
  - greedy if problems are small
  - beam search if problems too big for greedy
  - many dead necessitates complete searches
  - many duplicates necessitates duplicate dropping
- real-time only if you have real-time constraints

Bounded Suboptimal Search

Bounded Suboptimal Search: Outline

- weighted A*

  larger $w \neq$ faster search

- optimistic search

  selecting an appropriate optimism

  handling duplicates effectively

next tutorial: skeptical search, A^*_ϵ, EES

1. Best-first Search on $f'(n) = g(n) + w \cdot h(n)$

$w \geq 1$

placing additional emphasis on h should encourage progress

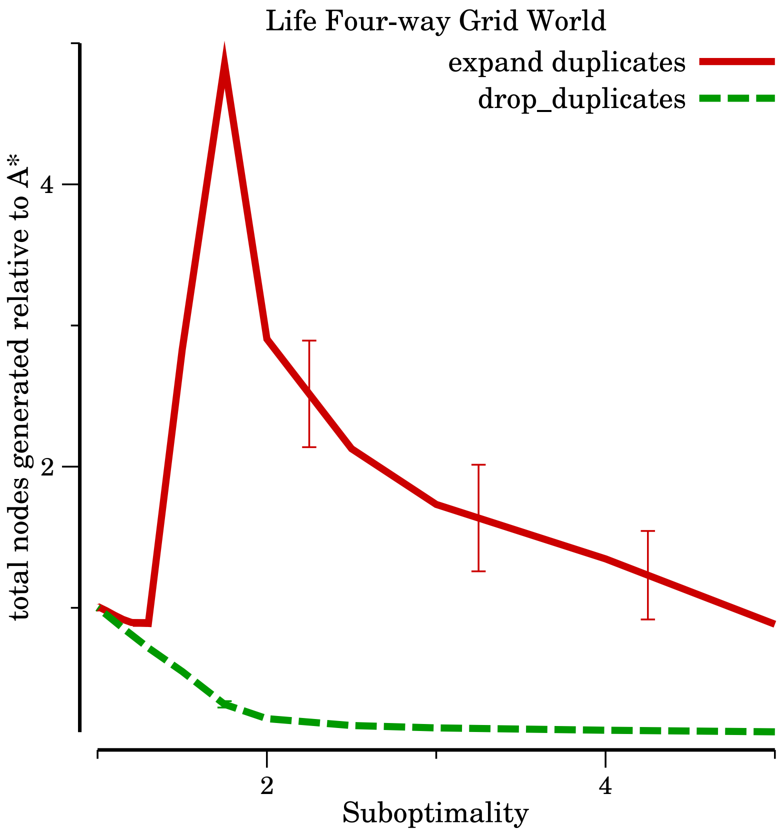
admissible h ensures bound.

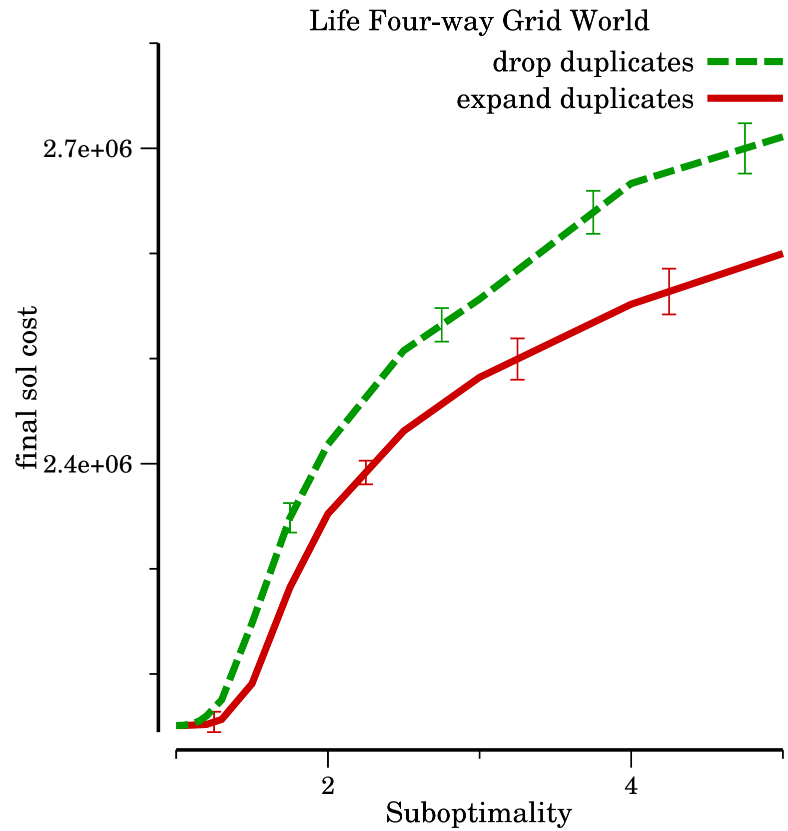
1. while *open* has nodes
2. remove n from *open* with minimum $f'(n)$
3. if n is a goal then return n
4. otherwise expand n, inserting its children into *open*
5. return failure

placing additional emphasis on h should encourage progress

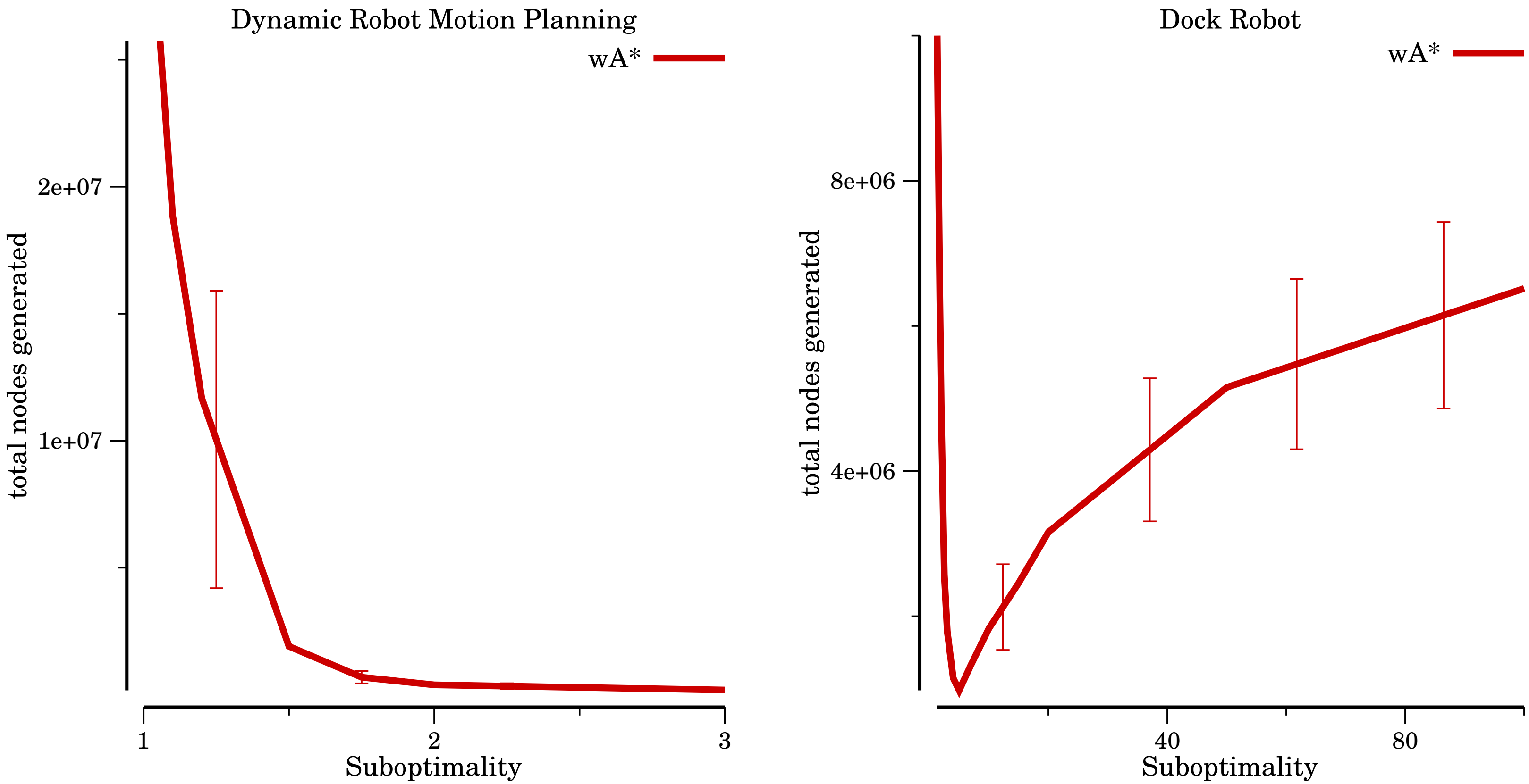
1. while *open* has nodes
2. remove n from *open* with minimum $f'(n)$
2a $f'(n) = g(n) + w \cdot h(n)$
2b. break ties on in favor of low $f(n)$
3. if n is a goal then return n
4. otherwise for each child c of n
4a. if c was ever expanded, discard it
4b. if c is in *open*, keep c with smallest g
4c. otherwise insert c into *open*
5. return failure

discarding duplicates greatly improves performance
at the cost of solution quality
only applicable with consistent heuristics



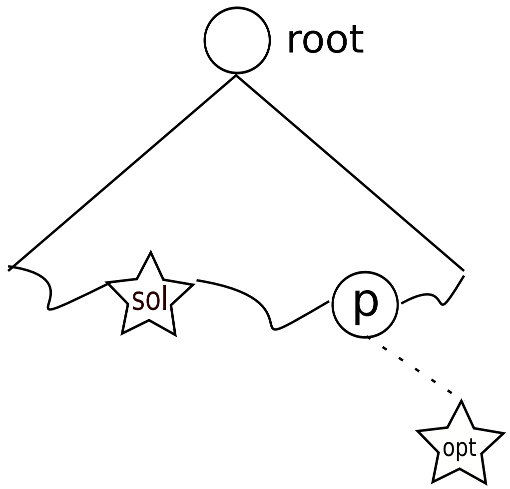


discard duplicates when possible



larger w does not always mean better performance

Bound for Weighted A*



$$f(n) = g(n) + h(n)$$
$$f'(n) = g(n) + w \cdot h(n)$$

- p is the deepest node on an optimal path to opt.

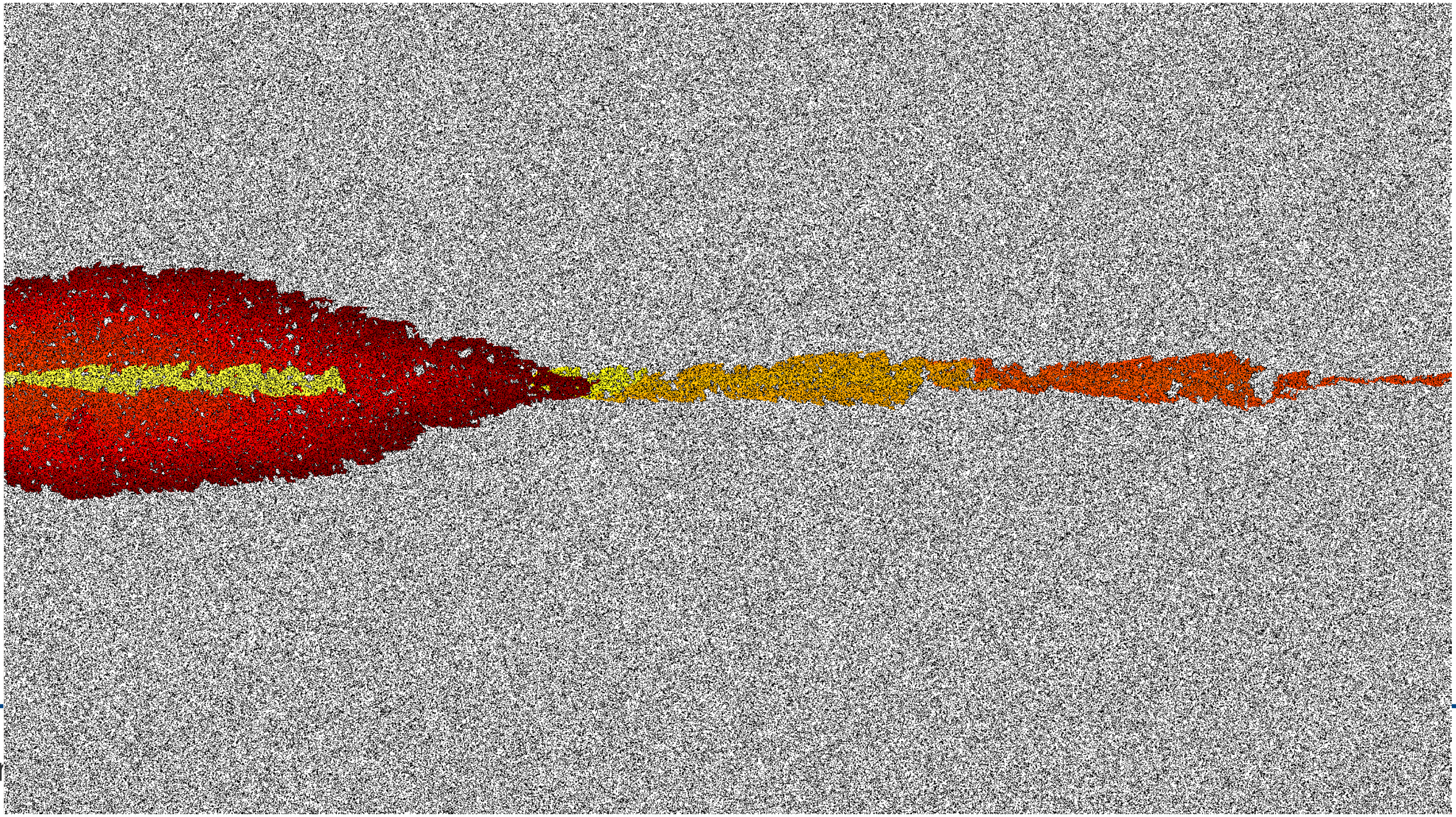
$$
\begin{array}{lllllll}
g(sol) & & & & & & \\
f'(sol) & \leq & f'(p) & & & & \\
 & & g(p) + w \cdot h(p) & \leq & w \cdot (g(p) + h(p)) & & \\
 & & & & w \cdot f(p) & \leq & w \cdot f(opt) \\
 & & & & & & w \cdot g(opt)
\end{array}
$$

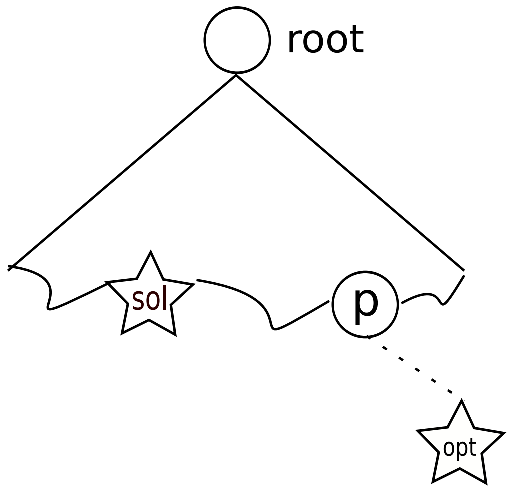
1. works for any $f'(p) \leq w \cdot f(p)$
2. $g(p) + w \cdot h(p) \ll w(g(p) + h(p))$!

- run weighted A^* with a high weight.
- expand node with lowest f value after a solution is found.

  continue until $w \cdot best_f \geq f(sol)$

  this 'clean up' guarantees solution quality.





- p is the deepest node on an optimal path to opt.
- $best_f$ is the node with the smallest f value.

$$\begin{aligned} f(p) &\leq f(opt) \\ f(best_f) &\leq f(p) \end{aligned}$$

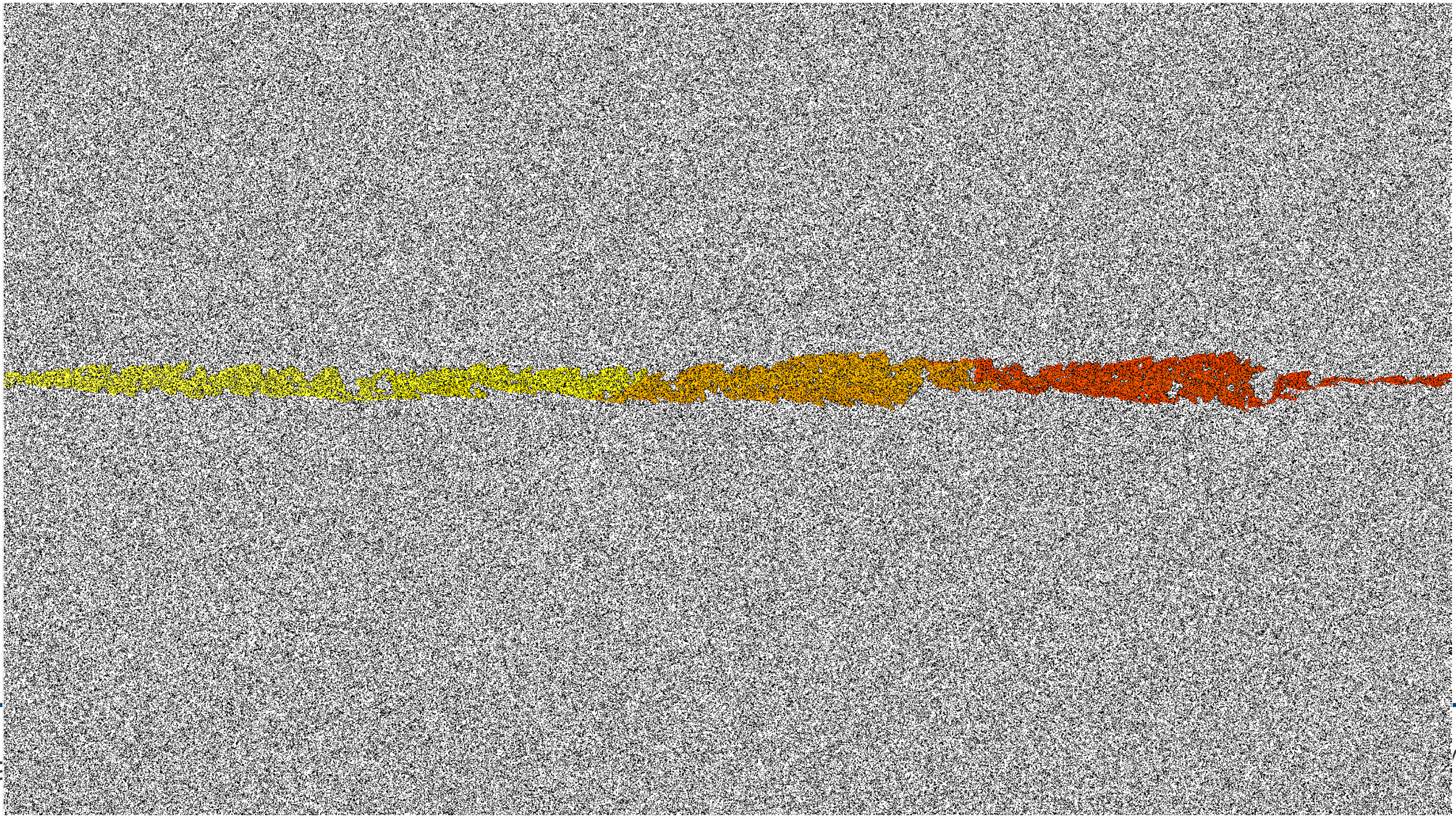
$best_f$ provides a lower bound on solution cost

Determine $best_f$ via priority queue sorted on f

1. run weighted A^* with weight $(bound - 1) \cdot 2 + 1$
2. expand node with lowest f value after a solution is found.

   Continue until $w \cdot best_f \geq f(sol)$

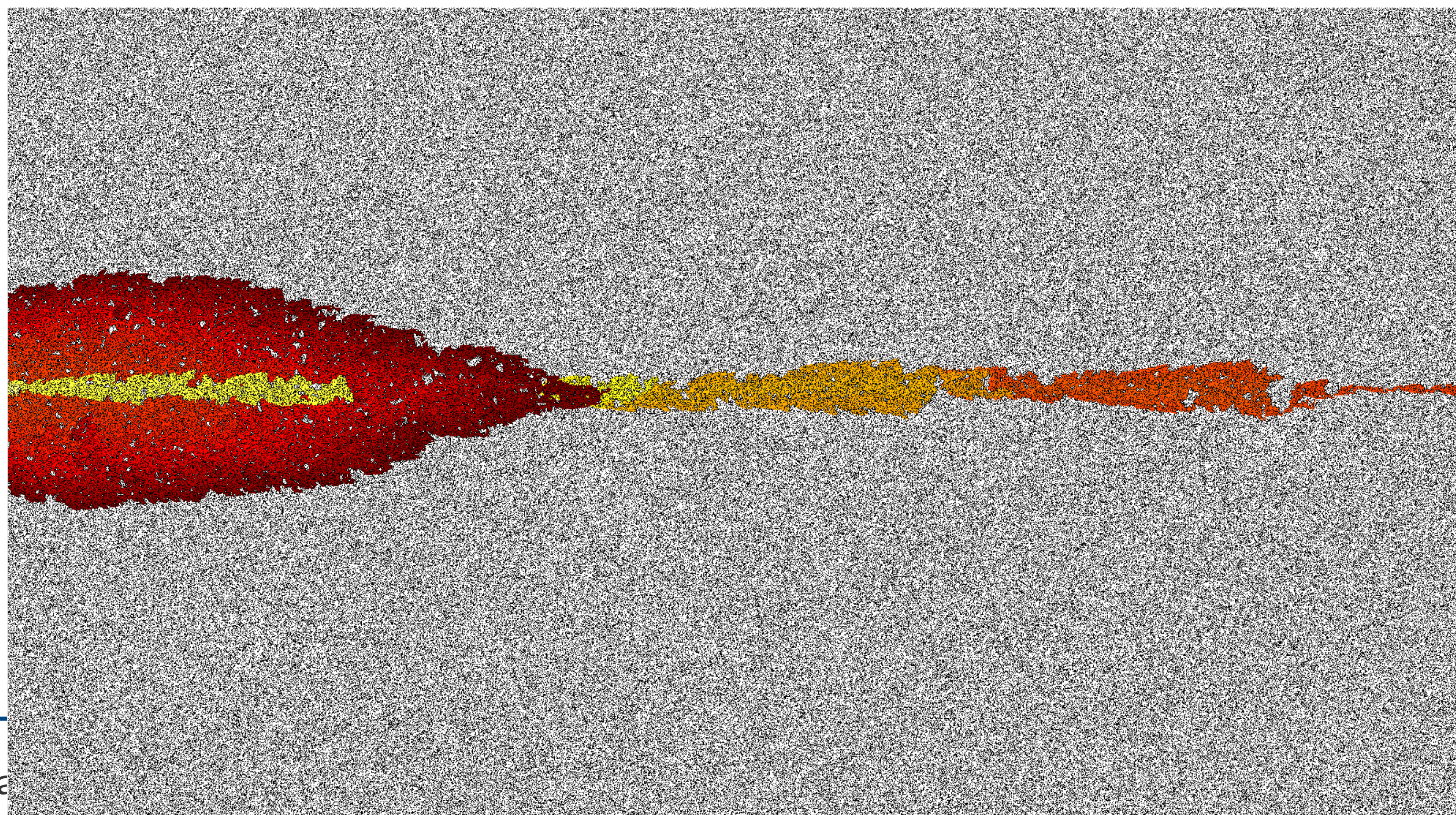
   this 'clean up' guarantees solution quality.

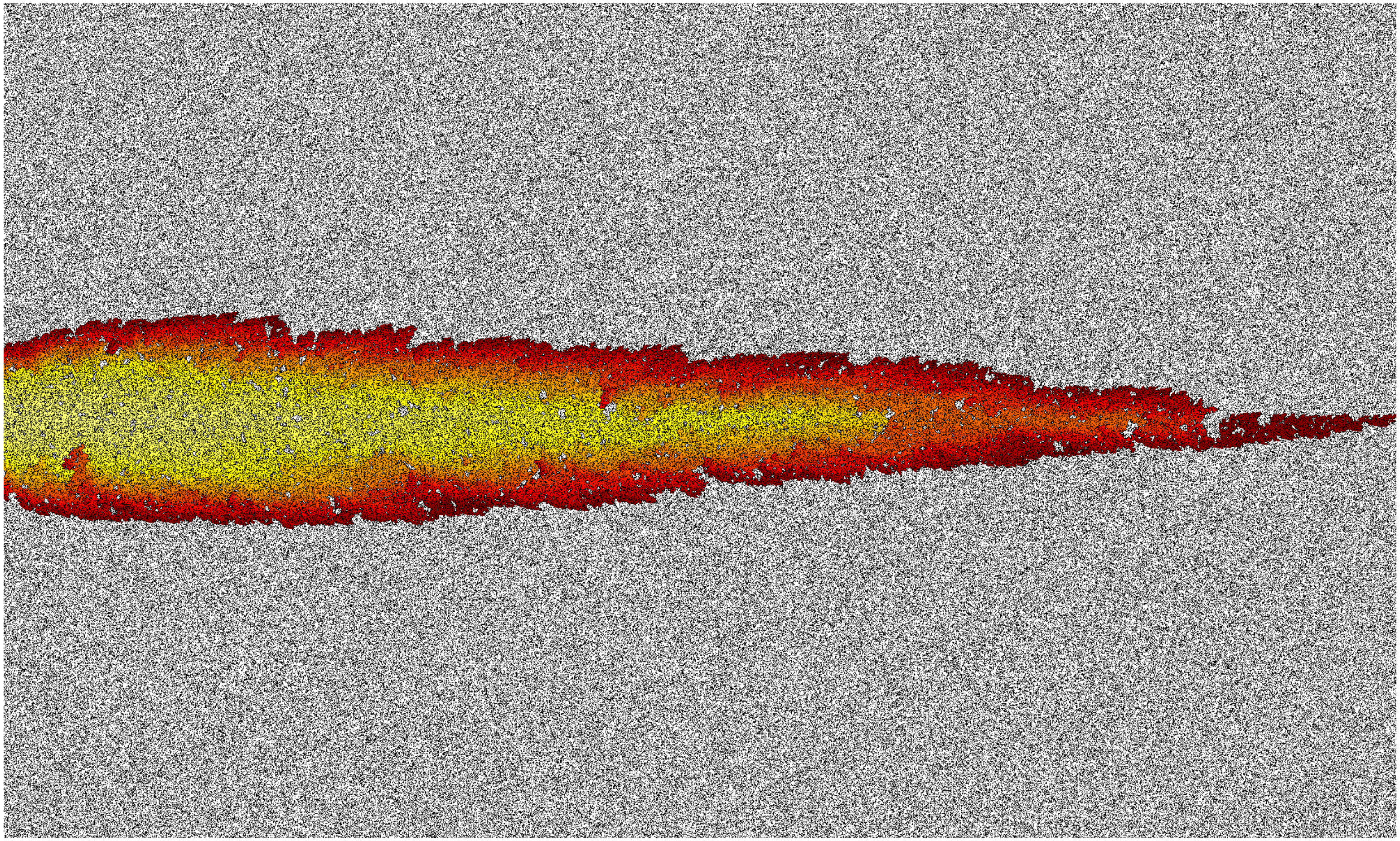


1. run weighted A^* with a high weight.
2. expand node with lowest f value after a solution is found.

   continue until $w \cdot best_f \geq f(sol)$

   this 'clean up' guarantees solution quality.

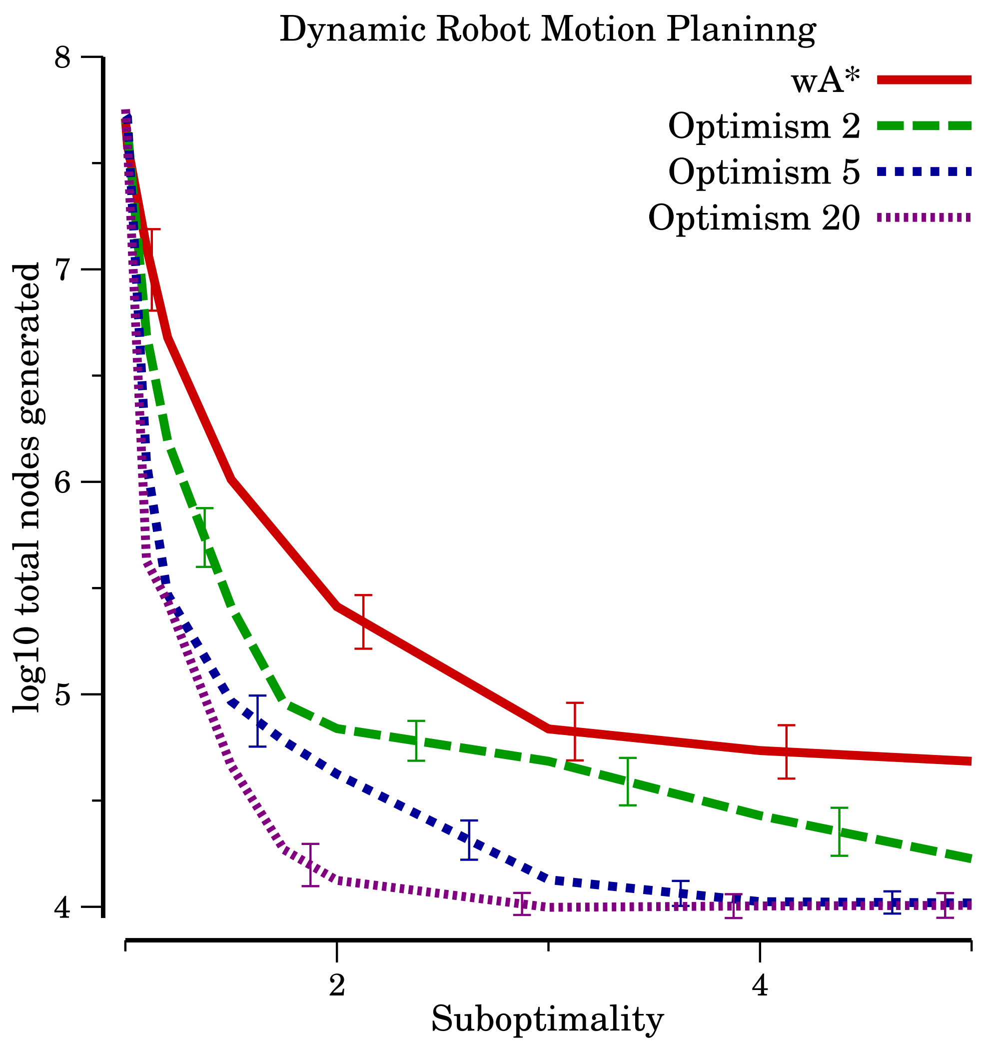




Thayer and Ruml, ICAPS-08

1. $w = (bound - 1) \cdot optimism$
2. while *open* and *clean* contain nodes
3.   if no incumbent has been found
4.   remove n from *open* with minimum $f'(n)$
5.   remove n from *clean*
5.   if n is a goal, set incumbent to n
6.   otherwise expand n, inserting its children into *open* and
7.   otherwise remove n from *clean* with minimum $f(n)$
8.   if $bound \cdot f(n) \leq f($incumbent$)$, return incumbent
9.   otherwise expand n, inserting its children into *open*
10. return failure

In practice, *clean* isn't constructed until an incumbent is found.



proper optimism depends on h accuracy

Section Summary

- use weighted A* as a first approach to a problem
- use optimistic search if you know weighted A* works well
- duplicate handling can be important
  - wA* can drop, requires consistent heuristic
  - other algorithms can delay
- solving problems and showing bounds can be separate steps

Anytime Search

Anytime Search Algorithms: Outline

- anytime repairing A*
- anytime weighted A*
- restarting weighted A*

come back next session for d-fenestration, size-cost search

Anytime Algorithms Based on Weighted A*

anytime weighted A*, Hansen and Zhou 1997

1. run weighted A*
2. if you find a goal, keep going.

anytime repairing A*, Likhachev et al, 2003

1. run weighted A*
2. if you find a duplicate, don't look at it just yet.
3. if you find a goal
4. dump duplicates into *open*, reduce w, keep going.

restarting weighted A*, Richter et al 2010

1. run weighted A*
2. if you find a goal, start over with a lower weight.

Anytime Algorithms Based on Weighted A*

anytime weighted A*, Hansen and Zhou 1997

1. run weighted A*
2. if you find a goal, keep going.

anytime repairing A*, Likhachev et al, 2003

1. run weighted A*
2. if you find a duplicate, don't look at it just yet.
3. if you find a goal
4. dump duplicates into *open*, reduce w, keep going.

restarting weighted A*, Richter et al 2010

1. run weighted A*
2. if you find a goal, start over with a lower weight.

Anytime Algorithms Based on Weighted A*

continued search, Hansen and Zhou 1997

1. run some complete suboptimal search
2. if you find a goal, keep going.

repairing search, Likhachev et al, 2003

1. run some complete parameterized search
2. if you find a duplicate, don't look at it just yet.
3. if you find a goal
4. dump duplicates into *open*, change parameter, keep going.

restarting search, Richter et al 2010

1. run some complete parameterized search
2. if you find a goal, start over with a different parameter.

Anytime Algorithms Based on Weighted A*

anytime weighted A*, Hansen and Zhou 1997

while *open* has nodes
    remove n from *open* with minimum $f'(n)$
    if n is a goal set n as incumbent
        otherwise for each child c of n
            if $f(c) < f(incumbent)$ insert c into *open*
return incumbent

anytime repairing A*, Likhachev et al, 2003
restarting weighted A*, Richter et al 2010

Anytime Algorithms Based on Weighted A*

anytime weighted A*, Hansen and Zhou 1997

anytime repairing A*, Likhachev et al, 2003

while *open* has nodes
    remove n from *open* with minimum $f'(n)$
    if n is a goal
        set n as incumbent
        empty *delay* into *open*
        reduce w
    otherwise for each child c of n
        if c was ever expanded, add it to *delay*
        otherwise insert c into *open*
return incumbent

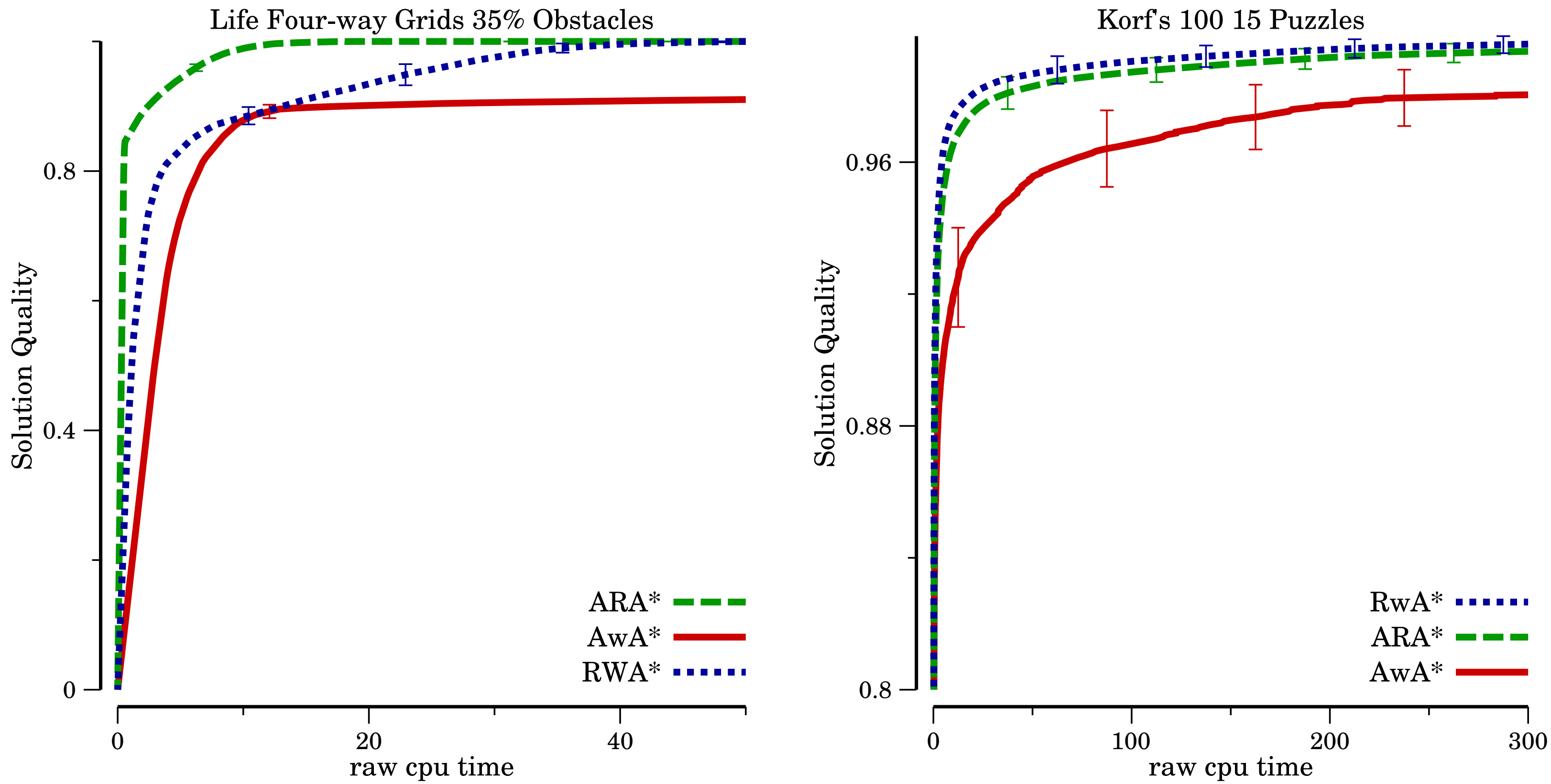
restarting weighted A*, Richter et al 2010

Anytime Algorithms Based on Weighted A*

anytime weighted A*, Hansen and Zhou 1997
anytime repairing A*, Likhachev et al, 2003
restarting weighted A*, Richter et al 2010

while *open* has nodes
    remove n from *open* with minimum $f'(n)$
    if n is a goal
        set n as incumbent
        reduce w
        if $w < 1$, return incumbent
        otherwise restart the search
    otherwise for each child c of n
        if $f(c) < f(incumbent)$ insert c into *open*
return incumbent

Anytime Algorithms Based on Weighted A*

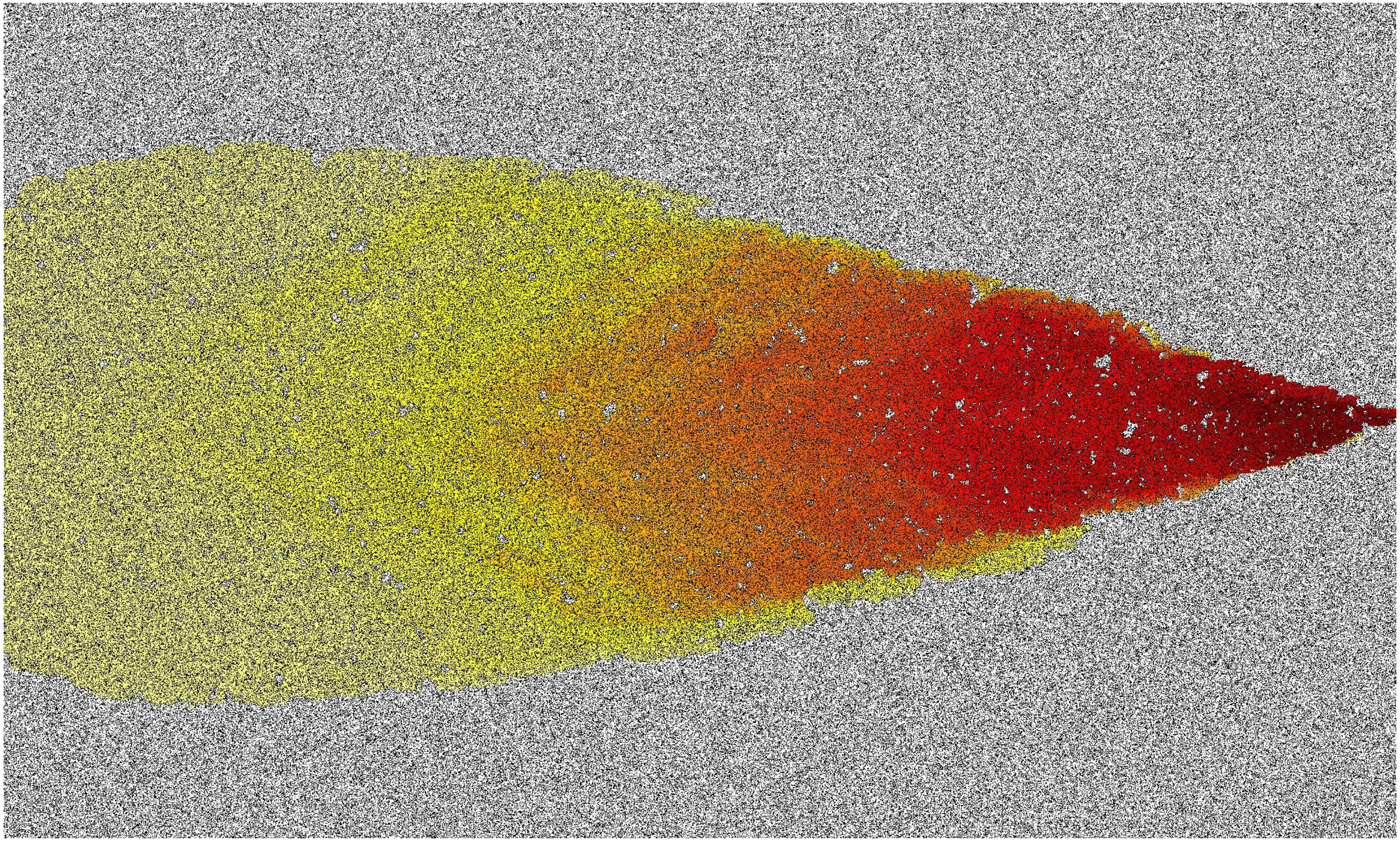


no one best anytime algorithm (or even framework)

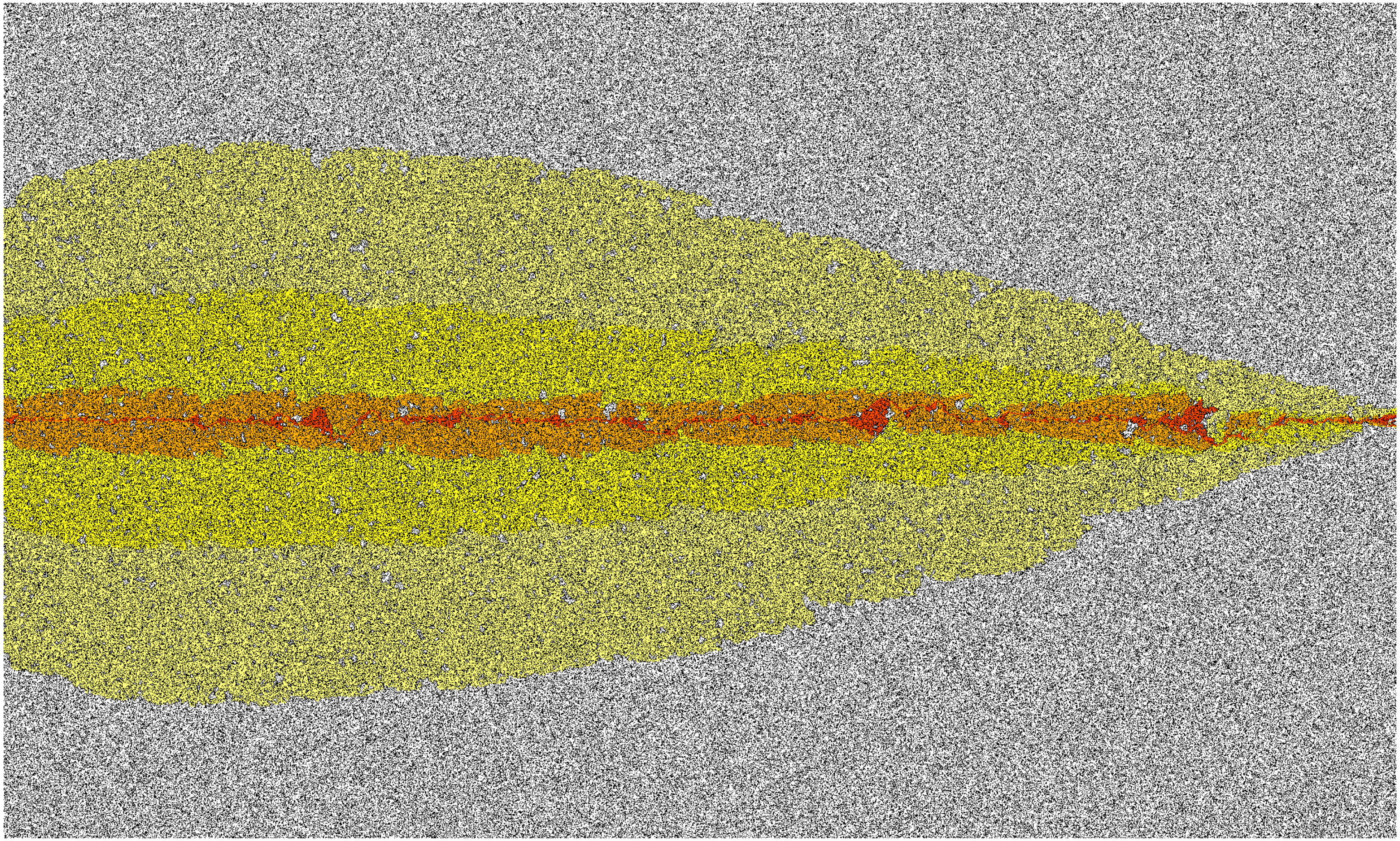
generally, try repairing first.

Anytime Algorithms Based on Weighted A*

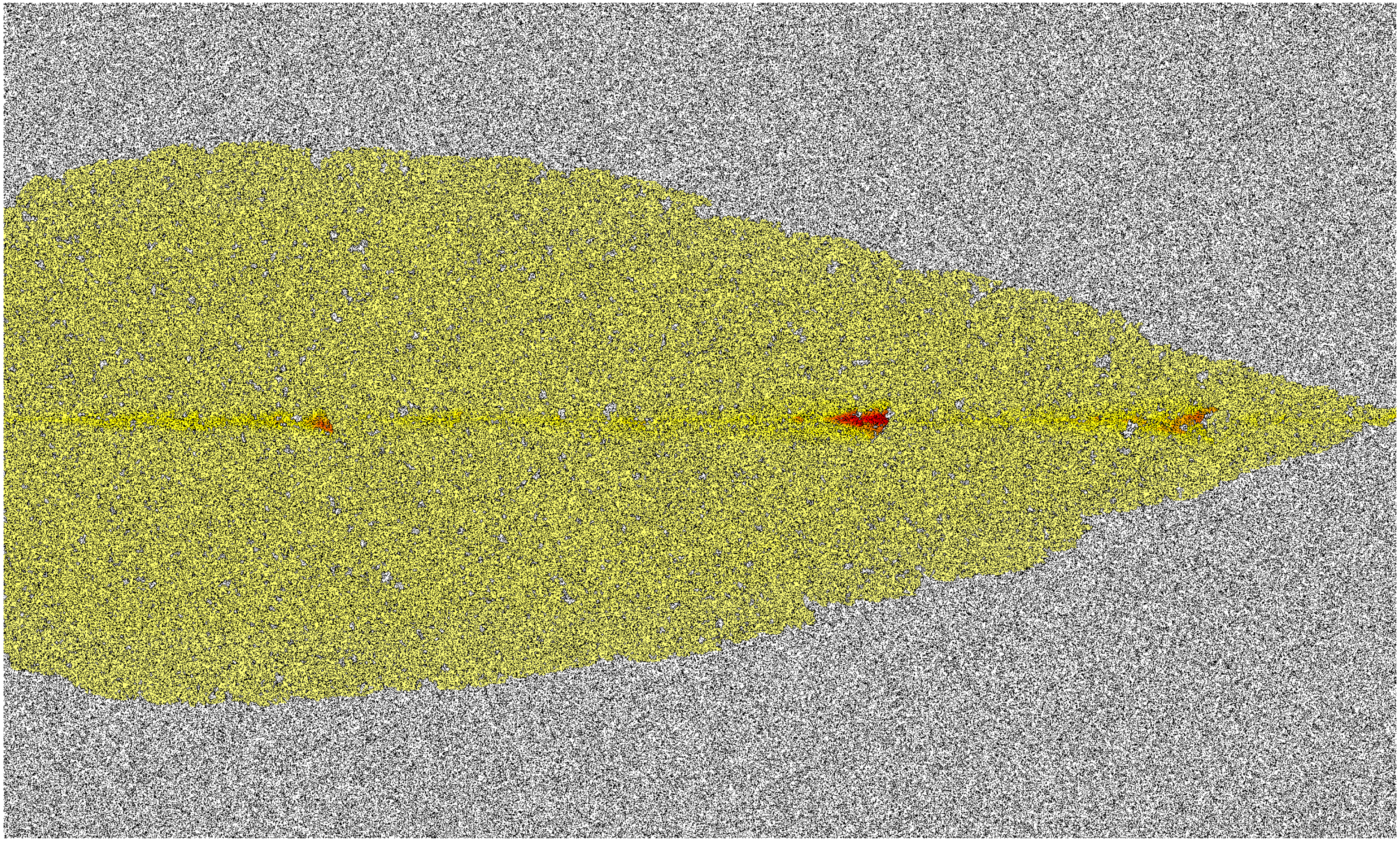
effort of anytime A*



effort of anytime repairing A*



effort of restarting A*



- ARA*, AwA*, RwA*

  describe general frameworks (wA* not important)

  continued – few duplicates, tight lower bound

  repairing – many duplicates, high initial bound

  restarting – low h bias, cheap expansions

- great for unknown deadlines

- for known deadlines, deadline aware search

  (Dionne et al, SoCS-11)

Summary

- suboptimal search — minimize solving time
    - no guarantees on solution quality
        - so use inadmissible heuristics
        - and drop duplicate states
- bounded suboptimal search — balance time and cost
    - bounds on solution quality
    - drop duplicates when possible
    - looser bounds $\neq$ always better performance
- anytime search — unknown deadlines
    - automatically trades time for quality
    - deadline agnostic
    - frameworks can be used with any complete algorithm

Bibliography

- J. E. Dorand and D. Michie,
  "Experiments with the Graph Traverser Program",
  Proceedings of the Royal Society of London. Series A,
  Mathematical and Physical Sciences, 1966.
- Ira Pohl, "Heuristic Search Viewed as Path Finding in a Graph",
  Artificial Intelligence volume 1, 1970.
- Elain Rich and Kevin Knight, "Artificial Intelligence", 1991.
- Sven Koenig and Xiaoxun Sun,
  "Comparing Real-Time and Incremental Heuristic Search for
  Real-Time Situated Agents",
  AAMAS-2008.
- Jordan T. Thayer and Wheeler Ruml,
  "Faster Than Weigthed A*:
  An Optimistic Approach to Bounded Suboptimal Search",
  ICAPS-2008.

Bibliography

- Eric A. Hansen, Shlomo Zilberstein and Victor A. Danilchenko, "Anytime Heuristic Search: First Results", University of Massachusetts, Amherst Technical Repart 97-50, 1997.
- Maxim Likhachev, Geoff Gordon and Sebastian Thrun "ARA*: Anytime A* with Provable Bounds on Sub-Optimality", NIPS-2003.
- Eric A. Hansen and Rong Zhou, "Anytime Heuristic Search", Journal of Artificial Intelligence Research volume 28, 2007.
- Silvia Richter, Jordan T. Thayer and Wheeler Ruml, "The Joy of Forgetting: Faster Anytime Search via Restarting", ICAPS-2010.
- Chris Wilt, Jordan T. Thayer, and Wheeler Ruml, "A Comparison of Greedy Search Algorithms", SoCS-2010.