Shortest Path Algorithms

Relaxation

(From AD book)
Relaxation is a technique for tightening an upper bound.

For each vertex $v \in V$ we maintain an attribute $v.d$ (*shortest path estimate*), which is an upper bound on the weight of a shortest path from source s to v.

The process of relaxing an edge (u, v) consists of testing whether we can improve the shortest path to v found so far by going through u and, if so, update $v.d$ and $v.\pi$. To initialize the shortest-path estimates and predecessors, call the initialization-procedure below:

Pseudocode:
(Page 648 - 649 in book)

```
INITIALIZE-SINGLE-SOURCE(G, s)
1  for each vertex v ∈ G.V
2      v.d = ∞
3      v.π = NIL
4  s.d = 0
```

```
RELAX(u, v, w)
1  if v.d > u.d + w(u, v)
2      v.d = u.d + w(u, v)
3      v.π = u
```

Dijkstra's algorithm relaxes each edge exactly once.
Bellman-Ford algorithm relaxes each edge $|V| - 1$ times.

Properties of shortest paths and relaxation

(Page 650 in book, proofs section 24.5)

Triangle inequality (Lemma 24.10)
For any edge $(u, v) \in E$, we have $\delta(s, v) \leq \delta(s, u) + w(u, v)$.

Upper-bound property (Lemma 24.11)
We always have $v.d \geq \delta(s, v)$ for all vertices $v \in V$, and once $v.d$ achieves the value $\delta(s, v)$, it never changes.

No-path property (Corollary 24.12)
If there is no path from s to v, then we always have $v.d = \delta(s, v) = \infty$.

Convergence property (Lemma 24.14)
If $s \rightsquigarrow u \rightarrow v$ is a shortest path in G for some $u, v \in V$, and if $u.d = \delta(s, u)$ at any time prior to relaxing edge (u, v), then $v.d = \delta(s, v)$ at all times afterward.

Path-relaxation property (Lemma 24.15)
If $p = \langle v_0, v_1, \ldots, v_k \rangle$ is a shortest path from $s = v_0$ to v_k, and we relax the edges of p in the order $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$, then $v_k.d = \delta(s, v_k)$. This property holds regardless of any other relaxation steps that occur, even if they are intermixed with relaxations of the edges of p.

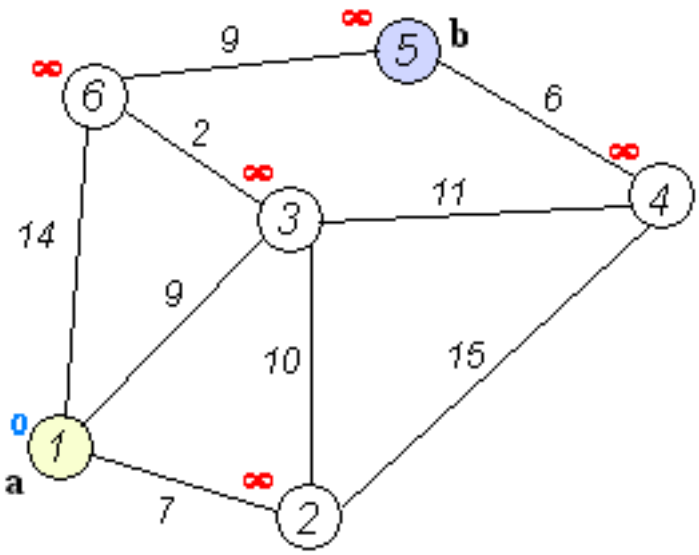
Predecessor-subgraph property (Lemma 24.17)
Once $v.d = \delta(s, v)$ for all $v \in V$, the predecessor subgraph is a shortest-paths tree rooted at s.

Dijkstra's algorithm

Dijkstra's original variant finds the shortest path between two nodes.
A more common variant fixes a single node as the "source" node and finds shortest paths from the source to all other nodes in the graph, producing a shortest-path tree.

Assumptions & properties:
* Works for both directed and undirected graphs
* All edges must have nonnegative weights.
* Graph must be connected.

Example



(1) → (2) → (3) → (6) → (5)

The closest distance from node (1) to (3) is 9. (another option could be 7+10 going through node 2)

The closest distance from node (3) to (6) is 2. (another option could be 14)
The closest distance from node (6) to (5) is 9.
The total distance from *a* to *b* is therefore 9 + 2 + 9 = 20

Pseudocode

(Page 658 in book)

```
DIJKSTRA(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  S = ∅
3  Q = G.V
4  while Q ≠ ∅
5      u = EXTRACT-MIN(Q)
6      S = S ∪ {u}
7      for each vertex v ∈ G.Adj[u]
8          RELAX(u, v, w)
```

Implementation

Extra information:
A node should contain an ID and index.
Has to examine the whole graph, before we can determine what the shortest path is.
Use backtracking to find the path taken. Keep note of what vertex the next vertex came from.

Dijkstra's algorithm written in C++:
http://www.reviewmylife.co.uk/blog/2008/07/15/dijkstras-algorithm-code-in-c/
http://www.geeksforgeeks.org/greedy-algorithms-set-6-dijkstras-shortest-path-algorithm/

Running time

(See book page 661-662 for explanation)

Using binary min-heap:
$O((V+E) \cdot \log(V))$
$O(E \cdot \log(V))$ if all vertices are reachable from the source.

Using Fibonacci heap:
$O(V \cdot \log(V) + E)$

Bellman-Ford algorithm

Like Dijkstra's Algorithm, Bellman–Ford is based on the principle of relaxation, in which an approximation to the correct distance is gradually replaced by more accurate values until eventually reaching the optimum solution.

Assumptions & properties:
* Works for negative weights
* Detects a negative cycle if any exist
* Finds shortest simple path if no negative cycle exists.
  If a graph contains negative-weight cycle, then some shortest paths may not exist.

For the bachelor project, we don't need to think about negative edges.

A* search algorithm

A* was developed in 1968 to combine heuristic approaches like Greedy Best-First-Search and formal approaches like Dijsktra's algorithm.

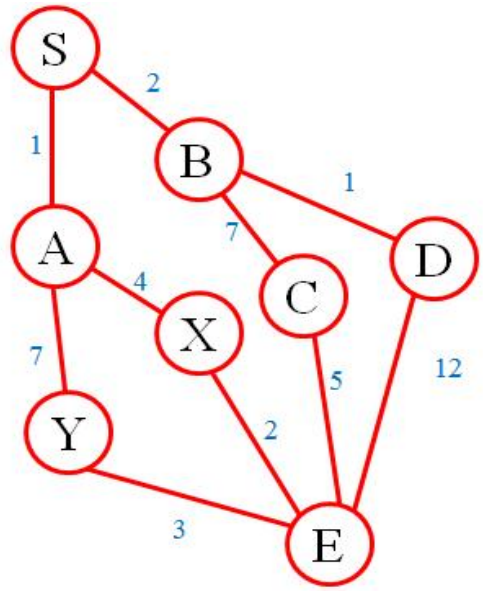
A* is almost exactly like Dijkstra's Algorithm, except we add in a heuristic. A* is built on top of the heuristic, and although the heuristic
itself does not give you a guarantee, A* can guarantee a shortest path.

It combines the pieces of information that Dijkstra's algorithm uses (favoring vertices that are close to the starting point)
and information that Greedy Best-First-Search uses (favoring vertices that are close to the goal).

A* can only take non-negative edges; therefore an edge can be examined only once.

Functions used:
$g(n)$ represents the exact cost of the path from the starting point to any vertex n.
$h(n)$ represents the heuristic estimated cost from vertex n to the goal.
Each time through the main loop, it examines the vertex n that has the lowest
$f(n) = g(n) + h(n)$.

Example:



- Values for h:
  A:5, B:6, C:4, D:15, X:5, Y:8

Expand S
{S,A} f=1+5=6
{S,B} f=2+6=8

Expand A
{S,B} f=2+6=8
{S,A,X} f=(1+4)+5=10
{S,A,Y} f=(1+7)+8=16

Expand B
{S,A,X} f=(1+4)+5=10
{S,B,C} f= (2+7)+4=13
{S,A,Y} f=(1+7)+8=16
{S,B,D} f= (2+1)+15=18

Expand X
{S,A,X,E} is the best path... (costing 7)

Heuristics

What is a heurestic?

An algorithm contains a heuristic function if it has some estimate of how far from the goal any vertex is.

A heuristic is a technique designed for solving a problem more quickly when classic methods are too slow,
or for finding an approximate solution when classic methods fail to find any exact solution. This is achieved
by trading optimality, completeness, accuracy, or precision for speed. In a way, it can be considered a shortcut.

The goal of a heuristic search is to reduce the number of nodes searched in seeking a goal (https://en.wikibooks.
org/wiki/Artificial_Intelligence/Search/Heuristic_search/Astar_Search)

A*'s use of the Heuristic

http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html
http://www.redblobgames.com/pathfinding/a-star/introduction.html

A* is guaranteed to find the shortest path if the heuristic is never larger than the true distance. As the heuristic becomes smaller, A* turns into Dijkstra's Algorithm. As the heuristic becomes larger, A* turns into Greedy Best First Search.

Speed or accuracy?

To get something quicker, we have to give up ideal paths.
If we want to switch between speed and accuracy, we can build a heuristic function that assumes the minimum cost to travel one grid space is 1 and then build a cost function that scales:

$$g'(n) = 1 + \alpha \cdot (g(n) - 1)$$

$\alpha = 0$:
Very fast, but accuracy suffers.
$\alpha = 1$:
The shortest path will be found but with slow speed.

(We can set α anywhere in between)

Implementation

Notes about performance, set representation, etc:
http://theory.stanford.edu/~amitp/GameProgramming/ImplementationNotes.html

Implementation in C++:
http://www.redblobgames.com/pathfinding/a-star/implementation.html#cplusplus

Sources

About Dijkstra's algorithm:

General theory + some lemmas + proof of correctness:
http://math.mit.edu/~rothvoss/18.304.3PM/Presentations/1-Melissa.pdf

About A* Search algorithm:

Introduction + comparison to Dijkstra's and Greedy Best-First-Search:
http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html

Books/links we may find useful:

* https://scholar.google.dk/

* "Encyclopedia of Artificial Intelligence"
  Stuart C. Shapiro, ed. (see articles on "Search" and "A* Search")

* "The Handbook of Artificial Intelligence, vol. 1"
  Barr and Feigenbaum, eds. (chapter 2 is on search)

* "Principles of Artificial Intelligence"
  Nils J. Nilsson (a bit dated, but good discussion of A*).

* "A* algorithm optimality proof when heuristic is always underestimating"
  http://stackoverflow.com/questions/10195780/a-algorithm-optimality-proof-when-always-underestimating

* "Easy C++ benchmarking"
  https://bruun.co/2012/02/07/easy-cpp-benchmarking

* Long pdf with theory about shortest paths:
  http://people.mpi-inf.mpg.de/~mehlhorn/ftp/Toolbox/ShortestPaths.pdf

Manhattan distance?

A* -> noget med at man har nogle stjerner (punkter) tæt på destination *t*.
Ud fra den stjerne der er tættest på, kan A* dermed vide hvilken retning den skal bevæge sig imod.

Extra notes

Complete search algorithm: a search algorithm that is guaranteed to find a solution if there is one.

Feedback fra Thorup

Bevise: man må aldrig besøge den samme knude 2 gange (dvs. ingen cykler, for så er Dijkstra forkert).
Bevise: distancen må aldrig blive mindre når du bevæger dig til det næste punkt
$(\forall\ u:\ \ p(u) \leq d(u,w) + p(w))$
Dette kan sikres, da vi måler den euklidiske afstand (vores heuristik.)

Argumenter for at algoritmerne virker.

Kan aldrig garantere en bedre køretid end O(m log m/n?)
Bonuspoint hvis vi også implementerer Fibonnaci heap. For så får man en lavere køretid end O(m log n) / O(E log V)
Vær sikker på at vores min-priority queue altid tager distancen $d(s,v) + p(v)$.

Sumfunktion som han talte om: $\sum_{i=1}^{d-1} c(v_i, v_{i+1}) + p(v_i) - p(v_{i+1})$.

$p(v)$ measures approximate distance to t (goal).