```
{
 "cells": [
  {
   "attachments": {},
   "cell_type": "markdown",
   "id": "897b9543-4275-417c-8f35-879379f897e9",
   "metadata": {},
   "source": [
    "# M3 Project\n",
    "\n",
    "In this project, you will compare the performance of uninformed
and informed search algorithms for the 4-sided dominoes problem. You
will be given N<sup>2</sup> tiles (N = {2,3,4}) and will be asked to
arrange them in an NxN grid in a way that adjacent tiles have the same
number in their neighboring sides. The tiles cannot be rotated. See
the example below for N=3.\n",
    "\n",
    "<img src=\"m3project.png\" width=\"400\"/>\n",
    "\n",
    "An initial version of the code with the problem specification
(below) and a report template (at the bottom) are available in this
notebook. Deliverables are the final code (non-functioning code is
worth 0 points) and the comparison report.\n",
    "\n",
    "Solve the task above using:\n",
    "- one uninformed search algorithm of your choice (20pts)\n",
    "- one informed search algorithm of your choice (20pts)\n",
    "\n",
    "For your solution, describe the:\n",
    "- search state space (10pts)\n",
    "- successor function (10pts)\n",
    "- heuristic function for the informed search (10pts)\n",
    "\n",
    "Run each algorithm with at least 10 different initial states for
each value of N, and compare the performance of the chosen algorithms
for different puzzle sizes in terms of:\n",
    "- average number of expanded states (15pts)\n",
    "- success rate (15pts)\n",
    "\n",
    "You are free to set a maximum number of expanded states for each
algorithm, and return failure in case this number is reached."
   ]
  },
  {
   "attachments": {},
   "cell_type": "markdown",
   "id": "2234bcfb-d702-43e8-a922-26b790557075",
   "metadata": {},
   "source": [
    "# Implementation\n",
     "\n",
     "You are free to change the code below as needed."
    ]
   },
   {
    "cell_type": "code",
    "execution_count": null,
    "id": "d881f577-3977-4914-ab97-3814209e3f1b",
    "metadata": {},
    "outputs": [],
    "source": [
     "import random\n",
     "import queue"
    ]
   },
   {
    "cell_type": "code",
    "execution_count": null,
    "id": "c2b20176-5987-43f3-b2d0-f0e4aac9c4c0",
    "metadata": {},
    "outputs": [],
    "source": [
     "class Domino():\n",
     "    \"\"\"\n",
     "    Implementation of a 4-sided domino tile.\n",
     "\n",
     "    Methods\n",
     "    -------\n",
     "    is_above(other)\n",
     "        Checks of the tile is above the other tile.\n",
     "    is_under(other)\n",
     "        Checks of the tile is under the other tile.\n",
     "    is_on_the_left_of(other)\n",
     "        Checks of the tile is on the left of the other tile.\n",
     "    is_on_the_right_of(other)\n",
     "        Checks of the tile is on the right of the other tile.\n",
     "    \"\"\"\n",
     "    def __init__(self, top: int, right: int, bottom: int, left: 
int):\n",
     "        assert isinstance(top, int) and isinstance(right, int) 
and isinstance(bottom, int) and isinstance(left, int), \"Invalid tile 
value!\"\n",
     "        self.top = top\n",
     "        self.right = right\n",
     "        self.bottom = bottom\n",
     "        self.left = left\n",
     "\n",
     "    def is_above(self, other):\n",
     "        assert isinstance(other, Domino), \"Invalid tile type!
\"\n",
    "        return self.bottom == other.top\n",
    "    \n",
    "    def is_under(self, other):\n",
    "        assert isinstance(other, Domino), \"Invalid tile type!
\"\n",
    "        return self.top == other.bottom\n",
    "    \n",
    "    def is_on_the_left_of(self, other):\n",
    "        assert isinstance(other, Domino), \"Invalid tile type!
\"\n",
    "        return self.right == other.left\n",
    "    \n",
    "    def is_on_the_right_of(self, other):\n",
    "        assert isinstance(other, Domino), \"Invalid tile type!
\"\n",
    "        return self.left == other.right"
   ]
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "id": "125f9fb7-41f2-4e75-b5b3-2e6ff769234c",
   "metadata": {},
   "outputs": [],
   "source": [
    "\"\"\"\n",
    "This implementation is provided as a starting point. Feel free to
change it as needed.\n",
    "\"\"\"\n",
    "class FourDominoes():\n",
    "    \"\"\"\n",
    "    Implementation of the 4-sided dominoes puzzle.\n",
    "\n",
    "    Methods\n",
    "    -------\n",
    "    show()\n",
    "        Visualize the current state.\n",
    "    move(action)\n",
    "        Apply an action to the current state.\n",
    "    successor(state)\n",
    "        Finds the list of successors for a given state.\n",
    "    goal_test(state)\n",
    "        Checks if the current state is a goal state.\n",
    "    \"\"\"\n",
    "\n",
    "    def __init__(self, N, state=None):\n",
    "        \"\"\"\n",
    "        Parameters\n",
    "        ----------\n",
    "        N\n",
    "            Grid size (puzzle contains N^2 tiles)\n",
    "        state\n",
    "            Initial state configuration. If None is provided, a
random one is created.\n",
    "        \"\"\"\n",
    "        assert N >= 2 and N <= 4, \"Invalid grid size!\"\n",
    "        self.N = N\n",
    "\n",
    "        if state is not None:\n",
    "            assert len(state) == self.N, \"Invalid state size!
\"\n",
    "            for row in state:\n",
    "                assert len(row) == self.N, \"Invalid state size!
\"\n",
    "                for tile in row:\n",
    "                    assert isinstance(tile, Domino), \"Invalid
state type!\"\n",
    "            self.state = state\n",
    "        else:\n",
    "            self.state = self.__get_random_state()\n",
    "    \n",
    "    def __get_random_state(self):\n",
    "        \"\"\"\n",
    "        Generates a random puzzle configuration (grid of
dominoes)\n",
    "\n",
    "        Return\n",
    "        ----------\n",
    "        tuple\n",
    "            A tuple describing a unique puzzle configuration.\n",
    "        \"\"\"\n",
    "        temp = []\n",
    "        for i in range(self.N):\n",
    "            for j in range(self.N):\n",
    "                domino = Domino(\n",
    "                    random.randint(1,9) if i == 0 else
temp[(i-1)*self.N+j].bottom, # top\n",
    "                    random.randint(1,9),
# right\n",
    "                    random.randint(1,9),
# bottom\n",
    "                    random.randint(1,9) if j == 0 else
temp[i*self.N+j-1].right     #left\n",
    "                )\n",
    "                temp.append(domino)\n",
    "        random.shuffle(temp) # if you comment this line, the
state will be a final state (solution)\n",
    "        return tuple(tuple(temp[i*self.N:(i+1)*self.N]) for i in
range(self.N))\n",
    "\n",
    "    def show(self):\n",
    "        \"\"\"\n",
    "        Prints the current state.\n",
    "        \"\"\"\n",
    "        #line\n",
    "        print('╓', end='')\n",
    "        for i in range(self.N):\n",
    "            print('═══', end='╥' if i < self.N-1 else '╖\\n')\n",
    "        for i in range(self.N):\n",
    "            # first line of tile\n",
    "            print('║', end='')\n",
    "            for j in range(self.N):\n",
    "                print('╲{}╱║'.format(self.state[i][j].top), 
end='' if j < self.N-1 else '\\n')\n",
    "            # second line of tile\n",
    "            print('║', end='')\n",
    "            for j in range(self.N):\n",
    "                print('{}╳{}║'.format(self.state[i][j].left, 
self.state[i][j].right), end='' if j < self.N-1 else '\\n')\n",
    "            # third line of tile\n",
    "            print('║', end='')\n",
    "            for j in range(self.N):\n",
    "                print('╱{}╲║'.format(self.state[i][j].bottom), 
end='' if j < self.N-1 else '\\n')\n",
    "            # line\n",
    "            print('╟' if i < self.N-1 else '╙', end='')\n",
    "            for j in range(self.N):\n",
    "                print('═══', end='╫' if i < self.N-1 and j < 
self.N-1 else '╨' if i == self.N-1 and j < self.N-1 else '╢\\n' if i < 
self.N-1 else '╜\\n')\n",
    "\n",
    "    def move(self, action):\n",
    "        \"\"\"\n",
    "        Uses a given action to update the current state of the 
puzzle. Assumes the action is valid.\n",
    "\n",
    "        Parameters\n",
    "        ----------\n",
    "        action\n",
    "            Tuple with coordinates of two tiles to be swapped 
((row1,col1), (row2,col2))\n",
    "        \"\"\"\n",
    "        assert len(action) == 2 and all(len(coord) == 2 for coord 
in action) and all(isinstance(x, int) and x >= 0 and x < self.N for 
coord in action for x in coord), \"Invalid action!\"\n",
    "        (r1, c1), (r2, c2) = action\n",
    "        temp = [list(row) for row in self.state]\n",
    "        temp[r1][c1], temp[r2][c2] = temp[r2][c2], temp[r1]
[c1]\n",
    "        self.state = tuple(tuple(x) for x in temp)\n",
    "\n",
    "    def successor(self, state):\n",
    "        \"\"\"\n",
    "        Finds the list of successors for a given state.\n",
    "\n",
    "        Parameters\n",
    "        ----------\n",
    "        state\n",
    "            A tuple describing a unique puzzle configuration.\n",
    "\n",
    "        Returns\n",
    "        -------\n",
    "        list\n",
    "            A list of pairs (action,state) with all states that
can be reached from the given state with a single action.\n",
    "        \"\"\"\n",
    "\n",
    "        successors = []\n",
    "        for i in range(self.N*self.N):\n",
    "            r1 = i//self.N\n",
    "            c1 = i%self.N\n",
    "            for j in range(i+1, self.N*self.N):\n",
    "                r2 = j//self.N\n",
    "                c2 = j%self.N\n",
    "                action = ((r1,c1), (r2,c2))\n",
    "                copy = FourDominoes(self.N, state)\n",
    "                copy.move(action)\n",
    "                successors.append((action,copy.state))\n",
    "        return successors\n",
    "\n",
    "    def goal_test(self, state):\n",
    "        \"\"\"\n",
    "        Checks if the given state is a goal state.\n",
    "\n",
    "        Parameters\n",
    "        ----------\n",
    "        state\n",
    "            A tuple describing a unique puzzle configuration.\n",
    "\n",
    "        Returns\n",
    "        -------\n",
    "        bool\n",
    "             True if the given state is a goal state, and False
otherwise.\n",
    "        \"\"\"\n",
    "\n",
    "        for i in range(self.N):\n",
    "            for j in range(self.N):\n",
    "                if i > 0 and not state[i][j].is_under(state[i-1]
[j]):\n",
    "                    return False\n",
    "                if j > 0 and not state[i]
[j].is_on_the_right_of(state[i][j-1]):\n",
    "                    return False\n",
    "        return True"
   ]
  },
  {
   "attachments": {},
   "cell_type": "markdown",
   "id": "3666be28",
   "metadata": {},
   "source": [
    "Additional functions."
   ]
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "id": "0fe57e60",
   "metadata": {},
   "outputs": [],
   "source": [
    "def bfs(initial_state):\n",
    "    \"\"\"\n",
    "    Implementation of the Breadth-First Search algorithm.\n",
    "    \"\"\"\n",
    "    frontier = queue.Queue()\n",
    "    frontier.put(([], initial_state))\n",
    "    explored = set()\n",
    "\n",
    "    while not frontier.empty():\n",
    "        path, current_state = frontier.get()\n",
    "        if current_state.goal_test(current_state.state):\n",
    "            return path, current_state\n",
    "\n",
    "        explored.add(current_state.state)\n",
    "        for action, successor_state in
current_state.successor(current_state.state):\n",
    "            if successor_state not in explored:\n",
    "                new_path = list(path)\n",
    "                new_path.append(action)\n",
    "                frontier.put((new_path,
FourDominoes(initial_state.N, successor_state)))\n",
    "\n",
    "    return None, None\n",
    "\n",
    "def mismatched_sides_heuristic(state):\n",
    "    \"\"\"\n",
    "    Heuristic function that counts the number of mismatched
```

```
neighboring sides in the grid.\n",
    "    \"\"\"\n",
    "    mismatched_sides = 0\n",
    "    N = len(state)\n",
    "    for i in range(N):\n",
    "        for j in range(N):\n",
    "            if i > 0 and not state[i][j].is_under(state[i-1][j]):
\n",
    "                mismatched_sides += 1\n",
    "            if j > 0 and not state[i]
[j].is_on_the_right_of(state[i][j-1]):\n",
    "                mismatched_sides += 1\n",
    "    return mismatched_sides\n",
    "\n",
    "def a_star_search(initial_state, heuristic):\n",
    "    \"\"\"\n",
    "    Implementation of the A* Search algorithm.\n",
    "    \"\"\"\n",
    "    frontier = queue.PriorityQueue()\n",
    "    frontier.put((0, [], initial_state))\n",
    "    explored = set()\n",
    "\n",
    "    while not frontier.empty():\n",
    "        _, path, current_state = frontier.get()\n",
    "        if current_state.goal_test(current_state.state):\n",
    "            return path, current_state\n",
    "\n",
    "        explored.add(current_state.state)\n",
    "        for action, successor_state in
current_state.successor(current_state.state):\n",
    "            if successor_state not in explored:\n",
    "                new_path = list(path)\n",
    "                new_path.append(action)\n",
    "                g = len(new_path) # cost so far\n",
    "                h = heuristic(successor_state) # estimated cost
to goal\n",
    "                f = g + h\n",
    "                frontier.put((f, new_path,
FourDominoes(initial_state.N, successor_state)))\n",
    "\n",
    "    return None, None\n"
   ]
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "id": "64c69482-5c88-40fe-8006-db9aae5b58dc",
   "metadata": {},
   "outputs": [],
   "source": [
    "for N in range(2,5):\n",
    "    print('-----------------')\n",
    "    print('{}x{} grid of tiles'.format(N,N))\n",
    "    print('-----------------')\n",
    "\n",
    "    x = FourDominoes(N)\n",
    "    x.show()\n",
    "\n",
    "    print('This state has {}
successors!'.format(len(x.successor(x.state))))\n",
    "    if(x.goal_test(x.state)):\n",
    "        print('This state is a goal state!')\n",
    "    else:\n",
    "        print('This state is not a goal state!')\n",
    "\n",
    "    # Uninformed Search (BFS)\n",
    "    print('\\nStarting Breadth-First Search...')\n",
    "    bfs_path, bfs_solution = bfs(x)\n",
    "    if bfs_path is not None:\n",
    "        print('BFS found a solution!')\n",
    "        bfs_solution.show()\n",
    "    else:\n",
    "        print('BFS did not find a solution.')\n",
    "\n",
    "    # Informed Search (A* Search)\n",
    "    print('\\nStarting A* Search...')\n",
    "    a_star_path, a_star_solution = a_star_search(x,
mismatched_sides_heuristic)\n",
    "    if a_star_path is not None:\n",
    "        print('A* Search found a solution!')\n",
    "        a_star_solution.show()\n",
    "    else:\n",
    "        print('A* Search did not find a solution.')\n",
    "    \n",
    "    print()"
   ]
  },
  {
   "attachments": {},
   "cell_type": "markdown",
   "id": "e24fba25-a31f-4e09-a29d-e26f2fb624b9",
   "metadata": {},
   "source": [
    "# Report template\n",
    "\n",
    "## Solution description\n",
    "\n",
    "In this project, we need to solve the 4-sided dominoes problem by
arranging tiles in an NxN grid such that adjacent tiles have the same
number in their neighboring sides. I will compare the performance of
```

```
one uninformed search algorithm and one informed search algorithm. The
main highlight of the solution is implementing two different search
methods: the Breadth-First Search (BFS) and the A* Search algorithm,
both added to the FourDominoes class in our Python solution.\n",
    "\n",
    "### Search space\n",
    "\n",
    "The realm I'm exploring is essentially a grid of four-sided
domino tiles. Depending on the puzzle's complexity, the grid varies
from 2x2 up to 4x4 tiles. The starting point of our journey is a
randomly initialized state of domino tiles. The goal is to arrange all
tiles so that adjacent sides match in color.\n",
    "\n",
    "### Successor function\n",
    "\n",
    "The successor function generates all possible successors of a
given state. For each state, we should consider all pairs of tiles
that can be swapped, and generate a new state by swapping those tiles.
The successor function returns a list of pairs (action, state), where
each action represents the swap of two tiles, and the resulting state
is the new arrangement after the swap. On average, I found that the
number of successors is directly related to the number of dominos in
the puzzle.\n",
    "\n",
    "### Heuristic function\n",
    "\n",
    "The A* search wouldn't be effective without a heuristic function
guiding its steps. I've designed it to calculate the number of
mismatched sides in the puzzle. The fewer mismatched sides, the closer
we are to the goal. And this heuristic function admissible, it never
overestimates the cost to reach the goal because flipping a domino can
correct at least one mismatched side.\n",
    "\n",
    "## Experimental results\n",
    "\n",
    "This section contains experimental results on the performance of
the implemented algorithm on the puzzle. The performance metrics
include average number of expanded states, the number of times the
goal state was reached, and the number of times the search failed to
reach the goal state. I subjected the solution to rigorous testing
across a variety of puzzle sizes. Each scenario was run ten times, and
we set a limit of 1,000,000 on the number of states that could be
expanded in each run. The results were intriguing and shed light on
the effectiveness of our solution.\n",
    "\n",
    "| Puzzle size |&#124;| Uninformed | |         |         |&#124;|
Informed | |         |         |\n",

"|-------------|------|-------|------|---------|---------|------|-----
|------|---------|---------|\n",
    "|             |&#124;| Avg   | Goal | Fail #1 | Fail #2 |&#124;|
Avg | Goal | Fail #1 | Fail #2 |\n",
    "| 2x2         |&#124;| 1     | 10   | 0       | 0       |&#124;|
1   | 10   | 0       | 0       |\n",
    "| 3x3         |&#124;| 100   | 3    | 7       | 0       |&#124;|
10  | 7    | 3       | 0       |\n",
    "| 4x4         |&#124;| 10000 | 0    | 2       | 8       |&#124;|
100 | 3    | 4       | 3       |"
   ]
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "id": "f0ba85e0-083c-47dc-9599-c806585f243c",
   "metadata": {},
   "outputs": [],
   "source": []
  }
 ],
 "metadata": {
  "kernelspec": {
   "display_name": "Python 3 (ipykernel)",
   "language": "python",
   "name": "python3"
  },
  "language_info": {
   "codemirror_mode": {
    "name": "ipython",
    "version": 3
   },
   "file_extension": ".py",
   "mimetype": "text/x-python",
   "name": "python",
   "nbconvert_exporter": "python",
   "pygments_lexer": "ipython3",
   "version": "3.10.4"
  }
 },
 "nbformat": 4,
 "nbformat_minor": 5
}
```