Computing Science (CMPUT) 455

Search, Knowledge, and Simulations

James Wright

Department of Computing Science
University of Alberta
james.wright@ualberta.ca

Fall 2021

Optional Material - More on Blind Search

Blind Search Extras

Optional material
Will not be on exams or quizzes
Highly recommend to review it anyway, to increase the depth and breadth of your learning

- Review main ideas, sample codes
- Depth-first Search (dfs)
- Breadth-first Search (bfs)
- Depth-limited dfs
- Iterative deepening dfs

Depth-first Search (DFS)

- Visit first child, then child-of-child, etc.
- Backtrack when no more children
- Goes (very) deep very quickly
- Minimal memory requirements - only path from root to current node
- Details e.g. `http://en.wikipedia.org/wiki/Depth-first_search`, or any algorithms textbook
  - See link from our resources page

Dfs for Treasure Search on Tree

- Simpler than on general graph (as e.g. in Cmput 204)

Depth-first search on tree
Returns (found, numNodesSearched)

```
def dfs(tree, node, treasure):
  numNodesSearched = 1
  if node == treasure:
    return True, numNodesSearched
  for child in tree[node]:
    found, childNodes = dfs(tree, child, treasure)
    numNodesSearched += childNodes
    if found:
      return True, numNodesSearched
  return False, numNodesSearched
```

Breadth-first Search (BFS)

- Blind graph search algorithm
- Data structure: queue (first-in-first-out, FIFO)
- Guaranteed to find solution with shortest number of steps in graph
- Expands nodes in “onion layers” around the root
- Main problem: queue typically gets very large very quickly
- Details e.g. `http://en.wikipedia.org/wiki/Breadth-first_search`, or any algorithms textbook

Bfs for Treasure Search on Tree

- Simpler than on general graph (as e.g. in Cmput 204)

Breadth-first search on tree
Returns (found, numNodesSearched)

```
def bfs(tree, start, treasure):
    numNodesSearched = 0
    queue = deque()
    queue.append(start)
    while len(queue) > 0:
        node = queue.popleft();
        numNodesSearched += 1
        if node == treasure:
            return True, numNodesSearched
        for child in tree[node]:
            queue.append(child)
    return False, numNodesSearched
```

Different Biases of Dfs and Bfs

- Dfs is fastest if the path to the treasure always leads through the *first child* near the root of the tree. Dfs goes deep on the first branch, then backtracks.
- Bfs is fastest if the path to the treasure is *short*. Bfs explores in order of increasing distance from the root
- On average over *all* nodes, those biases cancel out
- The expected number of steps is the same

Is there a Dfs with Bfs-like Behavior?

- Dfs needs very little memory
- Depth-first search can “get lost” in very deep searches, even if a shallow solution exists
- Bfs finds a shortest solution path, but needs much memory
- Can we combine the avantages of both?
- Yes, but there is a price to pay

Dfs with Depth Limit

- Idea: add a depth limit to dfs:
  `dfs(start, maxDepth)`
- This will stop search from going down very deep sequences
- Simple case: we know in advance the depth d_{sol} at which the solution will be found
- Just run `dfs(start,` d_{sol}`)`
- Avoid searching any nodes deeper in the tree
- Problem: in practice, may not know the value of d_{sol} beforehand
- Anyway, let's start with the case where d_{sol} is known

Analysis of DFS with Depth Limit

- Standard tree model: branching factor *b*, depth *d*
- Simplest analysis: Complete search of all levels including *d*.
- We already know how to count this:

$$c(b, d) = 1 + b + ...b^d = \sum_{i=0}^{d} b^i = \frac{b^{d+1} - 1}{b - 1}$$

- In expectation, we will only need about half of that
- In expectation we save about half of the last level
- (Optional) exercise: compute the exact expected number of nodes searched
- Hint: it is very slightly different from the “treasure search” before. We know there is no “treasure” on depths below *d*. What is the effect?

Iterative Deepening DFS (ID-DFS)

- What to do if d_{sol} is *not* known in advance?
- Iterative deepening idea: call dfs in a loop with increasing depth limits 0, 1, 2, ...
- `dfs(start, 0)`
- `if not found:  dfs(start, 1)`
- `if not found:  dfs(start, 2)`
- ...
- Stop as soon as goal found (or we run out of time...)
- Combines memory benefit of DFS with shortest path guarantee of BFS
- Overhead: needs to re-search lower levels in each iteration

Overhead of ID-DFS

- $c(b, d) = \frac{b^{d+1}-1}{b-1}$
- Iterative deepening cost, assuming complete searches of levels $0, 1, \ldots, d$. Cost: $idc(b, d) = c(b, 0) + c(b, 1) + \ldots c(b, d) = \sum_{i=0}^{d} \frac{b^{i+1}-1}{b-1} = \frac{1}{b-1}(\frac{b^{d+2}-b}{b-1} - (d+1))$
- relative cost $idc(b, d)/c(b, d)$ approximately $\frac{b^{d+2}}{(b-1)^2} / \frac{b^{d+1}}{b-1} = \frac{b}{b-1}$.
- For large b, the cost of re-searching lower levels is relatively small.
- The cost of the last level dominates
- Examples:
  $b = 2$, overhead factor 2.
  $b = 10$, overhead 11%.
  $b = 100$, overhead 1%.

Optional Material - More on Single-Agent Heuristic Search

Some Big Questions about Heuristic Search

- What are the important techniques in heuristic search today?
- What are the important applications?
- What are the main established techniques?
- How do the new techniques based on exploration and Monte Carlo methods work?
- What are the interesting research challenges?

Some Application Areas

- Single-agent search and puzzles
- Two player games
- Planning
  - Classical planning
  - Probabilistic planning, MDP, POMDP
  - Motion planning

Example - Linear Search Problem

- You are standing next to a river on a foggy day
- You want to find the (single) bridge to cross the river
- You don't know if the bridge is to the left or to the right
- It is so foggy you can only see the bridge when right in front of it
- What is a good strategy to find the bridge?
- When to turn around and try the other side?

Image source: `http://johngalbreathphotography.com/index/images/Travel`

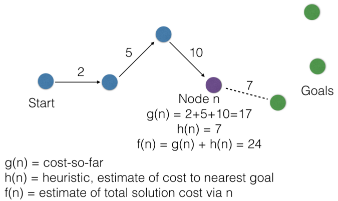
Classical Single Agent Search

- Typical heuristic search algorithms:
  - A*
  - Weighted A* (WA*)
  - Greedy best-first search (GBFS)
  - Branch-and-bound
- Local search algorithms:
  - Hill-climbing
  - GSAT, WalkSAT
  - Tabu search

Best-first Search

- “Informed” search algorithms:
  use heuristic to direct search towards goal
- Classic algorithms: A*, Greedy best-first search (GBFS), weighted A*
- Main difference: how to deal with solution costs vs speed
  - Optimal: find shortest path, use exact costs
  - Greedy search: focus on finding goal as quickly as possible, ignore costs
  - Bounded-optimal: Compromise, satisficing, consider both costs and search speed to some degree

Common Framework for Best-first Search Algorithms

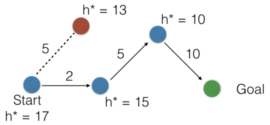


g(n) = 2+5+10=17
h(n) = 7
f(n) = g(n) + h(n) = 24

g(n) = cost-so-far
h(n) = heuristic, estimate of cost to nearest goal
f(n) = estimate of total solution cost via n

- $g(n)$ is cost of shortest known path from s to n
- $h(n)$ heuristic, estimate *cost-to-go* to closest goal
- $f(n)$ priority of expanding n
  - Usually a combination of g and h
- Best-first algorithms: expand node with smallest f-value

Perfect Heuristic and Hillclimbing



h*(n) = perfect heuristic, exact cost to nearest goal
5+13>17, not optimal action
2+15 = 17, optimal action

- $h^*(n)$: perfect heuristic, true distance to closest goal
- If you have $h^*(n)$, heuristic search is super easy:
- Repeat until goal: go to child with best h^* value
- This is called the *hillclimbing* strategy
- It is an example of *local search*
- You can hillclimb with any heuristic, but with h^* it works perfectly
- Decide next action locally, from a *current* state
- Compare with: random sampling

Desirable Properties of Heuristics

- *Admissible*:
  never overestimates the true cost

$$h(n) \leq h^*(n) \text{ for all } n$$

- *Consistent*:
  for any two neighbors u, v with edge cost $c(u, v)$

$$h(u) \leq c(u, v) + h(v)$$

- Consistency is stronger, implies admissibility
- Admissibility does not imply consistency

Reminder - Dijkstra's Algorithm for Shortest Paths

- Standard graph search algorithm, e.g. in Cmput 204
- Main ideas:
- Put each node n into a min-priority queue according to their best known distance from start ($g(n)$)
- Keep expanding smallest element from priority queue until expands goal state
- Update distance to a node when exploring an edge finds a new, shorter path (or the first path)
- Guaranteed to find a shortest path when edge costs are non-negative
- Blind search algorithm, uses no heuristics

Best-first Search Idea

- Similar to Dijkstra, but take heuristic into account
- $f(n)$ = priority of expanding n
- Put nodes into a min-priority queue according to their f-value
- keep expanding smallest-f node until solved
- Usually f is some combination of g and h
- See examples next slide

Popular Choices for $f(n)$

- $f = g$ is Dijkstra - ignores heuristic h
- $f = g + h$ the A* algorithm
- $f = h$ Greedy best-first search (ignore cost-so-far)
- $f = g + wh$ weighted A*, with some weight $w \geq 1$ on heuristic
- Sometimes you see $f = \alpha g + (1 - \alpha)h$, it is equivalent
- You can also use a combination of multiple different heuristics

Data Structures for Best-first Search

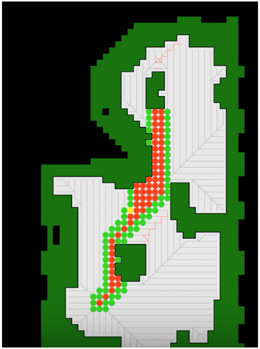


Image source:
`https://www.youtube.com/`

- *Open list*: a min-priority queue using *f* value
- *Closed list*: the nodes that have been expanded
- Depending on heuristic, may need to *re-expand* nodes in `Closed` if a shorter path is discovered later (as in Dijkstra)

Best-first Search Pseudocode

```
BestFirstSearch(G,s)
  Closed = {}, Open = {}
  Open.insert(s, h(s))
  # f(s) = h(s) for root because g(s) = 0
  while not Open.empty():
    v = Open.extract-min()
    Closed.insert(v)
    for u in adj(G, v):
      if not u in Closed $\cup$ Open:
        g(u) = g(v) + edge-cost(v,u)
        f(u) = g(u) + h(u)   # for A*
        Open.insert(u, f(u))
```

Comments on Best-first Search

- Code does not show the case where a new, cheaper path to a node is discovered
- Algorithms differ in how they handle this case
    - Ignore
    - Re-open: move node back from Closed into Open
    - Update node distance but don’t re-open it
    - It depends on properties of heuristic, and on whether we need optimal solutions

Iterative Deepening A* (IDA*)

- Similar idea to ID-DFS
- Depth-first search, stop recursion if given bound for f exceeded
- During depth-first search, keep track of smallest f-value above bound
- Use that smallest f-value as bound for next iteration
- No open list - less memory. No closed list needed either (but can use it)
- Similar problems with duplicate expansions

Branch and Bound

- Exact method for optimization problems
- Here: find a minimum cost solution
- Example: Traveling Salesperson Problem (TSP)
    - Salesperson needs to visit *n* cities
    - Given a start point, needs to return here at the end
    - Goal: optimize order of visits to minimize length of tour

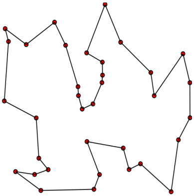


Image source: `https://upload.wikimedia.org`

Branch and Bound Algorithm Outline and Example

- Set of all possible solutions S
  - TSP: all permutations of other cities
- Upper bound u on cost of best solution.
  - Example: use any inexact method to get some good *initial solution*
  - TSP example: greedy - always go to the closest unvisited city
- Branch: Partition S into subsets $S_1, S_2, \cdots, S_k$
  - TSP example: pick the first city to visit. S_i = all tours that visit city i first

Branch and Bound Algorithm Outline and Example (2)

- Bound: for each S_i, find lower bound l_i on the cost of *any* solution in that set.
  - TSP: costs known for start of tour, plus admissible heuristic for visiting rest of cities
- Prune: if $l_i \geq u$, there can be no better solution in S_i
  - TSP: this is proof that path-so-far was bad

Branch and Bound Algorithm Outline and Example (3)

- If no pruning possible: recursion. Partition S_i into even smaller subsets
    - TSP: pick next city on tour
- End of recursion:
    - Complete single solution s
    - Compute its cost $c(s)$.
    - If $c(s) < u$, update u with new best-known solution
    - TSP example: complete tour is known

Branch and Bound Notes

- If no initial guess known: use $u = \infty$.
  No pruning until first real solution found
- In practice, the partitioning step often means refining a *partial solution*
  - TSP Example: fix next step in partial tour
- How to bound?
  - TSP example is typical
  - Cost of partial solution + admissible (lower) bound on cost of solving the “rest”

General Techniques for Dealing with Complex Problems

Four (related) “big ideas”

- Divide and conquer
- Approximation
- Abstraction
- Relaxation

Most of these ideas are discussed in Polya’s book as well

Divide and conquer

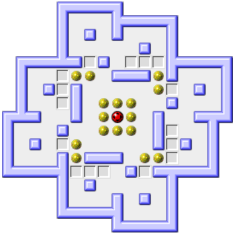


Image source:

`http://sneezingtiger.com/`

`sokoban/levels.html`

- Break problem into smaller sub-problems
- Solve them and combine solutions
- Examples: dynamic programming, branch and bound
- Example: Sokoban puzzle: solve each “room” separately

Approximation

- Cannot solve exactly? Find a “good” solution instead
- Example: irregularly shaped vehicle and obstacles
- Approximation: simpler polygons or circles
- Questions: how good is the approximation? Is the problem even still solvable?
- Sensor and numerical errors can change the problem

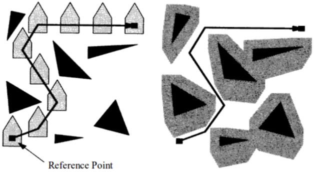


Image source: R. Mojtahedzadeh, MSc thesis, KTH, Sweden

Abstraction

- Problem too hard to solve exactly? Solve a related simpler problem instead
- Example: path-finding
  - Group clusters of nearby states into a single abstract state
  - E.g. “Edmonton” vs exact location within the city
- Example: ignore details of robot shape, treat it as a single point, or a sphere
- Uses:
  - For “good enough” solutions
  - For real-life problems that are too hard to model exactly;
  - For generating heuristics

Relaxation

- Idea: simplify some parts of the problem to make it easier to solve. Change to an "easier" state space.
- Example:
  - Knapsack problem - must either use one item completely, or not at all.
  - Relaxation: allow using a fraction of an item - relaxed problem is much easier and may help solve original
- Uses:
  - Find "relaxed solutions" close to good real solutions;
  - Get bounds on the best-possible solution;
  - Generate heuristics