Assignment Lecture 1: Searching and A*

Håkon Måløy

September 17, 2021

Searches, problem formulation

- A set of possible states.

This set contains all the states that the environment can be in. We call it the state space.

Searches, problem formulation

- A set of possible states.
- The initial state.

This is where the agent starts its search from.

Searches, problem formulation

- A set of possible states.
- The initial state.
- A set of one or more goal states.

Sometimes there's only one goal state (e.g. the city we want to go). Other times the goal can be defined by a property that can apply to many states (e.g. no dirt in any location).

Searches, problem formulation

- A set of possible states.
- The initial state.
- A set of one or more goal states.
- The actions available to the agent.

Given a state s, $ACTION(s)$ returns a finite set of actions that can be performed in s.

Searches, problem formulation

- A set of possible states.
- The initial state.
- A set of one or more goal states.
- The actions available to the agent.
- A transition model.

The transition model describes the outcome of each action: $RESULT(s, a)$ returns the state that results from doing action a in state s.

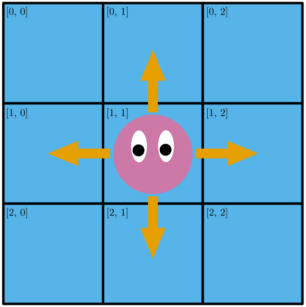
Searches, problem formulation

- A set of possible states.
- The initial state.
- A set of one or more goal states.
- The actions available to the agent.
- A transition model.
- An action cost function.

The action cost function gives the numeric cost of applying action a in state s so as to reach state s'. We denote it $ACTION\text{–}COST(s, a, s')$.

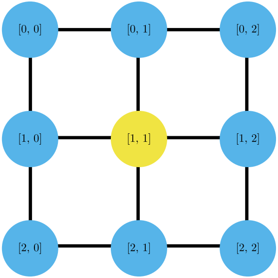
Representing the state space

- We start off from a grid world where we can move in the four cardinal directions, north, east, south and west.



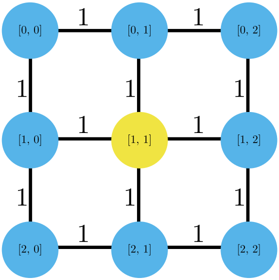
Representing the state space

- We start off from a grid world where we can move in the four cardinal directions, north, east, south and west.
- We represent the state space as a graph, where the vertices are states and the edges between vertices are actions.



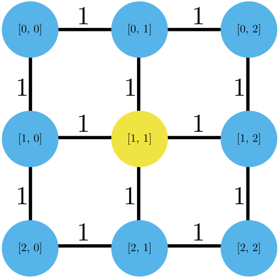
Representing the state space

- We start off from a grid world where we can move in the four cardinal directions, north, east, south and west.
- We represent the state space as a graph, where the vertices are states and the edges between vertices are actions.
- We can also show the cost associated with every action.



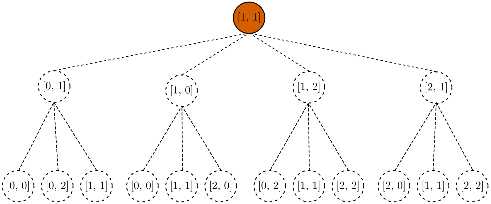
Representing the state space

- We start off from a grid world where we can move in the four cardinal directions, north, east, south and west.
- We represent the state space as a graph, where the vertices are states and the edges between vertices are actions.
- We can also show the cost associated with every action.
- Here, each edge indicates two actions, one in each direction.



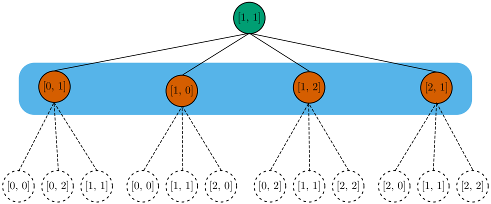
Traversing the search space

- Starting from the inital state the search tree has only one reached node.



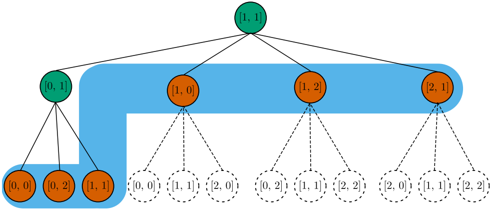
Traversing the search space

- Using the available actions for the initial state we expand to generate its child nodes.



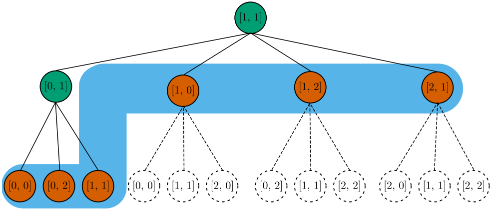
Traversing the search space

- We now choose one of these child nodes and expand it.



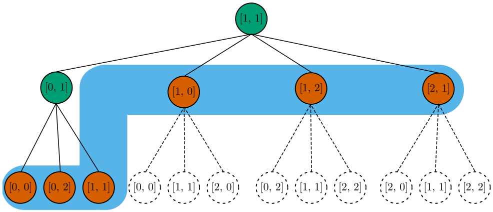
Traversing the search space

- We now choose one of these child nodes and expand it.
- Note the set of 6 unexpanded nodes. This is called the frontier.



Traversing the search space

- We now choose one of these child nodes and expand it.
- Note the set of 6 unexpanded nodes. This is called the frontier.
- Also note that evey child node from the root has the root as a child node.



Searching Algorithms

Which node should we expand?

Searching Algorithms

Which node should we expand?

This is the essence of searching and the main thing that changes in the algorithms we'll explore going forward.

Searching Algorithms - Breadth First Search

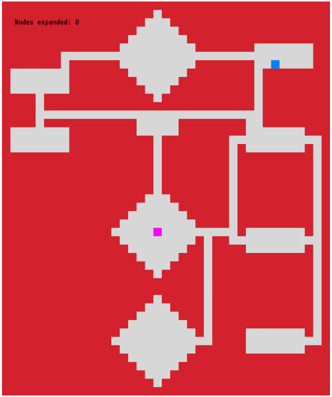
Algorithm 1 The Breadth First Algorithm

```
function BREADTH-FIRST-SEARCH(problem)
    node ← NODE(problem.INITIAL)
    if problem.IS-GOAL(node.STATE) then return node
    frontier ← a FIFO queue
    while not IS-EMPTY(frontier) do
        node ← POP(frontier)
        for each child in EXPAND(problem, node) do
            s ← child.STATE
            if problem.IS-GOAL(s) then return child
              add child to frontier

    return failure
function EXPAND(problem, node)
    s ← node.STATE
    for each action in problem.ACTIONS(s) do
        s′ ← problem.RESULT(s, action)
        cost ← node.PATH-COST + problem.ACTION-COST(s, action, s′)
        yield NODE(STATE = s', PARENT = node, ACTION = action, PATH-COST = cost)
```

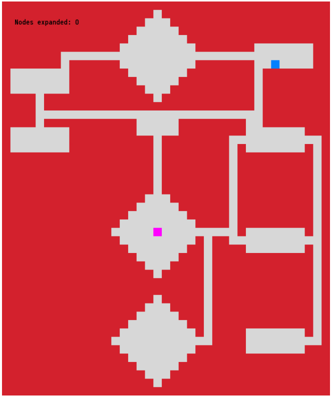
Searching Algorithms - Defining The Search Problem

- Initial state:
- Goal states:
- Actions:
- Transition model:
- Action cost function:
- State space:



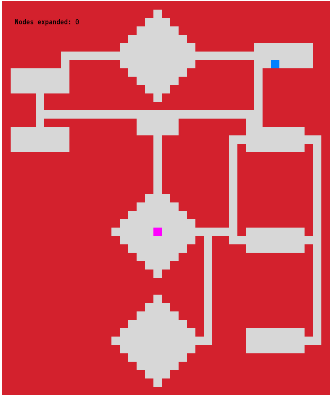
Searching Algorithms - Defining The Search Problem

- Initial state: The start position (pink square) [18, 27].
- Goal states:
- Actions:
- Transition model:
- Action cost function:
- State space:



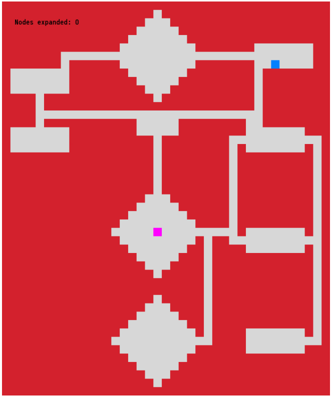
Searching Algorithms - Defining The Search Problem

- Initial state: The start position (pink square) [18, 27].
- Goal states: The goal position (blue square) [32, 7].
- Actions:
- Transition model:
- Action cost function:
- State space:



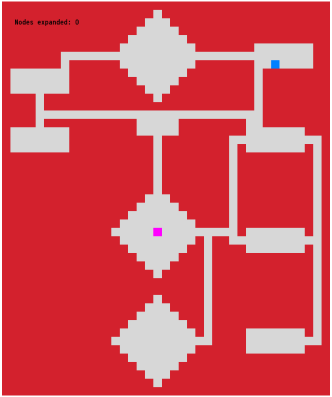
Searching Algorithms - Defining The Search Problem

- Initial state: The start position (pink square) [18, 27].
- Goal states: The goal position (blue square) [32, 7].
- Actions: Movement in the cardinal directions (if no obstacle in the way): North, East, South and West.
- Transition model:
- Action cost function:
- State space:



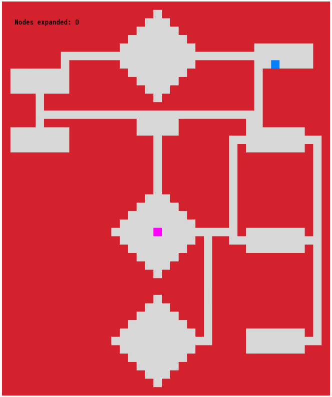
Searching Algorithms - Defining The Search Problem

- Initial state: The start position (pink square) [18, 27].
- Goal states: The goal position (blue square) [32, 7].
- Actions: Movement in the cardinal directions (if no obstacle in the way): North, East, South and West.
- Transition model: The result of moving in a direction. E.g $RESULT([18, 27], EAST) = [19, 27]$.
- Action cost function:
- State space:



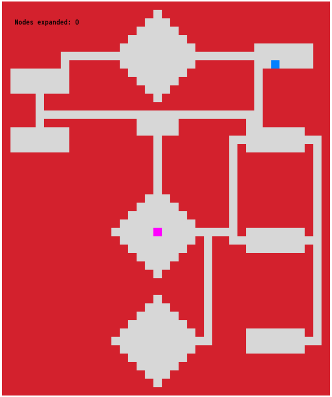
Searching Algorithms - Defining The Search Problem

- Initial state: The start position (pink square) [18, 27].
- Goal states: The goal position (blue square) [32, 7].
- Actions: Movement in the cardinal directions (if no obstacle in the way): North, East, South and West.
- Transition model: The result of moving in a direction. E.g $RESULT([18, 27], EAST) = [19, 27]$.
- Action cost function: The cost of making a move. Here all costs are 1.
- State space:



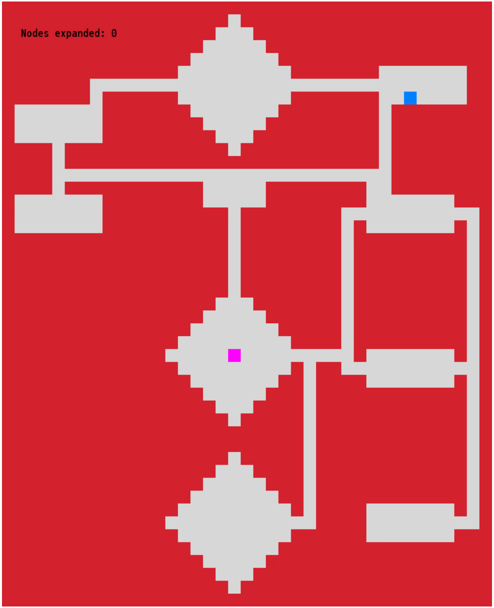
Searching Algorithms - Defining The Search Problem

- Initial state: The start position (pink square) [18, 27].
- Goal states: The goal position (blue square) [32, 7].
- Actions: Movement in the cardinal directions (if no obstacle in the way): North, East, South and West.
- Transition model: The result of moving in a direction. E.g $RESULT([18, 27], EAST) = [19, 27]$.
- Action cost function: The cost of making a move. Here all costs are 1.
- State space: All legal positions on the board.



Searching Algorithms - Breadth First Search

How will the algorithm progress?



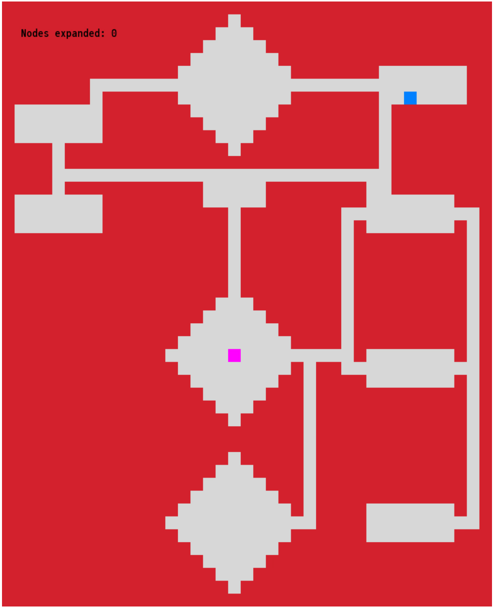
Searching Algorithms - Breadth First Search Graph Edition

Algorithm 2 The Breadth First Algorithm - Graph Search

```
function BREADTH-FIRST-SEARCH(problem)
    node ← NODE(problem.INITIAL)
    if problem.IS-GOAL(node.STATE) then return node
    frontier ← a FIFO queue
    reached ← {problem.INITIAL}
    while not IS-EMPTY(frontier) do
        node ← POP(frontier)
        for each child in EXPAND(problem, node) do
            s ← child.STATE
            if problem.IS-GOAL(s) then return child
            if s not in reached then
                add s to reached
                add child to frontier
    return failure
function EXPAND(problem, node)
    s ← node.STATE
    for each action in problem.ACTIONS(s) do
        s′ ← problem.RESULT(s, action)
        cost ← node.PATH-COST + problem.ACTION-COST(s, action, s′)
        yield NODE(STATE = s', PARENT = node, ACTION = action, PATH-COST = cost)
```

Searching Algorithms - Breadth First Search Graph Edition

How will the algorithm progress?



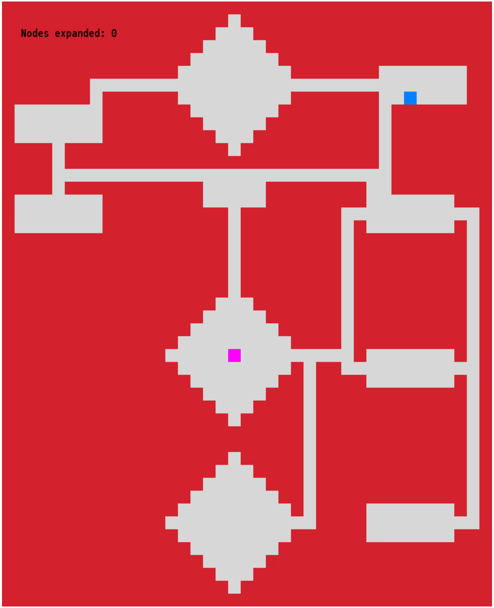
Searching Algorithms - Depth First Search Graph Edition

Algorithm 3 The Depth First Algorithm - Graph Search

```
function DEPTH-FIRST-SEARCH(problem)
    node ← NODE(problem.INITIAL)
    if problem.IS-GOAL(node.STATE) then return node
    frontier ← a LIFO queue
    reached ← {problem.INITIAL}
    while not IS-EMPTY(frontier) do
        node ← POP(frontier)
        for each child in EXPAND(problem, node) do
            s ← child.STATE
            if problem.IS-GOAL(s) then return child
            if s not in reached then
                add s to reached
                add child to frontier
    return failure
function EXPAND(problem, node)
    s ← node.STATE
    for each action in problem.ACTIONS(s) do
        s′ ← problem.RESULT(s, action)
        cost ← node.PATH-COST + problem.ACTION-COST(s, action, s′)
        yield NODE(STATE = s', PARENT = node, ACTION = action, PATH-COST = cost)
```

Searching Algorithms - Depth First Search Graph Edition

How will the algorithm progress?



Searching Algorithms - Can We Do Better?

Can we improve on BFS and DFS?

Searching Algorithms - Can We Do Better?

Can we improve on BFS and DFS?

Both Breadth First Search and Depth First Search expand a lot of nodes. How can we avoid this?

Searching Algorithms - Can We Do Better?

Can we improve on BFS and DFS?

Both Breadth First Search and Depth First Search expand a lot of nodes. How can we avoid this?

Introduce information about the progress!

Searching Algorithms - Can We Do Better?

Can we improve on BFS and DFS?

Both Breadth First Search and Depth First Search expand a lot of nodes. How can we avoid this?

Introduce information about the progress!

If we introduce some information about our progress (direction), the algorithm can avoid exploring unpromising nodes.

Searching Algorithms - Introducing Information

Heuristic

A heuristic gives the algorithm some information about whether or not it is heading in the right direction. It is often an estimate of how far the algorithm is from the goal.

Searching Algorithms - Introducing Information

Heuristic

A heuristic gives the algorithm some information about whether or not it is heading in the right direction. It is often an estimate of how far the algorithm is from the goal.

Useful Heuristic Functions:

A useful heuristic function should easy to calculate and sufficiently accurate for short-term goal approximation.

Searching Algorithms - Introducing Information

Heuristic

A heuristic gives the algorithm some information about whether or not it is heading in the right direction. It is often an estimate of how far the algorithm is from the goal.

Useful Heuristic Functions:

A useful heuristic function should easy to calculate and sufficiently accurate for short-term goal approximation.

- Manhattan Distance

Searching Algorithms - Introducing Information

Heuristic

A heuristic gives the algorithm some information about whether or not it is heading in the right direction. It is often an estimate of how far the algorithm is from the goal.

Useful Heuristic Functions:

A useful heuristic function should easy to calculate and sufficiently accurate for short-term goal approximation.

- Manhattan Distance
- Euclidian Distance

Searching Algorithms - A*

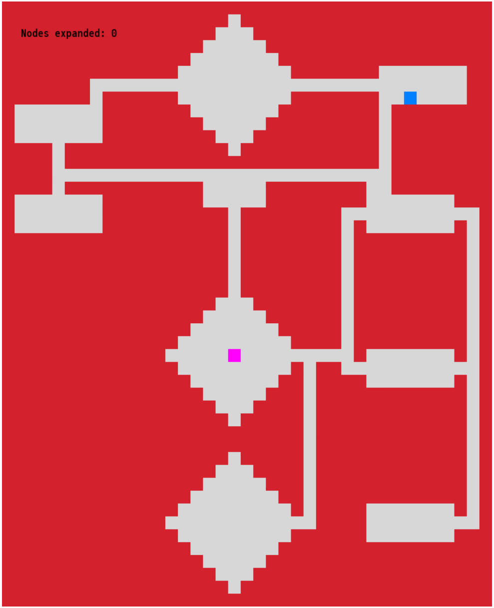
The same old algorithm, but with a priority queue and a heuristic function.

Algorithm 4 The A* Algorithm

```
function BREADTH-FIRST-SEARCH(problem)
    node ← NODE(problem.INITIAL)
    if problem.IS-GOAL(node.STATE) then return node
    frontier ← a PRIORITY queue ordered by f
    reached ← {problem.INITIAL}
    while not IS-EMPTY(frontier) do
        node ← POP(frontier)
        for each child in EXPAND(problem, node) do
            s ← child.STATE
            if problem.IS-GOAL(s) then return child
            if s not in reached and then
                add s to reached
                add child to frontier
    return failure
function EXPAND(problem, node)
    s ← node.STATE
    for each action in problem.ACTIONS(s) do
        s′ ← problem.RESULT(s, action)
        cost ← node.PATH-COST + problem.ACTION-COST(s, action, s′)
        heuristic ← problem.HEURISTIC-FN(s′, goal−node)
        f(s′) = cost + heuristic
        yield NODE(STATE = s′, PARENT = node, ACTION = action, PATH-COST = cost, f = f(s′))
```

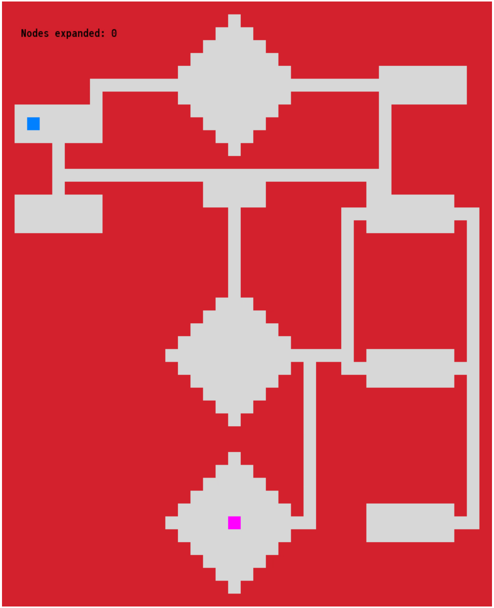
Searching Algorithms - A*

How will the algorithm progress?



Searching Algorithms - A* going the wrong way

How will the algorithm progress?



Searching Algorithms - Heuristics contd

Can we use any heuristic?

Searching Algorithms - Heuristics contd

Can we use any heuristic? NO

Searching Algorithms - Heuristics contd

Can we use any heuristic? NO A heuristic must be both admissible and consistent

Searching Algorithms - Heuristics contd

Admissibility

- An admissible heuristic is one that never overestimates the cost to reach the goal.

Searching Algorithms - Heuristics contd

Admissibility

- An admissible heuristic is one that never overestimates the cost to reach the goal.
- An example of an admissible heuristic is the Euclidian Distance.

Searching Algorithms - Heuristics contd

Admissibility

- An admissible heuristic is one that never overestimates the cost to reach the goal.
- An example of an admissible heuristic is the Euclidian Distance.
- The shortest point between two points is always a straight line.

Searching Algorithms - Heuristics contd

Consistency

- A heuristic is consistent if, for every node n and every successor n' of n generated by an action a, the estimated cost of reaching the goal from n is no greater than the step cost of getting to n' plus the estimated cost of reaching the goal from n'.

Searching Algorithms - Heuristics contd

Consistency

- A heuristic is consistent if, for every node n and every successor n' of n generated by an action a, the estimated cost of reaching the goal from n is no greater than the step cost of getting to n' plus the estimated cost of reaching the goal from n'.
- $h(n) \leq c(n, a, n') + h(n')$

Searching Algorithms - Heuristics contd

Consistency

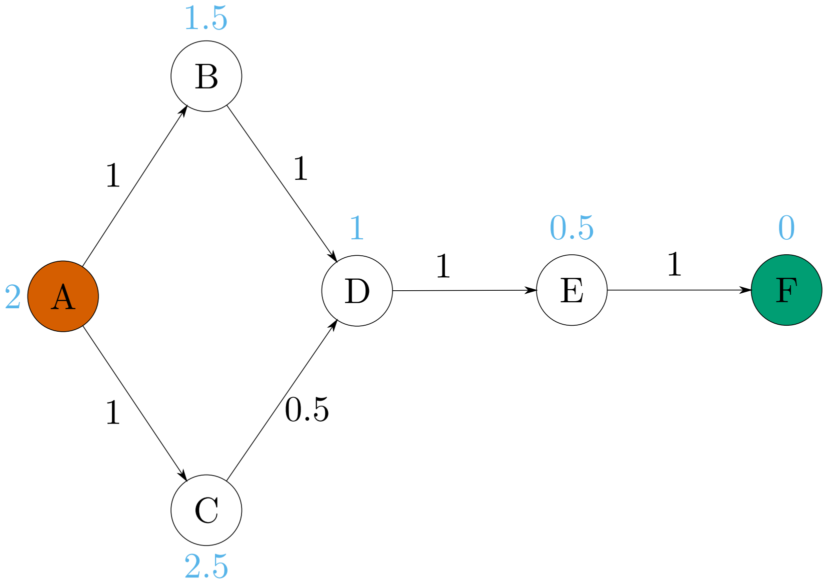
- A heuristic is consistent if, for every node n and every successor n' of n generated by an action a, the estimated cost of reaching the goal from n is no greater than the step cost of getting to n' plus the estimated cost of reaching the goal from n'.
- $h(n) \leq c(n, a, n') + h(n')$
- This is a form of the general triangle inequality where the triangle is formed by n, n' and the goal node G_n closest to n.

Searching Algorithms - Heuristics contd

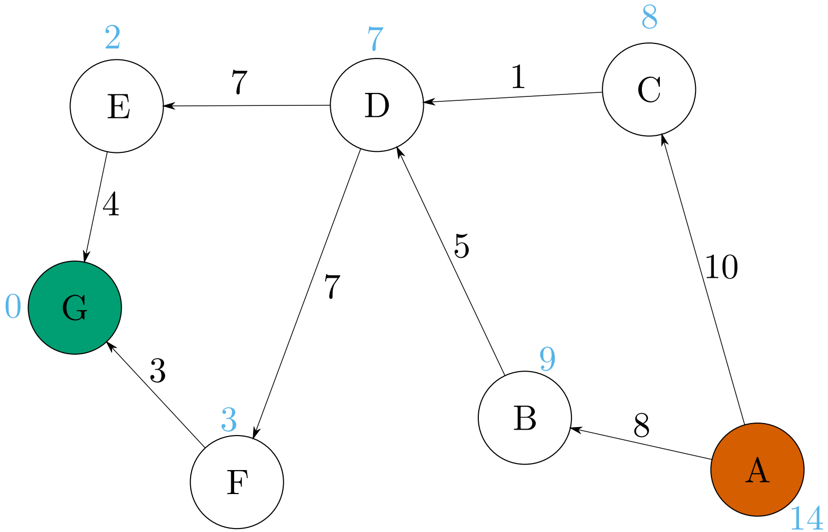
Consistency

- A heuristic is consistent if, for every node n and every successor n' of n generated by an action a, the estimated cost of reaching the goal from n is no greater than the step cost of getting to n' plus the estimated cost of reaching the goal from n'.
- $h(n) \leq c(n, a, n') + h(n')$
- This is a form of the general triangle inequality where the triangle is formed by n, n' and the goal node G_n closest to n.
- Every consistent heuristic is also admissible.

Searching Algorithms - A* Inconsistent Example



Searching Algorithms - A* Example



Assignment 2 - Problem overview

Task	Description
Part 1	**Pathfinding at Samfundet**
Task 1	Find the least costly path from Rundhallen to Strossa
Task 2	Find the least costly path from Strossa to Selskapssiden
Part 2	**Grids with different cell costs**
Task 3	Find the least costly path from Lyche to Klubben
Task 4	Find the least costly path from Lyche to Klubben with Edgar packed
Part 3	**Moving goal**(Optional)
Task 5	Find the least costly path to a moving goal

You need an A* implementation

You can either:

You need an A* implementation

You can either:

- Write it yourself (**recommended**)

You need an A* implementation

You can either:

- Write it yourself (**recommended**)
- Get it online

You need an A* implementation

You can either:

- Write it yourself (**recommended**)
- Get it online

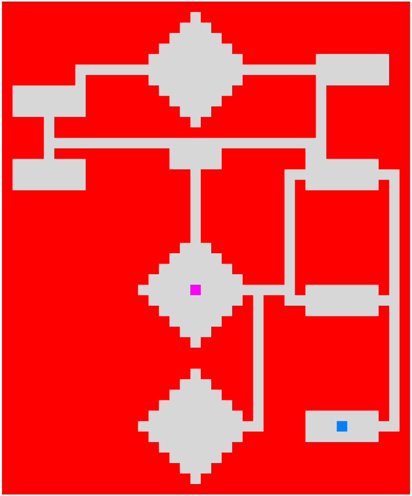
The book source code may be useful as inspiration for how to organize your code: `https://github.com/aimacode`

Grids with obstacles

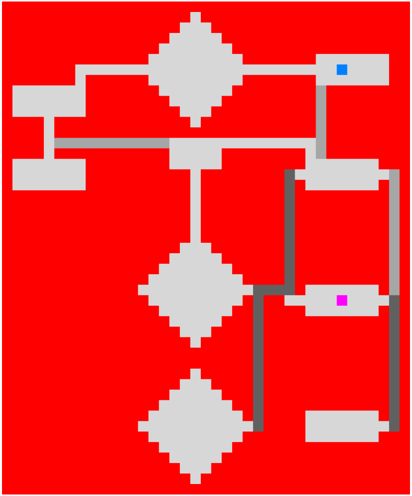
Table: Cell types and their associated costs.

	Char.	Description	Cost
	.	Flat Ground	1
	,	Stairs	2
	:	Packed Stairs	3
	;	Packed Room	4

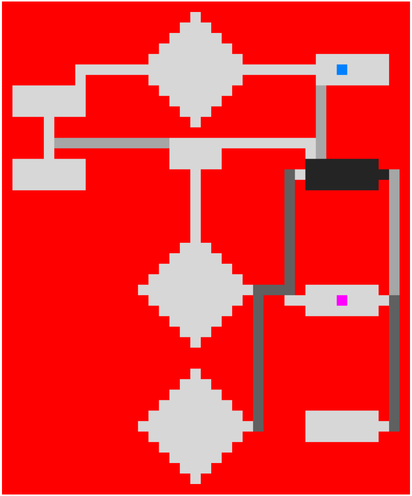
Three Versions of Samfundet



Three Versions of Samfundet



Three Versions of Samfundet



Deliverables

Deliverables

- Well-commented source code for a program that finds least-cost paths for the different boards and that visualizes the results.

Deliverables

- Well-commented source code for a program that finds least-cost paths for the different boards and that visualizes the results.
- Visualizations of the least-cost paths for the two tasks above.

Deliverables

- Well-commented source code for a program that finds least-cost paths for the different boards and that visualizes the results.
- Visualizations of the least-cost paths for the two tasks above.
- Delivery due: 30.09.2021

Programming Tips

Implement nodes as classes:

Programming Tips

Implement nodes as classes:

- Makes it possible to store node F-values and compare them on the fly.

Programming Tips

Implement nodes as classes:

- Makes it possible to store node F-values and compare them on the fly.
- We can store the parent of the node, enabling easy roll-up to find the best path.

Programming Tips

Use heapq.heapify as priority queue datastructure:

Programming Tips

Use heapq.heapify as priority queue datastructure:

- Automatically sorts entries on insertion.

Programming Tips

Use heapq.heapify as priority queue datastructure:

- Automatically sorts entries on insertion.
- You only need to implement comparison code in your node class:

```
def __lt__(self, other):
def __gt__(self, other):
def __eq__(self, other):
```

Programming Tips

Manhattan distance works well as an heuristic in this assignment.

Programming Tips

Manhattan distance works well as an heuristic in this assignment.

- We are working in a grid world, we can therefore be certain that the heuristic is admissible.

Programming Tips

Manhattan distance works well as an heuristic in this assignment.

- We are working in a grid world, we can therefore be certain that the heuristic is admissible.
- Proof: Remove the walls. Is there any legal path that gets to the goal node faster than the manhattan distance?

Good Luck

Good Luck