copyright GBL rev s 4/20

CSCE686 Advanced Algorithm Design

***3.4 Global Search - Breath First Search***

**Breath-first search (bfs)** is a graph search technique that considers all pos- sibilities at each level of the search. bfs expands the search graph frontier between generated nodes/states/vertices and un-generated nodes uniformly across the breath of the frontier. The metric related to uniformly is usually the number of edges from the root node of the search graph resulting in ex- plorations at equal search graph depths. Observe that a breath-first search generates a breath-first tree as it proceeds to explore the search graph. Note that P-Time complexity problems can be solved using breath-first search as well as depth-first search (examples?).

Best-first search (bfs,bf) explores/generates an OR search graph by expanding the most promising node (partial solution-state) chosen according to a specified first-order logic rule. Pearl describes best-first search as estimating the promise of node *n* by a "heuristic evaluation function” *f(n)* which, in general, may depend on the description of *n*, the description of the goal, the information gathered by the search up to that point, and most important, on any extra knowledge about the problem domaini.e., selection from a set of generated candidates to add to the partial solution state. Other authors have used "best-first search" to refer specifically to a search with a heuristic that attempts to predict how close the end of a path is to a solution, so that paths which are judged to be closer to a solution are extended first; a Greedy bfs. A bfs example is A\* with an added cost and an estimation function to a solution. The star refers to always finding the optimal solution. Note that efficient selection of the current best candidate for extension can be implemented using a priority queue. Others?

The abstract data structure relating to nodes to explore on the “frontier” can be reflected in a priority list of states with the most *merit*. If the generation of the *frontier* and selection of *frontier states* to expand is based on merit instead of layers of equal depth, a breath- first *best-ﬁrst* search is defined. We use the shorthand *bfs* to refer to this more general breath-first or best-first search algorithm domain. Note that a depth-first search, (dfs), uses the best-first frontier data structure as Stack!

Question: How to integrate problem domain heuristics into breath first search? Designing algorithm for what NPC application purpose?

Again, we need the time and space complexity of the problem domain (PD) and later in the design process the complexity of the algorithm domain (AD).

**Read: Pearl:** Chapters 2 (algorithms) & 3 (theory); **Talbi**, Sec. 2.3-2.6

**References:**

***CLR*** *pp 469-477; M&F; Chap 4;* ***N&N*** *pp 222-244;* ***B&B*** *pp 302-305;* ***M&L*** *pp 105-108;* ***Skiena*** *pp 90, 270, 276, 280;* ***R&N****, Part II*

***3.4.1*** ***Global Search - breath-first, best-first search (gs-bfs)***

- Global search bfs uses a breath ﬁrst search approach to implicitly or explicitly search the entire search space in order to obtain an optimal (or “satisﬁzing”) solution. With a breath ﬁrst best ﬁrst search, a frontier of partial solutions is progressively generated and the next candidate selected has the highest merit on the frontier; i.e., best ﬁrst. Here an “AND ” (divide-and-conquer) or “OR” structured search graph is assumed. The standard search functions from the greedy selection are used again along with delayed termination: (observe the similarities and diﬀerences with the global depth ﬁrst search; gs\_dfs with queue and gs\_dfs\_bt with stack)

•

• basic search functions/operations - set of candidates, next state generator, feasibility, selection, solution, objective (recursive state weighting function for evaluating partial solutions - search space state). Use **heuristic** (informed search) to direct search to promising search graph paths.

• delay termination - the entire state space is searched explicitly or im- plicitly to ﬁnd optimal solution(s) even though an non-optimal so- lution is found initially. Thus, a satisﬁcing criteria would result in accepting the ﬁrst solution found.

- the following reﬂects the algorithm domain speciﬁcations from design to function speciﬁcation for the global search bfs algorithm (observe that the requirements speciﬁcation is essentially the same as for general global search):

***algorithm domain requirements speciﬁcation form:***

*•*

*• name: Global-Search Breath First Search (Di, Do); gs-bfs*

*• domains: Di is set-of-candidates,*

*Do are sets of solutions (solution space of subsets)*

*• operations:*

*I(x); x in Di; x is a possible candidate from input set*

*O(x,z); x in Di, z in Do; z is an optimal (maximal) solution or set of*

*optimal solutions (or satisﬁzing solutions)*

***algorithm domain design speciﬁcation form:***

*• name: Global-Search-bfs (Di, Do); gs-bfs*

*• domains: Di is set-of-candidates, Do are the sets of solutions, Dp is set of partial solutions of “generated nodes”- setofsets, boolean*

*• imports: ADT set, list, queue, real/integer/character,*

*“other ADTs (types, sorts, ...) imported depend upon integration with*

*problem domain application”*

*• operations:*1

*I(x); x in Di*

*O(x,z); x in Di, z in Do;*

*“condition on z being an optimal (satisﬁzing) solution”*

*I’(x,y); x in Di, y in Dp; “condition on y being a partial solution in Dp*

*(usually by the feasibility check {implicit or explicit} and set union)”*

*Dp is the “open” list; Dc is the ”closed” list for examples from Pearl*

**–**

**–** *deﬁne state*

**– *next-state-generator***

*i)* ***selection*** *of a partial solution y in Dp based upon its superi- ority and put in Dc and delete from Dp*

*“based upon an ordered set reﬂecting explicitly/implicitly the ob- jective function and reﬂected in the merit of partial solutions in Dp; a best-ﬁrst search – a search guiding* ***heuristic*** *on OPEN list and CLOSED list ”*

*ii)* ***Generation*** *of* all *next states xj of y*

**– *feasibility*** *(xj , y) − > boolean [if true union (xj , y) and put result*

*in Dp]*

**– *solution*** *(y) − > boolean; z = y; delay termination and ﬁnd all*

*“optimal” solutions (if satisﬁzing accept one/ﬁrst solution)*

**–** *objective solution(Dp) − > “ordered set/well founded set over Dp”*

**– *heuristics***  *come from problem domain insight:*

***-- Attempt use PD next state generator to reduce set-of candidates* *ASAP (on the bfs frontier – OPEN list)***

***-- Attempt to generate a combination once and only once   
 in combinatorial problem domain (if not , use CLOSED list)***

***-- Attempt to generate early pruning (bounding) condition simple solution check***

1 regarding Pearl’s General Best First (GBF) search algorithm for ”AND” graphs, a solution base y is G*′′* , D*p* is subgraph

G*′*

**–** *algorithmic progress: breath ﬁrst search (best ﬁrst search) on search graph*

*•* ***axioms:*** *(later for testing? Operator on Operator predicate calculus forms)*

*•* ***comment:*** *need to also explicitly control equivalent dual states/nodes and progress with ”best” path (or graph) or subgraph (and or graph)*

***algorithm domain intermediate speciﬁcation form: (iterative)***

*• integrate heuristics into ﬁve basic search functions as appropriate*

*(Creative!))*

*• move towards function speciﬁcation with selection of data structures*

*+ algorithms = programs (eﬃciency)*

*• show and discuss explicit algorithm design decisions as gs-bfs design evolves (Creative!)*

*• indicate speciﬁc mapping of search algorithm data structure notation to functional data structure notation (Creative!)*

***algorithm domain function speciﬁcation form: (iterative)***

*•*

*•* ***function*** *global-search-bfs-iterative (Di) sets in Do*

*• Initial condition: clear(set Dp); deﬁne initial x in Di and associated initial state s*0 *in Dp; si is search state/node.*

*Dp is the “open” list; Dc is the ”closed” list*

*• body*

*while Di for each state not exhausted do gs-dfs-loop: “all nodes/states expanded”*

**–**

**– *next-state-generator***

*i)* ***selection***  *of a partial solution s(y) in Dp based upon a merit function fy .*

*“use priority queue for open list Dp. and a hash table for the closed list “*

*ii) “****Generation****” of all next states of s(y), delete s(y) from Dp*

**–** *if* ***feasibility*** *(s(xj , y) then s=union(xj , y) and union (s, Dp)*

**–** *if* ***solution(s)*** *then save z = s, z in Do, “current optimal solution”*

**–** *end gs-dfs while loop*

*if Do not empty set return all z’s (solutions);‘ “also could output each solution as generated”*

*additional reﬁnement of algorithm domain function speciﬁcation: (iterative, recursive, or objective oriented design into HOL;*

*eﬃcient data structures-algorithms developed)*

***example:*** *Pearl (p 48): iterative; uses explicit open and closed lists;*

*keeping track of parent states (nodes) for output data; reﬂect a tree search*

*- only one path through each node (state) for OR search graph.*

***3.4.2 Comments on gs-bfs* “**Understanding and Ability to Use”

* gs-bfs results in graph search (or tree search depending upon applica- tion). If the search graph is ﬁnite, then breath-ﬁrst search is guar- anted to ﬁnd the shallowest solution possible. As compared to dfs, more memory is required in order to maintain the frontier priority lisjumping from frontier node to frontier node along with the associated memory requirements makes this search technique diﬃcult for human reasoning as compared to dfs!
* various authors (Pearl, Nilsson) deﬁne speciﬁc data structures to hold partial solutions y in Dp, namely a LIFO Open list to store nodes on the frontier that have not been expanded and a LIFO Closed list to store nodes on the frontier that have been expanded. Using these data structures a more reﬁned gs-bfs function speciﬁcation can be deﬁned. In these cases the Open and Close list structures are in reality priority queues and are explicitly maintained in the implementation structure with subroutine calls generated for next-state generator, feasibility, solution and objective functions. Here x is usually a record format with the resulting lists of records in each y in Dp. The Open and Closed list structures can also be directly employed in the gs-dfs algorithm design (how? as a FIFO stack - see Pearl, B&B). Note that more eﬃcient algorithms may be appropriate in speciﬁc applications; i.e., problem domains.
* variations of gs-bfs for “OR” structures depend upon the form of the objective function and its impact of the updating of each states evaluation function. Examples include best ﬁrst (BF∗), additive cost functions (A∗, Pearl, p64, M&F, p105), order preserving recursive weight functions (Z∗). In the case of the previous two bfs techniques, the form of the evaluation function is related to the concept of recursive weighting. This constraint supports the algorithm’s ability to ﬁnd thoptimal solution (see Pearl, p 62). Find “reasonable” solution in limited time.
* ***The Heuristic estimates value of a node:*** *promise of a node, difficulty of solving the subproblem , quality of solution represented by node, the amount of information gained. Then f(n), the heuristic evaluation function****.*** *depends on n, the goal, the search so far (partial solultion), and the domain.*
* ***Properties of Heuristics:*** *admissibility, monotonicity, dominance, accuracy (Pearl).*
* ***Properties of Algorithms:*** *completeness, admissibility, dominance, Optimality - dominates all members of a class (Pearl, page 75), complexity (time, space)*

***3.4.3 Best-First Z\* algorithm (Pearl) with tests for duplicate nodes.***

1. Put the start node s on a list called OPEN of unexpanded nodes.

2. If OPEN is empty exit with failure; no solutions exists.

3. Remove/select the first OPEN node n at which f is minimum (break ties arbitrarily), and place it on a list called CLOSED to be used for expanded nodes.

4. Expand node n, generating all it’s successors with pointers back to n.

5. If any of n’s successors is a goal node, exit successfully with the solution obtained by tracing the path along the pointers from the goal back to s.

6. For every successor n’ on n: (feasibility check?)  
a. Calculate f (n’). (satisfy R.W.C.)  
b. if n’ was neither on OPEN nor on CLOSED, add it to OPEN. Attach a  
 pointer from n’ back to n. Assign the newly computed f(n’) to node n’.  
c. if n’ already resided on OPEN or CLOSED, compare the newly  
 computed f(n’) with the value previously assigned to n’. If the old   
 value is lower, discard the newly generated node. If the new value is lower, substitute it for the old (n’ now points back to n instead of to its previous predecessor). If the matching node n’ resided on CLOSED, move it back to OPEN.

1. Go to step 2.

***“Embed standard search elements in Z\* pseudo code”***

***3.4.4*** [***A\****](https://en.wikipedia.org/wiki/A*_search_algorithm) ***- an uniform cost search combined with informed best-first search***

A\* uses an additive recursive cost function (assuming a minimization problem): where f(n) = g(n) + h(n), g(n) is the cost of the path to node n and h(n) is an heuristic that estimates the distance (cost) to a goal solution fro node n ()partial solution). If h(n) always underestimate this distance, h(n) is **admissible** by definition. Then f(n) never overestimates the actual cost through the node n. Thus, A\* if admissible, will always find a least cost path to the solution. Because of the OPEN list of frontier nodes, the A\* search process will always visit a optimal path on the search frontier. **A\* is complete** (terminates and finds optimal solutions) as long as the maximum branching factor, b, is finite and every node movement generates a positive additive cost possibly resulting in a maximum path length, m. Of course the time complexity and space complexity can still remain exponentials (bm); if not informed, complexity same as bfs! Improving performance quality (efficiency) depends upon the domination of new heuristics over the old.

What if h is equal to h\* or zero? Heuristic are generated from relaxed problems because “Hypothesis: relaxed problems are easier to solve” and improve!

Note that A heuristic function h *dominates* h’ (more informed than h) if both are admissible and for every node n, h(n) is greater than h’(n).

An A\* search with a *dominating* heuristic function h has the property that any node it expands is also expanded by A\* with the dominated h’ (Hart, Nillson and Raphale, 1968)

***Another A\* functional view: “add standard search elements?”***

***(What are OPEN & CLOSED lists data structures?)***

Function A\*(initial, Expand, Goal, Cost, Heuristic)

q <- New-Priority-Queue()

Insert(initial, q, Heuristic(initial))

**while** q is not empty

**do** current <- Extract-Min(q)

**if** Goal(current) then **return** solution

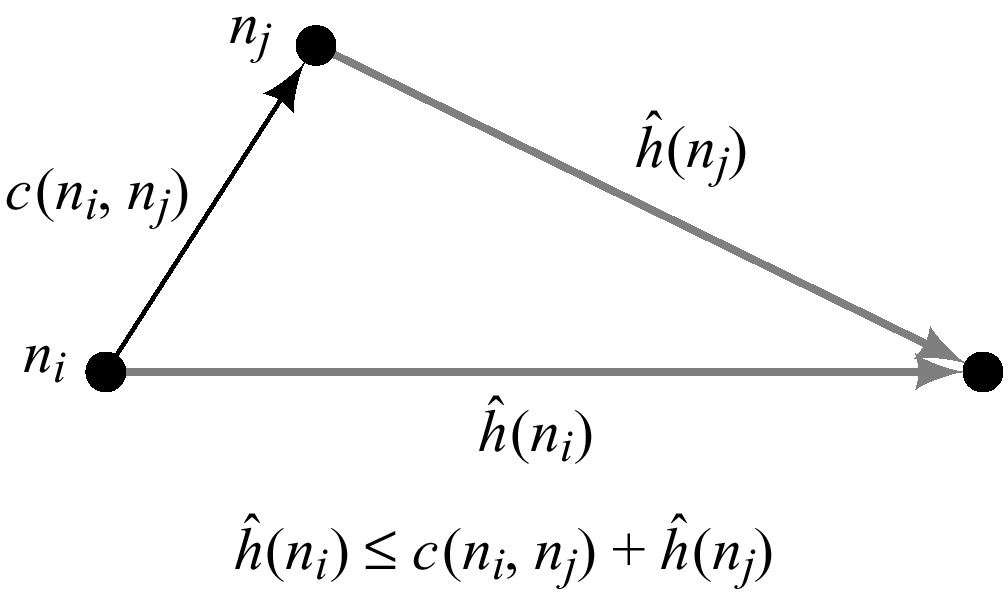
**for** each next in Expand(current)

**do** Insert(next, q, Cost(next) + Heuristic(next))

return failure

***If in A\*, h(n) is monotone/consistent: (see Pearl, Chapter 3)***

* If in the search graph the heuristic function satisfies triangle inequality for every n and its child node n’: h^(ni) is less or equal h^(nj) + c(ni,nj)
  + when h is monotone, the f values of nodes expanded by A\* are never decreasing.
* When A\* selected n for expansion it already found the shortest path to it.
* When h is monotone every node is expanded once (if check for duplicates).
* **If h is consistent, then A\* largely dominates every admissible deterministic search algorithm having access to the same h**! (Dechter and Pearl, 1983) – **A\* is optimal?** **A\* is optimal efficient?**
* But what if h is only admissible?
* In general, monotone heuristics maybe difficult to find (check!)

******

***h(n) heuristics for some problem examples***: (admissible?; monotonic?)

* **8-puzzle Problem (**[**http://www.brian-borowski.com/Software/Puzzle/**](http://www.brian-borowski.com/Software/Puzzle/) **)**
  + W(n): number of misplaced tiles
  + Manhatten distance, …
* **8- queen Problem on Chess Board (**[**http://www.cosc.canterbury.ac.nz/mukundan/dsal/NQP.html**](http://www.cosc.canterbury.ac.nz/mukundan/dsal/NQP.html) **)** 
  + Number of future feasible slots
  + Min number of feasible slots in a row,…
* **Travelling salesperson (**[**http://stackoverflow.com/questions/7159259/optimized-tsp-algorithms**](http://stackoverflow.com/questions/7159259/optimized-tsp-algorithms) **)**
  + Minimum spanning tree
  + Minimum assignment problem, …
* **Set-Covering/Hitting-set** 
  + Sum of min cost of items not yet covered
* **Generate/Find monotonic h(n)s for other NPC or PSPACE problems**

***3.4.5 Some Thoughts on A\* (or any BF\* algorithm)***

**Process:** As A\* traverses the graph, it follows a path of the lowest *known* cost, keeping a sorted priority queue (OPEN LIST) of alternate path segments along the way. The cost is f = g + h, where g is the known cost of the current partial solution and h is an estimate to a solution. If, at any point, a segment of the path being traversed has a higher cost than another encountered path segment, it abandons the higher-cost path segment and traverses the lower-cost path segment instead. This process continues until the goal is reached. The CLOSED LIST contains all those path nodes previously visited and expanded. Like all [informed search algorithms](http://en.wikipedia.org/wiki/Informed_search_algorithm), it first searches the routes that *appear* to be most likely to lead towards the goal. What sets A\* apart from a [greedy](http://en.wikipedia.org/wiki/Greedy_algorithm) [best-first search](http://en.wikipedia.org/wiki/Best-first_search) is that it also takes the distance already traveled into account; the g(n) part of the heuristic is the cost from the starting point, not simply the local cost from the previously expanded node.

The lower f(n) = g(n) + h(n) for a given node, the higher its priority. At each step of the algorithm, the node with the lowest f(n) value is removed from the queue, the h and f values of its neighbors are updated accordingly, and these neighbors are added to the queue. The algorithm continues until a goal node has a lower f value than any node in the queue (or until the queue is empty). (Goal nodes may be passed over multiple times if there remain other nodes with lower f values, as they may lead to a shorter path to a goal.) The f value of the goal is then the length of the shortest path, since h at the goal is zero in an admissible heuristic. If the actual shortest path is desired, the algorithm may also update each neighbor with its immediate predecessor in the best path found so far; this information can then be used to reconstruct the path by working backwards from the goal node. Additionally, if the heuristic is *monotonic* or *consistent* a *closed set* of nodes already traversed may be used to make the search more efficient.

**Incremental heuristic search:** such algorithms combine both incremental and heuristic search to speed up searches of sequences of similar search problems, which is important in domains that are only incompletely known or change dynamically.Incremental search has been studied at least since the late 1960s. Incremental search algorithms reuse information from previous searches to speed up the current search and solve search problems potentially much faster than solving them repeatedly from scratch.Similarly, heuristic search has been studied at least since the late 1960s. Heuristic search algorithms, often based on [A\*](http://en.wikipedia.org/wiki/A*), use heuristic knowledge in the form of approximations of the goal distances to focus the search and solve search problems potentially much faster than uninformed search algorithms.The resulting search problems, sometimes called dynamic path planning problems, are graph search problems where paths have to be found repeatedly because the [topology](http://en.wikipedia.org/wiki/Topology) of the graph, its edge costs, the start vertex or the goal vertices change over time. So far, three main classes of incremental heuristic search algorithms have been developed:

* The first class restarts A\* at the point where its current search deviates from the previous one (example: Fringe Saving A\*).
* The second class updates the h-values from the previous search during the current search to make them more informed (example: Generalized Adaptive A\* )
* The third class updates the g-values from the previous search during the current search to correct them when necessary, which can be interpreted as transforming the A\* search tree from the previous search into the A\* search tree for the current search (examples: Lifelong Planning A\*, [D\*](http://en.wikipedia.org/wiki/D*" \o "D*), D\* Lite).

All three classes of incremental heuristic search algorithms are different from other replanning algorithms, such as planning by analogy, in that *their plan quality does not deteriorate with the number of replanning episodes. Many robot applications.*

***A\* Educational Objectives in CSCE686:***

* Thoroughly understand A\* process and algorithm design
* Be able to trace simple examples of A\* execution.
  + Understand “admissibility” of heuristics, proof of completeness, and guaranteed optimality of solution path.
* Be able to criticize best first search design approaches given a problem domain (SEARCH landscape)

***3.4.6 An Iterative Deepening A\* Search Algorithm (IDA\*):***

**IDA\*** is a variant of the A\* search algorithm which uses iterative deepening to keep the memory usage lower than in A\*. It is an informed search based on ID(DFS): ***“Embed the standard search constructs in the functional form”***

function IDA\*(*initial*, Expand, Goal, Cost, Heuristic)

*limit* <- Heuristic(*initial*)

**loop**

**do** *result, limit* <- Contour (*initial, limit*)

**if** *result* then **return** *result*

**if** *limit = infinity* then r**eturn** failure

function Contour(*current, limit*)

*cost* <- Cost(*current*) + Heuristic(*current*)

**if** *limit < cost* then return null, *cost*

**if** Goal(*current*) then return solution, *cost*

*new-limit <- infinity*

**for** each *next* in Expand(*current*)

**do** *result, cost* <- Contour(*next, limit*)

**if** *result* then **return** solution, *cost*

*new-limit* <- min(*new-limit, cost*)

**return** failure, *new-limit*

***Computational Efficiency of A\* and IDA\****

* **Assume** tree-structured graph state space (b = branching factor, d = goal depth,

single goal state, each edge costs 1, and maximum error of ǫ)

* Any partial solution state n is more than ǫ/2 off of solution path has

f(n) = g(n) + h(n) > f\*.

* All states n on solution path have f(n) = g(n) + h(n) <= f\*.
* A\* and IDA\* visit O(dbǫ/2) states.
* A\* uses O(dbǫ/2) memory. IDA\* uses O(db). Memory to store states

***ID(DFS): Iterative Deepening Depth-First Search***

An iterative deepening search operates like a depth-first search, except slightly more constrained--there is a maximum depth which defines how many levels deep the algorithm can look for solutions. A **node** at the maximum level of depth is treated as **terminal**, even if it would ordinarily have successor nodes. If a search "fails," then the maximum level is increased by one and the process repeats. The value for the maximum depth is initially set at 0 (i.e., only the initial node).

**IDDFS**(root, goal) ***“Add standard search elements to this pseudo-code”***

{

depth = 0

repeat

{

result = **DLS**(root, goal, depth)

if (result is a solution)

return result

depth = depth + 1

}

}

**DLS**(node, goal, depth) ***“recursive”***

{

if (depth == 0 and node == goal)

return node

else if (depth > 0)

for each child in expand(node)

**DLS**(child, goal, depth-1)

else

return null

}

An interesting observation is that the nodes in this search are first checked in the same order they would be checked in a breadth-first-search; however, since nodes are deleted as the search progresses, much less memory is used at any given time.

The drawback to the iterative deepening search is that is can be painfully redundant, rechecking every node it has already checked with each new iteration. The algorithm can be enhanced to remember what nodes it has already seen, but this sacrifices most of the memory efficiency that made the algorithm worthwhile in the first place, and nodes at the maximum level for one iteration still need to be re-accessed and expanded in the following iteration. Still, when memory is at a premium, iterative deepening is preferable to a plain depth-first search when there is danger of looping or when the most efficient solution is desired.

*Comment: ID vs. breath-first search: the number of search states (nodes – partial solutions) visited by ID is at most bd+2 /(b-1)(b-1): again b = branching factor and there is a goal state within the distance d of the initial state. ID requires O(bd) memory while breath-first requires memory for O(bd) states! Again, the assumption is that the search graph is a tree. . What if there are repeating partial solution states!*

Comment: [*iterative lengthening search*](http://en.wikipedia.org/w/index.php?title=Iterative_lengthening_search&action=edit&redlink=1) *(ILS) uses increasing path-cost limits instead of depth-limits. It expands nodes in the order of increasing path cost; therefore the first goal it encounters is the one with the cheapest path cost. But ILS incurs substantial memory overhead that make it less useful than ID(DFS).*

*Comment: Observe that local heuristic search techniques to consider with gs\_bfs: (mementic):**beam search?*

* *Hill climbing , gradient descent, dfs, with random applications (Monte-Carlo)*
* *Simulated annealing, Tabu search*
* *Bio-inspired* 
  + *genetic algorithms, evolutionary algorithms*
  + *ant colonies, bees*
  + *neural nets,*
  + *artificial immune systems*
  + *…*

***Need some animated visualizations of gs\_bfs search algorithms!***

A\* animation \_ <http://www.youtube.com/watch?v=qo7B44zlYdQ>

<http://www.rci.rutgers.edu/~cfs/472_html/AI_SEARCH/SearchAnimations.html>

<http://commons.wikimedia.org/wiki/File:Astar_progress_animation.gif>

SMA\*: <http://dtai.cs.kuleuven.be/education/athens2011/Exercises/1-search/SMAstar.pdf>

Beam Search: (<http://jhave.org/algorithms/graphs/beamsearch/beamsearch.shtml> )

Linear/Binary search: (<http://www.cosc.canterbury.ac.nz/mukundan/dsal/appldsal.html> )

Breadth-first vs. depth first search (MST): <http://www.csse.monash.edu.au/~dwa/MELB/Graph.html>

Random BFS: <http://eprints.fri.uni-lj.si/1282/> “download”

General: <http://aima.eecs.berkeley.edu/index.html> **“AFIT AI course text”**

Sorting (great animation): <http://www.sorting-algorithms.com/>

***3.4.7 “When to use A\* or one of its variations?”***

**Basic Algorithmic Properties: (Pearl)**

* Completeness – algorithm terminates with a solution if it exists
* Admissible – algorithm guaranteed to find optimal solution if it exists
* Dominance – dominate algorithm expands fewer nodes
* Optimality – optimal algorithm dominates of members of “class”

**Admissible Heuristics Definition:**

* + Define h\*(n) heuristic as the true minimal cost to goal from n.
  + A heuristic h estimate is admissible if h(n) <= h\*(n) for all states n.
  + An admissible heuristic is guaranteed never to overestimate cost to goal.
  + An admissible heuristic is optimistic.
  + A\* with Admissible Heuristic Guarantees Optimal Path
  + Simple proof – see Pearl, Chapter 3
  + What if h(n) is monotonic? Faster?

**Is A\* Guaranteed to Terminate? Complete?**

* There are finitely many acyclic paths in the search tree. Why?
* A\* only ever considers acyclic paths.
* On each iteration of A\* a new acyclic path is generated because:
  + When a node is added the first time, a new path exists.
  + When a node is “promoted”, a new path to that node exists.
  + It must be new because it’s shorter – less cost.
  + So the very most work it could do is to look at *every* acyclic path in the graph – a finite number, but generally quite large
* A\* terminates. (with delayed termination finds optimal solution(s), Why?)

**Is A\* an Admissible Algorithm?**

* + If it has an admissible cost function.
  + R.W.C. ? ; f(n) = g(n) + h(n) where h(n) is estimate to solution

**A\* : The “Dark Side”**

* A\* can use lots of memory due to # of search nodes (states) expanded
* In principle it is of O(number of most possible problem domain states)
* Thus Order of complexity is space complexity but also time complexity
* For “really big” search spaces, A\* will run out of memory before finding sol
* How to store nodes on search frontier? Priority queue, … Why? (DS+CS=P)
* Use iterative deepening, problem domain heuristics; memory limiting algorithms – SMA\*, RBFS; or use DFS instead of BFS?
* SMA\* -- Simplified Memory-Bound A\*-- avoids repeated paths, …
* RBFS – Recursive Best-First Search – linear-space

## 

## *3.4.7.1 Variants of A\** - D\*, D\* Lite, Field D\*, ID, IDA\*, Fringe, Fringe saving A\*, Generalized Adaptive A\*, Lifelong Planning A\*, RTA\*, LRTA\*, SMA\*, Theta\*, A\* Beam Search

**Iterative Deepening** – ID (visualize search graph example)

- Depth limited search vs. iterative deepening search

- Only search to a specific search tree depth per some spatial or temporal epoch

or increase depth once all nodes/states have research that depth

- Assuming optimum solution exists high in search graph

- For Global BFS (A\*, Z\*, …) drives more search breath

- For Global DFS/BT -- drives BT process

- Finds optimum solution? Yes, but more search tree nodes can be expanded

- For SCP or MIS - large dimensions, what would be “good” tree depth to start?

**Comparing Iterative Deepening (ID) compared with A\***

* *From Russell and Norvig, Page 107, Fig 4.8*
* For 8-puzzle, average number ofstates expanded over 100

randomly chosen problems in which optimal path length:

4 steps …8 steps …12 steps

A\* using “Sum of Manhattan distances” as the heuristic 12.. 25.. 73

A\* using “number of misplaced tiles” as the heuristic 13.. 39.. 227

Iterative Deepening (heuristic = ?) 112.. 6,300..3,600, 000

**IDA\* with Memory Bounded Search (SM)**

• Iterative deepening A\*. Actually, quite different from A\*. Assuming costs integer.

1. Do loop-avoiding DFS, not expanding any node with

f(n) > 0. Did we find a goal? If so, stop.

2. Do loop-avoiding DFS, not expanding any node with

f(n) > 1. Did we find a goal? If so, stop.

3. Do loop-avoiding DFS, not expanding any node with

f(n) > 2. Did we find a goal? If so, stop.

4. Do loop-avoiding DFS, not expanding any node with

f(n) > 3. Did we find a goal? If so, stop.

…keep doing this, increasing the f(n) threshold by 1 each

time, until stop desired.

• IDA\* is Complete, Guaranteed to find optimal. More costly than A\*, BFS in general

“IDA\* in parallel?”

**A\* Beam Search** (visualize a search graph example)

* restrict BFS to a limited # of search paths in search tree via OPEN
* an approximation approach; may not find optimum solution down these paths
* need good selection criteria from set of candidates as in any BFS
* for SCP, consider ratio of cost per r item covered? And # of branches << # of sets?
* Beam search in parallel? Many different focus directions of search
* Local Beam Search
  + - Keep track of *k* states instead of one – Initially *k* random states
    - Next: determine all successors of *k* state
    - If any of successors is goal → finished Else select *k*  best from successors and repeat.
    - Major difference with random-restart search - Information is shared among *k* search threads.
* Can suffer from lack of diversity.- Stochastic variant: choose k successors proportionally to state success

### Real-Time A\* Search (RTA\*)

Simply repeating optimum search for each move ignores information from previous searches and results in infinite loops. In addition, since actions are committed based on limited information often the best move, may be due to undo the previous move. The principle of rationally is that *backtracking* should occur when the estimated cost of continuing the current path exceeds the cost of going back to a previous state plus the estimated cost of reaching the goal from the state Real-time A\* (RTA\*) implements the policy in constant time per move on a tree.

For each move, the f(n) = g(n) + h(n) value of each neighbour of the current state is computed where <n()> is now the cost of the edge from the current state to the neighbour, instead of from the initial state. The problem solver moves to the neighbour with the minimum f(n) value, and stores with the previous state the best f(n) value among the remaining neighbors. This represents the h(n) value of the previous state from the perspective of the new current state. This is repeated until a goal is reached. To determine the h(n) value of a previously visited state, the stored value is used, while for a new state the [heuristic evaluator](http://intelligence.worldofcomputing.net/ai-search/heuristic-evaluation-function.html) is called. Note that the heuristic evaluator may employ minimum lookahead search with [branch-and-bound](http://intelligence.worldofcomputing.net/ai-search/depth-first-branch-and-bound.html) as well.

In a finite problem space in which there exists a path to a goal from every state, RTA\* is on a tree, RTA\* makes locally-optimal decisions given the information it has seen so far.

**A\* compared to Uniform-Cost search**

If all the edges in the search graph do not have the same cost then breadth-first search generalizes to uniform-cost search. Instead of expanding nodes in order of their depth from the root, uniform-cost search expands nodes in order of their cost from the root. At each step, the next step n to be expanded is one whose cost g(n) is lowest where g(n) is the sum of the edge costs from the root to node n. The nodes are stored in a priority queue. This algorithm is also known as Dijkstra’s single-source shortest algorithm.

Whenever a node is chosen for expansion by uniform cost search, a lowest-cost path to that node has been found. The worst case time complexity of uniform-cost search is O(bc/m), where c is the cost of an optimal solution and m is the minimum edge cost. Unfortunately, it also suggests the same memory limitation as breadth-first search. Uniform-cost search is uninformed search: it doesn't use any domain knowledge. It expands the least cost node, and it does so in every direction because no information about the goal is provided. It can be viewed as a function f(n) = g(n) where g(n) is a path cost ("path cost" itself is a function that assigns a numeric cost to a path with respect to performance measure, e.g. distance in kilometers, or number of moves etc.). It simply is a cost to reach node n. **Uniform cost** is an uninformed search algorithm when **Best First** and **A\*** search algorithms are informed search algorithms, f(n)= g(n) + h(n). Informed means that it uses a heuristic function h(n) for deciding the expanding node.

**4.8.8 Additional Comments:**

**Example A\* Large Problems** (limited memory use?)

How to use A\* for solving MIS Problem

* What is the one main impact of (set of candidates)
* Can Q+ and Q- in the MIS DFS/BT algorithm be used?
* Minimize number of combinations/states checked; implies faster tree se
* How can OPEN and CLOSED lists be explicitly be used in DFS and DFS/BT?
* Only one child state opened at a time, no closed list possible?

How to use use A\* for SCP? Another Eval2? -- -SCP algorithm A\*?

- Can the Tableau in SCP algorithm be used?

- Need “good” data structure evolution and heuristics from NPC domain to limit search

- What if the A\* check for duplicate states is not executed?

- Paths with duplicate states may exist (requires more memory)

- Optimum found? Yes

How to use A\* for any NPC problem? Use CSCE686 A\* search algorithm design template?

**Other Questions:**

* Does Z\* terminate? Find *optimum* solution or *optimal* solution? Is the search algorithm complete?
* What if the Closed list is never employed in code?
* If goal nodes are known could do bidirectional search from top and bottom of search tree in parallel (practical?)
* Complexity performance of uninformed search techniques (Figure 3.17 in R&N, Reference 3) What does it tell us? Move towards O(p(n)PD complexity)

**A\* History:**

In 1964 Nils Nilsson invented a heuristic based approach to increase the speed of [Dijkstra's algorithm](http://en.wikipedia.org/wiki/Dijkstra%27s_algorithm). This algorithm was called A1. In 1967 Bertram Raphael made dramatic improvements upon this algorithm, but failed to show optimality. He called this algorithm A2. Then in 1968 Peter E. Hart introduced an argument that proved A2 was optimal when using a consistent heuristic with only minor changes. His proof of the algorithm also included a section that showed that the new A2 algorithm was the best algorithm possible given the conditions on g and h . He thus named the new algorithm in [Kleene star](http://en.wikipedia.org/wiki/Kleene_star) syntax to be the algorithm that starts with A and includes all possible version numbers. It should be noted however, that many researchers used the f = g + h BFS formulation for solving specific optimization problems years prior to 1964. Although, OR researchers used the additive cost function decades earlier on specific problems!

**Some A\* Web sites: (incompete! A\* pseudo code and in various languages)**

<http://www.briangrinstead.com/2.blog/astar-search-algorithm-in-javascript> A\* Demo/Java!

<https://gist.github.com/jamiees2/5531924> “A\* in Python”

<https://codereview.stackexchange.com/questions/97834/a-algorithm-in-c> “A\* in C++”  
  
<https://www.bedroomlan.org/projects/libastar/> “A\* in C”

[http://en.wikipedia.org/wiki/A\*\_search\_algorithm](http://en.wikipedia.org/wiki/A*_search_algorithm) “A\* Search - Wikipedia”  
  
<http://www.heyes-jones.com/astar.html> “A\* Tutorial”

<http://en.wikipedia.org/wiki/Incremental_heuristic_search> “Incremental heuristic search”  
  
<http://en.wikipedia.org/wiki/Ground_truth> “direct information observation”

<http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html> “A\* example”

<http://web.mit.edu/eranki/www/tutorials/search/> “A\* pathfinding”

***3.4.9 References: (incomplete – add some more!)***

1. Nils Nilsson. *Problem Solving Methods in Artificial Intelligence*, McGraw Hill (1971)
2. Judea Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison Wesley (1984) Chapters 2 & 3
3. Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. (2009) Prentice Hall, Chapters 3 & 4
4. Zbigniezw Michalewize and David Fogal, *How to Solve It:* *Modern Heuristics*, Springer (2004) Chapter 4
5. S. Koenig, M. Likhachev, Y. Liu and D. Furcy, “Incremental Heuristic Search in Artificial Intelligence,” Artificial Intelligence Magazine, 25(2), 99-112, 2004.
6. N. Deo and C. Pang, “Shortest-path algorithms: Taxonomy and Annotation,” Networks 14, 275–323, 1984.
7. P. Hart, N. Nilsson and B. Raphael, “A Formal Basis for the Heuristic Determination of Minimum Cost Paths,” IEEE Trans. Syst. Science and Cybernetics, SSC-4(2), 100-107, 1968.
8. X. Sun and S. Koenig, “The Fringe-Saving A\* Search Algorithm - A Feasibility Study,” In Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI), 2391-2397, 2007.
9. X. Sun, S. Koenig and W. Yeoh, “Generalized Adaptive A\*,” In Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS), 469-476, 2008.
10. S. Koenig, M. Likhachev and D. Furcy, “Lifelong Planning A\* ,”Artificial Intelligence Journal, 155, (1-2), 93-146, 2004.
11. A. Stentz, “The Focussed D\* Algorithm for Real-Time Replanning,” Proceedings of the International Joint Conference on Artificial Intelligence, 1652–1659, 1995.
12. S. Koenig and M. Likhachev. “Fast Replanning for Navigation in Unknown Terrain,” Transactions on Robotics, 21, (3), 354-363, 2005.
13. Buatan, et al, Best first Search – You Tube, 2016
14. Christopher Wilt and Wheeler Ruml, "Effective Heuristics for Suboptimal Best-First Search", Volume 57, pages 273-306, 2016
15. Guruji, et al, Time-Efficient A\* Algorithm for Robot Path Planning, ICIAMW 2016, Sci Direct

***3.4.10* *gs-bfs Algorithms for Various NPC problems***

”Using the previous format for dfs search, present the NPC problem for the bfs form reflecting a proper “engineering” algorithmic design:

**- Name, Description (English, formal{LP,ILP,MILP,0-1}, …),**

**- PD complexity,**

**-BFS algorithms and complexity (HEURISTICS),   
  
-Approximation algorithms and complexity,   
  
-Applications, References, Comments”**

***Some specific NPC problem domains:***

*A. MIS (find one?)*

*B. SCP ref;* [*http://www.springerlink.com/content/f5vr85k070n58615/*](http://www.springerlink.com/content/f5vr85k070n58615/)

*C. GCP ref:* [*http://wapedia.mobi/en/Chromaticnumber*](http://wapedia.mobi/en/Chromaticnumber)

*D. TSP ref:* [*http://portal.acm.org/citation.cfm?id=992867*](http://portal.acm.org/citation.cfm?id=992867)

*E. SAT ref:* [*http://dspace.mit.edu/bitstream/handle/1721.1/29242/*](http://dspace.mit.edu/bitstream/handle/1721.1/29242/)

*51589373.pdf ?sequence=1*

*F. 0-1 Knapsack ref:* [*http://ideas.repec.org/p/mse/wpsorb/b04024.html*](http://ideas.repec.org/p/mse/wpsorb/b04024.html)

***”gs-bfs with local search - more on local search using simulated annealing, Tabu search, and bio-inspired search later”***

***3.4.11 Appendices:*** (Other A\* Variations: **D\***, **Fringe** A\*, **GAA\*, SMA\* Theta, , LRTA***\**)  
“Many of these A\* variations can of course be applied to Z\* algorithms”  
Appendix I [3.4.11]

***D\*:*** is any one of the following three related [incremental search algorithms](http://en.wikipedia.org/wiki/Incremental_heuristic_search):

* The ***original D\*,*** by Anthony Stentz, is an informed incremental search algorithm.
* ***Focused D\**** is an informed incremental heuristic search algorithm by Anthony Stentz that combines ideas of [A\*](http://en.wikipedia.org/wiki/A*) and the original D\*. Focused D\* resulted from a further development of the original D\*.
* ***D\* Lite***is an incremental heuristic search algorithm by Koeing and Likhachev that builds on LPA\*, an incremental heuristic search algorithm that combines ideas of [A\*](http://en.wikipedia.org/wiki/A*) and Dynamic SWSF-FP.

The ***D\* algorithm*** works by iteratively selecting a node from the OPEN list and evaluating it. It then propagates the node's changes to all of the neighboring nodes and places them on the OPEN list. This propagation process is termed "expansion". In contrast to A\*, which follows the path from start to finish, D\* begins by *searching backwards from the goal node*. Each expanded node has a backpointer which refers to the next node leading to the target, and each node knows the exact cost to the target. When the start node is the next node to be expanded, the algorithm is done, and the path to the goal can be found by simply following the backpointers.   
 When an obstruction is detected along the intended path, all the points that are affected are again placed on the OPEN list, this time marked RAISE. Before a RAISED node increases in cost, however, the algorithm checks its neighbors and examines whether it can reduce the node's cost. If not, the RAISE state is propagated to all of the nodes' descendants, that is, nodes which have backpointers to it. These nodes are then evaluated, and the RAISE state passed on, forming a wave. When a RAISED node can be reduced, its backpointer is updated, and passes the LOWER state to its neighbors. These waves of RAISE and LOWER states are the heart of D\*.

The ***Focused D\**** is an extension of D\* which uses a heuristic to focus the propagation of RAISE and LOWER toward the robot. In this way, only the states that matter are updated, in the same way that A\* only computes costs for some of the nodes.

**D\* Lite** is not based on the original D\* or Focused D\*, but implements the same behavior. It is simpler to understand and can be implemented in fewer lines of code, hence the name "D\* Lite". Performance-wise, it is as good or better than Focused D\*. D\* Lite is based on ***Lifelong Planning A\****, which was introduced by Amita few years earlier.

All three search algorithms solve the same assumption-based path planning problems, including planning with the freespace assumption; example is where a robot has to navigate to given goal coordinates in unknown terrain. The algorithm makes assumptions about the unknown part of the terrain (for example: that it contains no obstacles) and finds a shortest path from its current coordinates to the goal coordinates under these assumptions. The robot then follows the path. When it observes new map information (such as previously unknown obstacles), it adds the information to its map and, if necessary, replans a new shortest path from its current coordinates to the given goal coordinates. It repeats the process until it reaches the goal coordinates or determines that the goal coordinates cannot be reached. When traversing unknown terrain, new obstacles may be discovered frequently, so this replanning needs to be fast. [Incremental (heuristic) search algorithms](http://en.wikipedia.org/wiki/Incremental_heuristic_search) speed up searches for sequences of similar search problems by using experience with the previous problems to speed up the search for the current one. Assuming the goal coordinates do not change, all three search algorithms are more efficient than repeated A\* searches.

D\* and its variants have been widely used for [mobile robot](http://en.wikipedia.org/wiki/Mobile_robot) and [autonomous vehicle](http://en.wikipedia.org/wiki/Autonomous_vehicle) [navigation](http://en.wikipedia.org/wiki/Navigation_research). Current systems are typically based on D\* Lite rather than the original D\* or Focused D\*. In fact, even Stentz's lab uses D\* Lite rather than D\* in some implementations.Such navigation systems include a prototype system tested on the Mars rovers [Opportunity](http://en.wikipedia.org/wiki/Opportunity_rover) and [Spirit](http://en.wikipedia.org/wiki/Spirit_rover) and the navigation system of the winning entry in the [DARPA Urban Challenge](http://en.wikipedia.org/wiki/DARPA_Urban_Challenge), both developed at [Carnegie Mellon University](http://en.wikipedia.org/wiki/Carnegie_Mellon_University). All D\* algorithms are incremental heuristic search”

Appendix II (More A\* variations) [3.4.11]

***Fringe A\**** is a middle ground between [A\*](http://en.wikipedia.org/wiki/A*) and the iterative deepening A\* variant (IDA\*). This method restarts A\* at the point where its current search deviates from the previous one. If *g*(*x*) is the cost of the search path from the first node to the current, and *h*(*x*) is the [heuristic](http://en.wikipedia.org/wiki/Heuristic_algorithm) estimate of the cost from the current node to the goal, then *ƒ*(*x*) = *g*(*x*) + *h*(*x*), and *h*\* is the actual path cost to the goal. Consider IDA\*, which does a [recursive](http://en.wikipedia.org/wiki/Recursion_(computer_science)) left-to-right [depth-first search](http://en.wikipedia.org/wiki/Depth-first_search) from the root node, stopping the recursion once the goal has been found or the nodes have reached a maximum value *ƒ*. If no goal is found in the first threshold *ƒ*, the threshold is then increased and the algorithm searches again i t iterates on the threshold.  
 There are three major inefficiencies with IDA\*. First, IDA\* repeat states when there are multiple (sometimes non-optimal) paths to a goal node - this is often solved by keeping a cache of visited states. IDA\* thus altered is denoted as memory-enhanced IDA\* (ME-IDA\*), since it uses some storage. Furthermore, IDA\* repeats all previous operations in a search when it iterates in a new threshold, which is necessary to operate with no storage. By storing the leaf nodes of a previous iteration and using them as the starting position of the next, IDA\*'s efficiency is significantly improved (otherwise, in the last iteration it would always have to visit every node in the tree).  
 Fringe search implements these improvements on IDA\* by making use of a data structure that is more or less two [lists](http://en.wikipedia.org/wiki/List_(computing)) to iterate over the frontier or fringe of the search tree. One list *now*, stores the current iteration, and the other list *later* stores the immediate next iteration. So from the root node of the search tree, *now* will be the root and *later* will be empty. Then the algorithm takes one of two actions: If *ƒ*(head) is greater than the current threshold, remove *head* from *now* and append it to the end of *later*; i.e. save *head* for the next iteration. Otherwise, if *ƒ*(head) is less than or equal to the threshold, expand *head* and discard *head*, consider its children, adding them to the beginning of *now*. At the end of an iteration, the threshold is increased, the *later* list becomes the *now* list, and *later* is emptied. “Incremental heuristic search”

An important difference here between fringe and A\* is that the contents of the lists in fringe do not necessarily have to be sorted - a significant gain over A\*, which requires the often expensive maintenance of order in its open list. Unlike A\*, however, fringe will have to visit the same nodes repeatedly, but the cost for each such visit is constant compared to the worst-case logarithmic time of sorting the list in A\*.

Appendix III [3.4.11]

The ***generalize Adaptive A\**** permits action costs that can increase and decrease. This class updates the h-values from the previous search during the current search to make them more informed. “Incremental heuristic search”

Appendix IV [3.4.11]

***SMA\* Search*** is one of the Memory bounded heuristic search that comes under Informed Search Strategies. The main advantage of this search is that it only makes use of available memory to carry out the search.

Appendix V [3.4.11]

***Theta\**** (angle path planning) as compared to Block A\* is more sensitive to bad heuristics for any angle path planning approach. This method checks for shortcuts during each vertex expansion in the search graph. Finally, [Block A\*](http://en.wikipedia.org/w/index.php?title=Block_A*&action=edit&redlink=1) uses a look-up table to quickly find piece-wise any-angle paths. While Theta\* find paths that are no longer than A\* paths. Unlike Theta\* and Field D\*, Block A\* is guaranteed to find the *ground-truth* optimal path given that Block A\* uses a sufficiently large look-up table.

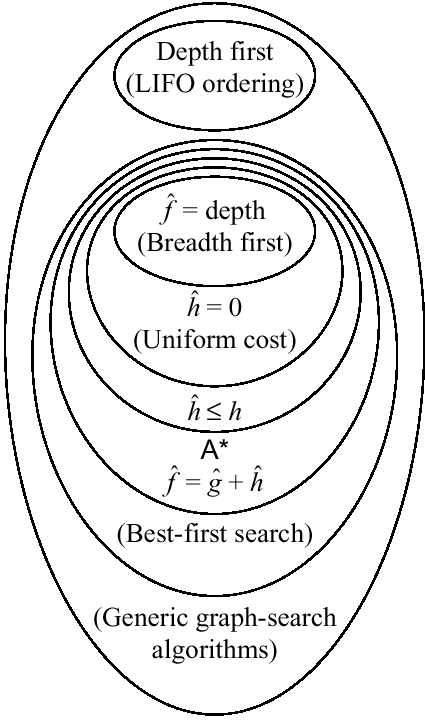
Appendix VI [3.4.11]

# *Learning Real-Time A\* (LRTA\*) Search*: If a problem is to be solved repeatedly with the same goal state but different intial state then one would like an algorithm that improves its performance over time. Learning Real-Time A\* (LRTA\*) is such an algorithm. It behaves almost identically to RTA\*, except that instead of stroring the second-best f value of a node as its new heuristic value, it stores the best value instead. Once one problem instance is solved, the stored heuristic values are saved and become the minimal value for the next problem instance.

While LRTA\* is less efficienct than RTA\* for solving a single problem instance, if it starts with admissible initial heuristic values over repeated trials, its heuristic values eventually coverge to their exact values. At which point the algorithm returns optimal solutions.

OTHERS?? (Yet there is another BFS mouse trap!)

***Relationships between some deterministic search algorithms (also include SMA\*, z,…?)***

**

***3.5* *Comments on the Design on Search algorithms using***

***Data Structure Transformation between***

***Algorithm Domain and Problem Domain***

*•* To integrate the algorithm domain date structure(s) as abstractly deﬁned by x in Di and z in Do with the explicit problem domain data structures is a creative aspect of the design synthesis process. An extensive variety of ADT structures can be selected including combinations of:

*- standard integers, reals, characters, boolean, ...*

*- stacks (LIFO, ...) or queues (priority, FIFO, ...)*

*- arrays or tables (hash, blocked, ...)*

*- sets/bags or sequences or strings of ... (subsets, subsequences, ...)*

*- lists or records*

*- trees (binary, red-black, B-, ...) or graphs (complete, ...)*

*•* Sets and lists along with sequences are initially selected with more speciﬁc ADT structures evolved from the reﬁnement process. The integration of a selected algorithm domain and problem domain usually results in a in- dexed decomposition of x and z which can be enumerated, split, or manipulated in the appropriate ADT manner. The process produces an unique design and reﬁnement leading to a relative eﬃcient implementation and eﬀective program. Of course, the design space itself is combinatoric lead- ing to a multitude of creative possibilities! We began with a variety of these ADT selections and domain integrations:

**–**

**–** *- shortest path problem/greedy algorithm-bfs - array*

**–** *- MST/greedy algorithm-dfs - sorted sequence*

**–** *- matroid problems/greedy algorithm work-dfs - sorted sequence*

**–** *- k-queens problem/gs-dfs - array/sequence (subsequence of columns)*

***3.6 Comments on***

***NPC Problem Domains and Deterministic Algorithm Domains***

*•* Evolving NPC problems such as SCP, MIS, Clique, Graph Coloring, TSP, Scheduling, Assignment require integrating with a priori graph search al- gorithm domains: Need problem domain English description and sym- bolic (math) non-ambiguous model including complexity of problem do-

main speciﬁcation.

**–**

**–** *- SCP/gs-dfs(bfs) algorithm - sets (table or sequence of sets reﬁne- ment)*

**–** *- MIS/gs-dfs(bfs) algorithm - sets (subsets of graph nodes explored/not- explored)*

**–** *- GCP/gs-dfs(bfs) algorithm - sets (subsets of graph nodes)*

**–** *- TSP/gs-dfs(bfs) alg - array, sets/sequence (subsets of graph nodes)*

**–** *- Routing, Scheduling problems/gs-dfs(bfs) - sets/sequence*

**–** *- Bin packing. Assignment. Knapsack/gs-dfs(bfs) - sets/subsets, tables  
- - VRP(CVRP)/(gs-dfs(bfs) – set of graph nodes, sets of demands/cost  
- - SAT ..  
- - Vertex Cover …  
- - …*

*•* uninformed deterministic search techniques: *uniformed search generally explicitly generates the entire search space(nodes/state)*

*- depth-ﬁrst search (dfs)*

*- depth-ﬁrst search with back-tracking (dfs/bt)*

*- breath-ﬁrst search (bfs)*

*- uniform-cost search*

*- depth limited search*

*- iterative deepening search (ID)*

*What is the time and search complexity of each of these uninformed deterministic search algorithms? (see R&N, Figure 3.17)*

*•* informed deterministic search techniques: *What heuristics from problem domain can be employed? What information from problem domain is*

*available for ... ? Is the overhead of the heuristic computation of low polynomial complexity? Even so, is it worthwhile.*

*- greedy dfs (local search yields optimal if PD has optimal substructure – P-Time*

*- depth-ﬁrst search dfs* ***(DFS\*, DFS/BT\*)***

*- best-ﬁrst search (BF\*, Z\*, A\*, ...)*

*- depth-limited search*

*- iterative deepening (ID, IDA\*. ...)*

*“Note that the algorithmic techniques for A\* modified algorithms could also be applied to Z\* and BF\* algorithms” Not much in the literature!*

*- beam search, a local or global selection of paths to explore based upon problem domain heuristics; unless ”good” heuristic information, has a limited amount of search exploration – defines beam width of select “best” values on the frontier for further exploration. Because, time and space complexity of queue storing and sorting can be very inefficient.*

*( <http://en.wikipedia.org/wiki/Beam_search> )*

*- recursive best-ﬁrst search RBFS; attempts to use* only linear mem- ory*; has to regenerate states/search; keeps track of ” good” F(n) nodes values, but has to regenerate paths to these nodes if they need to be ex- panded; possibly more eﬃcient that IDA\* in terms of nodes expanded (*[*http://eprints.fri.uni-lj.si/1282/*](http://eprints.fri.uni-lj.si/1282/) *)*

*- memory-bound A\* (MA\*); Simpliﬁed MA\* (*[*SMA\*);*](http://cs.txstate.edu/~ma04/files/CS5346/SMA%20search.pdf)  *attempt to use* all of available memory*; when memory limit reached, drops worst cost node on open list retaining path pointer to this state; can ”trash” between a limited number of search branches in memory; ”Of course, could just not use Closed list – does not check for repeated nodes” (*[*http://cs.txstate.edu/~ma04/files/CS5346/SMA%20search.pdf*](http://cs.txstate.edu/~ma04/files/CS5346/SMA%20search.pdf) *)*

*Which of all these above deterministic search algorithms are complete or admissible algorithms?*

*• And there are other variations with good performance for speciﬁc instantiated problems. Usually they are very speciﬁc algorithms because of* no free lunch theorem*, (NFL)* [*http://en.wikipedia.org/wiki/No\_free\_lunch\_in\_search\_and\_optimization*](http://en.wikipedia.org/wiki/No_free_lunch_in_search_and_optimization)

*• Educational Objective of Search: to solve NPC/PSPACE problem with associated well-disciplined algorithm domains: i.e.., to develop, study, analyze and integrate problem domain using deterministic provided search algorithm templates or evolve your own template based upon the variations provided above!! The pace continues and quickens.*

***QUIZ:*** *Make a table for each NP-C problem of interest. Indicate PD data structures, PD complexity, AD selection, AD data structures, and specific standard search elements. Of course, each AD selected would probably reflect a different search process and different data structures!*

*First try VRP (dfs/bt, bfs).*

***3.7 Divide and Conquer “AND/OR search Graph”  
 (This is a general best-first search)***

• if the underlying graph search structure is of an “AND/OR” struc- ture, then frontier state information is usually maintained as a set of explicit partial solution trees as compared to sets of explicit or implicit paths in an “OR” framework. Examples of the ‘AND” nature include global best ﬁrst (GBF) and an evolved constrained recursive-weighting

search (AO∗). Determination of the best solution base (subgraph) is usually embedded in the merit of an evaluation function f1. The superority of a node on the frontier of the solution base is usually deﬁned by a evaluation function f2, both of which are computable (primitive recursive) and are desired to be of small polynominal order or small dimension. (see Pearl for the underlying propagation of costs up the searchgraph

***Review*** *– Search is a essential computational algorithm used in many disciplines and applications. Embodies* ***deterministic*** *and stochastic search structures.*

*Generic simple algorithmic deterministic search approach:*

1. *Define a problem domain search space*
2. *Define an initial state and set of candidates to explore*
3. *Incrementally explore (dfs, bfs, …) partial solution paths from current state until goal (solution space) is reached. Check feasibility of each partial solution.*

*“Informed bfs/dfs search is better than uninformed - opens fewer   
 nodes/states (partial solutions)” Can proved faster algorithm completion time with associated PD heuristics that improve efficiency of search graph*

3.8 Search in Games