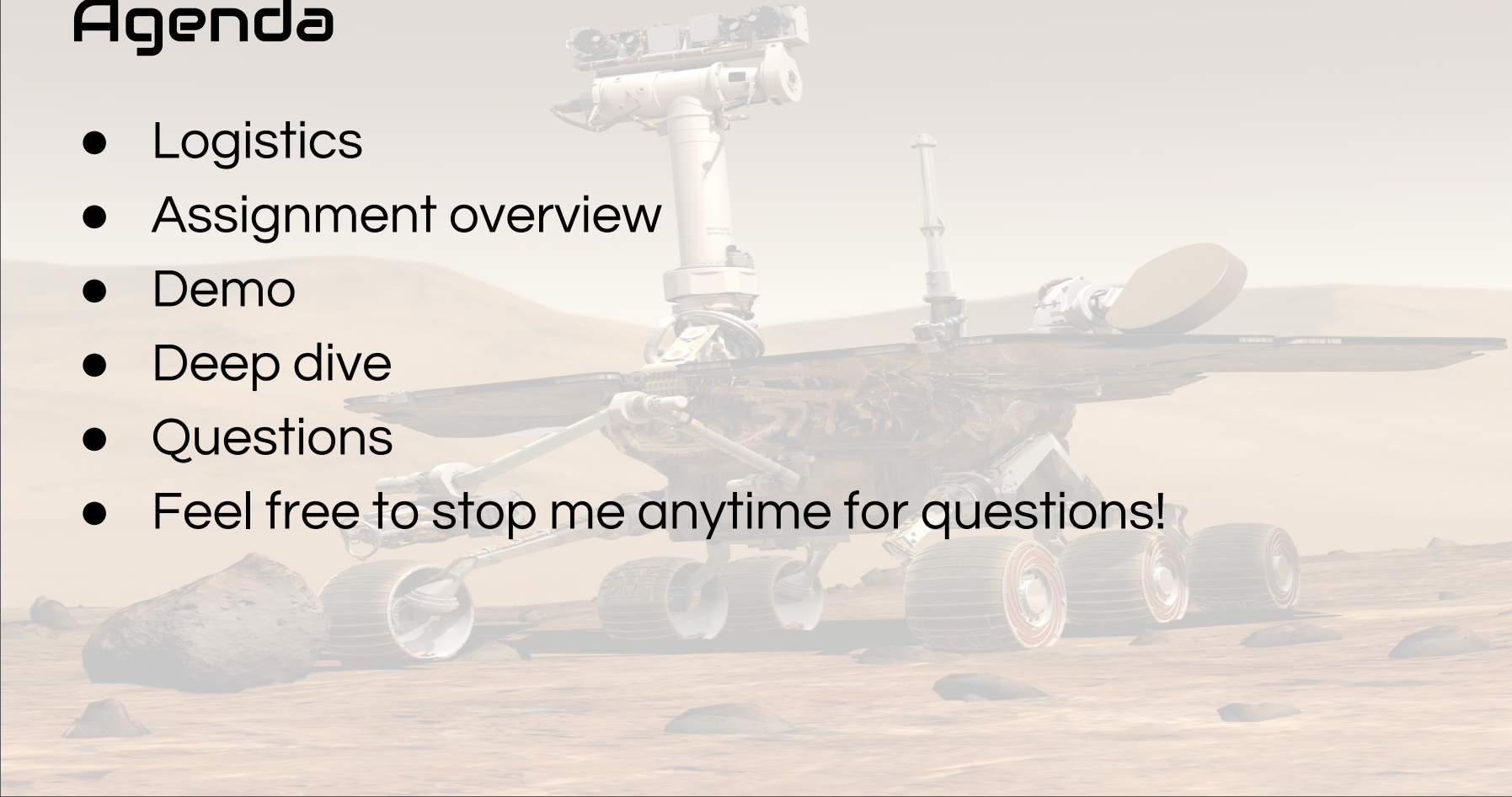
YEAH Hours
Trailblazer
Summer 2017

Aug 9, 2017

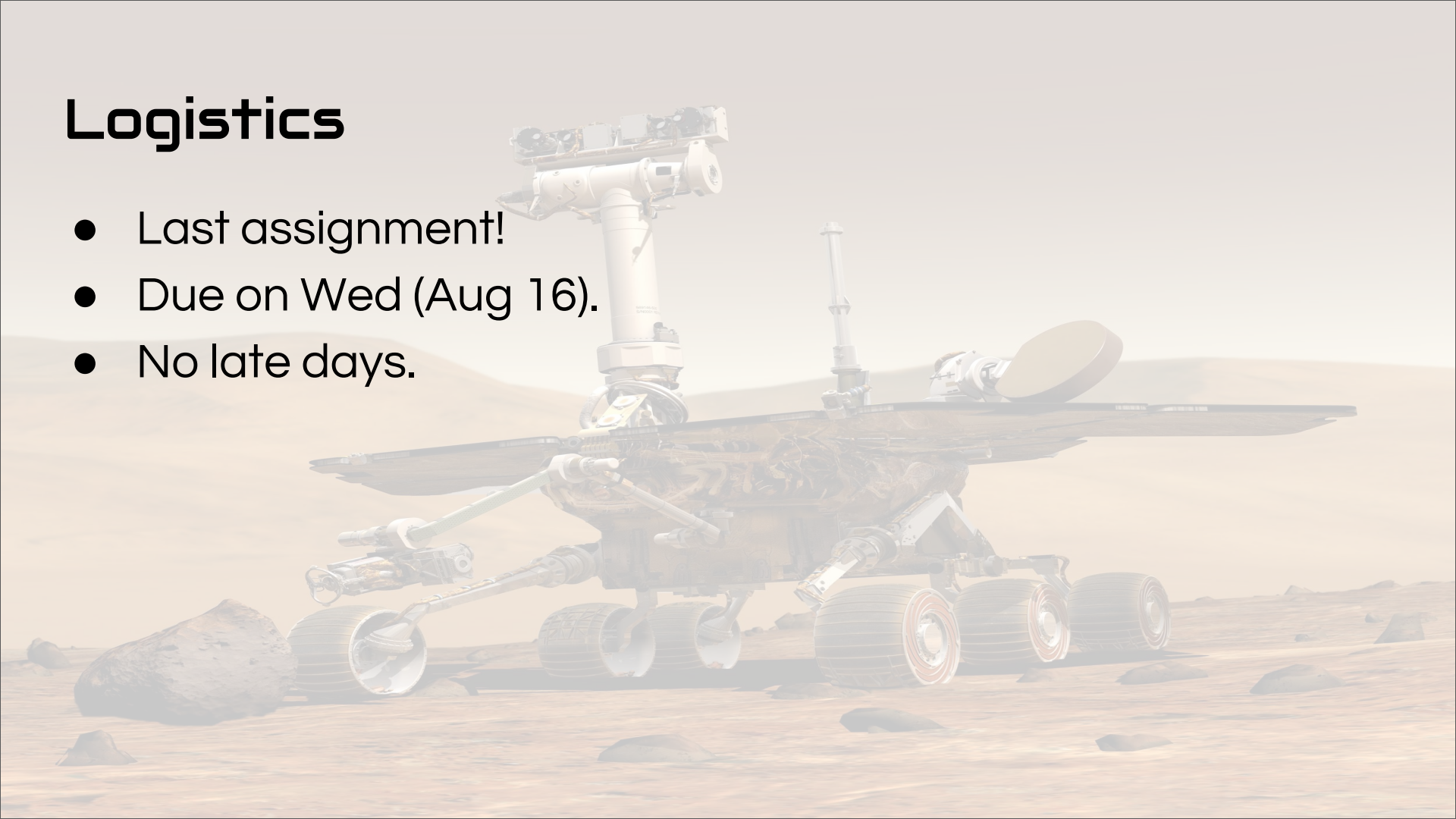
Agenda



- Logistics
- Assignment overview
- Demo
- Deep dive
- Questions
- Feel free to stop me anytime for questions!

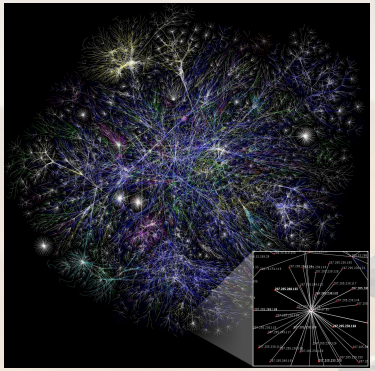
Logistics

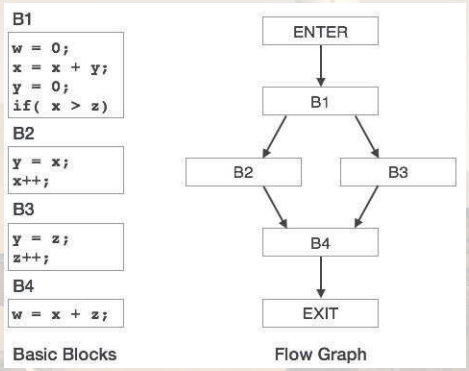
- Last assignment!
- Due on Wed (Aug 16).
- No late days.



Graphs... the final frontier

- They're everywhere

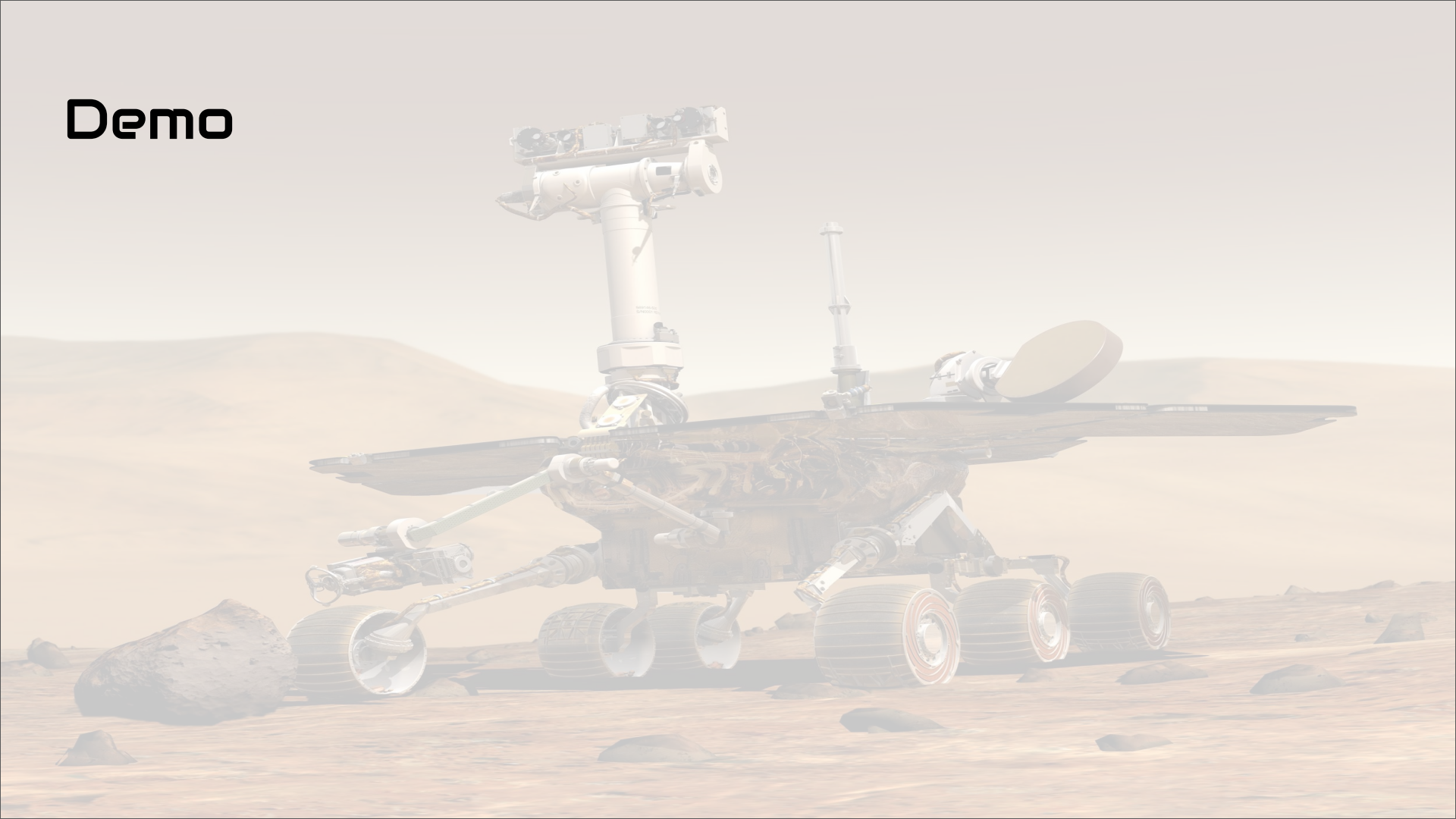




Internet Compilers Scheduling

And so much more - come up in every field

Demo



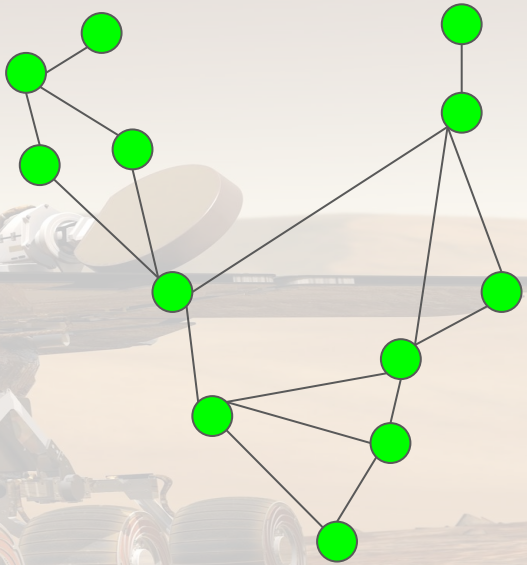
TODO

4 Algorithms
Nice handout here:
http://web.stanford.edu/class/cs106b/handouts/search.html

```
Path breadthFirstSearch(RoadGraph graph, RoadNode* start, RoadNode* end)
Path dijkstrasAlgorithm(RoadGraph graph, RoadNode* start, RoadNode* end)
            Path aStar(RoadGraph graph, RoadNode* start, RoadNode* end)
    Path alternateRoute(RoadGraph graph, RoadNode* start, RoadNode* end)
```

Terminology

- RoadGraph == BasicGraph
- RoadNode == Vertex
- Path == Vector<Vertex*>



RoadGraph

```
class RoadGraph {
    /* Returns the set of all the nodes adjacent to the given node. */
    Set<RoadNode*> neighborsOf(RoadNode* v) const;

    /* Given a start and end node, returns the edge that links them, or
     * nullptr if there is no such edge. */
    RoadEdge* getEdge(RoadNode* start, RoadNode* end) const;

    /* Returns the highest speed permitted on any road in the network. */
    double getMaxRoadSpeed() const;

    /* Returns the "straight-line" distance between the two nodes; that is,
     * the distance between them if you just drew a line connecting them. */
    double getCrowFlyDistance(RoadNode* start, RoadNode* end) const;
};
```

RoadNode

```
class RoadNode {
     // Name of the node, for testing and debugging
    string nodeName() const;

     // Outgoing edges from this node
    Set<RoadEdge*> outgoingEdges() const;

    // Should be one of Color::GRAY, Color::YELLOW, or Color::GREEN
     void setColor(Color color);

     // For debugging
    string toString() const;
};
```

Watching Your Algorithm Progress

- RoadNode Colors
- Grey is default
- Yellow is “enqueued” (BFS, Dijkstra)
- Green is visisted

BFS

```
bfs from v1 to v2:
    create a queue of paths (a vector), q
    q.enqueue(v1 path)
    while q is not empty and v2 is not yet visited:
        path = q.dequeue()
        v = last element in path
        if v is not visited:
            mark v as visited
            if v is the end vertex, we can stop after adding to
                the current path.
            for each unvisited neighbor of v:
                make new path with v's neighbor as last element
                enqueue new path onto q
```

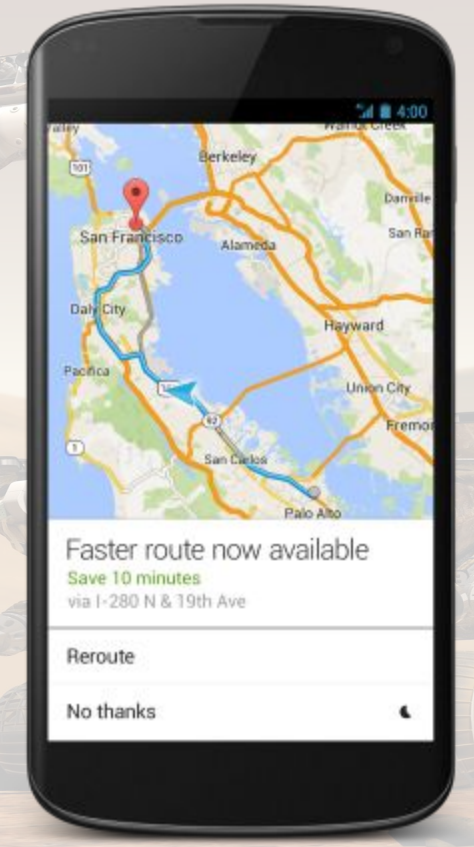
Dijkstra

- Chris will talk about tomorrow
- Search for weighted graph to find shortest path

A*

- Chris will talk about tomorrow
- Think about it like driving with a friend who's local - they have heuristic knowledge about what route to take
- If we know what the expected distance is then we can try paths that get us closer faster
- getCrowFlyDistance and getMaxRoadSpeed useful here

Alternative



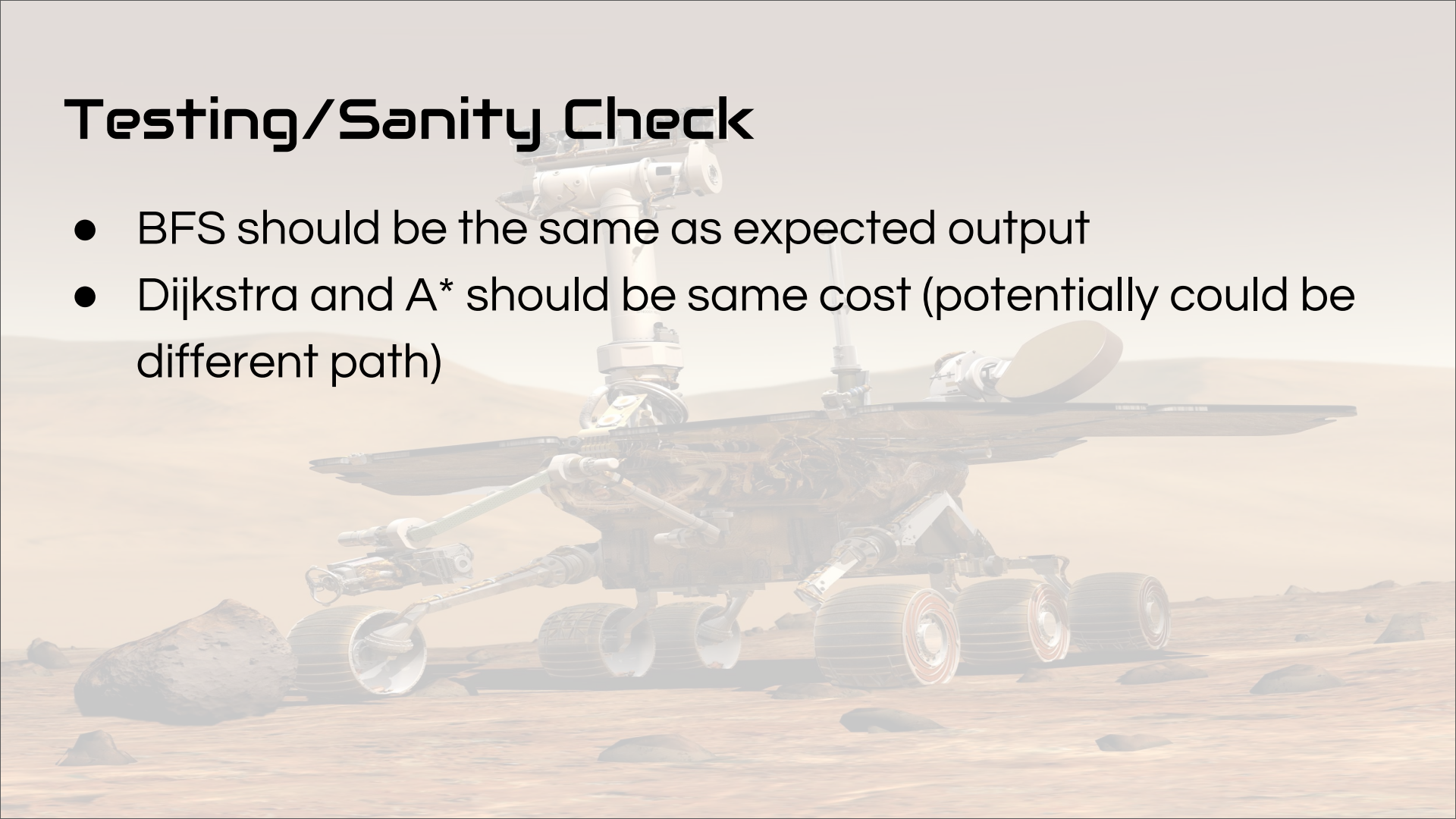
Alternative

- First find the shortest path (i.e. Dijkstra)
- Then remove edges from that path and calculate path that ignores that edge
- Find lowest cost path that's sufficiently different
- Sufficiently different: SUFFICIENT_DIFFERENCE threshold

$$\text{diff} = \frac{\text{\# of nodes in alt. path } \textbf{not} \text{ in main path}}{\text{\# of nodes in alt. path}}$$

Testing/Sanity Check

- BFS should be the same as expected output
- Dijkstra and A* should be same cost (potentially could be different path)



Creative

- Create your own map!

Suggestions

- In order from easiest to hardest
- Pick small map/routes at first and trace through for debugging


Good Luck!