Planning

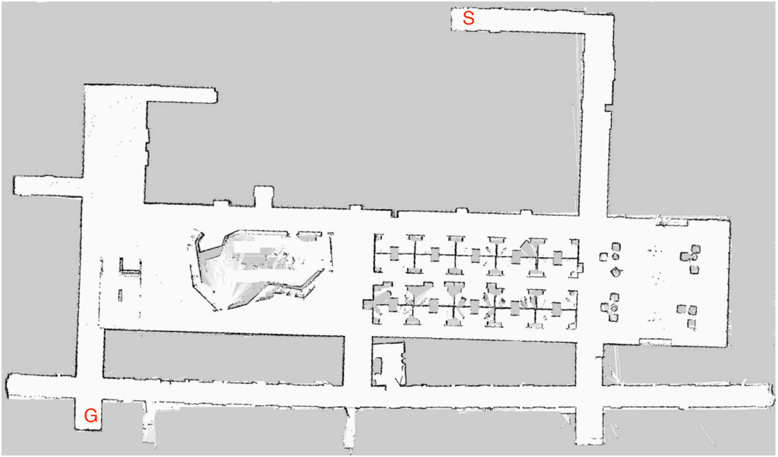
Stefano Carpin

Department of Computer Science and Engineering
School of Engineering
University of California, Merced
https://sites.ucmerced.edu/scarpin
https//robotics.ucmerced.edu

Planning definition

- Planning: the problem of selecting a sequence of actions to achieve an assigned goal
  - not a unique definition accepted by everyone
- often formulated in terms of state spaces
- goal must be specified, but starting location may or may not be assigned
- Plan: a sequence of actions or a function determining which action to take (e.g., based on current state, current time, uncertainty, etc...)
- so *planning* is the process, and *plan* is the result
- planning is where AI meets robotics...

Prototypical Planning Problem



Go from S to G

Three Different Planning Problems: Deterministic Planning

- Deterministic planning: given a start state $\mathbf{x}_s$, a goal state $\mathbf{x}_g$ (or a set of goal states), and a *state transition equation* $f(\mathbf{x}, \mathbf{u})$, determine a sequence of inputs $\mathbf{u}_1, \mathbf{u}_2, \ldots, \mathbf{u}_n$ to transform the start state in the goal state.

$$\mathbf{x_s} = \mathbf{x_0} \xrightarrow{\mathbf{u}_1} \mathbf{x}_1 = f(\mathbf{x}_0, \mathbf{u}_1) \xrightarrow{\mathbf{u}_2} \mathbf{x}_2 = f(\mathbf{x}_1, \mathbf{u}_2) \xrightarrow{\mathbf{u}_3} \ldots \qquad \mathbf{x}_n = f(\mathbf{x}_{n-1}, \mathbf{u}_n) = \mathbf{x}_g$$

  - the sequence $\mathbf{u}_1, \mathbf{u}_2, \ldots, \mathbf{u}_n$ is the plan
- *deterministic* means that the outcome of each action is fully predictable
- problem reduces to *graph search*

Three Different Planning Problems: Feedback Planning

- Feedback Planning: given a start state x_s and a goal state x_g and the objective is to determine a feedback function (also called *policy*) $\pi : X \rightarrow U$ defining for each state the action to take.
- implicitly assumes state is known (see how π is defined)
- considers the case where the outcome of actions is unpredictable
  - must know which action to execute from *any* possible state
- usually solved using a different formalism, i.e., Markov Decision Processes

Three Different Planning Problems: Planning in Belief Spaces

- Planning in belief spaces: given x_s and x_g the hypothesis of state observability no longer holds (current state cannot be determined with certainty)
- most generic case
- outcome of actions is not predictable and state is not precisely known
- must formulate plans based on uncertain information (belief)
- significantly harder than the previous two cases (we will not cover it).

Discrete Models

- state space set is discrete, finite, and known a priori: $S = \{x_1, x_2, \ldots, x_n\}$
- For each state $x_i \in S$, a finite set of actions (or inputs) $U(x)$ is defined
  - $U(x)$ is the set of actions that can be executed in state x
  - different states may have different set of actions
  - set of all actions: $U = \cup_{x \in S} U(x)$
  - with a slight abuse of notation, often write U instead of $U(x)$
  - U is also finite

Features of Planning Algorithms

Completeness : a planning algorithm is *complete* if it will always find a solution, if one exists

Uninformed : a planner that has only access to the problem structure (states, input, start/goal)

Informed : a planner that uses additional information to expedite the planning process (e.g., domain specific heuristics about how *good* a state is)

Open Loop Planning

Builds upon discrete model, adding a time invariant state transition equation

$$x_t = f(x_{t-1}, u_t) \qquad x_i \in S, u_t \in U(x_{t-1}).$$

Alternative formulation:

$$K = \{(x, u) \in S \times U \mid x \in S \wedge u \in U(x)\}$$

Then $f : K \to S$

Open loop planning must be used with caution...

The Planning Graph

For the above problem definition, a *planning graph* can be defined. It is a directed graph $G = (V, E)$ defined as follows:

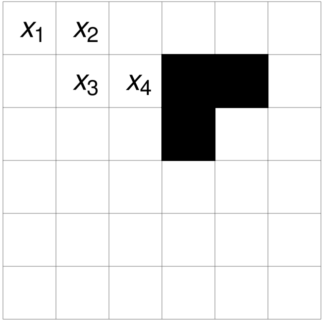
1. Vertices set V: on vertex per state
2. Edges set E: if there exists $u \in U(x_i)$ such that $x_j = f(x_i, u)$, then we add a directed edge from x_i to x_j
   - edges set is one-to-one to with K

Definition

Let $G = (V, E)$ be a graph, and $x_1 \in V$, $x_n \in V$ be two vertices. A path p between x_1 and x_n is a sequence of vertices $x_1, x_2, \ldots, x_n$ such that $(x_i, x_{i+1}) \in E$ for $1 \leq i \leq n - 1$.

One-to-one mapping between paths and plans.

Example: grid world



Unweighted Graph Search/Planning Problems

Directed Graph Search Problem: *Given a directed graph $G = (V, E)$, and vertices $x_s \in V$, $x_g \in V$, determine a path from x_s to x_g or return* failure *if no path can be found.*

Not a YES/NO problem (if a path exists, it must be returned)

Weighted Graph Search/Planning Problems

Definition

Let $G = (V, E)$ be a weighted graph, and let $c : E \to \mathbb{R}$ be its cost function. Let $p = x_1, x_2, \ldots, x_n$ be a path in G. The cost of path p is the sum of the costs of its edges, i.e.,

$$c(p) = \sum_{i=1}^{n-1} c(x_i, x_{i+1}).$$

Weighted Directed Graph Search Problem: *Given a directed weighted graph $G = (V, E)$ with non-negative cost function $c : E \to \mathbb{R}_{\geq 0}$, and vertices $x_s \in V$, $x_g \in V$, determine a path of minimum cost from x_s to x_g or return* failure *if no path can be found.*

Minimum cost is unique, but path of minimum cost is not necessarily unique.

Common Traits in Discrete Planning Algorithms

We will see four different planning problems. They all share the following:

- all algorithms start from x_s
- vertices to be processed are stored in a data structure that is often referred to as the *OPEN* list or queue
  - may or may not be prioritized
- after a node is removed from the *OPEN* data structure (expanded), it is moved into a container data structure called *CLOSED*
  - *CLOSED* can at times be removed, but it is useful to reason about the correctness
- *parent* of a vertex: backpointer to the vertex from which it was discovered; *null* if it has not been discovered yet;
- *visited*: binary flag to mark vertices that have not yet been discovered; useful to avoid re-visiting vertices (could also use *parent* for the same purpose)
- *spanning tree*: tree rooted at x_s showing how the graph has been visited

Breadth First Search – BFS

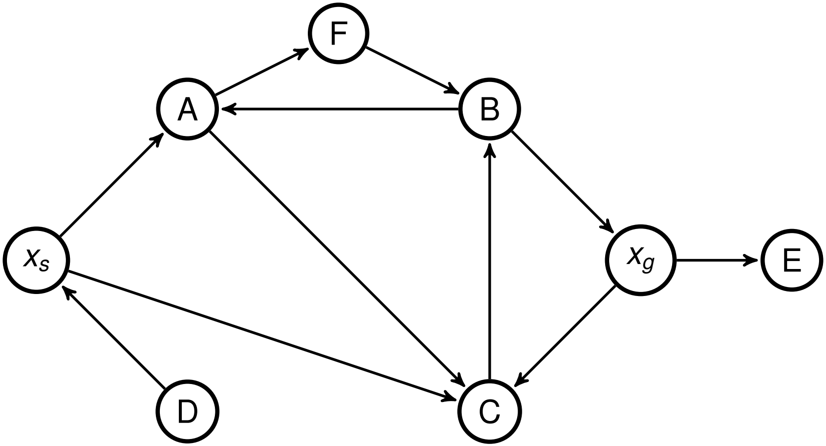
```
Data: G = (V, E), x_s ∈ V, x_g ∈ V
Result: Path from x_s to x_g if it exists, or FAILURE
foreach x ∈ V do
    x.parent ← null;
    x.visited ← false;
OPEN.initializeEmpty();
CLOSED.initializeEmpty();
OPEN.insert(x_s);
x_s.visited ← true;
while not OPEN.empty() do
    x ← OPEN.remove();
    CLOSED.insert(x);
    foreach x′ ∈ V such that (x, x′) ∈ E do
        if x′.visited =false then
            x′.visited ← true;
            x′.parent ← x ;
            if x′ = x_g then
                return ExtractPath(x_s, x_g);
            else
                OPEN.insert(x′);
return FAILURE;
```

Algorithm 1: BFS/DFS algorithm

BFS Features

- *OPEN* is a FIFO container (queue)
- complexity: $\mathcal{O}(|V| + |E|)$
- finds a shortest plan in terms of actions (every action/edge costs the same)
- ExtractPath walks its way backwards from x_g to x_s via the backpointers
- BFS is complete

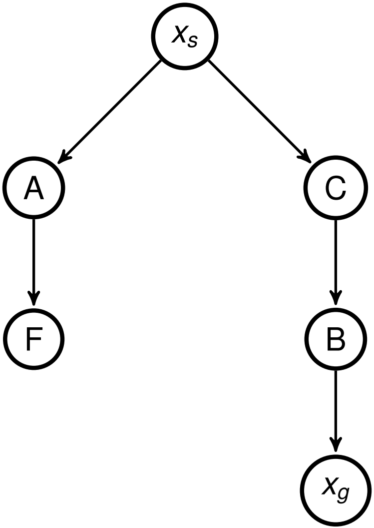
Example



Example

Step	$OPEN$	x_s	A	B	C	D	E	F	x_g
0	x_s	N/T	N/F	N/F	N/F	N/F	N/F	N/F	N/F
1	A,C	N/T	x_s/T	N/F	x_s/T	N/F	N/F	N/F	N/F
2	C,F	N/T	x_s/T	N/F	x_s/T	N/F	N/F	A/T	N/F
3	F,B	N/T	x_s/T	C/T	x_s/T	N/F	N/F	A/T	N/F
4	B	N/T	x_s/T	C/T	x_s/T	N/F	N/F	A/T	N/F
5	$\varnothing$	N/T	x_s/T	C/T	x_s/T	N/F	N/F	A/T	B/T

Rooted Tree (BFS)



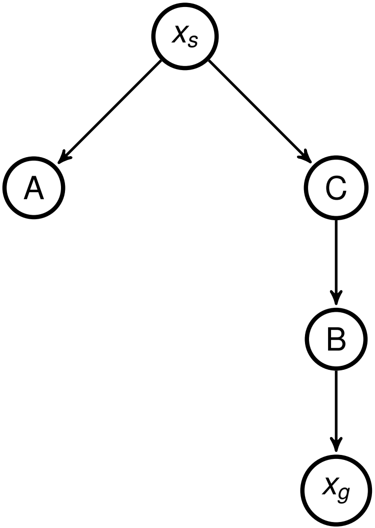
Depth First Search (DFS)

- identical pseudocode, but *OPEN* is now a LIFO (stack)
- DFS is complete when graph is finite
- complexity: $\mathcal{O}(|V| + |E|)$
- cannot say anything about the length of the computed plan

Example

Step	*OPEN*	x_s	A	B	C	D	E	F	x_g
0	x_s	N/T	N/F	N/F	N/F	N/F	N/F	N/F	N/F
1	A,C	N/T	x_s/T	N/F	x_s/T	N/F	N/F	N/F	N/F
2	B,A	N/T	x_s/T	C/T	x_s/T	N/F	N/F	N/F	N/F
3	A	N/T	x_s/T	C/T	x_s/T	N/F	N/F	N/F	B/T

Rooted Tree(DFS)

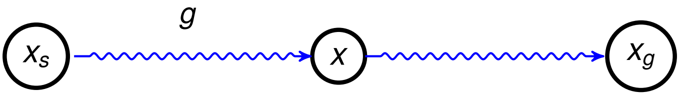


Dijkstra's Algorithm

- solves more realistic case of weighted graphs
  - returns a path of minimal cost
- implemented inside some of ROS planners
- developed around ideas widely used in many other planning algorithms
- still an *uninformed* algorithm

Key Ideas

- for each vertex, a cost to come is determined (indicated as g)
  - it is the cost to come from the source vertex x_s
- costs are iteratively refined (lowered)
  - *label correcting algorithm*
- $g(x)$ is the lowest cost to come from x_s to x discovered *so far*
- shortest/optimal path *must* be composed by optimal/shortest subpaths



Key Ideas and Algorithmic Details

- start exploring from x_s
- OPEN is a priority queue prioritized by the cost to come g
- as the graph is explored, look for *shortcuts*, i..e, ways to decrease g
- once the optimal cost of a vertex is determined, freeze it and do not consider the vertex anymore

Dijkstra's Algorithm

Data: $G = (V, E)$, $x_s \in V$, $x_g \in V$, $c : E \to \mathbb{R}_{\geq 0}$
Result: Shortest path from x_s to x_g if it exists, or FAILURE

foreach $x \in V$ **do**
    $x.g \leftarrow \infty$; $x.parent \leftarrow$ **null**;
$x_s.g \leftarrow 0$;
OPEN.initializeEmpty();
OPEN.insert(x_s);
CLOSED.initializeEmpty();
while not *OPEN*.empty() **do**
    $x \leftarrow$ *OPEN*.remove();
    CLOSED.insert(x);
    if $x = x_g$ **then**
        return ExtractPath(x_s, x_g);
    foreach $x' \in V$ *such that* $(x, x') \in E$ **do**
        if $x'.g = \infty$ **then**
            $x'.parent \leftarrow x$;
            $x'.g \leftarrow x.g + c(x, x')$;
            OPEN.insert(x');
        else if $x.g + c(x, x') \leq x'.g$ **then**
            $x'.g \leftarrow x.g + c(x, x')$;
            x'.parent$\leftarrow x$;
            OPEN.rebalance(x');
return FAILURE;

Properties

- when x_g is removed from OPEN, an optimal path has been found. Why?
- every time a node is rediscovered, an attempt is made to lower it costs
- must rebalance OPEN because cost to go of nodes changes
- Key difference with BFS/DFS: Dijkstra's algorithm does not terminate when x_g is discovered for the first time, but when x_g is removed from OPEN (compare pseudocode)

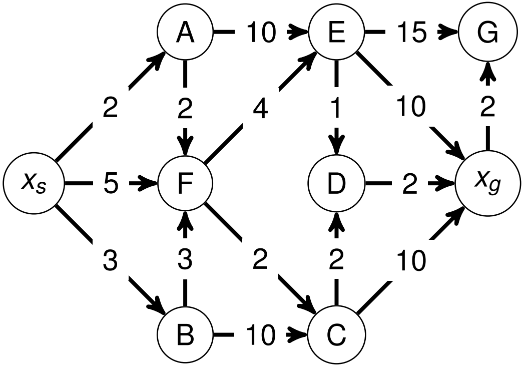
Correctness and Complexity

- computational complexity depends on the data structure used to implement OPEN
  - e.g., $\mathcal{O}((V+E)\log V)$ if using a binary heap or $\mathcal{O}(V\log V + E)$ if using a Fibonacci heap

Theorem

If a path between x_s and x_g does not exist, Dijkstra's algorithm returns FAILURE. If a between x_s and x_g exists, Dijkstra returns a path of minimum cost, as per the path cost formerly defined.

Example



Example

Step	OPEN	x_s	A	B	C	D	E	F	G	x_g
0	$x_s/0$	$N/0$	N/∞	N/∞	N/∞	N/∞	N/∞	N/∞	N/∞	N/∞
1	$A/2, B/3, F/5$	$N/0$	$x_s/2$	$x_s/3$	N/∞	N/∞	N/∞	$x_s/5$	N/∞	N/∞
2	$B/3, F/4, E/12$	$N/0$	$x_s/2$	$x_s/3$	N/∞	N/∞	$A/12$	$A/4$	N/∞	N/∞
3	$F/4, E/12, C13$	$N/0$	$x_s/2$	$x_s/3$	$B/13$	N/∞	$A/12$	$A/4$	N/∞	N/∞
4	$C/6, E/8$	$N/0$	$x_s/2$	$x_s/3$	$F/6$	N/∞	$F/8$	$A/4$	N/∞	N/∞
5	$D/8, E/8, x_g/16$	$N/0$	$x_s/2$	$x_s/3$	$F/6$	$C/8$	$F/8$	$A/4$	N/∞	$C/16$
6	$E/8, x_g/10$	$N/0$	$x_s/2$	$x_s/3$	$F/6$	$C/8$	$F/8$	$A/4$	N/∞	$D/10$
7	$x_g/10, G/23$	$N/0$	$x_s/2$	$x_s/3$	$F/6$	$C/8$	$F/8$	$A/4$	$E/23$	$D/10$
8	$G/23$	$N/0$	$x_s/2$	$x_s/3$	$F/6$	$C/8$	$F/8$	$A/4$	$E/23$	$D/10$

Exploration Tree

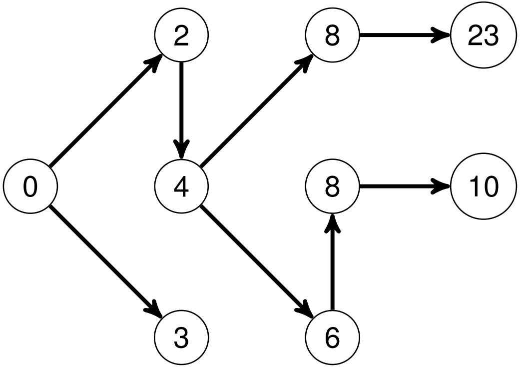


Figure: Tree produced by Dijkstra's algorithm.

Single Source Shortest Path

```
Data: G = (V, E), x_s ∈ V, c : E → ℝ_{≥0}
Result: Shortest from x_s to each vertex in V reachable from x_s
foreach x ∈ V do
    x.parent ← null;
    x.g ← ∞;
x_s.g ← 0;
OPEN.initializeEmpty();
CLOSED.initializeEmpty();
OPEN.insert(x_s);
while not OPEN.empty() do
    x ← OPEN.remove();
    CLOSED.insert(x);
    foreach x′ ∈ V such that (x, x′) ∈ E do
        if x′.g = ∞ then
            x′.g ← x.g + c(x, x′);
            x′.parent ← x ;
            OPEN.insert(x′);
        else if x.g + c(x, x′) ≤ x′.g then
            x′.g ← x.g + c(x, x′);
            x′.parent← x ;
            OPEN.rebalance(x′);
```

Algorithm 3: Single source shortest path algorithm

Single Source Shortest Path Tree

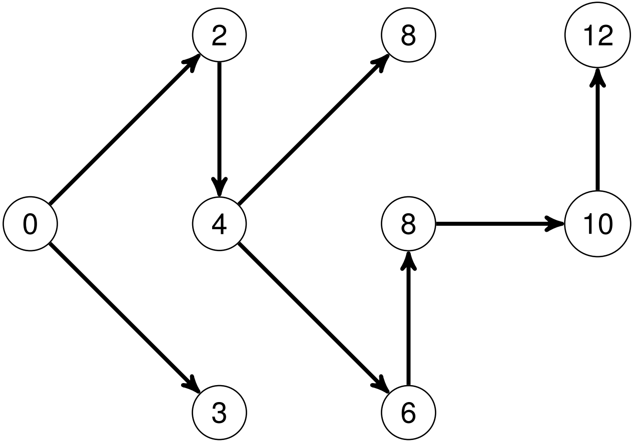


Figure: Tree produced by Single Source Shortest Path algorithm.

A*

- The most fundamental algorithm in search based AI
- informed method: assumes the availability of an heuristic telling us *how good* a vertex is
  - *how good* is an estimate of the cost to go h from a vertex to the goal

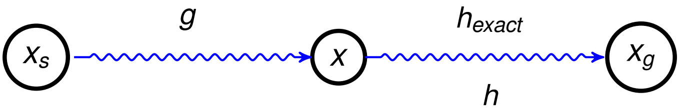


Figure: A node x along one path from x_s to x_g. The cost to come g is the cost of the best path (discovered so far) between x_s and x. h_{exact} is the cost of the shortest path between x and x_g, whereas h is an estimate of h_{exact}.

Important Quantities

- g: lowest cost-to-come discovered so far
- h_{exact}: lowest cost to go (unknown)
- $g + h_{exact}$: cost of the shortest path from x_s to x_g passing through x (after x is extracted from OPEN)
- h: estimate for h_{exact} (underestimation – more details later)
- $f = g + h$: is always an estimation of the cost of the shortest path from x_s to x_g passing through x

Admissible Heuristics

Definition

Let $G = (V, E)$ be a weighted graph where $c : V \to \mathbb{R}_{\geq 0}$ is the cost function. Let $x_g \in V$ be a goal vertex and for each vertex $v \in V$ let $c(p_{v,x_g})$ be the cost of a shortest path from v to x_g. An admissible heuristic is a function $h : V \to \mathbb{R}_{\geq 0}$ such that for each vertex $h(v) \leq c(p_{v,x_g})$.

- $h(v) = 0$ for each $v \in V$ is always an admissible heuristic because $c(p_{v,x_g}) \geq 0$ for each vertex v
- Dijkstra's algorithm can be seen as A* with $h(v) = 0$ for each vertex
- "ideal heuristic": $h(v) = c(p_{v,x_g})$. The closer the gap, the better.

Consistent Heuristic

Definition

Let $G = (V, E)$ be a weighted graph where $c : V \to \mathbb{R}_{\geq 0}$ is the cost function, and let $x_g \in V$ be a goal vertex. A function $h : V \to \mathbb{R}_{\geq 0}$ is a consistent heuristic if

1. for couple of vertices $u, v \in V$ with $(v, u) \in E$, then $h(v) \leq c(v, u) + h(u)$;
2. $h(x_g) = 0$.

- does not over-estimate the cost of an edge
- consistent implies admissible, but admissible does not imply consistent

On heuristics: Admissible but not Consistent

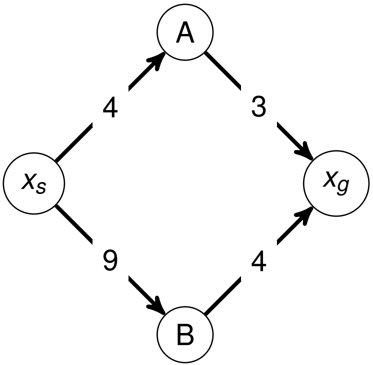


Figure: A weighted directed graph where every edge is associated with a non-negative cost.

Node	h
x_s	6
x_g	0
A	1
B	1

Table: Inconsistent estimate for the graph

Consistent vs Admissible

- admissible is a requirement for correctness
- a consistent heuristic is more efficient (i.e., the algorithm expands less nodes)
- if admissible but not consistent, A* implementation is more complex

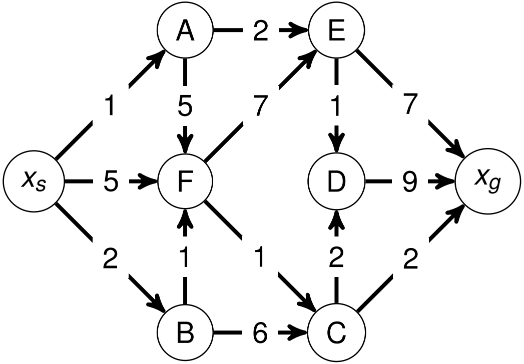
A* with Consistent Heuristic

- must keep two attributes per node: g and $f = g + h$
  - g is always the minimum cost of the shortest path discovered so far
- OPEN is prioritized by f
- everything else is like in Dijkstra's algorithm
- if $h(v) = 0$ for every vertex, A* behaves exactly like Dijkstra's algorithm

A* with Consistent Heuristic

```
Data: G = (V, E), x_s ∈ V, x_g ∈ V, c : E → ℝ_{≥0}
Result: Shortest path from x_s to x_g if it exists, or FAILURE
foreach x ∈ V do
    x.parent ← null ; x.g ← ∞ ; x.f ← ∞;
x_s.g ← 0 ; x_s.f ← x_s.g + h(x_s);
OPEN.initializeEmpty();
CLOSED.initializeEmpty();
OPEN.insert(x_s);
while not OPEN.empty() do
    x ← OPEN.remove();
    CLOSED.insert(x);
    if x = x_g then
        return ExtractPath(x_s, x_g);
    foreach x′ ∈ V such that (x, x′) ∈ E do
        if x′.g = ∞ then
            x′.g ← x.g + c(x, x′);
            x′.f ← x′.g + h(x′);
            x′.parent ← x ;
            OPEN.insert(x′);
        else if x.g + c(x, x′) ≤ x′.g then
            x′.g ← x.g + c(x, x′);
            x′.f ← x′.g + h(x′);
            x′.parent← x ;
            OPEN.rebalance(x′);
return FAILURE;
```

A* Example

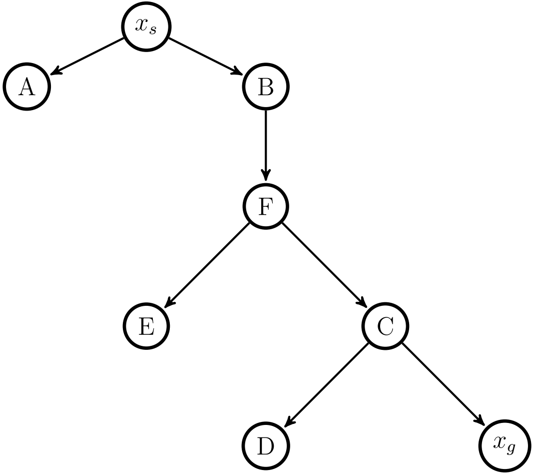


Node	h
x_s	5
x_g	0
A	6
B	3
C	1
D	4
E	4
F	2

A* Example

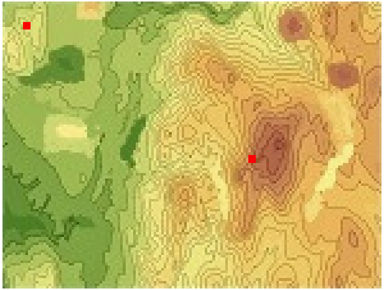
OPEN	x_s	A	B	C	D	E	F	x_g
x_s	$N/0/5$	$N/\infty/\infty$	$N/\infty/\infty$	$N/\infty/\infty$	$N/\infty/\infty$	$N/\infty/\infty$	$N/\infty/\infty$	$N/\infty/\infty$
B, A, F	$N/0/5$	$x_s/1/7$	$x_s/2/5$	$N/\infty/\infty$	$N/\infty/\infty$	$N/\infty/\infty$	$x_s/5/7$	$N/\infty/\infty$
F, A, C	$N/0/5$	$x_s/1/7$	$x_s/2/5$	$B/8/9$	$N/\infty/\infty$	$N/\infty/\infty$	$B/3/5$	$N/\infty/\infty$
C, A, E	$N/0/5$	$x_s/1/7$	$x_s/2/5$	$F/4/5$	$N/\infty/\infty$	$F/10/14$	$B/3/5$	$N/\infty/\infty$
x_g, A, D, E	$N/0/5$	$x_s/1/7$	$x_s/2/5$	$F/4/5$	$C/6/10$	$F/10/14$	$B/3/5$	$C/6/6$

A* Spanning Tree

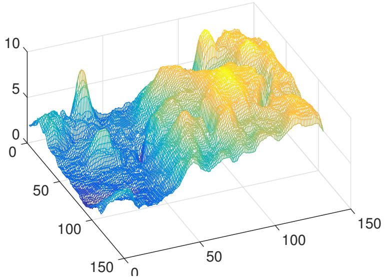


Comparing Different Planners

Elevation map (top view)



Elevation map (lateral view)

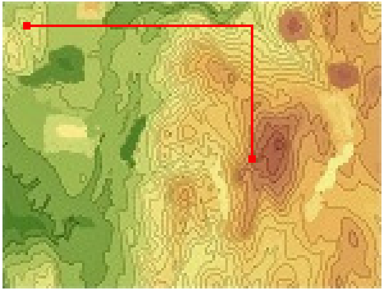


Elevation of a pixel $e(x)$:

Edge cost: downhill: 1; uphill $c(x, x') = 1 + K(e(x') - e(x))$

Comparing Different Planners: BFS

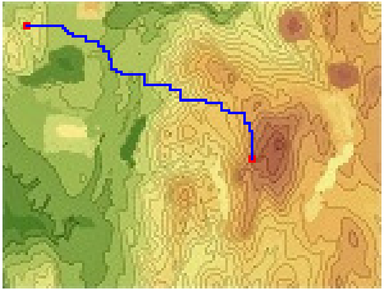
Path found by BFS



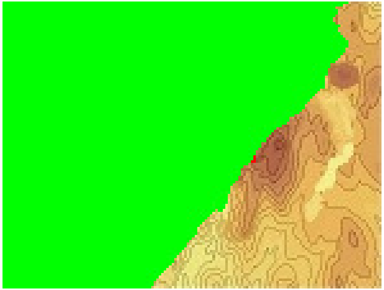
Nodes expanded by BFS

Comparing Different Planners: Dijkstra

Path found by Dijkstra

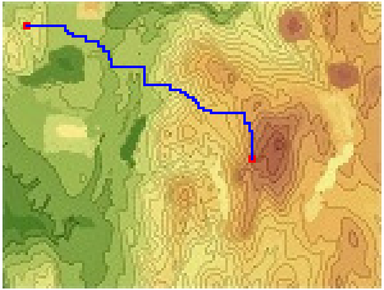


Nodes expanded by Dijkstra

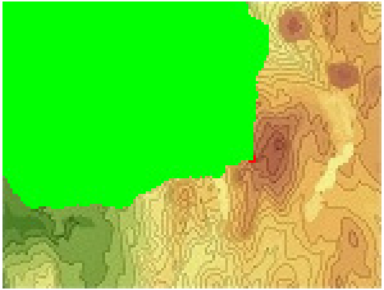


Comparing Different Planners: A*

Path found by A*



Nodes expanded by A*



Navigation Functions

- encode a family of plans
    - for each state identifies the action execute
- intuition: for each state, define a non-negative function: $\psi : V \to \mathbb{R}_{\geq 0}$
- do gradient descent on ψ: among all edges outgoing from x, pick

$$e' = \arg \min_{(x,y) \in e(x)} \{\psi(y)\}$$

- pervasively used for obstacle avoidance (often in combination with other planners)

Navigation Function

Definition

Let $G = (V, E)$ be a planning graph and let $x_g \in V$ a goal vertex. A navigation function $\psi : V \to \mathbb{R}_{\geq 0}$ is *feasible* if it satisfies the following three conditions.

1. $\psi(x_g) = 0$.
2. It $x \in V$ is a vertex from which there is no path to x_g, then $\psi(x) = \infty$.
3. if x is a state from which x_g can be reached, then for $y = e'(x)$ we have $\psi(y) < \psi(x)$.

Navigation Functions

6	5	4	5	6	5
5	4	3			4
4	3	2		2	3
3	2	1	0	1	2
4			1	2	3
5	4	3	2	3	4

Figure: Navigation function for a grid world